Discovering vulnerabilities using data-flow analysis and machine learning

Demonstrated for PHP applications

J. J. Kronjee

Student: 850638968
Date: 23/03/2018

Open Universiteit
www.ou.nl

DISCOVERING VULNERABILITIES USING DATA-FLOW ANALYSIS AND MACHINE LEARNING

DEMONSTRATED FOR PHP APPLICATIONS

by

J. J. Kronjee

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering

Student number:	850638968	
Course code:	IM9906	
Supervisor:	Dr. A. J. Hommersom	
Thesis committee:	Dr. A. J. Hommersom	Open Universiteit
	Dr. ir. H. P. E. Vranken	Open Universiteit

Open Universiteit
www.ou.nl

ACKNOWLEDGEMENTS

I would like to take this as an opportunity to thank everyone who has supported me in writing this thesis.

First of all, I would like to express my gratitude to my supervisors Dr. A.J. Hommersom and Dr. ir. Harald Vranken. You have been excellent advisers to me. Your valuable comments and suggestions as well as your critical questions have been truly helpful during this project. I hope I finally managed to nail down that definition of AUC-PR.

I would also like to thank my parents, who bought me my first computer and paid for internet dial-up without complaints (even though the bills may have been a bit higher than expected). It was during this period that I got interested in software development and security, which made me decide to start a career in IT. My parents are also the ones that motivated me to pursue a MSc degree, which, in retrospect, was not a bad idea.

I would also like to thank Trillian, my wife, for her support. I know that in the last two years I have spent much of our time together on this research, so thank you for stomaching my incoherent ramblings about heavily biased data sets, for your love, and for applying those missing Oxford commas (including the one in this sentence). I could not have done this without you.

Finally, I would like to thank our pets (Fenrir, Gwendolyn, and especially Rhys) for providing the soundtrack to my research.

Jorrit Kronjee

CONTENTS

List of Figures

List of Tables

SUMMARY

While software projects are ever growing in size, vulnerability discovery is still mostly done by manual analysis. Since manual analysis requires time and intimate knowledge of the system, new methods are required to assist analysts with vulnerability discovery.

Recently, researchers have started looking at using a combination of static code analysis and machine learning to find vulnerabilities. By extracting certain features (e.g. API symbols), code can be represented as vectors and the similarity between vectors can be determined to discover previously unknown vulnerabilities.

Studies in this area have mostly been focusing on static languages like C and C++ with a limited set of vulnerability types that are specific to these languages, while dynamic languages have not seen a lot of research. One possible reason that dynamic programming languages are understudied is that their features such as run-time source inclusion, weak typing, and implicit object creation, are all complications for static code analysis.

In this research, we have explored whether machine learning techniques and static code analysis can be used to detect vulnerable code in one specific dynamic programming language, namely PHP. The focus of the research has been on web application vulnerabilities, namely *SQL injection (SQLi)* and *Cross-Site Scripting (XSS)*, and the main contribution of this research is tooling that is able to detect vulnerable PHP code using a probabilistic classifier and features extracted with data-flow analysis.

To train and test our probabilistic classifiers we constructed a data set by combining mined samples from the *National Vulnerability Database (NVD)* and generated test cases from NIST's *SAMATE* project.

For the data set transformation, we used *control flow graphs (CFG)* to extract features from our data set. CFGs are often used in data-flow analysis. We defined and used three techniques (*Reaching Definitions*, *Constants as a feature*, and *Taint Analysis*) that leverage data-flow analysis for our feature extraction.

Once our data sets were transformed and our probabilistic models were created, our models were tested with the test set and evaluated by using the *AUC-PR* score to rate the performance of our models. The performance of individual features was measured using model-based ranking and the χ^2-test.

We also tested our tooling against other static code analysis tools and found that our tooling outperformed all of them for both the SQLi and XSS data sets. Finally, we have used our tooling to find previously unknown vulnerabilities and were able to find an SQL injection in a commonly used photo-sharing web application.

Our tooling, named *WIRECAML*, will be made available as open-source software under the MIT license.

1

INTRODUCTION

As we increasingly rely on software technology for all parts of our lives, efforts to secure these systems become more crucial. Unfortunately, the success of these efforts is repeatedly undermined by subtle flaws in implementations. A prominent example of such a flaw is the Heartbleed vulnerability [1] found in the cryptographic library OpenSSL. The library provides the basis for HTTP encryption on a large number of systems, but a single missing bounds check in its code turned into a disastrous data leak. In effect, attackers gained the ability to read sensitive information from an estimated 24%-55% of the most popular one million websites serving encrypted pages, while ironically, servers not offering encryption remained immune. This example highlights the central role the quality of the underlying program code plays for the security of computer systems.

In total, efforts for the discovery of these kinds of vulnerabilities result in the disclosure of around 5400 vulnerabilities per year on average, as measured over the period 2006 – 2015 [2]. A small number of these vulnerabilities are called zero-day vulnerabilities. These are undisclosed software vulnerabilities that attackers exploit to gain control or adversely affect software systems. They are known as *zero-day*, because once such a flaw becomes known, the software vendor has zero days to mitigate the issue.

Finding zero-day vulnerabilities has become increasingly commercialized by black markets where brokers connect buyers, such as government agencies, with exploit sellers. The average price for a zero-day exploit lies between $40,000 and $160,000 [3]. In 2015, the number of zero-day vulnerabilities that were discovered went up to 54, which is an increase of 125% compared to 2014 [4]. Given the price of these vulnerabilities, it is not surprising that the market has evolved to meet the demand.

To date, the vast majority of critical vulnerabilities are found by manual analysis of code by security experts. Examples are the previously mentioned Heartbleed vulnerability found in the OpenSSL library, but also the Shellshock [5] vulnerability in the GNU bash shell and the vulnerabilities CacheBleed [6] and DROWN [7], which were again found in the OpenSSL library. Vulnerability discovery is 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 to help analysts with vulnerability discovery.

Academic work in this area has mostly focused on the development of formal methods such as model checking and symbolic execution, which allow properties of the code to

be verified in an automated deductive fashion. Although these methods are powerful in a theoretical setting, in practice they have been shown to be hard to scale for the large software projects that we use today. In addition, instead of trying to assist the analyst with their work, most methods strive for full automation, which is a considerably more difficult and most likely futile task. For these reasons, the results of academic work have only had a limited role in real-world vulnerability discovery [8, 9]. Furthermore, most of the studies that have been conducted in this field have been focusing on static languages like C and C++ with a limited set of vulnerability types that are specific to these languages (buffer overflow, "use after free," memory disclosure, etc.). This is odd as 4 out of 10 languages in the TIOBE index [10] top 10 - an index that measures the popularity of programming languages - are considered dynamic. It is no surprise that more research in this area has been suggested [11].

One possible reason that dynamic programming languages are understudied is what separates them from static languages. Although there is no exact definition of what a dynamic programming language is, in general, a dynamic language ensures the safety of its abstractions at run-time as opposed to a static language which performs these checks at compile-time [12]. Dynamic language features like run-time source inclusion, weak typing, and implicit object creation are all complications for static code analysis [13, 14].

Recently, researchers have started looking at using machine learning [15–18] to find vulnerabilities based on models that can be learnt by a machine. By extracting certain features (e.g. API symbols), code can be represented as vectors and the similarity between vectors can be determined to discover previously unknown vulnerabilities. Other, albeit similar, approaches using graphs of abstract syntax trees or so-called 'code property graphs' [19] have also been shown to be successful.

This research focuses on the application of supervised machine learning techniques to discover vulnerabilities in software written in dynamic languages. In the next chapters, we introduce the reader to the field of vulnerability discovery and its techniques. We will also give a brief explanation of machine learning and how these techniques have been applied to vulnerability discovery in previous studies. With this background information at hand, we proceed to form our research question, explain the research method, and discuss how our research contributes to the field.

1.1. VULNERABILITY DISCOVERY

The Microsoft Security Response Center defines a security vulnerability as a weakness in a product that could allow an attacker to compromise the integrity, availability, or confidentiality of that product [20].

- *Integrity* refers to the trustworthiness of a resource. An attacker that exploits a weakness in a product to modify it silently and without authorization is compromising the integrity of that product.

- *Availability* refers to the possibility to access a resource. An attacker that exploits a weakness in a product, denying appropriate user access to it, is compromising the availability of that product.

- *Confidentiality* refers to limiting access to information on a resource in order to protect the information from disclosure to unauthorized parties. An attacker that ex-

> ploits a weakness in a product to access non-public information is compromising the confidentiality of that product.

The Internet Security Glossary [21] states that systems can have vulnerabilities during the three stages of a software life cycle: (a) vulnerabilities in design or specification; (b) vulnerabilities in implementation; (c) vulnerabilities in operation and management. For this proposal, we focus our efforts on vulnerabilities in design and implementation by concentrating on those flaws that are visible and fixable in program code. We will further limit our scope on those programs that can be considered web applications, i.e. applications that use a client-server architecture in which the client (or user interface) runs in a web browser. In the following subsections we will explore current techniques used for vulnerability discovery as well as provide some examples of web application vulnerabilities.

1.1.1. TECHNIQUES

Various techniques are used for vulnerability discovery in software: manual inspection, dynamic analysis (e.g. fuzzing) and static (code) analysis. Each of these techniques have their limitations. As source code projects grow, manual inspection becomes too labor-intensive to be feasible, which is why static and dynamic analyses are often used as aids in finding vulnerabilities. We will describe both techniques in more detail.

DYNAMIC ANALYSIS

Dynamic analysis is performed by testing an application during run-time and can be applied with or without the availability of source code. Dynamic analysis can be done by using a defined set of malicious patterns or by randomly generating test patterns. The former is usually called a *vulnerability scanner*, the latter is a *fuzzer*.

Vulnerability scanning is done by submitting various known malicious patterns in an automated fashion as input for the application and analyzing its output. If the output contains the pattern of a known vulnerability (e.g. an SQL injection attack results in an SQL parse error), the scanner will store both the input and the output for manual inspection. Vulnerability scanners are often used by penetration testers to do a quick test because they require little setup and output a convenient report that details the vulnerabilities found.

Although vulnerability scanners are popular [22], they suffer from severe limitations. Since a vulnerability scanner is testing dynamically with no knowledge of the code branches within the targeted application, it cannot guarantee complete code coverage. Furthermore, because it is dependent on a defined set of patterns, it will not be able to find any vulnerabilities outside of that known set. To combat this, fuzzers can be used to generate semi-random input data and then, similar to a vulnerability scanner, check for some undesirable result, e.g. a program crash. The main problem with fuzzing is that it generally only finds very simple faults. Because the input data it generates is random, it is likely that it does not pass some of the internal checks within an application, e.g. a checksum in a file that needs to be verified before the application runs any other logic. This is why some fuzzers can add instrumentation to the application to discover new code branches [23]. However, this still requires that the fuzzer actually finds the right input to satisfy the constraint. Despite these disadvantages, fuzzers have found vulnerabilities in many different applications [24] and are considered to be an invaluable tool by many vulnerability researchers.

Another approach in dynamic analysis to solve constraints and therefore maximize code coverage is a technique called *concolic testing*. Concolic testing, which is a port-

manteau of *concrete* and *symbolic*, combines symbolic execution with an SMT solver or constraint solver to generate new concrete inputs, i.e. test cases. Although concolic testers have had some commercial success [25], they too suffer from a number of limitations. Programs that are non-deterministic or where constraints cannot be solved in practice (e.g. having to invert a one-way hash function to satisfy a condition) can all lead to poor code coverage.

STATIC ANALYSIS

Unlike dynamic analysis, static analysis is performed by analyzing the code without running the application. The advantage of static program analysis is that the analysis can consider all paths in the program, not just those paths exercised at run-time. Typically, for static code analysis, a predefined set of rules is used to find vulnerabilities such as the use of insecure library functions, buffer overflows or insufficient data validation. This approach obviously limits the vulnerabilities that can be discovered to the rules that were defined.

One example of a basic static code analysis tool is flawfinder [26] from David Wheeler. Flawfinder examines C/C++ source code and reports possible security weaknesses sorted by risk level. It does this by checking the code for potentially dangerous functions like `strcpy()`, which does an unbounded copy of a string from *source* to *destination*. The function does not check whether the destination buffer is big enough to store the source string, which can easily lead to a buffer overflow if the programmer did not validate this beforehand. Tools like flawfinder do not actually check for a validation and leave it up to the user to verify. It is not hard to imagine that this leads to numerous false positives, which can be a little disheartening to the researcher who has to check each reported potential vulnerability manually.

To provide better results, many tools employ a technique called *taint checking* as part of the static analysis. The main idea behind taint checking is that any variable that can be modified (either directly or indirectly) by the user has to be considered *tainted* as it has the potential to become a security vulnerability. Variables that are derived from tainted variables become tainted as well (e.g. `welcome_txt = "Welcome " + name`).

Static code analyzers use taint checking by finding potentially vulnerable functions (so-called *sinks*) and then trace back their parameters to see if they were tainted. If the parameter is a constant variable set by the programmer, the analyzer will not report it, but if it believes it could be modified by the user, it will list it as a potential vulnerability.

For dynamic languages, static code analysis is more complicated due to the features found in these languages. These features are generally triggered during run-time. For instance, a feature such as run-time source inclusion will mean that the application will decide during run-time which source code will be loaded by the application. Although a static code analysis tool can reason whether the source code will be loaded, this will always remain an approximation.

1.1.2. EXAMPLES OF WEB APPLICATION VULNERABILITIES

As the focus of the research will be on web application vulnerabilities, this section will try to provide some understanding of the different kinds of web application vulnerabilities.

In the introduction of section 1.1 we have defined what a vulnerability does. The Open Web Application Security Project (OWASP) maintains a popular taxonomy (i.e. classification) for web application vulnerabilities [27]. The current version is from 2013 and distinguishes between 10 classifications.

In the following sections we will highlight two of those classifications: *Cross-Site Scripting* and *Injection* (in the form of *SQL injection*). These are commonly seen issues in web applications and can both be remedied by enforcing input validation.

CROSS-SITE SCRIPTING

OWASP describes cross-site scripting, commonly abbreviated to XSS, as a type of attack where an attacker uses a web application to send malicious code, generally in the form of a browser script, to a different end user. Since it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site.

There are two main categories of XSS attacks: *stored* and *reflected*. With stored attacks, the malicious script is permanently stored, such as in a database. The victim retrieves the script from the server when it requests the stored information. An example of code vulnerable to a stored XSS is shown in listing 1.1, which is a vulnerability that existed in versions of phpmyadmin prior to 4.2.6. The vulnerability was exploitable by using specially crafted MySQL table comments that included a malicious script. An example of a malicious script would be an AJAX call that sends the contents of `document.cookie` to a website that the attacker controls. The attacker could then use the session cookie to log in as the victim.

The development team fixed the vulnerability by applying `htmlspecialchars()` on line 2 to the tainted source variable.

```
1 $browse_table_label = '<a href="sql.php?' . $tbl_url_query
2     . '&amp;pos=0" title="' . $current_table['TABLE_COMMENT'] . '">'
3     . $truename . '</a>';
```

Listing 1.1: CVE-2014-4954 - XSS due to unescaped table comment [28]

With reflected XSS the malicious script is not retrieved from storage, but is instead reflected back by the server upon receiving a request, such that some of the input that was part of the request is included within the response. When a user is tricked into clicking on a malicious link, the injected code is included in the request that goes to the vulnerable web site, which sends the attack back to the user's browser. As with stored XSS, the browser will execute the script as the source it came from is considered trusted.

SQL INJECTION

Injection flaws occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization. SQL injections (SQLi) are a specific form of injection flaws that target SQL interpreters.

One example of an SQLi is shown in listing 1.2, which is a vulnerability that existed in versions of phpmyadmin 3.5.x before 3.5.8.2 and 4.0.x before 4.0.4.2. Due to a missing validation of passed POST parameters (represented in the listing as the variable `$scale`), it was possible to inject SQL statements that would run with the privileges of the control user. This gives read and write access to the tables of the configuration storage database, and possibly read access to some table of the *mysql* database if the control user had the necessary privileges. The vulnerability was fixed by adding a numeric validation check to the `$scale` variable.

```
$pmd_table = PMA_backquote($GLOBALS['cfgRelation']['db']) . '.' .
    PMA_backquote($GLOBALS['cfgRelation']['designer_coords']);
$pma_table = PMA_backquote($GLOBALS['cfgRelation']['db']) . '.' .
    PMA_backquote($cfgRelation['table_coords']);

if (isset($exp)) {

  $sql = "REPLACE INTO " . $pma_table . " (db_name, table_name,
    pdf_page_number, x, y) SELECT db_name, table_name, " . $pdf_page_number
     . ", ROUND(x/" . $scale . ") , ROUND(y/" . $scale . ") y FROM " .
    $pmd_table . " WHERE db_name = '" . $db . "'";

  PMA_query_as_controluser($sql,TRUE,PMA_DBI_QUERY_STORE);
}
```

Listing 1.2: CVE-2013-5003 - SQLi due to missing validation checks [29]

1.2. MACHINE LEARNING

In the following sections, we will describe various classifiers in the field of machine learning. Classifiers are used to identify to which class an observation belongs, on the basis of a training set containing observations whose classes are known.

The goal is to create a function with the training set that will predict the correct class based on a set of feature values. Features are parameters that make up the observation. For instance, if we want a classifier distinguishing between a human and a cat, then things like heart rate, number of legs, and IQ could all be used as features.

