Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives

Ibéria Medeiros
University of Lisboa, Faculty of Sciences, LaSIGE
ibemed@gmail.com

Nuno F. Neves
University of Lisboa, Faculty of Sciences, LaSIGE
nuno@di.fc.ul.pt

Miguel Correia
University of Lisboa, Instituto Superior Técnico, INESC-ID
miguel.p.correia@ist.utl.pt

ABSTRACT

Web application security is an important problem in today's internet. A major cause of this status is that many programmers do not have adequate knowledge about secure coding, so they leave applications with vulnerabilities. An approach to solve this problem is to use source code static analysis to find these bugs, but these tools are known to report many false positives that make hard the task of correcting the application. This paper explores the use of a hybrid of methods to detect vulnerabilities with less false positives. After an initial step that uses taint analysis to flag candidate vulnerabilities, our approach uses data mining to predict the existence of false positives. This approach reaches a trade-off between two apparently opposite approaches: humans coding the knowledge about vulnerabilities (for taint analysis) versus automatically obtaining that knowledge (with machine learning, for data mining). Given this more precise form of detection, we do automatic code correction by inserting fixes in the source code. The approach was implemented in the WAP tool [1] and an experimental evaluation was performed with a large set of open source PHP applications.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and protection; D.2.4 [**Software/ Program Verification**]: Validation; D.2.4 [**Software/ Program Verification**]: Statistical methods

Keywords

Web applications; security; input validation vulnerabilities; false positives; source code analysis; automatic protection; software security; data mining.

1. INTRODUCTION

In two decades of existence, the Web evolved from a platform to access hypermedia to a framework for running complex *web applications*. These applications appear in many forms, from small home-made to large-scale commercial services such as Gmail, Office 365, and Facebook. From the

---

[1] http://awap.sourceforge.net/

security point of view, web applications have been a major source of problems. For example, a recent report indicates an increase of web attacks of around 33% in 2012 [35].

An important reason for the insecurity of web applications is that many programmers lack the appropriate knowledge about secure coding, so they leave applications with flaws [7]. However, the mechanisms for web application security fall in two extremes. On one hand, there are techniques that put the programmer aside, e.g., web application firewalls and other runtime protections [11, 24, 38]. On the other hand, there are techniques that discover vulnerabilities but put the burden of removing them on the programmer, e.g., black-box testing [1, 3, 12] and static analysis [13, 15, 28].

The paper explores an approach for automatically protecting web applications while keeping the programmer in the loop. The approach consists in analyzing the web application source code searching for vulnerabilities and inserting fixes in the same code to correct these flaws. The programmer is kept in the loop by being allowed to understand where the vulnerabilities were found and how they were corrected. This contributes directly for the security of web applications by removing vulnerabilities, and indirectly by letting the programmers learn from their mistakes. This last aspect is enabled by inserting fixes that follow common security coding practices, so programmers can learn these practices by seeing the vulnerabilities and how they were removed.

In this paper we explore the use of a novel hybrid of methods to detect vulnerabilities. Static analysis is an effective mechanisms to find vulnerabilities in source code, but tends to report many false positives (non-vulnerabilities) due to its undecidability [16]. This problem is particularly difficult with languages such as PHP that are weakly typed and not formally specified [25]. Therefore, we complement a form of static analysis – taint analysis – with the use of data mining to predict the existence of false positives. This approach combines two apparently opposite approaches: humans coding the knowledge about vulnerabilities (for taint analysis) versus automatically obtaining that knowledge (with supervised machine learning supporting data mining). Interestingly this dichotomy has been present for long in another area of security, intrusion detection. As its name suggests, signature- or knowledge-based intrusion detection relies on knowledge about intrusions coded by humans (signatures), whereas anomaly-based detection relies on models of normal behavior created using machine learning. Nevertheless, anomaly-based detection has been much criticized [33] and has very limited commercial used today. We show that the combination of the two broad approaches of human-coded knowledge and learning can be effective for vulnerability detection.

The paper also describes the design of the *Web Application Protection* (WAP) tool, a proof of concept prototype that implements our approach. WAP analyzes and removes input validation vulnerabilities from code[2] written in PHP 5, which according to a recent report is used by more than 77% of the web applications [14]. Currently WAP assumes that the background database is MySQL, DB2 or PostgreSQL, although it can be extended to support others.

Designing and implementing WAP was a challenging task. The tool does taint analysis of PHP programs, a form of data flow analysis. To do a first reduction of the number of false positives, the tool performs global, interprocedural and context-sensitive analysis, which means that data flows are followed even when they enter new functions and other modules (other files). This involves the management of several data structures, but also to deal with global variables (that in PHP can appear anywhere in the code, simply by preceding the name with *global* or through the $_GLOBALS array) and resolving module names (which can even contain paths taken from environment variables). Handling object orientation with the associated inheritance and polymorphism was also a considerable challenge.

To predict the existence of false positives we introduce the novel idea of assessing if the vulnerabilities detected are false positives using data mining. To do this assessment, we measure attributes of the code that we observed to be associated with the presence of false positives, and use a classifier to flag every vulnerability as false positive or not. We explore the use of several classifiers: ID3, C4.5/J48, Random Forest, Random Tree, K-NN, Naive Bayes, Bayes Net, MLP, SVM, and Logistic Regression. Classifiers are automatically configured using machine learning based on labeled vulnerability data.

The tool does not detect and remove every possible vulnerability, something that is not even possible due to the mentioned undecidability, but to cover a considerable number of classes of vulnerabilities: SQL injection (SQLI), cross-site scripting (XSS), remote file inclusion, local file inclusion, directory traversal/path traversal, source code disclosure, insecure use of *eval*, and OS command injection. The first two continue to be among the highest positions of the OWASP Top 10 of web application security risks in 2013 [40], whereas the rest are also known to have a high risk. The tool can be extended with more classes of flaws, but this set is enough to demonstrate the concept.

We evaluated the tool experimentally by running it with both simple synthetic code with vulnerabilities inserted on purpose and with 35 open PHP web applications available in the internet, adding up to more than 2,800 files and 470,000 lines of code. Our results suggest that the tool is capable of finding and correcting the vulnerabilities from the classes it was programmed to handle.

The main contributions of the paper are: (1) an approach for improving the security of web applications by combining detection and automatic correction of vulnerabilities in web applications; (2) a combination of taint analysis and data mining techniques to identify vulnerabilities with low false positives; (3) a tool that implements that approach for web applications written in PHP with several database management systems; (4) a study of the configuration of the data mining component and an experimental evaluation of the tool with a considerable number of open source PHP applications.

---

[2]We use the terms PHP code, script, and programs interchangeably in the paper.

2. RELATED WORK

There is a large corpus of related work so we just summarize the main areas by discussing representative papers, while leaving many others unreferenced for lack of space.

Detecting vulnerabilities with static analysis.

Static analysis tools automate the auditing of code, either source, binary or intermediate. In the paper we use the term *static analysis* in a narrow sense to designate static analysis of source code to detect vulnerabilities [13, 15, 28, 34].

The most interesting static analysis tools do *semantic analysis* based on the abstract syntax tree (AST) of a program. *Data flow analysis* tools follow the data paths inside a program to detect security problems. The most commonly used data flow analysis technique for security analysis is *taint analysis*, which marks data that enters the program as *tainted* and detects if it reaches sensitive functions (e.g., *mysql_query* in PHP). Taint analysis tools like CQUAL [28] and Splint [9] (both for C code) use two qualifiers to annotate source code: the *untainted* qualifier indicates either that a function/parameter returns trustworthy data (e.g., a sanitization function) or that a parameter of a function only accepts trustworthy data (e.g., *mysql_query*); the *tainted* qualifier means that a function or a parameter return non-trustworthy data (e.g., functions that read user input).

Pixy [15] uses taint analysis for verifying PHP code, but extends it with *alias analysis* that takes into account the existence of aliases, i.e., of two or more variable names that are used to denominate the same variable. SaferPHP uses taint analysis to detect certain semantic vulnerabilities in PHP code: denial of service due to infinite loops and unauthorized operations in databases [34]. WAP also does taint analysis and alias analysis for detecting vulnerabilities, although it goes further by also correcting the code. Furthermore, Pixy does only module-level analysis, whereas WAP does global analysis (i.e., the analysis is not limited to a module/file, but can involve several).

Vulnerabilities and data mining.

Data mining has been used to predict the presence of software defects [2, 5, 17]. These works were based on code attributes such as numbers of lines of code, code complexity metrics, and object-oriented features. Some papers went one step further in the direction of our work by using similar metrics to predict the existence of vulnerabilities in source code [20, 32, 37]. They used attributes such as past vulnerabilities and function calls [20], or code complexity and developer activities [32]. On the contrary of our work, these others did not aim to detect bugs and identify their location, but to assess the quality of the software in terms of prevalence of defects/vulnerabilities.