Classification is considered to be part of supervised learning, that is learning where a training set of labeled observations is available. The corresponding unsupervised procedure is generally known as clustering, and involves grouping unlabeled data into categories based on certain criteria.

Probabilistic classifiers are among the most popular classifiers used in the machine learning community and increasingly in many applications. They are able to predict, given a sample input, the most likely class that the sample belongs to based on a probability distribution over a set of classes.

In chapter 1.3 we will explain how probabilistic classifiers can be used to discover vulnerabilities by reviewing two different studies where machine learning and vulnerability discovery were combined. In this section, we will give a high-level introduction to four probabilistic classifiers that were used in these studies, namely *Decision Tree*, *Random Forest*, *Naive Bayes* and *Logistic Regression*. In addition, we will give an introduction to *Tree-augmented Naive Bayes* which is a generalization of Naive Bayes.

1.2.1. DECISION TREES AND RANDOM FOREST

Decision trees have a flowchart-like structure that classifies observations of a data set by going down the tree from the root to a leaf node that will provide the classification. Each node in the tree has a test of some feature of the observation and each branch descending from that node corresponds to one of the possible values of that feature.

Decision trees can be further extended by predicting the probability of a class, which is determined by the fraction of training samples of the same class in a leaf.

An example of a decision tree is given in figure 1.1 which shows a tree that decides

whether or not you should go outside. In this case, you should not go out when it is raining.

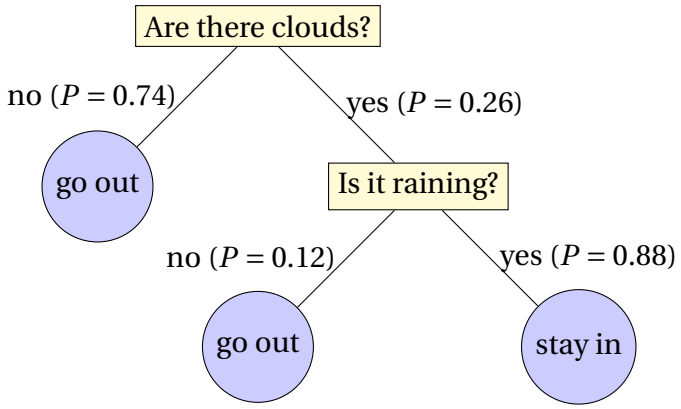

Figure 1.1: Decision tree for whether to go outside

Decision trees are particularly well-suited for data that contains observations that are represented by features that have discrete output values. In figure 1.1 we have used *yes* and *no* as output values, but it is easy to see that this can be extended to other values. For instance, we could introduce a feature *humidity* that has three values: *high*, *normal*, and *low*.

Decision tree learning is the construction of a decision tree from a data set by means of an algorithm. Algorithms for constructing decision trees usually work top-down, by choosing a variable at each step that best splits the remaining set of items, where the definition of 'best' depends on the algorithm. Notable decision tree algorithms are ID3, C4.5, CART, CHAID, and MARS [30–32].

Decision tree algorithms are likely to create over-complex trees that overfit their training set. One approach to combat this is using a technique called *pruning*. Pruning reduces the size of decision trees by removing sections of the tree that provide little power to classify observations. For instance, if the raininess in figure 1.1 is not a strong predictor of whether we should stay in, it could be replaced by a leaf node, creating a simpler, more generalized decision tree.

Random (decision) forests grow a collection of decision trees with the training set and output a class that is the mode of the classes determined by the collection. Random forests use a technique called *feature bagging* (or *feature **b**ootstrap **ag**gregation*). Feature bagging is a way to decrease the error due to variance[1] by generating additional bootstrap samples for training using random subsets of features with repetitions.

If we consider a training data set Z and a prediction function $f(x)$, we can create a prediction function $f^{*b}(x)$ with each bootstrap sample Z^{*b}, where $b = 1,2,\ldots,B$ and B is the size of the collection of bootstrap samples. The classification can then be determined by using the majority vote over B decision trees.

$$f_{\text{bag}}(x) = \text{Majority}(\{f^{*1}(x), f^{*2}(x), \ldots, f^{*B}(x)\})$$

The reason random subsets of features instead of subsets of the data set are used is to reduce correlation of the trees: if subsets of the data set were used as bootstrap samples

[1]The error due to variance is the amount by which a prediction for the given training set differs from the expected predicted value over all the training sets derived from the same distribution.

and a few features are very strong predictors, these features will be selected in many of the decision trees, causing them to become correlated [33]. Feature bagging is used to reduce this correlation.

1.2.2. NAIVE BAYES

To explain the existence of *Naive Bayes*, we first need to explain the limitations of a Bayes classifier. Bayes' rule describes the probability of an event, based on conditions that are possibly related to that event. So if we consider that A and B are events, $P(A)$ and $P(B)$ the probabilities of these events occurring, and $P(B|A)$ the conditional probability of event B given event A, we can state Bayes' rule by the following equation:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

with $P(B) \neq 0$.

Let us now consider a supervised learning problem where we want to approximate target function $f : X \rightarrow Y$ (or $P(Y|X)$). In this scenario, Y is a boolean-valued variable and X is a vector containing n boolean attributes ($X = \langle X_1, X_2, \ldots, X_n \rangle$). We write x for $X = x$. The Bayes' classifier is described by:

$$h_{\text{Bayes}}(x) = \underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y | X = x)$$

The problem is that estimating $P(Y|X)$ requires a lot of samples as we need to determine the probability for each particular configuration of X for each particular Y. Since X is a boolean vector of size n and Y can take on 2 distinct values, they can each take on 2^n and 2 possible values respectively. It is clear to see that this is unrealistic in most practical machine learning settings. For example, if X is a vector containing 50 boolean features, we would need to estimate the probabilities for more than 2 quadrillion parameters.

The Naive Bayes classifier reduces this complexity by making the (rather naive) assumption that given the label, the features are independent of each other. That is,

$$P(X = x | Y = y) = \prod_{i=1}^{n} P(X_i = x_i | Y = y)$$

With this assumption, it becomes clear that we now only need estimations for $2n + 1$ parameters (including the class prior), which is a dramatic reduction. By using our earlier definition of the Bayes classifier, we can now also define the Naive Bayes (NB) classifier:

$$\begin{aligned} h_{\text{NB}}(x) &= \underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y | X = x) \\ &= \underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y) P(X = x | Y = y) \\ &= \underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y) \prod_{i=1}^{n} P(X_i = x_i | Y = y) \end{aligned}$$

Even though this assumes that all features are independent given the class label, it has been shown that Naive Bayes performs quite well in practice, even when strong feature dependencies are present [34].

It is important to note that calculating class probabilities this way tends to push predicted values towards 0 or 1 [35] and should not be used as accurate estimates. However, if the application is to simply determine which class is most likely correct, then even if the estimates are grossly inaccurate, Naive Bayes will make the right decision so long as the correct class is more probable than the other class.

Because the Naive Bayes classifier is less computationally intense than some other classifiers, it can be used to quickly discover relationships between features. It is used often to do initial exploration of data after which the results can be applied to create more accurate models with more sophisticated algorithms.

1.2.3. TREE-AUGMENTED NAIVE BAYES

Tree-augmented Naive Bayes (TAN) [36] is a variation of Naive Bayes where the assumption of attribute independence is dropped. Instead, a tree structure is used where each attribute only depends on the class and one other attribute from the feature set. A maximum weighted spanning tree that maximizes the likelihood of the training data is used to perform classification.

Figures 1.2 and 1.3 shows Bayesian network representations of the types of model that NB and TAN respectively create.

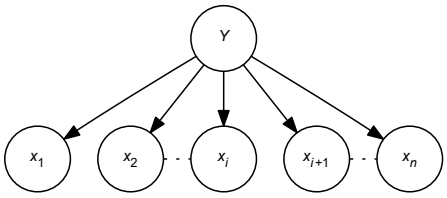

Figure 1.2: Naive Bayes network

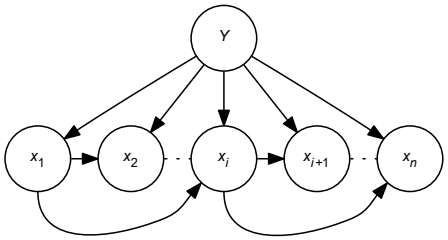

Figure 1.3: TAN network

Building the tree is done using the following steps.

1. Compute the *mutual information* between each pair of features
2. Build a complete undirected graph in which the vertices are the features $X_1, X_2, \ldots, X_n$. The edges are weighted according to the pairwise mutual information.
3. Build a maximum weighted spanning tree.
4. Transform the resulting undirected tree to a directed one by choosing a root variable and setting the direction of all edges to be outward from it.
5. Construct a TAN model by adding an arc from the class variable to all other variables

The parent of each feature X_i (excluding Y) is indicated as $\pi(X_i)$ and the parent of the class variable is ø. It assumes that features are independent given the class and their parents and classifies a single sample with feature values $x_1, x_2, \ldots, x_n$ by using:

$$\underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y) \prod_{i=1}^{n} P(x_i | y, \pi(x_i))$$

Due to the dropped attribute independence assumption, TAN considerably reduces the bias of naive Bayes at the cost of an increase in variance. Empirical results [36] show that it substantially reduces the number of misclassifications compared to naive Bayes on all data sets examined more often than not.

1.2.4. LOGISTIC REGRESSION

A logistic regression model or logit model [37] is a linear model for binary classification. In linear regression, we have a linear sum, where the output could be any $\mathbb{R}$. In logistic regression, this linear sum is restricted by the logistic function, also called the *sigmoid function*. It is an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, which is why it is particularly suited for outputting probabilities of a binary class. We define the sigmoid function as follows:

$$g(z) = \frac{1}{1+e^{-z}}$$

where e is Euler's number. Its graph looks like this:

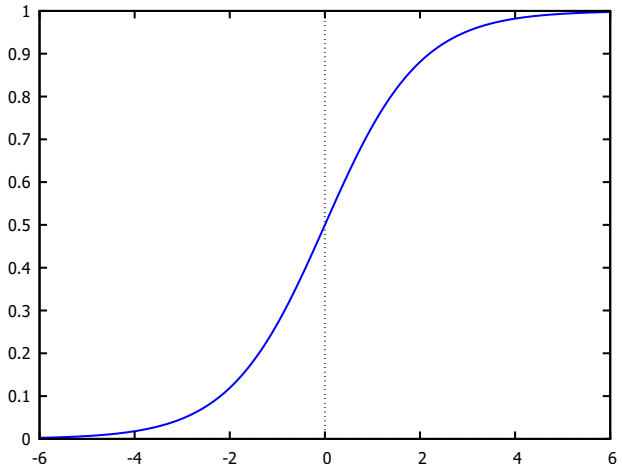

Figure 1.4: Sigmoid function $g(z) = \frac{1}{1+e^{-z}}$

In logistic regression z is defined as the linear sum of weights and features, such that

$$z = w_0x_0 + w_1x_1 + ... + w_nx_n = \sum_{j=0}^{n} w_jx_j = W^TX$$

where $W = \langle w_0, w_1, \ldots, w_n \rangle$ is the vector of weights to be estimated and $X = \langle x_0, x_1, \ldots, x_n \rangle$ are the features. This makes our logistic regression function:

$$\phi(X) = g(W^TX) = \frac{1}{1+e^{-W^TX}}$$

To summarize, we still got a linear model but it is modeling the probabilities on a non-linear scale. To estimate the weights, a maximum likelihood estimator can be used.

Let us assume that:

$$P(y = 1|X;W) = \phi(X)$$
$$P(y = 0|X;W) = 1 - \phi(X)$$

Or more compactly:

$$P(y|x;W) = (\phi(X))^{y}(1-\phi(X))^{1-y}$$

Assuming we have m observations, we can then write down the likelihood of the parameters as

$$L(W) = \prod_{i=1}^{m}(\phi(X_i)^{y_i}(1-\phi(X_i))^{1-y_i}$$

The goal is to find the set of weights W where we maximize L considering all m observations. We use an iterative algorithm called *Gradient Ascent* to find the maximum L. By using the partial derivative of L, Gradient Ascent takes steps proportional to the positive of the gradient, such that one approaches a local maximum of L.

Logistic regression can make use of one or more features that may be either continuous or categorical. Logistic regression was developed by statistician David Cox [38] for the purpose of modeling trials where the outcome is binary.

Logistic regression is used in various fields, such as the medical field, social sciences and machine learning. For instance, logistic regression could be used to predict coronary heart diseases using a vector of covariates, including age, blood cholesterol level, systolic blood pressure, relative weight, blood hemoglobin level and smoking [37].

1.3. MACHINE LEARNING AND VULNERABILITY DISCOVERY

A recent survey from Ghaffarian et al. [39] split work in the area of machine learning and vulnerability discovery into three main categories:

- *Vulnerability Prediction Models based on Software Metrics* utilize data mining, machine learning, and statistical analysis techniques to predict vulnerable software artifacts (such as source code files, object oriented classes, binary components, etc.) based on common software engineering metrics. The main idea of these approaches is borrowed from the field of software quality and reliability assurance, where limited resources for software testing and verification, demands a guiding model to enable more efficient software testing plans.

- *Anomaly Detection Approaches* refer to the problem of finding patterns in data that do not conform to the normal and expected behavior; often referred to as *anomalies* or *outliers*. In the context of software quality assurance, anomaly detection methods are aimed to identify software defects by finding locations in source code that do not conform to the usual or expected code patterns for APIs. A simple example of such API usage patterns are the function pair `malloc()` and `free()` in C.

- *Vulnerable Code Pattern Recognition* analyzes and extracts features from program source code, where it strives to extract models and patterns of vulnerable code. The common theme in this category is to gather a large data set of vulnerable samples, process them to extract feature vectors from each sample, and utilize 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; including: conventional code parsers, static data-flow

and control-flow analysis, dynamic analysis, text mining on program source code, and so on.

In Related Work we will list examples of research in these categories.

In the following two sections we discuss two papers from earlier research where machine learning is used to discover vulnerabilities. Although they focus on different feature sets, in both cases supervised learning with probabilistic classifiers is used. In section 1.4 we will draw our own conclusions from these papers and propose an alternative feature set for our research.

1.3.1. CODE METRICS

Shin et al. [16] showed that there is a correlation between *CCD metrics* and vulnerable code, where CCD metrics are defined as complexity, code churn (i.e. the number of lines added, deleted, or modified from one version to the next), and developer activity. They argued that complexity can make code difficult to understand and to test for security. Frequent or large amounts of code change can introduce vulnerabilities. Poor developer collaboration can diminish project-wide secure coding practices. We will describe these three metrics in a bit more detail.

For complexity metrics, Shin et al. hypothesize that (1) vulnerable files have a higher intrafile complexity than neutral files, that (2) vulnerable files have a higher coupling than neutral files, and that (3) vulnerable files have a lower comment density than neutral files. Intrafile complexity is defined as a combination of lines of code and cyclomatic complexity metrics of a file. Coupling is defined as the fan-in and fan-out metrics[2] and lastly, the comment density is the ratio of lines of comments to lines of code.

For code churn metrics, they hypothesize that vulnerable files have a higher code churn than neutral files. Code churn can be counted in terms of the number of check-ins into a version control system and the number of lines that have been added or deleted by code change.

For developer activity, Shin et al. create a developer network, which looks at how developers within a project are connected to each other. In this developer network, two developers are considered connected if they have both made a change to at least one file in common during the period of time under study. The result is a graph where each node represents a developer and edges are based on whether or not they have worked on the same file during the same release. Central developers are developers that are well-connected to other developers relative to the entire group. This allows them to define their first hypothesis for developer activity, which is that vulnerable files are more likely to have been worked on by noncentral developers than neutral files.

This network also allows the authors to define clusters. A file that is between two clusters was worked on by two groups of developers, and those two groups did not have many other connections in common. This helps to define their second hypothesis which is vulnerable files are more likely to be changed by multiple, separate developer clusters than neutral files.

[2]The authors use a particular definition of these metrics. They define fan-in as the number of inputs to a function, such as parameters and global variables. Likewise, fan-out is defined as the number of function calls and global variable assignments within the function.

For the third hypothesis a contribution network is created which is a graph with two types of nodes: developers and files. An edge exists where a developer made changes to a file, where the weight is equal to the number of check-ins that developer made to the file. If a file has high centrality, then that file was changed by many developers who made changes to many other files. This is referred to as an "unfocused contribution". Files with an unfocused contribution would not get the attention required to prevent the injection of vulnerabilities. Therefore: their third hypothesis is that vulnerable files are more likely to have an unfocused contribution than neutral files.

In total the authors put together a list of 28 metrics for their hypotheses. They retrieved these metrics from two large projects: the Mozilla Firefox web browser and the Linux kernel and tested whether they supported their hypotheses. At least 24 of the 28 metrics supported the hypotheses and discriminate successfully between vulnerable and neutral files for both projects.

Knowing that these metrics could be used as indicators, the authors used machine learning techniques to see whether they could predict vulnerabilities. Logistic regression and four other classification techniques (J48 decision tree, Random forest, Naive Bayes and Bayesian network) were used to predict vulnerable files. A file is classified as vulnerable when the outcome probability is greater than 0.5.

Overall, 80 predictions were performed across all eight releases (R4-R11) of Firefox and 100 predictions for the Linux kernel. In the univariate predictions, they identified the metrics that supported the hypotheses in at least 50 percent of the total predictions. For Firefox, five of the 28 metrics met the criteria and only one of the 28 metrics for the Linux kernel did; however, relying on a single metric should not be recommended.