Shar and Tan presented PhpMinerI and PhpMinerII, two tools that use data mining to assess the presence of vulnerabilities in PHP programs [29, 30]. These tools extract a set of attributes from program slices, then apply data mining algorithms to those attributes. The data mining process is not really done by the tools, but by the WEKA tool [41]. More recently the authors evolved this idea to use also traces or program execution [31]. Their approach is an evolution of the previous works that aimed to assess the prevalence of vulnerabilities, but obtaining a higher accuracy. WAP is quite different because it has to identify the location of vulnerabilities in the source code, so that it can correct them with fixes.

Moreover, WAP does not use data mining to identify vulnerabilities but to predict if vulnerabilities found by taint analysis are really so or if, on the contrary, they are false positives.

Correcting vulnerabilities.

We propose to use the output of static analysis to remove vulnerabilities automatically. We are aware of a few works that use approximately the same idea of first doing static analysis then doing some kind of protection, but mostly for the specific case of SQL injection and without attempting to insert fixes in a way that can be replicated by a programmer. AMNESIA does static analysis to discover all SQL queries – vulnerable or not– and in runtime checks if the call being made satisfies the format defined by the programmer [10]. Buehrer et al. do something similar by comparing in runtime the parse tree of the SQL statement before and after the inclusion of user input [6]. WebSSARI does also static analysis and inserts runtime guards, but no details are available about what the guards are or how they are inserted [13]. Merlo et al. present a tool that does static analysis of source code, performs dynamic analysis to build syntactic models of legitimate SQL queries, and generates code to protect queries from input that aims to do SQLI [19]. None of these works use data mining or machine learning.

Dynamic analysis and protection.

Static analysis is only one among a set of techniques to detect vulnerabilities. *Dynamic analysis* or *testing* techniques find bugs or vulnerabilities while executing the code [12]. *Web vulnerability scanners* use vulnerability signatures to detect if they exist in a web site, but this approach has been shown to lead to high ratios of false negatives [36]. *Fuzzing* and *fault/attack injection tools* also search for vulnerabilities but they try a wide range of possible inputs, instead of just vulnerability signatures [3, 1].

There are also *dynamic taint analysis* tools that do taint analysis in runtime. For example, PHP Aspis does dynamic taint analysis of PHP applications with the objective of blocking XSS and SQLI attacks [22]. Similarly to AMNESIA and [6], CANDID compares the structure of a SQL query before and after the inclusion of user input [4]. However, it does not do static analysis but dynamic analysis to infer the original structure of the query.

3. INPUT VALIDATION VULNERABILITIES

This section presents briefly the vulnerabilities handled by the WAP tool. The main problem in web application security lies in the improper validation of user input, so this is the kind of vulnerabilities we currently consider. Inputs enter an application through *entry points* (e.g., $_GET) and exploit a vulnerability by reaching a *sensitive sink* (e.g., *mysql_query*). Most attacks involve mixing normal input with metacharacters or metadata (e.g., ', OR), so applications can be protected by placing *sanitization functions* in the paths between entry points and sensitive sinks.

SQL injection (SQLI) vulnerabilities are caused by the use of string-building techniques to execute SQL queries. Figure 1 shows PHP code vulnerable to SQLI. This script inserts in a SQL query (line 4) the username and password provided by the user (lines 2, 3). If the user is malicious he can provide as username *admin' - -* , causing the script to execute a query that returns information about the user *admin* without the need

```
1: $conn = mysql_connect("localhost","username","password");
2: $user = $_POST['user'];
3: $pass = $_POST['password'];
4: $query = "SELECT * FROM users WHERE username='$user'
AND password='$pass' ";
5: $result = mysql_query($query);
```

Figure 1: Login PHP script that is vulnerable to SQLI.

of providing a password: *SELECT * FROM users WHERE username='admin' - - ' AND password='foo'*

This vulnerability can be removed either by sanitizing the inputs (e.g., preceding with a backslash metacharacters such as the prime) or by using prepared statements. We opted by the former because it requires simpler modifications to the code. Sanitization depends on the sensitive sink, i.e., on the way in which the input is used. For SQL and the MySQL database, PHP provides the *mysql_real_escape_string* function. The username could be sanitized in line 2: *$user = mysql_real_escape_string($_POST['user']);* (the same should be done in line 3).

Cross-site scripting (XSS) attacks execute malicious code (e.g., JavaScript) in the victim's browser. Differently from the other attacks we consider, a XSS attack is not against a web application itself, but against its users. There are three main classes of XSS attacks depending on how the malicious code is sent to the victim (reflected or non-persistent, stored or persistent, and DOM-based) but we describe only reflected XSS for briefness. A script vulnerable to XSS can have a single line: *echo $_GET['username'];*. The attack involves convincing the user to click on a link that accesses the web application, sending it a script that is reflected by the *echo* instruction and executed in the browser. This kind of attack can be prevented by sanitizing the input and/or by encoding the output. The latter consists in encoding metacharacters such as < and > in a way that they are interpreted as normal characters, instead of HTML metacharacters.

We present the other vulnerabilities handled by WAP only briefly for lack of space (a longer explanation can be found on a companion web page [18]). A remote file inclusion (RFI) vulnerability allows attackers to embed a remote file containing PHP code in the vulnerable program. Local file inclusion (LFI) differs from RFI by inserting in a script a file from the file system of the web application, not a remote file. A directory traversal or path traversal (DT/PT) attack consists in an attacker accessing unpredicted files, possibly outside the web site directory. Source code disclosure (SCD) attacks dump source code and configuration files. An operating system command injection (OSCI) attack consists in forcing the application to execute a command defined by the attacker. An insecure *eval* vulnerability allows an attacker to supply code that is executed by an *eval* statement.

4. THE APPROACH

The approach proposed involves detecting and correcting vulnerabilities in the source code, which is tightly related to *information flows*: detecting problematic information flows in the source code; modifying the source code to block these flows. The notion of information flow is central to two of the three main security properties: confidentiality and integrity [27]. Confidentiality is related to private information flowing to public objects, whereas integrity is related to untrusted data flowing to trusted objects. Availability is an exception as it is not directly related to information flow.

The approach proposed in the paper is, therefore, about *information-flow security* in the context of web applications.
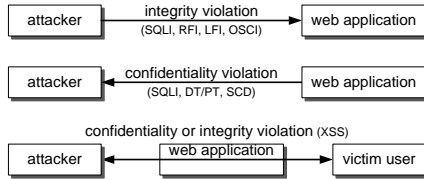
Figure 2: Information flows that exploit web vulnerabilities. The arrows indicate information flows.

We are mostly concerned with the server-side of these applications, which is normally written in a programming language such as PHP, Java or Perl. Therefore, the problem can be considered to be a case of language-based information-flow security, which is a topic much investigated in recent years [26, 13, 21]. Attacks against web vulnerabilities can be expressed in terms of violations of information-flow security. Figure 2 shows the information flows that exploit each of the vulnerabilities of Section 3. The information flows are labeled with the vulnerabilities that usually permit them. There are a few rarer cases that are not represented (e.g., RFI or LFI can also be used to violate confidentiality, but this is not usual). XSS is different from other vulnerabilities because the victim is not the web application itself, but a user.

Our approach can be considered to be a way of enforcing information-flow security at language-level. The tool detects the possibility of existing the information flows represented in the figure, and modifies the web application source code to prevent them.

The approach can be implemented as a sequence of steps:

1. Taint analysis: parse the source code; generate the abstract syntax tree (AST); do taint analysis based on the AST and generate trees describing candidate vulnerable control-flow paths (starting in an entry point and finishing in a sensitive sink);

2. Data mining: obtain attributes from the candidate vulnerable control-flow paths and use a classifier to predict if each candidate vulnerability is a false positive or not;

3. Code correction: given the control-flow paths trees of vulnerabilities predicted not to be false positives, identify the vulnerabilities, the fixes to insert and the places where they have to be inserted; assess the probabilities of the vulnerabilities being false positives and modify the source code with the fixes (depending on the probabilities);

4. Feedback: provide feedback to the programmer based on the data collected in the previous steps (vulnerable paths, vulnerabilities, fixes, false positive probability).

5. TAINT ANALYSIS

The taint analyzer is a static analysis tool that operates over an AST. The ASTs are created by a lexer and a parser for PHP 5 that we created using ANTLR (ANother Tool for Language Recognition) [23]. ANTLR also builds tree walkers. These classes are used to navigate through the nodes of the tree programmatically and collect data about the program represented by the AST.

Taint analysis is performed using the tree walkers to navigate through the AST. In the beginning of the taint analysis all *symbols* (variables, functions) are *untainted* unless they are an entry point (e.g., $a in $a = $_POST['a']). The tree walkers build a *tainted symbol table* (TST) in which every cell is a

program statement from which we want to collect data. Each cell contains a subtree of the AST plus some data. For each symbol several data items are stored, e.g., the symbol name and the line number of the statement. While the tree walkers are building the TST, they also build a *tainted execution path tree* (TEPT). Each branch of the TEPT corresponds to a tainted variable (never an untainted variable) and contains one node for each statement that passes the taintedness from the entry point until the tainted variable.