In multivariate predictions, the models using code churn, developer activity, and combined CCD metrics supported the hypotheses in over 50 percent of the total predictions for both projects. Even though the models using complexity metrics supported the hypotheses for Firefox, none of the predictions were successful for the Linux kernel, which indicates that the effectiveness of complexity metrics to predict vulnerabilities is low. History metrics such as code churn and developer activity metrics provided higher prediction performance than complexity.

Given these results, it is unclear whether CCD metrics can really be used for vulnerability discovery. The authors themselves conclude that considering the large size of the two projects, the quantity of files and the lines of code to inspect or test based on the prediction results are still large.

1.3.2. TEXT MINING BUG DATABASES

Instead of using CCD metrics, Wijayasekara et al. [18] propose to use bug databases to search for so-called Hidden Impact Bugs (HIBs). HIBs can be defined as vulnerabilities identified as such only after the related bug had been disclosed to the public. These software bugs are disclosed to the public via bug databases, before being identified as a vulnerability. Because the security implication of the bug has not been correctly identified, the fix may not be applied in a timely fashion.

In order to find vulnerabilities, they use text mining techniques that utilizes the textual description of the bugs that were reported to publicly available databases. Their methodology was tested on Linux bug reports that were reported to the Redhat Bugzilla bug database [40] within the time period from January 2006 to April 2011, and Linux kernel vulnerabili-

ties that were reported in the MITRE CVE [41] database during the same time period. HIBs were identified as vulnerabilities that had at least 2 weeks of impact delay, where the impact delay was defined as the time between the moment of public disclosure of the bug via a patch to the time a CVE was assigned in the MITRE database.

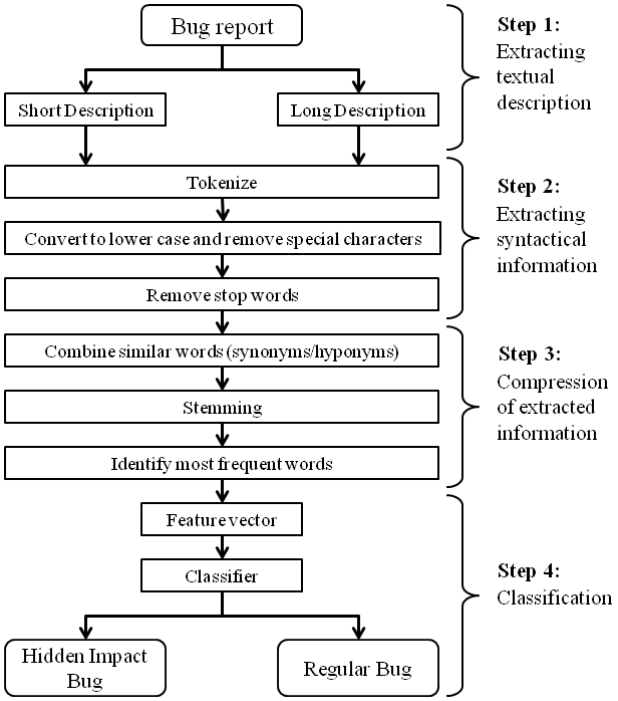

Figure 1.5: Vulnerability identification using bug reports

Figure 1.5 shows the process that utilizes the textual description of reported bugs to generate a feature vector that will be used by a classifier. The overall process is separated into four major steps.

In **Step 1** two descriptions are extracted: (1) the title provided by the reporter that is around 5 - 10 words in length and (2) a more detailed description of the bug which may include how to recreate it, e.g. code snippets, memory dumps, etc.

In **Step 2**, using text mining techniques, the most important and recurring syntactical information is extracted from the bug descriptions by representing them as a *bag-of-words* where we store the number of times each unique keyword is found. This disregards numbers, special characters and *stop words* in the English language. Stop words include pronouns ("I", "he", "she"), articles ("a", "an", "the"), prepositions ("after", "to", "but"), conjunctions ("and", "but", "when"), and other frequently appearing words. These words carry very little information when taken out of context, so can be disregarded. Thus a document D_i containing n unique words, where t_{ij} is the number of times term t_j appeared in document i, can be represented as:

$$D_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\}$$

From this, we can construct a Term-Document Matrix (TDM) of size $N \times M$ where N is the number of documents and M the number of unique words.

$$TDM = \begin{bmatrix} D_1 = \{t_{11}, t_{12}, \ldots, t_{1M}\} \\ \ldots \\ D_N = \{t_{N1}, t_{N2}, \ldots, t_{NM}\} \end{bmatrix}$$

The main problem faced when using the TDM is, as the number of documents (N) increases, the number of unique words (M) also increases. This results in a large matrix that can be extremely sparse, since many of the unique words might not appear in most of the documents.

To combat this, compression of the extracted information is performed in **Step 3**. The first stage of compression is identifying and combining synonyms into a single dimension

in the TDM. The second stage in the compression step of the text mining process is deconstructing words that have been transformed, for example by pluralizing or by adding a gerund, into their basic form. This process is called stemming. The third and final stage of the compression step is further reducing the TDM dimensionality by discarding keyword sets that appear in less than P% of the documents.

Finally, in **Step 4**, a feature space is created from the extracted set of keywords. Each dimension of the feature space consists of a set of words that carry similar information. This feature vector is used by a classifier to perform the final classification whether a given bug is a potential HIB or a regular bug.

Three different classifiers were tested on the textual information extracted from bug reports: Naive Bayes (NB), Naive Bayes Multinomial (NBM), and Decision Tree (DT). The NB classifier only considers the presence or absence of a keyword in a document, whereas the NBM classifier considers the number of times each keyword occurred in each document.

The classification was performed using 10-fold cross validation. Naive Bayes classifier showed the best True Positive (TP) rate (88%), however, the True Negative (TN) rate was low (46%). Similarly the decision tree had a very low TP rate (28%) but the highest TN rate (99%). For Naive Bayes Multinomial, both the TP rate as well as the TN rate were higher (78% and 90% respectively). The authors argued that, although these results may be relatively low, they were still better than a random guess (1.2%).

For all three classifiers, the authors also calculated the Bayesian detection rate (BDR), which is done with the following equation:

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

The BDR of DT (0.40) showed that more than a third of bugs that were classified by DT as vulnerabilities were actual vulnerabilities, despite the low TP rate. Similarly, the NBM classifier reported a BDR of 0.09, which means that just under 1 out of 10 bugs classified as vulnerabilities were actual ones. Due to the very high False Positive (FP) rate of the NB classifier, only 1 out of 50 bugs that were classified as vulnerabilities were actual vulnerabilities.

1.3.3. AN ALTERNATIVE APPROACH

As we have now seen, researchers have had some success using metadata like code churn, developer activity, and bug reports to train their models, rather than examining the code itself.

However, these techniques require the availability of such metadata and depend upon its quality. A lot of open-source projects only have one maintainer which may make a measure like developer activity a useless indicator. Code churn is dependent on the maturity of the project (i.e. the number of features that are still left to implement) and this can differ from project to project. The same applies to the quality of bug reports.

It is also unclear from the research by Shin et al. and Wijayasekara et al. why certain probabilistic classifiers perform much better on a data set than other classifiers given the same features. Finally, neither study reports on new vulnerabilities that were found using their method.

Consequently, even though this metadata might serve as a good indicator that something could contain a vulnerability, only the code itself can give conclusive proof that a

vulnerability exists. An example of research where the code was used as the data source is Yamaguchi et al. [17], wherein they propose to use *abstract syntax trees (AST)* that are extracted directly from the source code to find vulnerabilities. An abstract syntax tree is an ordered tree where the inner nodes represent operators (e.g. additions or assignments) and leaf nodes correspond to operands (e.g. constants or identifiers). An example of an AST (from Yamaguchi et al. [19]) is shown in figure 1.6 with the related code shown in listing 1.3.

```
1 void foo()
2   {
3     int x = source();
4     if (x < MAX)
5     {
6       int y = 2 * x;
7       sink(y);
8     }
9   }
```

Listing 1.3: Sample code for AST

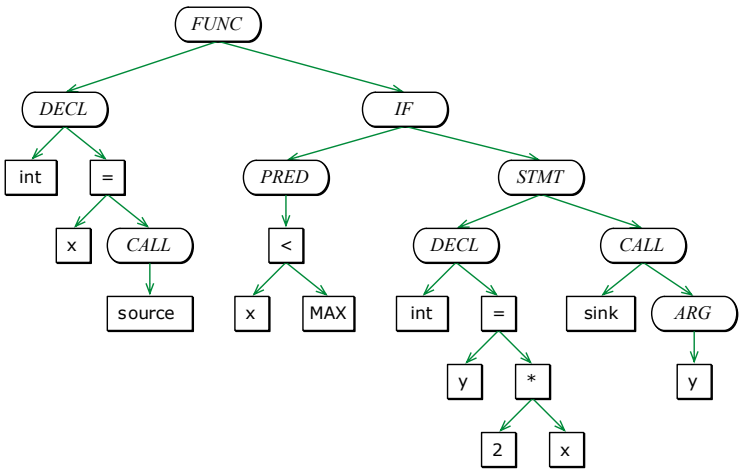

Figure 1.6: Abstract syntax tree for listing 1.3

Yamaguchi et al. translated AST graphs into vectors and then used a natural language processing technique called *latent semantic analysis* to find similarities between known vulnerabilities and potential vulnerabilities in the code.

Although Yamaguchi et al. did manage to find new vulnerabilities in various open-source projects using this method, the authors did realize the limitations of their research as they concluded in their paper that their method only considers the statistics of the source code rather than its true semantics. Nonetheless, they believed that this method could still be beneficial as it could discover potential vulnerabilities.

1.4. PROPOSED FEATURES

Based on the previous research of Yamaguchi et al. as outlined in the previous section (1.3.3), we propose to use the AST as the source for our feature set. We believe that the AST could provide a lot of information about the code that would prove useful, such as whether a token is a function or a variable and which functions were applied to a variable. Using ASTs for our features also means that our research falls under the third category, namely *Vulnerable Code Pattern Recognition*, which we had defined in the introduction of section 1.3.

We can use taint analysis to detect both SQLi and XSS vulnerabilities. Let us consider the example from listing 1.2 again. An SQL injection was possible because the function `PMA_query_as_controluser` was given a tainted parameter called `$scale`. In order for an SQL injection to work, there are three conditions that need to be met:

1. the function needs to be a potentially vulnerable function (PVF)
2. the variable needs to be tainted, i.e. not sanitized by a sanitization function
3. the variable provided to the function needs to contain data from a source

Some parts that are needed to check these conditions are readily available in the AST. For instance, from the AST we can determine that the variable `$scale` is being used as a parameter for a call to function `PMA_query_as_controluser`. This means that we can provide the presence or absence of variables and functions as part of the data set to our probabilistic classifiers. It is not hard to imagine that by applying this method our machine learning algorithm will learn which function is likely to be a PVF given enough examples. This will allow us to check for condition 1.

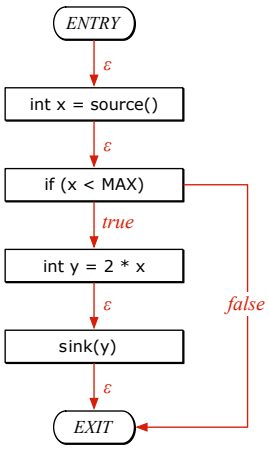

Figure 1.7: Control flow graph for listing 1.3

For condition 2, to check whether a function that is applied to a variable is a sanitization function, we can employ the same mechanism as for PVFs. Given enough examples, the algorithm should be able to figure out which functions are likely to affect the outcome if a line is vulnerable or not.

To determine if both functions were applied to a variable, we can construct a *control flow graph (CFG)*, such as figure 1.7, from the AST. By considering the entire preceding execution path, we can use the presence or absence of functions in that path as binary features.

For the final condition, the AST can provide us information on if the variable was set using a literal, such as a number or a string. In that case, the data is from a trusted source and the variable is untainted. If the variable was set using a source such as another variable or function however, and this source was not a literal, the variable must be considered tainted and anything that is set using this variable must be considered tainted as well. Using this method we can determine if a variable contains data from a source.

We have now established that ASTs can be used as a data source to discover SQLi and XSS vulnerabilities. Our challenge lies in finding data transformations that create the most predictive power and can still be used by probabilistic classifiers. This is what our research will entail.

1.5. THESIS OUTLINE

This thesis consists of six chapters, of which five remain and are organized as follows. In chapter 2 we will address the research design by defining the research questions and giving a high-level overview of our research method. Our research will start off by describing the data sets in chapter 3, including statistical analysis and procedures for extracting vulnerable code. Using the data sets, we will describe the way we can extract features using the generated ASTs in chapter 4. In particular, we will be extracting features commonly used in *Data-Flow Analysis*. Furthermore, we will rank these features on usefulness. We will proceed to evaluate various probabilistic classifiers by comparing models in chapter 5. The best performing model will be compared to other static code analyzers. We will also try to discover unknown vulnerabilities using our model. Finally, in chapter 6 the limitations of the presented work are discussed, and conclusions are drawn. We will close off by discussing possible directions for future research in the area of ML-based vulnerability discovery.

2

RESEARCH DESIGN

2.1. RESEARCH QUESTION

To help vulnerability researchers with the ever increasing complexity of software projects, new approaches for assisted vulnerability discovery are needed. Dynamic programming languages have not received a lot of attention in academic studies using static code analysis due to their dynamic nature.

Various studies [15–19] have shown that machine learning can be used to find vulnerabilities in an automated fashion. Our research aims to extend on these studies by employing machine learning techniques in the field of vulnerability discovery. The research question we will try to answer is:

Can machine learning with features extracted from control flow graphs be used for vulnerability detection in software written in a dynamic language, such as PHP?

To answer this question, we divide the question up into the following subquestions:

- **RQ1**: *When using the same feature set, how does the performance of various probabilistic classifiers compare?*

  In section 1.2 we have defined various probabilistic classifiers (Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Tree Augmented Naive Bayes) that we will use for comparison. The *AUC-PR* score is used to rate the performance of our models. AUC-PR works well in cases where the data set is highly skewed [42], which we expect to be the case as we believe that most of the code will be non-vulnerable.

  AUC-PR works as follows: consider that Z is a random variable indicating the probability assigned to an example by the classifier, Y is a random variable that indicates the true label for the same example, and c is a threshold value. We can then define the PR-curve as the following set of coordinates:

  $$PR(c) = \{(Recall(c), Prec(c)), -\infty < c < \infty\}$$

  with precision ($Prec(c) = P(Y = 1|Z > c)$) and recall ($Recall(c) = P(Z > c|Y = 1)$). AUC-PR is then the area under the PR-curve, i.e. the integral.

- **RQ2**: *Which features extracted from the CFG perform best for vulnerability detection?*

  The features that we will be extracting are functions, constants, and whether or not the sample is tainted. Feature performance can be measured by applying univariate statistical tests, such as the χ^2-test, which will give a low ranking to those features that are the most likely to be independent of class and therefore irrelevant for classification.

  In addition, we can use model-based ranking to show the usefulness of extracting constants and the taintedness of a sample.

- **RQ3**: *By using machine learning and features extracted from control flow graphs, can we correctly classify Injection and Cross-Site Scripting vulnerabilities as described by OWASP [27]?*

  To limit our scope, we will focus our efforts on the two vulnerability types that we have described before (see section 1.1.2). These two vulnerability types can both be discovered using taint analysis. Other vulnerability types will most likely require different features to discover. For instance, OWASP's *Insecure Direct Object References* refers to an attacker being able to directly access a resource where access should have been restricted. This would require our model to understand which resource is restricted, which is not relevant for Injection or Cross-Site Scripting attacks.

- **RQ4**: *Can our classifier be used to find previously unknown vulnerabilities?*

  The goal of this research is to create a classifier with a high precision and recall. As part of the testing procedure, we will be verifying whether positives are false or real. It is uncertain whether we will be able to discover any real positives, as some projects simply might not contain any new vulnerabilities of the type we are looking for. Nevertheless, it is essential to verify positives as false positives also impact the *precision* ($Prec = \frac{TP}{TP+FP}$).

2.2. RESEARCH METHOD

To answer the question whether probabilistic classification to find vulnerabilities can be considered useful, we require software code with known vulnerabilities as a data set. We will use several popular open-source software (OSS) projects in which vulnerabilities have been found and subsequently fixed, as these are the easiest to obtain. To identify these projects, we will use the National Vulnerability Database (NVD) [2]. The NVD will also provide the vulnerability classification which we will require for our classifier. In addition, we will be using the samples from NIST's SAMATE [43] data set which consists of generated vulnerable and non-vulnerable test cases.

To be able to distinguish between vulnerable and non-vulnerable code snippets, the software patches that are linked to the vulnerability reports will be used to identify vulnerable lines of code. Any other code will be considered non-vulnerable. Finding these patches and normalizing them will require manual work as the NVD and other vulnerability databases do not use a standard format for patches but instead link directly to project's source code repository (if available). For this reason, we will limit our scope to only a small number of projects.

Project selection will be done based on the following criteria:

1. The project needs to be written in PHP
2. The project needs to have had one or both of the vulnerabilities of the types defined in RQ3. Preferably, there are multiple instances of the vulnerability types available within the project. In some cases, one patch can fix multiple instances.
3. Patches for these vulnerabilities need to be publicly available.

Once a data set is created, we divide the set in three subsets: a training, a tuning, and a testing set. The training set will be used to train the models and the tuning set to tweak the model parameters. It is important that the testing set is kept separate until we are ready to evaluate the model because using it prematurely might cause us to accidentally overfit the model on the test set and skew the results.

For feature extraction, we require a parser that is able to convert the code into an AST. To extract these ASTs, we will use *phply* [44], a PHP parser for Python that can build ASTs. The subsequent transformations using *data-flow analysis* will be developed as described in chapter 4. These transformations will be applied separately to our tuning and test sets.

Although our research is focused on PHP, we will try to select language-neutral features, so that our approach can be applied to other dynamic languages in later research.

We will write our tooling in Python for which a plethora of machine learning libraries have already been written. We will be using the scikit-learn [45] library which contains various probabilistic classifiers as well as tools for feature extraction and normalization, choosing features, model selection and validation.

Once our probabilistic models have been created, and assuming that all our models are able to output a class probability, our models will be tested with the test set and evaluated by using the *AUC-PR* score to rate the performance of our models.

A high-level flow of our research method is shown in figure 2.1. It should be clear that picking features and training the model is an iterative process, as the model's performance will make it apparent which features and transformations work best.

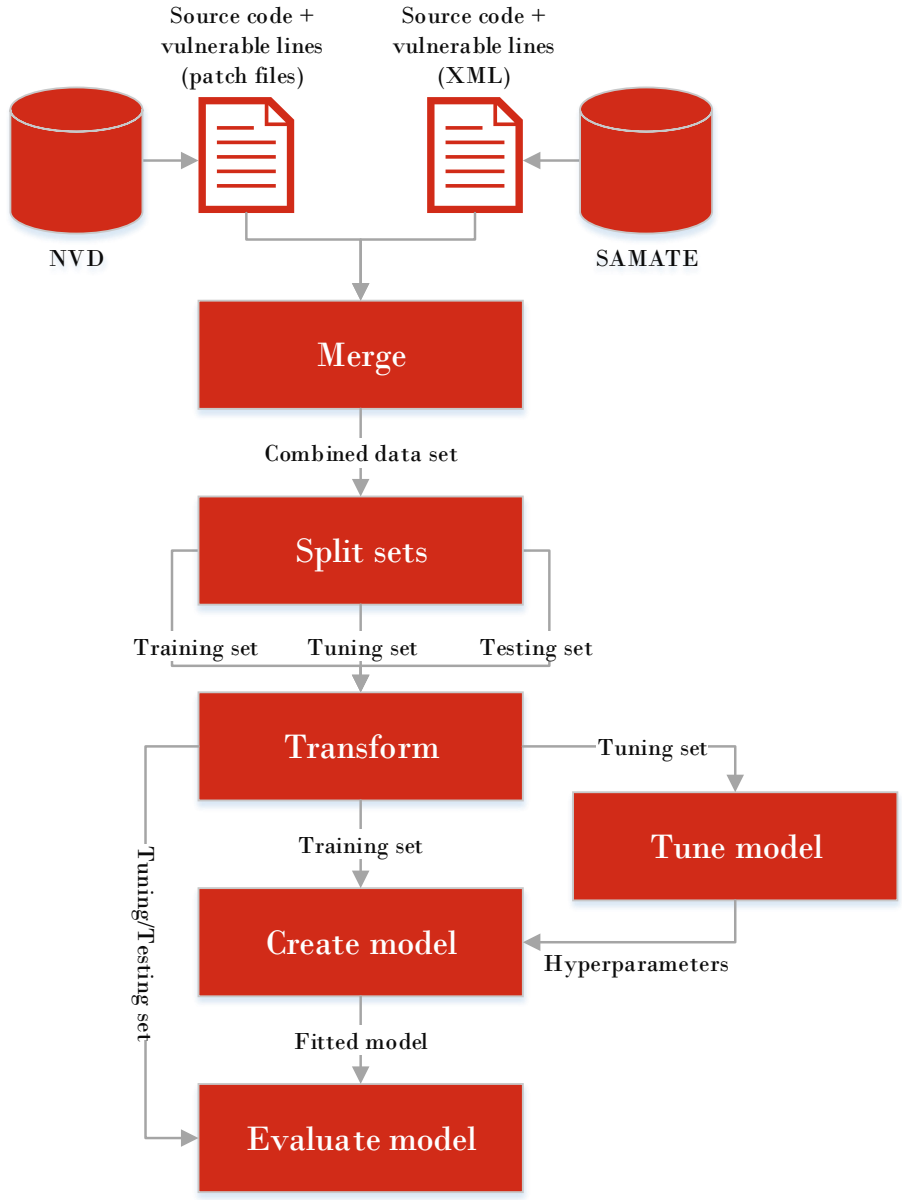

Figure 2.1: High-level flow of our research method

2.3. RESEARCH CONTRIBUTION

In this research, we explore whether machine learning techniques can be used to detect vulnerable code in a dynamic programming language, namely PHP. The main contribution of this research will be tooling that is able to detect vulnerable PHP code using a probabilistic classifier and features extracted with data-flow analysis. To train and test our classifiers we will construct a data set by combining mined samples from the National Vulnerability Database and generated test cases from the SAMATE project. We will test the performance of these classifiers as well as test the performance of individual features. We will also test our tooling against other static code analysis tools. Finally, we will try to find previously unknown vulnerabilities. Our tooling, named *WIRECAML* [46], will be made available as open-source software under the MIT license.

3

DATA DESCRIPTION

3.1. NATIONAL VULNERABILITY DATABASE

The *National Vulnerability Database (NVD)* [2] is a product of the National Institute of Standards and Technology (NIST) Computer Security Division and is sponsored by the Department of Homeland Security's (DHS) National Cyber Security Division. The NVD is a vulnerability database that integrates publicly available U.S. government vulnerability resources with references to industry resources. The NVD is based upon the Common Vulnerabilities and Exposures (CVE) [41] standard vulnerability dictionary and provides access to this information via, among other methods, XML feeds, which is what we will be using to extract relevant information.

A common way to classify vulnerabilities is to use the *Common Weakness Enumeration (CWE)* [47]. CWE is a formal list or dictionary of common software weaknesses that can occur in software's architecture, design, code or implementation that can lead to exploitable security vulnerabilities. The NVD integrates CWE into the scoring of CVE entries, allowing us to identify XSS and SQLi vulnerabilities by using CWE codes CWE-79 and CWE-89 respectively. We will be using the NVD files of the years 2002 (when NVD started) to 2016 to generate our data set.

The NVD does not store the language the application was written in, so we have to create our own list of PHP projects using various sources (such as the description fields in the NVD, our own experience with PHP projects, and various PHP project list sites). Once we have parsed the XML files, we can filter the vulnerabilities based on our list and create our data set.

Figure 3.1 shows the pervasiveness of XSS and SQLi vulnerabilities in PHP projects. We found many more instances of CWE-79 and CWE-89 vulnerabilities than any other vulnerabilities.

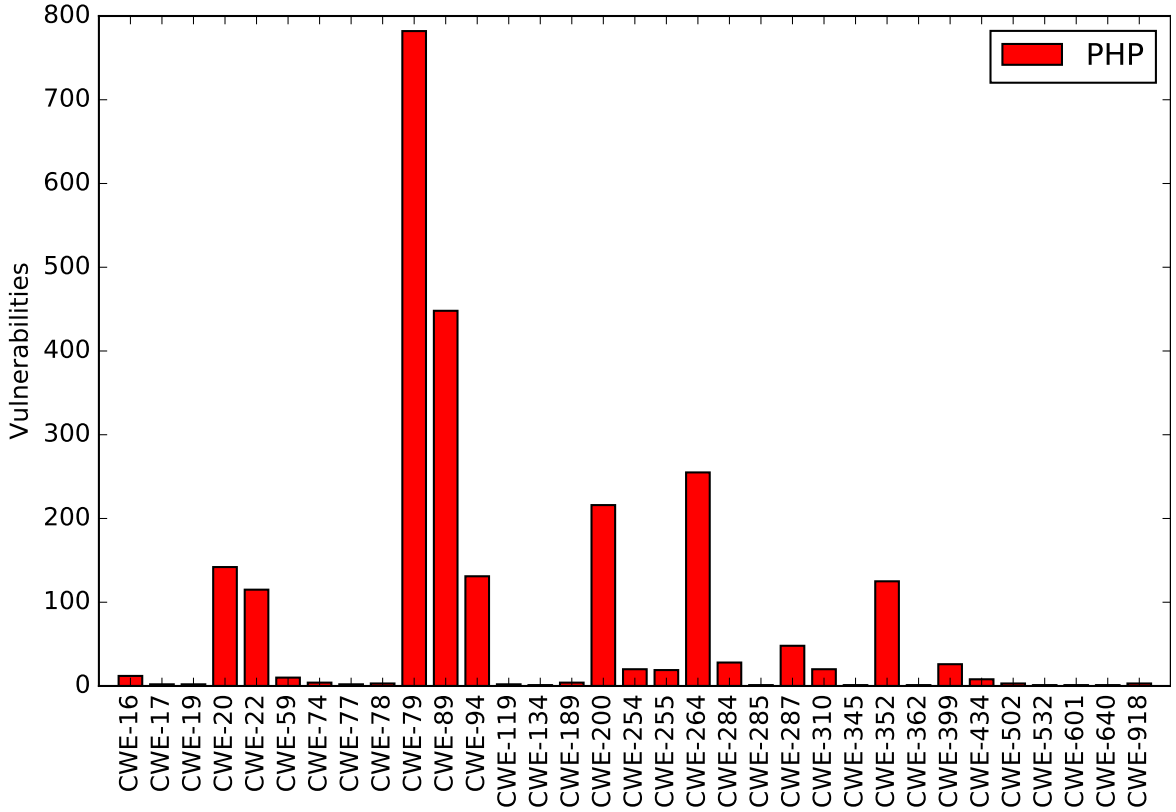

Figure 3.1: Number of vulnerabilities by type (NVD 2002 - 2016)

For the data set we require samples of vulnerable and non-vulnerable lines of code. Although the vulnerabilities contained in the NVD regularly list a git link (or another type of DVCS link), this is not a given. In the case where a link containing the commit ID is given, we can use `git checkout <commit ID>^` to find the vulnerable version and use `git diff <commit ID>^ <commit ID>` to create a patch.

However, for many vulnerabilities where no links were given we had to go through the commit history to find the patch. Especially for older vulnerabilities we found that, as the project had transitioned from subversion or CVS to git, parts or all of the commit history was lost.

In the end we were able to create samples using 28 SQLi vulnerabilities and 81 XSS vulnerabilities. The SQLi vulnerabilities were found in the following OSS projects: *Joomla!* [48], *moodle* [49], *MyBB* [50], *Piwigo* [51], *Tiki Wiki CMS Groupware* [52], *Typo3* [53], *WordPress* [54], and *Zen Cart* [55]. The XSS vulnerabilities were found in the following projects: *Kajona* [56], *moodle*, *ownCloud* [57], *phpMyAdmin* [58], and *WordPress*.

3.2. SAMATE

To extend our data set, we use the test cases from the *NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project* [43]. The SAMATE project is dedicated to improving software assurance by developing methods to enable software tool evaluations, measuring the effectiveness of tools and techniques, and identifying gaps in tools and methods.

SAMATE offers a data set that allows users, researchers, and developers to evaluate tools and test their methods. This data set includes synthetic PHP test cases for SQLi and XSS vulnerabilities that were generated by a tool [59] created by Bertrand Stivalet [60]. The test cases are separated into safe and unsafe categories. All samples are contained in a single file without external dependencies.

During our research, we discovered that some XSS samples were miscategorized as 'unsafe' while in fact, they were 'safe'. Furthermore, we discovered that for the SQLi samples the sink was mislabeled. Both issues have been reported to the author including patches [61, 62] to fix them. Table 3.1 lists the resulting number of samples.

Category	SQLi (CWE-89)	XSS (CWE-79)
safe	8640	6176
unsafe	912	3904

Table 3.1: Number of generated samples

3.3. PREPARING OUR DATA SET

To create the sets, we built an in-memory list of files that are part of the set. This list was then split into three parts; a training set, a tuning set, and a testing set where the split was 70%, 10% and 20% respectively. We used the training set to train our models, the tuning set to test the models and tune their parameters, and the testing set to evaluate our models.

We also constructed a separate list of tuples containing the file names and vulnerable lines. For the generated test cases from the SAMATE data set (section 3.2), we had an XML file containing the file names and vulnerable line numbers. For the NVD data set (section 3.1) we used the acquired patch files. We considered the lines that would be removed by the patch file as the vulnerable lines.

The non-vulnerable files, i.e. the files that do not contain any vulnerables lines, were sampled. This means that we only used a randomized subset of these files to build our sets. The reasons for sampling were two-fold; 1. sampling the non-vulnerable files was possible as we had many more non-vulnerable files compared to vulnerables files, and 2. if we did not sample, we would run into memory issues during the transformation step.

The training/tuning/testing sets and the list of vulnerable lines were then 'pickled'. 'Pickling' [63] is the process whereby a Python object hierarchy is converted into a byte stream. This byte stream can then be stored into a file and be subsequently 'unpickled', which is the inverse operation whereby the byte stream is converted back into an object hierarchy. This has two advantages for us; 1. we do not have to rescan the directories which saves processing time and 2. our sets remain fixed.

Having created our sets, we proceeded with the transformation step detailed in chapter 4.

4

FEATURE EXTRACTION

In section 1.4 we have stated that we will use *control flow graphs* (CFG) to extract features from our data set. These CFGs will be programmatically generated from ASTs that we have extracted from the source code using *phply* [44].

4.1. CONTROL FLOW GRAPH

A control flow graph [64] represents all paths that might be traversed through a program during its execution. 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.

In a CFG, apart from the basic blocks, two specially designated blocks exist: entry and exit blocks. The entry block allows the control flow to enter into the graph, likewise the control flow leaves through the exit block. This is how a control flow graph can depict how different program units or applications process information between different ends in the context of the system.

CFGs are often used in *data-flow analysis*. In the next sections we will be showing three techniques (Reaching Definitions, Constants as a feature, and Taint Analysis) that leverage data-flow analysis for our feature extraction.

4.2. REACHING DEFINITIONS

Reaching definitions for each program point are all definitions that have been made and not overwritten, when program execution reaches this point along some path. For example, consider the following code:

```
1 $x = 5;
2 $y = 1;
3 while ($x > 1) {
4   $y = $x * $y;
5   $x--;
6 }
```

Listing 4.1: A reductive code sample

We can easily see in our sample that all of the definitions reach the entry of line 4. The definitions on line 1 and 2 reach there on the first iteration. For line 5, only the definitions on line 1, 4 and 5 reach there, because the definition on line 4 'kills' the reach of the definition on line 2.

To programmatically determine the reaching definitions for each line we can use the algorithm in listing 4.2, which we have adopted from the *Dragon Book* [65]. In this book, blocks of code are considered. For simplicity of implementation, we have changed a block to mean a line and have modified the algorithm to reflect that.

```
for each line L do {
  IN[L] = Ø;
  OUT[L] = GEN[L];
}
change = true;
while change do {
  change = false;
  for each line L do {
    IN[L] = ⋃P is a predecessor of L OUT[P];
    oldout = OUT[L];
    OUT[L] = GEN[L] ⋃ (IN[L]−KILL[L]);
    if (OUT[L] ≠ oldout)
      change = true;
  }
}
```

Listing 4.2: An Iterative Algorithm for Computing Reaching Definitions

The algorithm maintains four sets for each line. $GEN[L]$ represents the set of all definitions (usually just one definition) inside L that are visible immediately after line L. $KILL[L]$ is the union of the definitions in all the other basic blocks of the flow graph that are killed by the individual state machine L. $OUT[L]$ represents the union of all definitions created by that line and all definitions that were not killed by that line. $IN[L]$ represents all incoming definitions from preceding lines. The algorithm runs repeatedly until no more changes in the OUT sets are found.

For our analysis we represent the four sets as bit vectors instead of sets to minimize memory usage. This means that for each assignment found in the code, a unique bit flag is assigned. We traverse the CFG to determine the predecessors for each line.

Consider the CFG in figure 4.1:

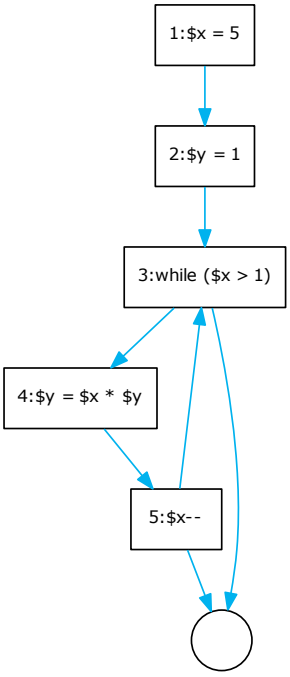

Figure 4.1: CFG for listing 4.1

$$
\begin{aligned}
IN[1] &= \emptyset, GEN[1] = \{x_1\} \\
OUT[1] &= \{x_1\}, KILL[1] = \{x_5\} \\
IN[2] &= \{x_1\}, GEN[2] = \{y_2\} \\
OUT[2] &= \{x_1, y_2\}, KILL[2] = \{y_4\} \\
IN[3] &= \{x_1, x_5, y_2, y_4\}, GEN[3] = \emptyset \\
OUT[3] &= \{x_1, x_5, y_2, y_4\}, KILL[3] = \emptyset \\
IN[4] &= \{x_1, x_5, y_2, y_4\}, GEN[4] = \{y_4\} \\
OUT[4] &= \{x_1, x_5, y_4\}, KILL[4] = \{y_2\} \\
IN[5] &= \{x_1, x_5, y_4\}, GEN[5] = \{x_5\} \\
OUT[5] &= \{x_5, y_4\}, KILL[5] = \{x_1\}
\end{aligned}
$$

Figure 4.2: Reaching Definitions for listing 4.1

By simply checking the edges directly connected to the node, we can see that both line 2 and 5 are predecessors for line 3, while other lines only have one predecessor. Figure 4.2 shows the result of the algorithm. In the figure, definitions are named after the variable and the line they were defined on, so x_1 is the definition of variable x on line 1, etc.