The taint analysis consists in traveling though the TST. If a variable is tainted this state is propagated to symbols that depend on it, e.g., function parameters or variables that are updated using it. This requires updating the TEPT with the variables that become tainted. On the contrary, the state of a variable is not propagated if it is untainted or if it is an argument of a PHP sanitization function. The process finishes when all symbols are analyzed this way.

During the analysis, whenever a variable that is passed to a sensitive sink becomes tainted, the code corrector is activated in order to prepare the correction of the code. However, the code is updated and stored in a file only at the end of the process, when the analysis finishes and all the corrections that have to be made are known.

WAP does *global*, *interprocedural* and *context-sensitive* analysis. Interprocedural means that it analyzes the propagation of taintedness when functions are called, instead of analyzing functions in isolation. The analysis being global means that the propagation of taintedness is also propagated when functions in different modules are called. Being context-sensitive means that the result of the analysis of a function is propagated only to the point of the program where the call was made (context-insensitive analysis propagates results to all points of the program where the function is called) [15].

Table 1 shows the functions used to fix SQL injection vulnerabilities (the rest can be found at [18]). For SQLI the tool uses sanitization functions provided by PHP (column on the right hand side of the table), but also replaces some problematic, deprecated, tainted sensitive sinks (*mysql_db_query*, *mysqli_execute*) by non-deprecated functions with similar functionality (*mysql_query*, *mysqli_stmt_execute*).

Vuln.	Entry points	Sensitive sinks	Sanitization functions
SQLI	$_GET $_POST $_COOKIE $_REQUEST HTTP_GET_VARS HTTP_POST_VARS HTTP_COOKIE_VARS HTTP_REQUEST_VARS	mysql_query mysql_unbuffered_query mysql_db_query	mysql_real_escape_string
		mysqli_query mysqli_real_query mysqli_master_query mysqli_multi_query	mysqli_real_escape_string
		mysqli_stmt_execute mysqli_execute	mysqli_stmt_bind_param

Table 1: Sanitization functions used to fix PHP code for SQLI vulnerabilities.

6. PREDICTING FALSE POSITIVES

The static analysis problem is known to be related to Turing's halting problem, so undecidable for non-trivial languages [16]. In practice this difficulty is solved by making only partial analysis of some language constructs, leading static analysis tools to be unsound. In the analysis made by WAP this problem can appear, for instance, with string manipulation operations. The issue is what to do to the state of a tainted string (resp. untainted) that is processed by operations that, e.g., return a substring or concatenate it with another string. Both operations can untaint (resp. taint) the string, but it is uncertain if they do it or not. We opted by letting the string

tainted (resp. tainting it), which may lead to false positives but not false negatives.

The analysis might be further refined by considering, for example, the semantics of string manipulation functions, as in [39]. However, coding explicitly more knowledge in a static analysis tool is hard and requires a lot of effort, which typically has to be done for each class of vulnerabilities ([39] considers a single class of vulnerabilities, SQL injection). Moreover, the humans who code the knowledge have first to obtain it, which can be complex.

Data mining allows a different approach. Humans label samples of code as vulnerable or not, then machine learning techniques are used to configure the tool with the knowledge acquired though that process. Data mining then uses that data to analyze the code. The key idea is that there are symptoms in the code, e.g., the presence of string manipulation operations, that suggest that flagging a certain pattern as a vulnerability may be a false positive. The assessment has mainly two steps:

1. *definition of the classifier* – pick a representative set of vulnerabilities identified by the taint analyzer, verify if they are false positives or not, extract a set of attributes, analyze their statistical correlation with the presence of a false positive, evaluate candidate classifiers to pick the best for the case in point, define the parameters of the classifier;

2. *classification of vulnerabilities* – given the classifier, for every vulnerability found by WAP classify it as a false positive or not; this step is done by the WAP tool, whereas the other is part of the configuration of the tool.

6.1 Classification of vulnerabilities

Any process of classification involves two aspects: the *attributes* that allow classifying a sample, and the *classes* in which these samples are classified.

We identified the *attributes* by analyzing manually the vulnerabilities found by WAP's taint analyzer. We studied these vulnerabilities to understand if they were false positives. This involved both reading the source code and executing attacks against each vulnerability found to understand if it was attackable (true positive) or not (false positive). This data set is further discussed in Section 6.3.

From this analysis we found three main sets of attributes that led to false positives:

String manipulation: attributes that represent PHP functions or operators that manipulate strings. These are: substring extraction, concatenation, addition of characters, replacement of characters, and removal of white spaces. Recall that a data flow starts at an entry point, where it is marked tainted, and ends at a sensitive sink. The taint analyzer flags a vulnerability if the data flow is not untainted by a sanitization function before reaching the sensitive sink. These string manipulation functions may result in the sanitization of a data flow, but the taint analyzer does not have enough knowledge to change the status from tainted to untainted, so if a vulnerability is flagged it may be a false positive. The combinations of functions/operators that untaint a data flow are hard to establish, so this knowledge is not simple to retrofit into the taint analyzer.

Validation: set of attributes related to validation of user inputs, often involving an if-then-else construct, a know source of undecidability. We define the following attributes: data type (calls to is_int(), is_string()), is value set (isset()), control

pattern (preg_match()), test of belong to a white-list, test of belong to a black-list, error and exit functions that output an error if the user inputs do not pass a test. Similarly to what happens with string manipulations, any of these attributes can sanitize a data flow and lead to a false positive.

SQL query manipulation: attributes that have to do with the insertion of data in SQL queries (SQL injection only). We define attributes to: string inserted in a SQL aggregate function (AVG, SUM, MAX, MIN, etc.), string inserted in a FROM clause, test if data is numeric, and data inserted in a complex SQL query. Again any of these constructs can sanitize data of an otherwise considered tainted data flow.

For the string manipulation and validation sets the possible values for the attributes were two, corresponding to the presence (Y) or no presence (N) of at least one of these constructs in the sequence of instructions that propagates input from an entry point to a sensitive sink. The SQL query manipulation attributes can take a third value, not assigned (NA), when the vulnerability observed is other than SQLI.

We use only two classes to classify the vulnerabilities flagged by the taint analyzer: Yes (it is a false positive) and No (it is not a false positive, but a real vulnerability). Table 2 shows some examples of candidate vulnerabilities flagged by the taint analyzer, one per line. For each candidate vulnerability the table shows the values of some of the attributes (Y or N) and the class, which has to be assessed manually (supervized machine learning). The data mining component is configured using data like this.

6.2 Classifiers and metrics

As already mentioned, our data mining component uses machine learning algorithms to extract knowledge from a set of labeled data. This knowledge is afterwards used to classify candidate vulnerabilities detected by the taint analyzer as false positives or not. In this section we present the machine learning algorithms that we studied to identify the best to classify candidate vulnerabilities, which is the one implemented in WAP. We also discuss the metrics used to evaluate the merit of the classifiers.

Machine learning classifiers.

We studied machine learning classifiers of three classes:

Graphical/symbolic algorithms. The algorithms of this category represent knowledge using a graphical model. ID3, C4.5/J48, Random Tree and Random Forest are classifiers in which the graphical model is a decision tree. They use the information gain rate metric to decide how relevant an attribute is to classify an instance in a class, which is represented as a leaf of the tree. An attribute with a small information gain has a big entropy (degree of impurity of attribute or information quantity that the attribute offers to the obtention of the class), so it is less relevant for a class than another with a higher information gain. C4.5/J48 is a evolution of ID3 that does pruning of the tree, i.e., removes nodes with less relevant attributes (with a bigger entropy). The Bayesian Network is an acyclic graphical model, where the nodes are represented by random attributes from the data set.

Probabilistic algorithms. This category includes Naive Bayes (NB), K-Nearest Neighbor (K-NN) and Logistic Regression (LR). They classify an instance in the class that has the highest probability. NB is a simple statistical classifier based on the Bayes formula and the assumption of conditional independence of the probability distributions of the attributes. K-NN

Potential vulnerability		String manipulation					Validation						SQL query manipulation				Class
Type	Webapp	Extract substring	String concat.	Add char	Replace string	Remove whitesp.	Type checking	IsSet entry point	Pattern control	While list	Black list	Error / exit	Aggreg. function	FROM clause	Numeric entry point	Complex query	
SQLI	CurrentCost	Y	Y	Y	N	N	N	N	N	N	N	N	Y	N	N	N	Yes
SQLI	CurrentCost	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	Yes
SQLI	CurrentCost	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	No
XSS	emoncms	N	Y	N	Y	N	N	N	N	N	N	N	NA	NA	NA	NA	Yes
XSS	Mfm-0.13	N	Y	N	Y	Y	N	N	N	N	N	N	NA	NA	NA	NA	Yes
XSS St.	ZiPEC 0.32	N	Y	N	N	N	N	N	N	N	N	N	NA	NA	NA	NA	No
RFI	DVWA 1.0.7	N	N	N	N	N	N	N	N	Y	N	Y	NA	NA	NA	NA	Yes
RFI	SRD	N	N	N	Y	N	N	Y	N	N	N	N	NA	NA	NA	NA	No
RFI	SRD	N	N	N	Y	N	Y	Y	N	N	N	N	NA	NA	NA	NA	No
OSCI	DVWA 1.0.7	N	Y	N	Y	N	N	N	N	N	Y	N	NA	NA	NA	NA	Yes
XSS St.	vicnum15	Y	N	N	N	N	N	N	Y	N	N	N	NA	NA	NA	NA	Yes
XSS	Mfm-0.13	N	N	N	N	N	N	N	N	N	Y	N	NA	NA	NA	NA	Yes

Table 2: Attributes and class for some vulnerabilities.

classifies an instance in the class of its neighbors. LR uses regression analysis to classify an instance.