Once we have constructed the Reaching Definitions sets, we can determine the *Use-Definition chains (UD chains)* for each definition. UD chains consist of a use of that variable, U, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. UD chains are constructed using the IN set for a particular definition. Table 4.1 shows the UD chains for listing 4.1.

Use	UD chain
y_4	y_2, y_4
x_5	x_1, x_5

Table 4.1: UD chain for listing 4.1

For our feature set, we are using UD chains to create new features. We consider each line to be a sample that could be either vulnerable or not vulnerable. We create a CFG from the code and run the Reaching Definitions algorithm (listing 4.2) after which we determine the UD chain for each line. We can now determine for each line which functions may have been used for that variable. As we have explained in section 1.4, for an SQLi or XSS vulnerability to work, there needs to be a potentially vulnerable function and a lack of sanitization, which usually comes in the form of a function as well. Using the presence of functions as features, we can create a feature set, where each function is a feature and a value of 1 means the function is present (where value 0 means the function was not used for that variable on that particular line).

To illustrate this further, let us consider a slightly more complicated example.

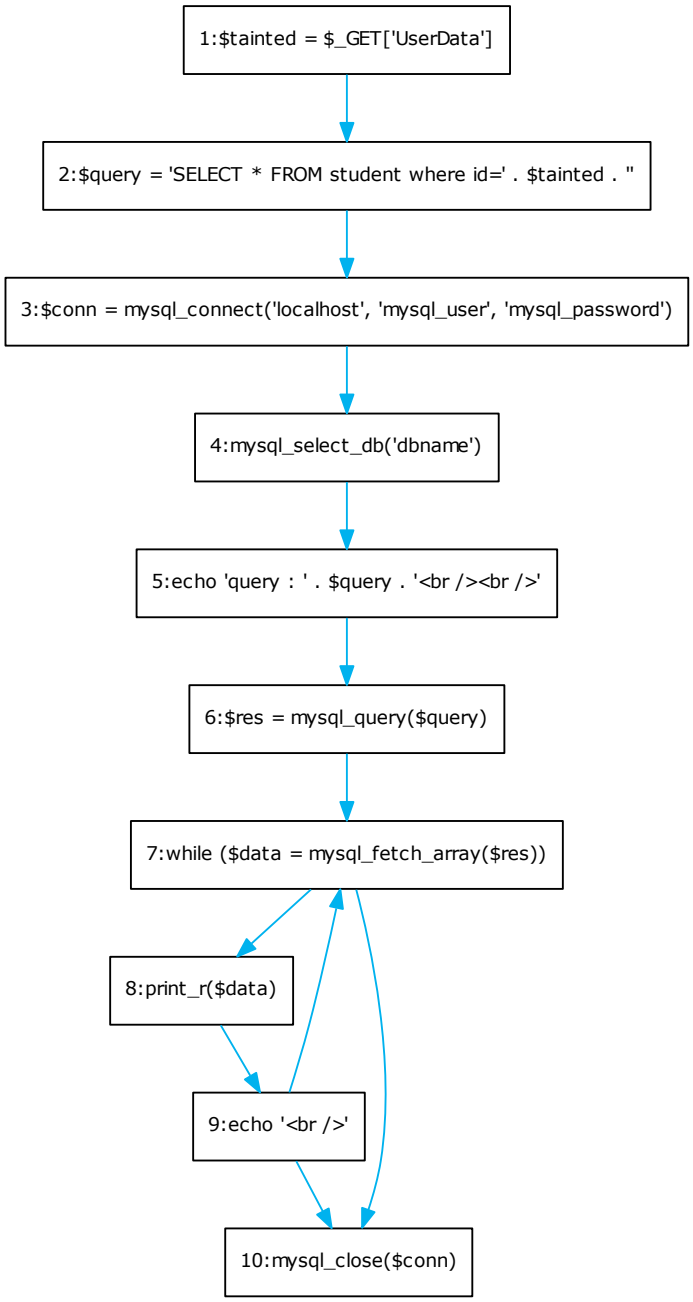

Figure 4.3: A slightly more complicated example

Figure 4.3 shows a program that accepts user input, runs a query with the supplied input and displays the results. It is also a vulnerable program as the input is not sanitized. We create a data set where each line is a sample and mark the lines that are in the path of the sink as vulnerable[1]. This is where the flaw is actually exploited.

Using UD chains we can create a matrix such as table 4.2. The matrix shows which func-

[1]We have also tried only marking the line containing the sink but we got slightly better results when considering the whole path as vulnerable

line	echo	mysql_ close	mysql_ connect	mysql_ fetch_array	mysql_ query	mysql_ select_db	print_r	vulnerable
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 4.2: Data set for figure 4.3

tions each line depends upon. Line 7, for instance, uses `mysql_fetch_array`, which uses a parameter `$res`, which is defined on line 6 by function `mysql_query`. Lines 1–2 are all zeroes as they are not dependent upon any functions.

This matrix is also an appropriate format for our probabilistic classifiers, where the vulnerable column is used as the class and the other columns are used as the features. There is, however, another feature we can extract using this method.

4.3. CONSTANTS AS A FEATURE

During our testing we discovered that another feature could be extracted that may have some predictive power: constants. In PHP the function `filter_var()` uses a constant as its second parameter to specify what kind of filtering is required. For instance, `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.

It is clear that the presence of these constants may indicate some form of sanitization taking place. By making all constants features using our method outlined in section 4.2, we can include these in a way that could be applied to other dynamic languages. A code base where custom sanitization functions with constants as parameters were used will also benefit in this manner.

Finally, there is one other feature that we can extract from our CFG using *Taint Analysis*, which we will discuss in the next section.

4.4. TAINT ANALYSIS

Tainted data is the 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. Using a set of suitable operations, tainted data can become untainted (sanitized), removing its harmful properties. Vulnera-

bilities like SQLi and XSS can be seen as instances of this general class of taint-style vulnerabilities.

For our data set, variables can either be tainted or untainted. Since our approach is language-neutral, our analysis does not use a database of sensitive sink functions. This means that we will have to identify the taint value of the variables for all lines, not just the ones with sinks. We can accomplish this by reusing our work on Reaching Definitions in section 4.2.

We assume a variable is untainted when its type is a *float*, *int*, *double* or *bool*. This also applies when a variable is cast 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 truth table 4.3.

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

Table 4.3: Taint rules

For each line we determine one taintedness feature. In the case where a line contains multiple variables, the rules in table 4.3 are followed. Simply put, if one variable is tainted, the line is considered tainted.

Similarly to the sinks, we will not be able to determine which function is a sanitization function due to our language-neutral approach. This means that variables that should have become untainted will remain tainted. This may limit the usefulness of the taintedness feature. We will see how its performance compares to the other features in section 5.3.

4.5. COMBINING FEATURES

The final step in our feature extraction process is combining the features from the previous sections (functions, constants, and taintedness) into a single matrix. In table 4.4 we see a simplified version of such a matrix.

line	⋯	FILTER_ SANITIZE_ STRING	FILTER_ SANITIZE_ EMAIL	mysql_ fetch_ array	mysql_ query	⋯	tainted	vulnerable
1	⋯	1	0	0	0	⋯	1	0
2	⋯	0	0	0	0	⋯	0	0
3	⋯	0	0	0	0	⋯	0	0
4	⋯	1	0	0	0	⋯	1	0
5	⋯	0	0	0	0	⋯	0	0
6	⋯	1	0	0	1	⋯	1	1
7	⋯	1	0	1	1	⋯	1	1
8	⋯	1	0	1	1	⋯	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.4: Transformed set with all features

The columns `FILTER_SANITIZE_STRING` and `FILTER_SANITIZE_EMAIL` represent constants. Similar to the function features, these are set to 1 when these constants were used for the variable on that particular line. It is important to note that the number of features is not fixed as it depends on the number of functions and constants that the source code contains. The columns `mysql_fetch_array` and `mysql_query` represent functions and the column 'tainted' states if the variable on the line is tainted. The 'line' column is not used for training, but is retained to be able to find the line once a sample is classified as vulnerable. As stated before, the 'vulnerable' column represents the class variable.

Having transformed our sets, we proceeded with the model training and evaluation step detailed in chapter 5.

5

MODEL EVALUATION

In section 1.2 we have defined various probabilistic classifiers (Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Tree Augmented Naive Bayes (TAN)) that we will use for comparison. In addition we will use a dummy classifier which will serve as a random baseline. An overview of the classifiers is provided in table 5.1.

Classifier	Implementation	Library
Dummy	DummyClassifier	scikit-learn
Decision Tree	DecisionTreeClassifier	scikit-learn
Naive Bayes	BernoulliNB	scikit-learn
Random Forest	RandomForestClassifier	scikit-learn
TAN	TAN	Weka
Logistic Regression	LogisticRegression	scikit-learn

Table 5.1: Classifiers

For all classifiers except one, we will be using the implementations of the scikit-learn library [45]. Since scikit-learn has no implementation of the TAN classifier, we will use the TAN classifier of the Weka library [66].

5.1. TUNING HYPERPARAMETERS

Hyperparameters are the parameters of a model training algorithm and are set prior to the learning process. With hyperparameter tuning the goal is to find the optimal hyperparameter set for a learning algorithm.

To optimize the hyperparameters we use a technique called *grid search* (or *parameter sweep*), which is simply an exhaustive search through a manually specified subset of hyperparameters. As we are trying to optimize our models for the AUC-PR value, we use the AUC-PR value on the tuning set as a performance metric.

Different model training algorithms require different hyperparameters. We use the hyperparameters and values in table 5.2 for our grid search. The definitions for these hyperparameters are in appendix A.

Grid search then trains the models by determining the Cartesian product of the defined hyperparameter values and evaluating the model's performance for each of the parameter combinations. Hyperparameters that we have not defined will remain on their default settings. The grid search algorithm outputs the hyperparameter values for the model that achieved the highest score. TAN does not have any hyperparameters, but for comparison, we have included TAN in the results.

Classifier	Hyperparameters
Dummy	strategy ∈ {most_frequent, uniform, stratified}
Decision Tree	max_depth ∈ {5, 15, 30, ∅}
	min_samples_leaf ∈ {1, 2, 10, 50, 100}
	max_features ∈ {log2, sqrt, ∅}
	class_weight ∈ {balanced}
Naive Bayes	alpha ∈ {0.001, 0.003, 0.01, 0.03, 0.1, 1, 3, 10, 30, 100}
Random Forest	n_estimators ∈ {300, 500}
	max_depth ∈ {5, 15, 30, ∅}
	min_samples_leaf ∈ {1, 2, 10}
	max_features ∈ {log2, sqrt}
	class_weight ∈ {balanced}
TAN	∅
Logistic Regression	penalty ∈ {l1, l2}
	C ∈ {0.001, 0.01, 0.1, 1, 10, 100}
	class_weight ∈ {balanced}

Table 5.2: Hyperparameter space

Our grid search algorithm determined the hyperparameter settings in tables 5.3 and 5.4 for our classifiers.

Classifier	Hyperparameters	AUC-PR tuning set
Dummy	strategy = most_frequent	0.51
Decision Tree	max_depth = 30	0.85
	min_samples_leaf = 2	
	max_features = ∅	
Naive Bayes	alpha = 0.001	0.64
Random Forest	n_estimators = 300	0.81
	max_depth = ∅	
	min_samples_split = 5	
	min_samples_leaf = 1	
	max_features = sqrt	
TAN	∅	0.69
Logistic Regression	penalty = l1	0.83
	C = 0.1	

Table 5.3: Hyperparameter settings for SQLi

Classifier	Hyperparameters	AUC-PR tuning set
Dummy	strategy = most_frequent	0.51
Decision Tree	max_depth = ∅	0.82
	min_samples_leaf = 10	
	max_features = ∅	
Naive Bayes	alpha = 0.003	0.69
Random Forest	n_estimators = 500	0.85
	max_depth = ∅	
	min_samples_leaf = 2	
	max_features = sqrt	
TAN	∅	0.83
Logistic Regression	penalty = l2	0.81
	C = 0.01	

Table 5.4: Hyperparameter settings for XSS

We will use these parameter settings throughout the following sections.

5.2. CLASSIFIER PERFORMANCE

For RQ1 (*When using the same feature set, how does the performance of various probabilistic classifiers compare?*) we have tested the performance of 5 probabilistic classifiers against the testing set. We have used the following classifiers: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Tree Augmented Naive Bayes. To set a baseline, we have also included a dummy classifier.

The dummy classifiers are shown in figures 5.1 and 5.2. We are using the 'most_frequent' parameter for these classifiers, which will output the most likely result every time, meaning, it will classify everything non-vulnerable. This may not be a very useful classifier, but it sets a baseline for the classifiers that follow.

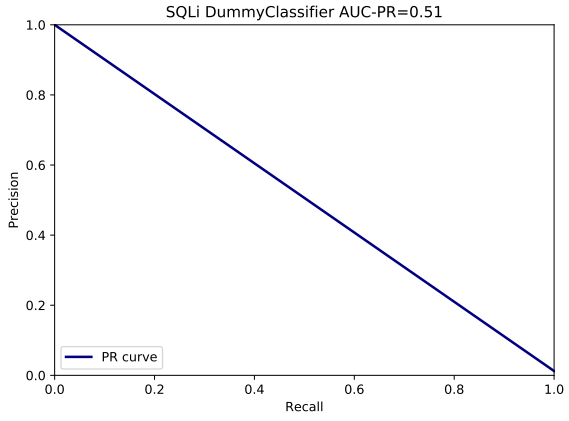

Figure 5.1: Dummy classifier for SQLi

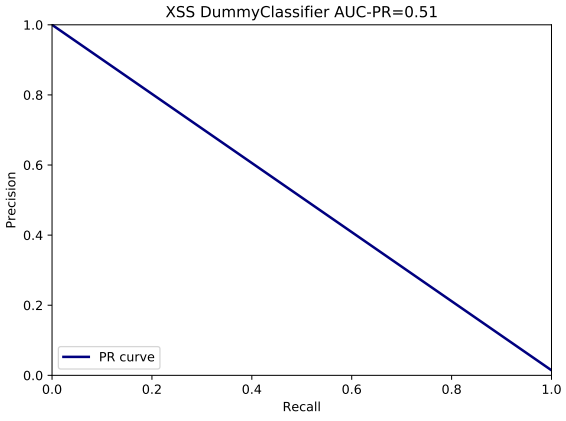

Figure 5.2: Dummy classifier for XSS

The Decision Tree classifiers shown in figures 5.3 have performed the best on the testing set. The AUC-PR scores are even better than the AUC-PR scores of the tuning set. Consid-

ering that XSS attacks are more diverse (stored vs. reflected, multiple sinks), it is possible that the performance of the XSS classifier was affected by this extra complexity.

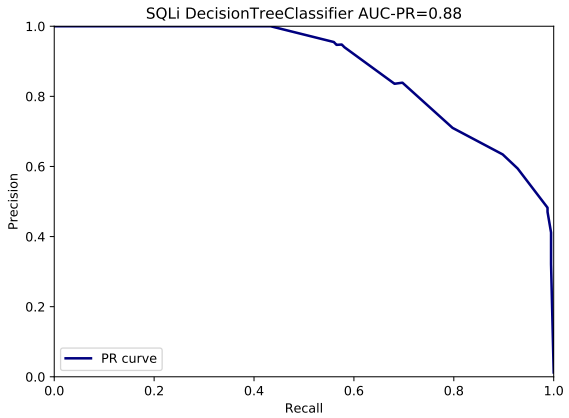

Figure 5.3: Decision Tree classifier for SQLi

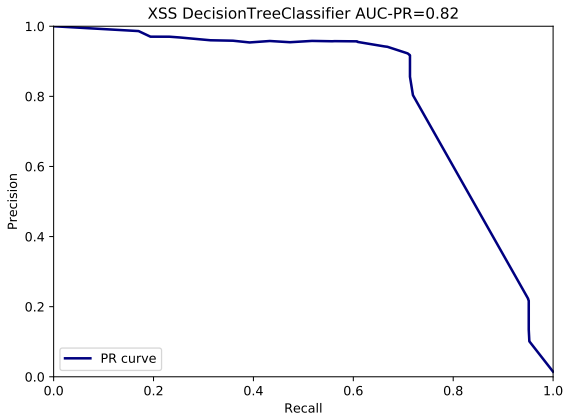

Figure 5.4: Decision Tree classifier for XSS

The Naive Bayes classifiers are shown in figures 5.5 and 5.6. Even though these perform much better than the baseline, their performance is worse in comparison to the other classifiers. The performance is however exactly the same as it was on the tuning set.

When we look at the shape of the graphs, we see a jagged pattern, which most likely stems from the fact that Naive Bayes pushes its predictions to the extremes, as previously described in section 1.2.2.

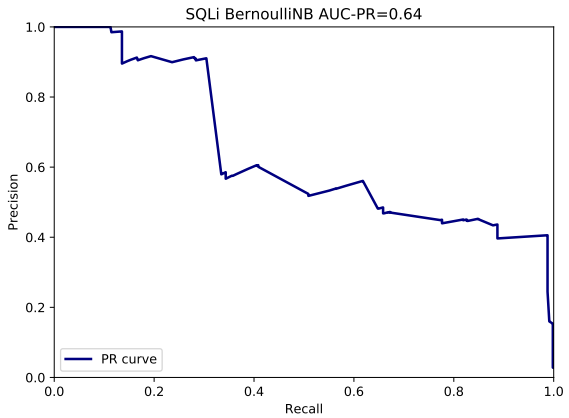

Figure 5.5: Naive Bayes classifier for SQLi

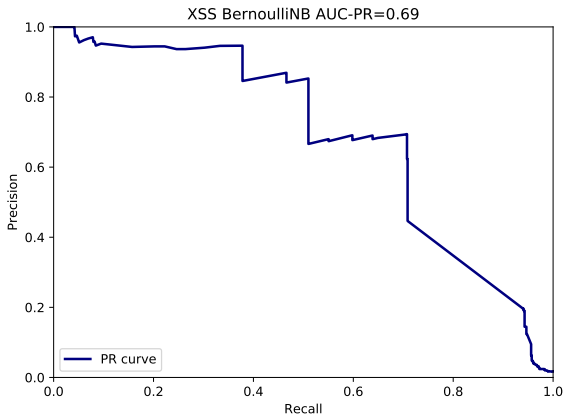

Figure 5.6: Naive Bayes classifier for XSS

The performance of the Random Forest classifiers (shown in figures 5.7 and 5.8) is very similar to the Decision Tree classifiers, both in the shape of the curves and the AUC-PR scores. This is not a surprise since a Random Forest is basically a "forest of decision trees", so their behaviors should be similar.

Compared to the AUC-PR scores for the tuning set, the SQLi RF classifier scored slightly better (0.81 vs. 0.85) and the XSS RF classifier scored slightly worse (0.85 vs. 0.82).

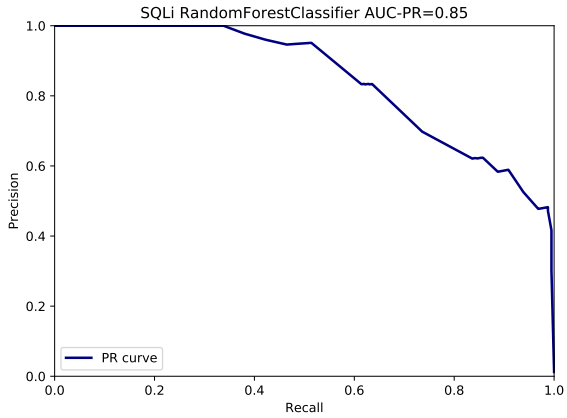

Figure 5.7: Random Forest classifier for SQLi

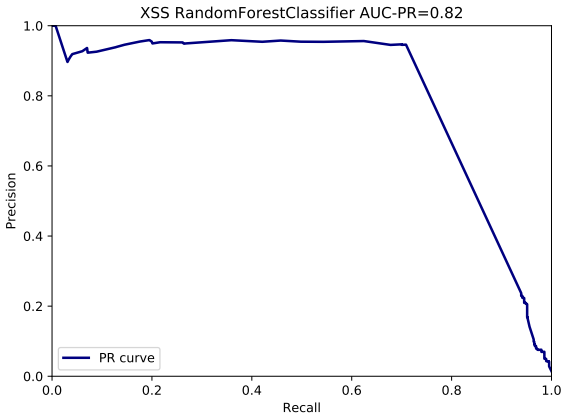

Figure 5.8: Random Forest classifier for XSS

For the TAN classifiers (shown in figures 5.9 and 5.10) we ran into out-of-memory errors when using the full training and testing set. To reduce the size of the data set, we selected the top 1,000 features based on the χ^2-test and filtered out the other features. We then used the same filter for the testing set. In earlier experiments, this kind of filtering had limited impact on the results of the other classifiers.

The performance of the TAN classifiers is not as good as the tree classifiers but still better than their simpler counterpart - Naive Bayes. Looking at the performance on the tuning set, the SQLi TAN classifier has performed better (0.69 vs. 0.75) and the XSS TAN classifier has performed slightly worse (0.83 vs. 0.81).

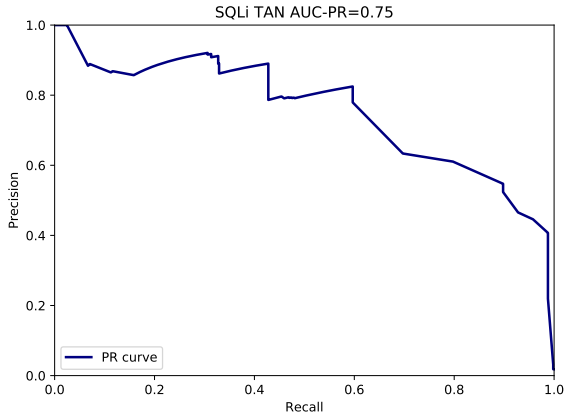

Figure 5.9: TAN classifier for SQLi

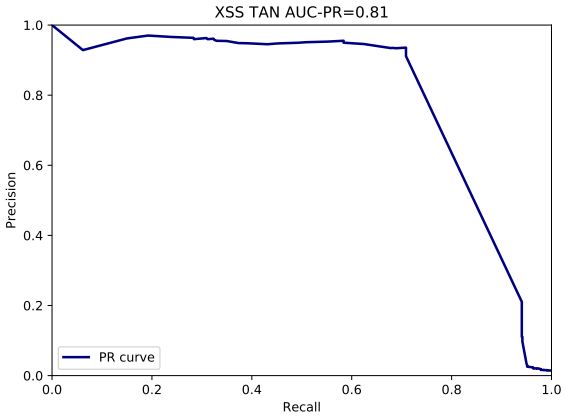

Figure 5.10: TAN classifier for XSS

The Logistic Regression classifiers are shown in figures 5.11 and 5.12. The performance of the classifiers is very comparable to the tree classifiers, where the SQLi LR classifier scores about as well as the SQLi DT classifier and the XSS LR classifier scores worse than the DT and RF XSS classifiers.

Compared to the AUC-PR scores for the tuning set, the SQLi LR classifier scored slightly better (0.83 vs. 0.87) and the XSS LR classifier scored slightly worse (0.81 vs. 0.79).

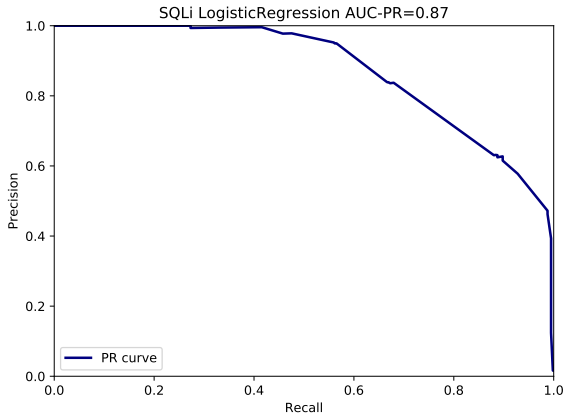

Figure 5.11: Logistic Regression classifier for SQLi

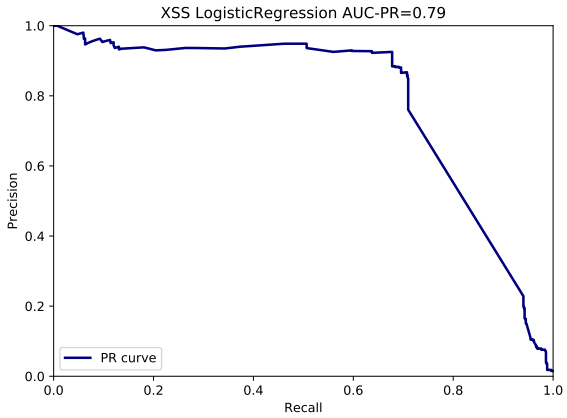

Figure 5.12: Logistic Regression classifier for XSS

Since the Decision Tree classifiers have shown the best performance on the testing set, we will be using those classifiers to answer the other research questions.

5.3. FEATURE PERFORMANCE

For RQ2 (*Which features extracted from the CFG perform best for vulnerability detection?*) we propose two methods to determine feature performance: model-based ranking and the χ^2-test. The following two sections are an elaboration of these methods.

5.3.1. MODEL-BASED RANKING

In sections 4.2, 4.3, and 4.4 we have introduced three types of features (functions, constants and taintedness). To see their usefulness, we build models where we leave out one of these feature types to establish their contribution to the model's performance.

In section 2.1 we have explained that we will be using the AUC-PR score to compare models. To create the models we use the Decision Tree algorithm. Figures 5.13 and 5.14 show the results. The PR-curve named 'all' is the combination of all feature types and is considered the baseline.

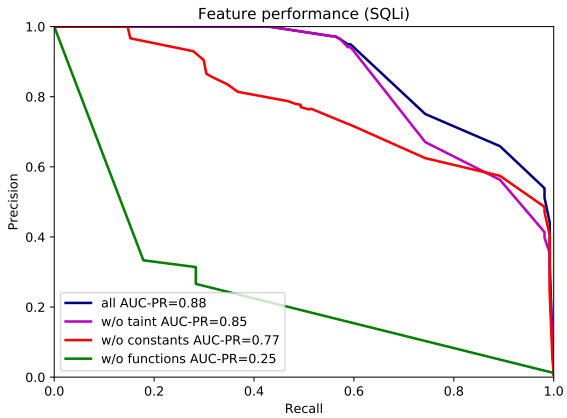

Figure 5.13: SQLi model-based feature ranking

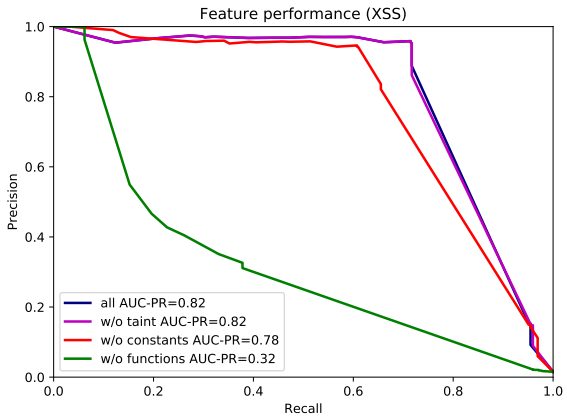

Figure 5.14: XSS model-based feature ranking

Although all feature types seem to add to the model's performance, the feature type

taintedness seems to have little to negligible impact. This may have to do with the method used as our taint analysis does not recognize sanitization functions causing most variables to be tainted. It could also be because the taintedness does not add any new information that would increase the predictive power of the model. To see if that is the case, we will be looking into the individual features in the following section.

5.3.2. χ^2-TEST

To test the performance of individual features we apply the χ^2-test [67]. The χ^2-test measures dependence between the feature and the class variable, so using this function filters out the features that are the most likely to be independent of class and therefore irrelevant for classification.

To measure the χ^2-test statistic for a feature, we apply the χ^2-test to features using the following definitions and formula:

$$
\begin{aligned}
O_i &= \text{the number of observations of that feature given class } i \\
N &= \text{total number of observations of that feature} \\
P_i &= \text{the expected theoretical probability}^{1} \text{ of class } i \\
E_i &= NP_i = \text{the expected theoretical frequency of class } i
\end{aligned}
$$

$$
\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}
$$

where $k = 2$ as we have a vulnerable and a non-vulnerable class.

In table 5.5 we show the results of the χ^2-test which we have applied on the training set. We have limited the results to the top 25 highest scoring features for readability purposes. In actuality the number of features are 6472 for the SQLi samples and 5423 for the XSS samples. Functions are denoted by a trailing (). The other features are constants.

Our specifically engineered feature based on taint analysis (section 4.4) is absent due to underperformance as we had already seen in section 5.3.1. For SQLi it ended up in position 114 and for XSS in position 250 and therefore did not make this list. This means that it has less predictive power than we had hoped.

[1] This is simply the number of samples of class i divided by the total number of samples.

SQLi	χ^2	XSS	χ^2
mysql_query()	51813.43	echo()	15646.38
mysql_fetch_array()	25268.18	checked_data	14130.25
filter_var()	7733.48	filter_var()	9425.43
FILTER_SANITIZE_SPECIAL_CHARS	5460.85	mysql_real_escape_string()	6477.21
FILTER_SANITIZE_FULL_SPECIAL_CHARS	5297.84	ENT_QUOTES	5775.33
FILTER_SANITIZE_EMAIL	4643.13	http_build_query()	2903.74
sprintf()	3224.22	FILTER_SANITIZE_EMAIL	2832.44
mysql_real_escape_string()	2982.49	FILTER_SANITIZE_MAGIC_QUOTES	2716.72
shell_exec()	1598.68	FILTER_SANITIZE_FULL_SPECIAL_CHARS	2572.81
fgets()	1243.34	FILTER_SANITIZE_SPECIAL_CHARS	2271.79
stream_get_contents()	992.27	htmlentities()	1970.08
system()	950.25	rawurlencode()	1910.52
fread()	703.10	addslashes()	1894.10
popen()	543.95	FILTER_SANITIZE_NUMBER_INT	1486.99
unserialize()	444.04	urlencode()	1218.76
fopen()	259.46	preg_replace()	1159.15
echo()	170.30	shell_exec()	1057.27
false	104.42	htmlspecialchars()	519.02
get_string()	95.29	system()	473.24
true	85.93	stream_get_contents()	455.60
define()	47.19	fgets()	454.01
optional_param()	37.41	fread()	408.19
PARAM_INT	35.78	false	275.66
get_context_instance()	35.45	get_string()	236.08
mysql_connect()	29.58	true	231.25

Table 5.5: Top 25 features for SQLi and XSS

For both SQLi and XSS the top result is a sink which is to be expected. `mysql_query()` is considered a sink for SQLi as it can execute a possibly malicious query that is given as a parameter. `echo()` is considered a sink for XSS as it can output a XSS string that will be parsed by the browser. It is also no surprise to see a sanitization function like `filter_var()` including its filter parameter constants high up in the list.

The fact that `mysql_connect()` is relatively low on the list may have a good reason. This function is generally defined only once in an include file, which is then included throughout the code base to provide a database connection. WIRECAML contains an include preprocessor to make sure that these files are concatenated before they are parsed. However, the file names that are included may contain variables that can only be determined at run-time (run-time source inclusion). It is likely that this is the reason that `mysql_connect()` does not show up as often in execution paths as one would expect.

There are also some peculiarities. For instance, it is unclear why `mysql_real_escape_string()` is on position 4 for XSS as it does little to filter XSS attacks. It is also puzzling why the boolean constants `true` and `false` made it in this list as these should have no predictive power for Injection attacks.

5.4. COMPARING TO OTHER TOOLS

In RQ3 (*By using machine learning and features extracted from control flow graphs, can we correctly classify Injection and Cross-Site Scripting vulnerabilities as described by OWASP?*) we ask if we are able to correctly classify SQLi and XSS vulnerabilities. In order to test this,

we have decided to compare our models to existing static code analysis tools to test correctness.

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 analyzers [68].

For our comparison, we evaluated the 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?). We decided on four tools (*Pixy* [69], *RIPS* [70], *WAP* [71, 72], and *Yasca* [73]) to compare with our own tool.

5.4.1. PIXY

Pixy is a scanner 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 [74], although there has been no active development since 2014. We are using the version with commit ID 3f81106 for our comparison.

5.4.2. RIPS

RIPS is the self-proclaimed most popular static code analysis tool to automatically detect vulnerabilities in PHP applications. By tokenizing 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 [75] to overcome the technical limitations of the open-source version. Due to budgetary restrictions, we will only be evaluating the open-source version (version 0.55).

5.4.3. WAP

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 analyzing source code. More precisely, it does taint analysis to detect the input validation vulnerabilities. The aim of the taint analysis is to track malicious inputs inserted by entry points (`$_GET` and `$_POST` arrays) and to verify if they reach some sensitive sink. After detection, the tool uses data mining to confirm if the vulnerabilities are real or false positives. As a final step, real vulnerabilities are corrected with the insertion of fixes (small pieces of code) into the source code. WAP was written by Ibéria Medeiros in 2014 and has been part of the OWASP project since 2015. We are using version 2.1 for our comparison.

5.4.4. YASCA

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 are using version 3.0.5 for our comparison.

5.4.5. 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:

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

It is important to note that the F_1-score will be a weighted average over both the vulnerable and non-vulnerable classes, since the data set is biased (there are many more non-vulnerable samples than vulnerable ones).

As our tool does output a probability of the vulnerable class, we determine the threshold that maximizes the F_1-score of the model's predictions using the tuning set. 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 maximizes the F_1-score. This threshold c will then be used to determine the F_1-score for the testing set.