Neural network algorithms. This category has two algorithms: Multi-Layer Perceptron (MLP) and Support Vector Machine (SVM). These algorithms are inspired on the functioning of the neurons of the human brain. MLP is an artificial neural network classifier that maps sets of input data (values of attributes) onto a set of appropriate outputs (our class attribute, Yes or No). SVM is a evolution of MLP.

Classifier evaluation metrics.

To evaluate the classifiers we use ten metrics that are computed based mainly on four parameters of each classifier. These parameters are better understood in terms of the quadrants of a confusion matrix (Table 3). This matrix is a cross reference table where its columns are the observed instances and its rows the predicted results (instances classified by a classifier). For example, the cell *False negative (fn)* contains the number of instances that were wrongly classified as *No (not FP)*, but were false positives, i.e., should be classified in the class *Yes (FP)*. The evaluation metrics and tests are:

True positive rate of prediction (tpp). Measures how good the classifier is at predicting false positives. $tpp = tp/(tp+fn)$.

False positive rate of prediction (fpp). Measures how the classifier deviates from the correct classification of a candidate vulnerability as false positive. $fpp = fp/(fp + tn)$.

Precision of prediction (prfp). Measures the actual false positives that are correctly predicted in terms of percentage of total number of false positives. $prfp = tp/(tp + fp)$.

True positive of detection (tpd). Measures how the classifier is good at detecting real vulnerabilities. $tpd = tn/(tn + fp)$.

False positive of detection (fpd). Measures how the classifier deviates from the correct classification of a candidate vulnerability that was a real vulnerability. $fpd = fn/(fn + tp)$.

Precision of detection (prd). Measures the actual vulnerabilities (not false positives) that are correctly predicted in terms of a percentage of the total number of vulnerabilities. $prd = tn/(tn + fn)$.

Accuracy (acc). Measures the total of instances well classified. $acc = (tp + tn)/(tp + tn + fp + fn)$.

Precision (pr). Measures the actual false positives and vulnerabilities (not false positives) that are correctly predicted in terms of a percentage of total number of cases. $pr = average(prfp, prd)$.

Kappa statistic (kappa). Measures the concordance between the classes predicted and observed. Can be stratified in five categories: worst, bad, reasonable, good, very good, excellent.

		Observed	
		Yes (FP)	No (not FP)
Predicted	Yes (FP)	True positive (tp)	False positive (fp)
	No (not FP)	False negative (fn)	True negative (tn)

Table 3: Confusion matrix (to be filled for each classifier).

$kappa = (po - pe)/(1 - pe)$, where $po = acc$ and $pe = (P * P' + N * N')/(P + N)^2$ to $P = (tp + fn)$, $P' = (tp + fp)$, $N = (fp + tn)$ and $N' = (fn + tn)$.

Wilcoxon signed-rank test (wilcoxon). Test to compare classifier results with pairwise comparisons of the metrics *tpp* and *fpp* or *tpd* and *fpd*, with a benchmark result of *tpp, tpd > 70%* and *fpp, fpd < 25%* [8].

6.3 Selection of classifiers

Machine learning classifiers can have different performances. Here we use the previous metrics to select the best classifiers for our case. Our current data set has 76 vulnerabilities labeled with 15 attributes: 14 to characterize the candidates vulnerabilities and 1 to classify it as being false positive (Yes) or real vulnerability (No). For each candidate vulnerability, we used a version of WAP to collect the values of the 14 attributes and we manually classified them as false positives or not. Needless to say, understanding if a vulnerability was real or a false positive was a tedious process. The 76 potential vulnerabilities were distributed by the classes Yes and No with 32 and 44 instances, respectively. Figure 3 shows the number of occurrences of each attribute.

The 10 classifiers are implemented in WEKA, an open source data mining tool [41]. We use the tool for training and testing the ten classifiers with a commonly-used *10-fold cross validation* estimator. This estimator divides the data into 10 buckets, trains the classifier with 9 of them and tests it with the 10th. This process is repeated 10 times to test every bucket with the classifier trained with the rest. This method accounts for heterogeneities in the data set.