We will only be using the SAMATE data set for our evaluation, as we noticed that none of the tools were able to cope with the larger NVD data set. Our tool will be trained with a training set that is extracted from the SAMATE data set. The testing 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 have decided to only compare files and ignore the line number. Since the SAMATE data set only contains one vulnerability per file, we believe that this will not skew the results.

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

5.4.6. RESULTS

We will start with the SQLi data set of which the weighted averages are shown in table 5.6. It shows that our model 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 5.8) it becomes clear that both RIPS and WAP were struggling to find the vulnerable samples.

Yasca scores 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 5.7).

SAMATE SQLi data set ($c = 0.97$)

	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 5.6: Weighted averages (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 5.7: F_1-scores for class 'not vulnerable' (SQLi)

	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 5.8: F_1-scores for class 'vulnerable' (SQLi)

As we have seen in earlier results, our model scores lower for XSS. This apparently also applies to other tools (table 5.9), which is why our model still scores the best overall. We note that for the non-vulnerable class RIPS scores better in table 5.10. 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 5.11).

SAMATE XSS data set ($c = 0.83$)

	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 5.9: Weighted averages (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 5.10: F_1-scores for class 'not vulnerable' (XSS)

	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 5.11: F_1-scores for class 'vulnerable' (XSS)

5.5. DISCOVERING UNKNOWN VULNERABILITIES

For RQ4 (*Can our classifier be used to find previously unknown vulnerabilities?*) we have manually verified whether positive samples found were false or real. For this purpose, we have used the latest versions of the projects mentioned in section 3.1: 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). We chose these projects as the training set contains many of the functions and constants that these projects use. We believe that this will increase the chance that results are relevant. We used the latest versions because that will make any vulnerabilities we find more significant.

For each project we ran two Decision Tree models; one that finds XSS vulnerabilities and one that finds SQLi vulnerabilities. The output of each model was a CSV file containing the file name, line number and probability for each sample. These needed to be manually inspected to confirm whether or not they are exploitable.

For the output, we decided to only ignore the samples where the probability that they are vulnerable is 0 ($P(y = 1) = 0$). This resulted in 487,548 possible positives. This may seem like a lot, but it is only 9% of the total NCLOC[3], which number 5,183,277 lines. Another reason to only ignore the $P(y = 1) = 0$ samples is that it gave us the flexibility to start our inspection with likely positives but still evaluate less likely samples if we suspected a correlation. During our inspection we started with analyzing the samples most likely to be vulnerable first.

We set a time limit of 3 days for manual inspection and ended up inspecting 1,646 samples of which we found 28 samples suspicious. These samples are from Joomla!, moodle, Piwigo, Tiki Wiki CMS Groupware, and Zen Cart. To verify if these are actual vulnerabilities, we installed these OSS projects on a virtual machine running Apache and MySQL and tried to find a suitable exploit. Again, we set a time limit for ourselves of 2 days to find possible exploits.

We quickly discovered that, even though at first glance input parameters seemed to be taken directly from the browser, these input parameters were actually being sanitized 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 sanitized versions. This sanitization could not have been picked up during our transformation step, as array operations are not supported by our tooling. After further examination, we discovered that for most samples some form of sanitization is in place making them false positives except for one sample of Piwigo.

5.5.1. THE PIWIGO VULNERABILITY

Piwigo also uses a generic sanitization mechanism where quotes are added to input parameters using PHP's `addslashes()` function. `addslashes()` 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 it is often used as a filter to prevent SQL injections, it does not provide complete protection against it.

[3]Non-Comment Lines Of Code (NCLOC) as measured by *phploc* [76]

```
1  if (isset($_POST['delete']) and isset($_POST['tags']))
2  {
3    if (!isset($_POST['confirm_deletion']))
4    {
5      $page['errors'][] = l10n('You need to confirm deletion');
6    }
7    else
8    {
9      $query = '
10 SELECT name
11   FROM '.TAGS_TABLE.'
12   WHERE id IN ('.implode(',', $_POST['tags']).')
13 ;';
14     $tag_names = array_from_query($query, 'name');
15
16     delete_tags($_POST['tags']);
17
18     $page['infos'][] = l10n_dec(
19       'The following tag was deleted', 'The %d following tags were deleted
     ',
20       count($tag_names)
21       )
22       .' : '.implode(', ', $tag_names);
23   }
```

Listing 5.1: Piwigo tag deletion snippet

In listing 5.1 we see how the input parameter `$_POST['tags']` is processed without validation, even though it was only sanitized by `addslashes()` earlier.

Because it is assumed that `$_POST['tags']` contains an array of integers, 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 5.1.

The security risk of this vulnerability is estimated as low with a CVSS[4] score of 3.8 because exploitation requires the attacker to be authenticated as administrator. The vulnerability has been reported [77] and the vendor has released a fix.

5.5.2. EVALUATION

During the analysis that we have outlined in the previous sections it became clear that not being able to recognize array elements in an array is a definite shortcoming of our tooling. 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.

Another area of improvement is regular expressions. Regular expressions are commonly used for either validation or sanitization, which is why both models flagged lines containing functions related to these expressions, such as `preg_match()` and `preg_replace()`.

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

However, in most cases, these regular expressions were formed correctly and could not be exploited. Testing these regular expressions automatically could provide an interesting set of new features for our data set. For instance, we could use the regular expression to test a set of strings commonly used in SQLi and XSS vulnerabilities and see whether the expression is able to match or filter these. The results could be used as extra features for our data set.

We have also run into a few odd bugs. In some cases lines were marked incorrectly, where it seems that the sample was actually referring to a different line in the file. We have also seen instances where lines were marked as vulnerable even though they did not contain any variables, tainted or otherwise.

Despite these challenges, we have been able to find an SQLi vulnerability (section 5.5.1) with our tooling within a reasonable time frame. Furthermore, this vulnerability was not identified with a commercial vulnerability scanner. This means that our tooling can help in assisting a vulnerability researcher or pentester. Adding the aforementioned improvements will further reduce the number of false positives and increase the chances that a vulnerability researcher will find a relevant vulnerability.

6

Conclusion

Despite the popularity of dynamic languages, little research has been done in the area of assisted vulnerability discovery for dynamic languages. This thesis has investigated whether the application of machine learning techniques and static code analysis can be used to detect vulnerable code in one specific dynamic programming language, namely PHP. In section 2.1 we formed our main research question and divided it up into four subquestions. To answer these subquestions we have built tooling called WIRECAML [46] which is able to test the performance of the individual features and the performance of the classifiers. It can also be used to discover vulnerabilities in PHP applications. The answers to our research subquestions are summarized below.

- **RQ1**: *When using the same feature set, how does the performance of various probabilistic classifiers compare?*

  We have answered this question in section 5.2, where we have tested the performance of 5 probabilistic classifiers against the testing set. For the SQLi testing set, Decision Tree performed the best (AUC-PR=0.88), followed by Logistic Regression (AUC-PR=0.87), Random Forest (AUC-PR=0.85), TAN (AUC-PR=0.75), and Naive Bayes (AUC-PR=0.64). For the XSS testing set, Decision Tree and Random Forest did equally well (AUC-PR=0.82), followed by TAN (AUC-PR=0.81), Logistic Regression (AUC-PR=0.79), and Naive Bayes (AUC-PR=0.69). All classifiers scored better than the baseline set by the dummy classifier (AUC-PR=0.51 for both data sets).

  Since the Decision Tree classifiers have shown the best performance on the testing set, we have used these classifiers to answer the subsequent research questions.

- **RQ2**: *Which features extracted from the CFG perform best for vulnerability detection?*

  We have answered this question in section 5.3 where we generated a list of the top 25 performing features for the SQLi and XSS data sets. We used two approaches: model-based ranking (section 5.3.1) and χ^2-test (section 5.3.2).

  For the three feature types (functions, constants and taintedness), we used model-based ranking by creating models without that feature type and comparing AUC-PR scores. We found that all feature types seemed to add to the model's performance but the feature type taintedness seemed to have little to negligible impact.

To test the performance of individual features we applied the χ^2-test, the results of which are shown in table 5.5. For both SQLi and XSS the top result is a sink and various sanitization functions were also shown in the list. Our taintedness feature was absent due to underperformance. For SQLi taintedness ended up in position 114 and for XSS it was in position 250, and therefore it did not make the list of the top 25 features.

- **RQ3**: *By using machine learning and features extracted from control flow graphs, can we correctly classify Injection and Cross-Site Scripting vulnerabilities as described by OWASP [27]?*

  Yes, our tooling can correctly classify SQLi and XSS vulnerabilities. We have answered this question in section 5.4, where we compared our models to four static code analysis tools to test correctness using an F_1-score.

  For both the SQLi and XSS data sets our models scored the best with a weighted F_1-score of the combined classes of 0.94 and 0.64 respectively. Our models also scored best on the F_1-scores per class, except for the 'not vulnerable' class for the XSS data set, where RIPS scored better. This is because RIPS was not able to detect any XSS vulnerabilities at all and thus had a perfect recall score.

- **RQ4**: *Can our classifier be used to find previously unknown vulnerabilities?*

  Yes, our classifier can be used to find previously unknown vulnerabilities. We have answered this question in section 5.5, where we manually verified positive samples of 11 open-source software projects. As we had more samples than we could inspect, we timeboxed our work and found one vulnerability within this period. The vulnerability was found in Piwigo 2.9.2 and allows remote attackers that are authenticated as administrator to inject SQL code into a query.

We can now proceed to answer the main research question.

Can machine learning with features extracted from control flow graphs be used for vulnerability detection in software written in a dynamic language, such as PHP?

Yes, we have demonstrated that machine learning in combination with features extracted from control flow graphs and abstract syntax trees can be used for vulnerability detection in PHP applications. We have also shown that this approach performs better than 4 existing tools that were built for this purpose. Moreover, we have been able to find an SQL injection in the latest version of a photo-sharing web application using our tooling.

6.1. LIMITATIONS

The results presented in this work demonstrate the merits of combining control flow graphs and machine learning for vulnerability discovery. Nonetheless, our approach has several limitations, both of inherent and of technical nature.

First, due to the complexity of tracking arrays, our transformation functions ignore their existence. This can be problematic when one element of the array is set to a constant, another element is a tainted but sanitized variable, and a third is tainted and unsanitized.

Depending on the order, the entire array may be considered to have any of those characteristics.

Second, due to the fact that our data set consists of multiple vulnerabilities in different versions of the same application and considering that these application versions share a lot of the same code, our data set contains duplicate samples. It is unclear how, if at all, this affected our results. Strategies to combat duplication can be complicated as duplicated code can be a few lines scattered across a function.

Third, our research may have a sampling bias as we only have used samples from open-source applications, and no applications from the commercial sector. This is due to the fact that both the code as well as the vulnerability data of commercial applications are not publicly accessible. Moreover, assuming that all application code is available is a limitation in our approach as an application may use third-party plugins and components for which the source code may be unavailable.

Finally, when comparing to other tools (section 5.4), we had limited ourselves to freely available open-source versions of SQLi/XSS vulnerability scanners. It is unclear how well our tool [46] would do in comparison to commercially available tools such as Fortify SCA [78] or the commercial version of RIPS [75]. Considering that these tools use sophisticated rule sets, they are likely to score better than their free counterparts.

6.2. RESEARCH CONTRIBUTIONS

Throughout this research we have made a number of contributions to open-source projects. Here is the list of these contributions:

- We were originally planning on using *ANTLR* [79] to create abstract syntax trees from PHP code. ANTLR did not have a PHP grammar for Python, so we created one [80] which we have shared and has since been included in the ANTLR project. We ended up using phply [44] instead due to performance issues we had with ANTLR.

- During our research, we also used the *Brier score*[1] as an alternative to AUC-PR. We discovered, however, that the implementation of the Brier score function in scikit-learn [45] exhibits a bug when only one class is predicted. We have reported this bug and provided a fix [81]. At the time of writing this bug is still open.

- As mentioned in section 3.2 we discovered that some XSS samples in the SAMATE data set were miscategorized as 'unsafe' while in fact, they were 'safe'. Furthermore, we discovered that for the SQLi samples the sink was mislabeled. Both issues have been reported to the author of the test case generator including patches [61, 62] to fix them.

- In section 5.5.1 we have described a vulnerability in the photo-sharing web application, Piwigo. This vulnerability has been reported [77] and the vendor has released a fix, which will be included in Piwigo 2.9.3.

[1] Across all items in a set of predictions, the Brier score measures the mean squared difference between the predicted probability and the actual outcome.

6.3. RELATED WORK

Our work combines the field of machine learning with vulnerability discovery for PHP web applications. In section 1.3 we have defined three categories which we will use to describe related work in this section.

The first category is *Vulnerability Prediction Models based on Software Metrics*, where software metrics are used as features for vulnerability prediction. The justification to use these particular metrics may be that they are often either readily available or easily obtained for software projects. On the other hand, the purpose of these metrics is to act only as a guiding model for better planning and allocation of resources in software engineering teams, making it unclear if software metrics are good indicators for the existence of software vulnerabilities.

For instance, in section 1.3.1 we had introduced the research of Shin et al. [16], where the authors had created a set of software metrics called CCD metrics, which represent complexity, code churn, and developer activity. Even though their metrics supported the hypotheses in over 50 percent of the total predictions, the authors had to conclude that the number of false positives was too high to be useful.

Doyle and Walden [82] analyzed relationships between software metrics and vulnerabilities in 14 open source PHP web applications. The authors used static analysis tools to measure a variety of metrics including a metric proposed by the authors named *Security Resources Indicator (SRI)*. The results show that no single metric is suitable to distinguish high vulnerability web applications from low vulnerability ones.

In later research, Walden et al. [83] performed a study where the performance of predicting vulnerable software components based on software metrics was compared to text-mining techniques using a hand-curated data set of vulnerabilities gathered from three large and popular open-source PHP web applications. According to various experiments by the authors, the prediction technique based on text mining performed better on average (i.e. higher recall and precision) and the difference was statistically significant. Even though this empirical study can not be generalized to all software projects and all software metrics, it indicates that other methods (such as text mining) may yield better results than software metrics as features.

Then there are *Anomaly Detection Approaches*, which can be used to find vulnerabilities due to improper API usage, as well as logic vulnerabilities due to missing checks. One of the limitations of anomaly detection approaches is that they are only effectively applicable for mature software systems. This is due to the basic assumption that missing checks or improper API usages are rare events and the majority of conditions applied to security objects as well as API usages in a software project are correct, which is an assumption that generally only holds true in mature software projects.

Another limitation of anomaly detection is its inability to distinguish vulnerabilities from defects. For instance, Yamaguchi et al. [84] proposed a system named Chucky for automatic detection of missing checks in source code. The authors report that Chucky identified several missing checks in all open-source projects that it evaluated, where almost all of the top 10 reported anomalies for each function either contain a defect or a security vulnerability. This approach may require more manual inspection by security researchers to determine if reported anomalies are actually vulnerabilities or just defects.

Unsurprisingly, we believe that *Vulnerable Code Pattern Recognition* is the most promising approach for vulnerability discovery. Our inspiration came from Yamaguchi et al. [15,

17] who proposed a method for assisted discovery of vulnerabilities in source code by introducing the concept of 'vulnerability extrapolation', which aims at identifying unknown vulnerabilities based on programming patterns observed in known security vulnerabilities. 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 authors to deconstruct a known vulnerability to a graph query and extrapolate it to a code base, such that functions potentially suffering from the same flaw can be suggested to the analyst.

In later research, Yamaguchi et al. [19, 85] proposed a new graph representation named the 'code property graph', for modeling and discovering vulnerabilities by means of traversals over this graph. The code property graph merges concepts of classic program analysis, namely 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 an existing vulnerability. Although this is a novel approach to assist a security researcher, it does require 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 [86, 87] did investigate 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 SQL-injection (SQLi) and cross-site scripting (XSS) attacks. 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 sanitization statements. To evaluate the effectiveness of the attributes, the authors developed a prototype tool called PhpMinerI and performed experiments on eight open source web applications based on PHP. The authors also compared the performance of PhpMinerI against Pixy. On average, Pixy discovered more vulnerabilities but also produced many more false positives compared to PhpMinerI.

In later research, Shar et al. [88] predicted vulnerabilities using hybrid code attributes. Dynamic analysis was incorporated into static analysis to improve the classification accuracy. Although these earlier works only targeted SQLi and XSS vulnerabilities, the authors stressed that the work should be extended to address other types of vulnerabilities as well.

Shar et al. [89] further extended their previous work by adding *Remote Code Execution (RCE)* and *File Inclusion (FI)* to their vulnerability scope and employing a technique called 'static backward program slicing' in order to extract different execution paths from program slices. They also modified their feature set to include 10 static and 22 dynamic attributes and use 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 labeled training data. These were then implemented in a new and renamed version of PhpMiner.

Considering the similarity of the research of Shar et al. to ours, we have tried to obtain a copy of PhpMiner and its data set for comparison, but have not been successful. The website that used to host it has been taken down and we were not able to reach the authors.

6.4. FUTURE WORK

It is our hope that the approach as introduced in this thesis will provide a fruitful ground for further research. In particular, starting from the work presented, the following directions seem interesting for further research in vulnerability discovery.

First, we believe that our approach is language-neutral, so that it can be applied to other dynamic languages. In order to accomplish this, a new data set needs to be acquired and a new language parser needs to be added to WIRECAML.

Second, in our research we have limited our scope to XSS and SQLi vulnerabilities. These are vulnerabilities that are normally detected with taint analysis and control flow graphs. Shar et al. [89] were able to extend their vulnerability scope with Remote Code Execution and File Inclusion, which can leverage the same methods and seem like logical next steps for our research.

Third, during our analysis in section 5.5 we have seen that the inability to track arrays can generate false positives. Jovanovic et al. describe how they have tracked array variables in their research [69], which we were not able to implement due to time constraints. This may be an interesting next step for our research. PHP is also becoming increasingly more object-oriented which poses an additional challenge for tracking variables.

Fourth, during our evaluation in section 5.5.2 we made the comment that regular expressions could be used to further enrich our feature set. Regular expressions are used for validation or sanitization purposes in many dynamic languages. Further work could include testing the regular expression with a set of strings commonly used in SQLi and XSS vulnerabilities and using the results of these tests as extra features.

Fifth, our data set is very biased due to the fact that each source code line is a sample and there are obviously many more non-vulnerable lines than vulnerable ones. To combat this, we have sampled the non-vulnerable files to reduce the size of our data set. Since the sampling is completely random, this may mean that we have lost valuable samples in favor of duplicates, which in turn impacts the performance of our classifiers. A more intelligent approach would be that duplicate files or duplicate lines are removed from the non-vulnerable samples. Considering that our NVD data set consists of only a few projects, removing duplicates may provide enough of a reduction to make random sampling redundant.

Sixth, in our experiments with machine learning algorithms we have limited ourselves to a small set of probabilistic classifiers. In the area of supervised learning, neural networks have gained a dominating role. For our data set we had to transform directed graphs into vectors so that they are suitable for the classifier, but using a neural network that supports these graph structures (such as a Recurrent Neural Network (RNN)) may provide better performance.

Finally, not unlike other empirical studies, our results are limited to the applied machine learning techniques, the data set, and the setup used. One way of refuting, confirming or improving our results is to replicate the experiments with different data sets, different machine learning techniques, and tweaks to the feature sets. This can be done as we provide both the data sets and the tool via our repository.

A

HYPERPARAMETER DEFINITIONS

Parameter	Definition
alpha	Additive (Laplace/Lidstone) smoothing parameter. Smoothing is used to avoid overfitting the data by smoothing probabilities in such a way that samples that were not observed are still given a small probability.
C	In Logistic Regression λ is used to specify the weight of regularization, which is used to avoid overfitting. In scikit-learn $C = 1/\lambda$, so a smaller C means more regularization.
max_depth	The maximum depth of the tree. If $\varnothing$, then nodes are expanded until all leaves are pure (pure is when one class dominates a leaf node). Setting this value can be used to control the size of the tree to prevent overfitting.
max_features	The size of the random subsets of features to consider when splitting a node. If "sqrt", then max_features = $\sqrt{\text{n\_features}}$, where n_features represents the total number of features of the data set. If "log2", then max_features = $\log_2$ n_features and if $\varnothing$, then max_features = n_features.
min_samples_leaf	The minimum number of samples required to be at a leaf node. A very small number will usually mean the tree will overfit, whereas a large number will prevent the tree from learning the data.
n_estimators	The number of trees in the forest.
penalty	Used to specify the norm used in the regularization. The difference between L1 and L2 is that L2 is the sum of the square of the weights, while L1 is the sum of the weights.
strategy	If "most_frequent", the classifier always predicts the most frequent label in the training set. If "uniform", the classifier generates predictions uniformly at random. If "stratified", the classifier generates predictions by respecting the training set's class distribution.

Table A.1: Hyperparameter definitions

BIBLIOGRAPHY

ACADEMIC REFERENCES

[1] Zakir Durumeric et al. “The Matter of Heartbleed”. In: ACM Press, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755.

[6] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “Cachebleed: A Timing Attack on Openssl Constant Time Rsa”. In: Cryptology ePrint Archive, Report 2016/224, 2016.

[7] Nimrod Aviram et al. “DROWN: Breaking TLS Using SSLv2”. In: *Proceedings of the 25th USENIX Security Symposium*. 2016. URL: https://drownattack.com/drown-attack-paper.pdf.

[8] Sean Heelan. “Vulnerability Detection Systems: Think Cyborg, Not Robot”. In: *IEEE Security & Privacy* 3 (2011), pp. 74–77. URL: http://www.computer.org/csdl/mags/sp/2011/03/msp2011030074-abs.html.

[11] Fabian Yamaguchi. “Pattern-Based Vulnerability Discovery”. University Göttingen, 2015. URL: http://hdl.handle.net/11858/00-1735-0000-0023-9682-0.

[12] Laurence Tratt. “Dynamically Typed Languages”. In: *Advances in Computers* 77 (2009), pp. 149–184. DOI: 10.1016/S0065-2458(09)01205-4.

[13] Magnus Madsen. “Static Analysis of Dynamic Languages”. Aarhus, 2015. URL: http://plg.uwaterloo.ca/~mmadsen/thesis.pdf.

[15] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. “Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning”. In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*. WOOT’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 13–13.

[16] Yonghee Shin et al. “Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities”. In: *IEEE Transactions on Software Engineering* 37.6 (Nov. 2011), pp. 772–787. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.81.

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

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

[19] Fabian Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: *Security and Privacy (SP), 2014 IEEE Symposium On*. IEEE, 2014, pp. 590–604.

[25] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: *Queue* 10.1 (2012), p. 20. URL: http://dl.acm.org/citation.cfm?id=2094081.

[30] Tom M. Mitchell. “Machine Learning”. In: McGraw-Hill, 1997, pp. 52–80. ISBN: 0-07-115467-1.

[31] G. V. Kass. “An Exploratory Technique for Investigating Large Quantities of Categorical Data”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 29.2 (1980), pp. 119–127. ISSN: 0035-9254. DOI: 10.2307/2986296. JSTOR: 2986296.

[32] Jerome H. Friedman. “Multivariate Adaptive Regression Splines”. In: *The annals of statistics* (1991), pp. 1–67. JSTOR: 2241837.

[33] Tin Kam Ho. “A Data Complexity Analysis of Comparative Advantages of Decision Forest Constructors”. In: *Pattern Analysis & Applications* 5.2 (June 2002), pp. 102–112. ISSN: 1433-7541. DOI: 10.1007/s100440200009.

[34] Pedro Domingos and Michael Pazzani. “On the Optimality of the Simple Bayesian Classifier under Zero-One Loss”. In: *Machine learning* 29 (2-3 1997), pp. 103–130. URL: http://link.springer.com/article/10.1023/A:1007413511361.

[35] Alexandru Niculescu-Mizil and Rich Caruana. “Predicting Good Probabilities with Supervised Learning”. In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML ’05. Bonn, Germany: ACM, 2005, pp. 625–632. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102430.

[36] Nir Friedman, Dan Geiger, and Moises Goldszmidt. “Bayesian Network Classifiers”. In: *Machine Learning* 29 (2-3 Nov. 1, 1997), pp. 131–163. ISSN: 0885-6125, 1573-0565. DOI: 10.1023/A:1007465528199. URL: https://link.springer.com/article/10.1023/A:1007465528199.

[38] D. R. Cox. “The Regression Analysis of Binary Sequences”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 20.2 (1958), pp. 215–242. ISSN: 0035-9246. JSTOR: 2983890.

[39] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. “Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey”. In: *ACM Computing Surveys* 50.4 (Aug. 25, 2017), pp. 1–36. ISSN: 03600300. DOI: 10.1145/3092566. URL: http://dl.acm.org/citation.cfm?doid=3135069.3092566.

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

[45] F. Pedregosa et al. “Scikit-Learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

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

[64] Frances E. Allen. “Control Flow Analysis”. In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. URL: http://doi.acm.org/10.1145/800028.808479.

[65] Alfred V. Aho and Alfred V. Aho, eds. *Compilers: Principles, Techniques, & Tools*. 2nd ed. OCLC: ocm70775643. Boston: Pearson/Addison Wesley, 2007. 1009 pp. ISBN: 978-0-321-48681-3.

[67] Karl Pearson. “On the Criterion That a Given System of Deviations from the Probable in the Case of a Correlated System of Variables Is Such That It Can Be Reasonably Supposed to Have Arisen from Random Sampling”. In: *Breakthroughs in Statistics*. Springer Series in Statistics. Springer, New York, NY, 1992, pp. 11–28. ISBN: 978-0-387-94039-7 978-1-4612-4380-9. DOI: 10.1007/978-1-4612-4380-9_2. URL: https://link.springer.com/chapter/10.1007/978-1-4612-4380-9_2.

[69] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities”. In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, 6–pp. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1624016.

[71] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. “Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives”. In: ACM Press, 2014, pp. 63–74. ISBN: 978-1-4503-2744-2. DOI: 10.1145/2566486.2568024. URL: http://dl.acm.org/citation.cfm?doid=2566486.2568024.

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

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

[84] Fabian Yamaguchi et al. “Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery”. In: ACM Press, 2013, pp. 499–510. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516665. URL: http://dl.acm.org/citation.cfm?doid=2508859.2516665 (visited on 05/23/2016).

[85] Fabian Yamaguchi et al. “Automatic Inference of Search Patterns for Taint-Style Vulnerabilities”. In: IEEE, May 2015, pp. 797–812. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.54. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7163061.

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

[87] Lwin Khin Shar and Hee Beng Kuan Tan. “Predicting SQL Injection and Cross Site Scripting Vulnerabilities through Mining Input Sanitization Patterns”. In: *Information and Software Technology* 55.10 (Oct. 2013), pp. 1767–1780. ISSN: 09505849. DOI: 10.1016/j.infsof.2013.04.002. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584913000852.

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

[89] Lwin Khin Shar, Lionel C. Briand, and Hee Beng Kuan Tan. “Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning”. In: *IEEE Transactions on Dependable and Secure Computing* 12.6 (Nov. 1, 2015), pp. 688–707. ISSN: 1545-5971. DOI: 10.1109/TDSC.2014.2373377. URL: http://ieeexplore.ieee.org/document/6963442/.

OTHER SOURCES

[2] National Vulnerability Database. *NVD - Statistics Search*. 2016. URL: https://web.nvd.nist.gov/view/vuln/statistics (visited on 06/26/2016).

[3] Stefan Frei. *The Known Unknowns*. Dec. 5, 2013. URL: https://www.nsslabs.com/blog/measuring-the-known-unknowns-in-cyber-security/.

[4] Symantec. *Internet Security Threat Report 2016*. 2016. URL: https://www.symantec.com/security-center/threat-report (visited on 06/26/2016).

[5] MITRE. *CVE - CVE-2014-6271*. 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271 (visited on 06/26/2016).

[9] Michal Zalewski. *Lcamtuf's Blog: Symbolic Execution in Vuln Research*. Apr. 2, 2015. URL: http://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html (visited on 07/10/2016).

[10] Tiobe - The Software Quality Company. *TIOBE Index*. URL: http://www.tiobe.com/tiobe_index (visited on 09/25/2017).

[14] Paul Biggar and David Gregg. *Static Analysis of Dynamic Scripting Languages*. 2009. URL: https://paulbiggar.com/research/wip-optimizer.pdf (visited on 07/23/2016).

[20] Microsoft Security Response Center. *Definition of a Security Vulnerability*. URL: https://msdn.microsoft.com/en-us/library/cc751383.aspx (visited on 07/10/2016).

[21] Robert Shirey. *Internet Security Glossary, Version 2*. 2016. URL: https://tools.ietf.org/html/rfc4949 (visited on 06/06/2016).

[22] SecTools. *Vulnerability Scanners – SecTools Top Network Security Tools*. URL: http://sectools.org/tag/vuln-scanners/ (visited on 07/23/2016).

[23] Michal Zalewski. *Technical "Whitepaper" for Afl-Fuzz*. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 07/23/2016).

[24] Michael Rash. *A Collection of Vulnerabilities Discovered by the AFL Fuzzer (Afl-Fuzz)*. URL: https://github.com/mrash/afl-cve (visited on 07/23/2016).

[26] David A. Wheeler. *Flawfinder Home Page*. URL: http://www.dwheeler.com/flawfinder/ (visited on 07/30/2016).

[27] OWASP. *Category:OWASP Top Ten Project - OWASP*. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (visited on 07/30/2016).

[28] MITRE. *CVE - CVE-2014-4954*. 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4954 (visited on 07/22/2016).

[29] MITRE. *CVE - CVE-2013-5003*. 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5003 (visited on 08/28/2016).

[37] David Freedman. *Statistical Models: Theory and Practice*. OCLC: 497573530. Cambridge; New York: Cambridge University Press, 2009. ISBN: 978-0-511-60336-5 978-0-511-60414-0 978-0-511-81586-7. URL: http://dx.doi.org/10.1017/CBO9780511815867 (visited on 10/10/2017).

[40] *Red Hat Bugzilla*. URL: https://bugzilla.redhat.com/ (visited on 08/28/2016).

[41] MITRE. *CVE - Common Vulnerabilities and Exposures (CVE)*. URL: https://cve.mitre.org/ (visited on 08/21/2016).

[43] *SAMATE - Software Assurance Metrics And Tool Evaluation Project Main Page*. URL: https://samate.nist.gov/Main_Page.html (visited on 08/27/2017).

[44] Stanisław Pitucha. *Phply: PHP Parser Written in Python Using PLY*. URL: https://github.com/viraptor/phply.

[46] Jorrit Kronjee. *WIRECAML: Weakness Identification Research Employing CFG Analysis and Machine Learning*. URL: https://github.com/jorkro/wirecaml (visited on 02/03/2018).

[47] *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/ (visited on 08/27/2017).

[48] *Joomla!* URL: https://www.joomla.org/ (visited on 08/27/2017).

[49] *Moodle - Open-Source Learning Platform*. URL: https://moodle.org/ (visited on 08/27/2017).

[50] *MyBB - Free and Open Source Forum Software*. URL: https://mybb.com/ (visited on 08/27/2017).

[51] *Piwigo Is Open Source Photo Gallery Software for the Web | Piwigo.Org*. URL: http://piwigo.org/ (visited on 08/27/2017).

[52] *Tiki Wiki CMS Groupware*. URL: https://tiki.org/HomePage (visited on 08/27/2017).

[53] *TYPO3 - The Enterprise Open Source CMS*. URL: http://typo3.org/ (visited on 08/27/2017).

[54] *Blog Tool, Publishing Platform, and CMS — WordPress*. URL: https://wordpress.org/ (visited on 08/27/2017).

[55] *Zen Cart*. URL: https://www.zen-cart.com/ (visited on 08/27/2017).

[56] *Kajona*. URL: https://www.kajona.de/ (visited on 08/27/2017).

[57] *ownCloud.Org*. URL: https://owncloud.org/ (visited on 08/27/2017).

[58] *phpMyAdmin*. URL: https://www.phpmyadmin.net/ (visited on 08/27/2017).

[59] Bertrand Stivalet. *PHP-Vuln-Test-Suite-Generator: PHP Synthetic Test Cases Generator*. URL: https://github.com/stivalet/PHP-Vuln-test-suite-generator.

[61] *Pull Request #1 · Stivalet/PHP-Vuln-Test-Suite-Generator*. URL: https://github.com/stivalet/PHP-Vuln-test-suite-generator/pull/1 (visited on 03/17/2018).

[62] *Pull Request #2 · Stivalet/PHP-Vuln-Test-Suite-Generator*. URL: https://github.com/stivalet/PHP-Vuln-test-suite-generator/pull/2 (visited on 03/17/2018).

[63] *Pickle — Python Object Serialization — Python 3.5.4 Documentation*. URL: https://docs.python.org/3.5/library/pickle.html (visited on 10/22/2017).

[66] Eibe Frank, Mark Hall, and Ian H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Fourth Edition. Morgan Kaufmann, 2016.

[68] *Source Code Security Analyzers - SAMATE*. URL: https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html (visited on 07/02/2017).

[70] *RIPS - Free PHP Security Scanner Using Static Code Analysis*. URL: http://rips-scanner.sourceforge.net/ (visited on 07/01/2017).

[72] *WAP - Web Application Protection*. URL: http://awap.sourceforge.net/ (visited on 07/01/2017).

[73] Michael Scovetta. *Yasca*. URL: http://www.scovetta.com/yasca.html (visited on 05/17/2017).

[74] Oliver Klee. *Pixy: Pixy Is a Scanner Static Code Analysis Tools That Scans PHP Applications for Security Vulnerabilities*. URL: https://github.com/oliverklee/pixy.

[75] *RIPS - Static Code Analysis for PHP Security Vulnerabilities*. URL: https://www.ripstech.com/ (visited on 07/01/2017).

[76] Sebastian Bergmann. *Phploc: A Tool for Quickly Measuring the Size of a PHP Project*. URL: https://github.com/sebastianbergmann/phploc (visited on 02/09/2018).

[77] MITRE. *CVE - CVE-2018-6883*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6883 (visited on 02/13/2018).

[78] *Static Analysis, Static Application Security Testing, SAST | Micro Focus*. URL: https://software.microfocus.com/en-us/software/sca (visited on 11/18/2017).

[79] Terence Parr. *ANTLR*. URL: http://www.antlr.org/ (visited on 03/17/2018).

[80] *Grammars-v4: Grammars Written for ANTLR V4*. URL: https://github.com/antlr/grammars-v4 (visited on 03/17/2018).

[81] *Pull Request #8459 · Scikit-Learn/Scikit-Learn*. URL: https://github.com/scikit-learn/scikit-learn/pull/8459 (visited on 03/17/2018).