Table 4 shows the evaluation of the classifiers. The first observation is the rejection of the K-NN and Naive Bayes algorithms by the Wilcoxon signed-rank test. The rejection of the K-NN algorithm is explained by the classes Yes and No not being balanced, where the first class has less instances (32) than the second class (44), which leads to unbalanced numbers of neighbors and consequently to wrong classifications. The Naive Bayes rejection seems to be due to its naive assumption that attributes are conditionally independent and the weak occurrence of certain attributes.
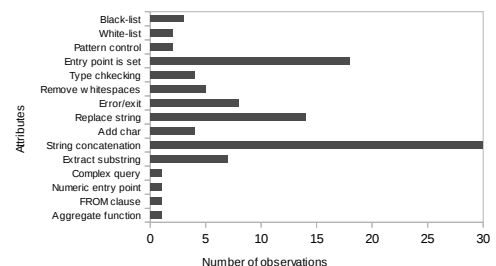
Figure 3: Number of occurrences registered by attribute in the original data set.

Measures (%)	ID3	C4.5/J48	Random Forest	Random Tree	K-NN	Naive Bayes	Bayes Net	MLP	SVM	Logistic Regression
tpp	75.0	81.3	78.1	78.1	71.9	68.8	78.1	75.0	81.3	84.4
fpp	0.0	13.6	4.5	0.0	0.0	13.6	13.6	0.0	4.5	2.3
prfp	100.0	81.3	92.6	100.0	100.0	78.6	80.6	100.0	92.9	96.4
tpd	100.0	86.4	95.5	100.0	100.0	86.4	86.4	100.0	95.5	97.7
fpd	25.0	18.8	21.9	21.9	28.1	31.3	21.9	25.0	18.8	15.6
prd	84.6	86.4	85.7	86.3	83.0	79.2	84.4	84.6	87.5	89.6
acc	89.5	82.2	88.2	90.8	82.9	78.9	82.9	89.5	89.5	92.1
(% #)	68	64	67	69	63	60	63	68	68	70
pr	91.0	84.2	88.6	92.0	86.8	78.9	82.8	91.0	89.8	92.5
kappa	77.0 very good	67.0 very good	75.0 very good	81.0 excellent	63.0 very good	56.0 good	64.0 very good	77.0 very good	78.0 very good	84.0 excellent
wilcoxon	accepted	accepted	accepted	accepted	rejected	rejected	accepted	accepted	accepted	accepted

Table 4: Evaluation of the machine learning models applied to the original data set.

The first four columns of the table are the decision tree models. These models select for the tree nodes the attributes that have higher information gain. The C4.5/J48 model prunes the tree to achieve better results. The branches that have nodes with weak information gain (higher entropy), i.e., the attributes with less occurrences, are removed (see Figure 3). However, an excessive tree pruning can result in a tree with too few nodes to do a good classification. This was what happened in our study, where J48 was the worst decision tree model. The results of ID3 validate our justification because this model is the J48 model without tree pruning and we can observe this model has better results comparing its accuracy and precision with J48: 89.5% against 82.2% and 91% against 84.2%, respectively. The best of the tree decision models is the Random Tree. The table shows that this model has the highest accuracy (90.8% that represents 69 of 76 instances well classified) and precision (92%), and the kappa value is in accordance (81% - excellent). This result is asserted by the 100% of prpf that tells us that all false positive instances were well classified in class Yes; also the 100% of tpd tells us that all instances classified in class No were well classifier.

The Bayes Net classifier is the third worst model in terms of kappa, which is justified by the random selection of attributes to the nodes of its acyclic graphical model. The selected attributes have high entropy so they insert noise in the model that results in bad performance.

The last three columns of Table 4 correspond to three models with good results. MLP is the neural network with best results and, curiously, with the same results as ID3. Logistic Regression (LR) was the classifier with best results. Table 5 shows the confusion matrix of LR, with values equivalent to those in Table 4. This model presents the highest accuracy (92.1%, which corresponds to 70 of 76 instances well classified), precision (92.5%) and an excellent value of kappa (84%). The prediction of false positives (first 3 rows of the Table 4) is very good with a great true positive rate of prediction ($tpp = 84.6\%$, 27 of 32 instances), very low false alarms ($fpp = 2.3\%$, 1 of 44 instances) and an excellent precision of prediction of false positives ($prfp = 96.4\%$, 27 of 28 instances). The detection of vulnerabilities (next 3 rows of the Table 4) is also very good, with a great true positive rate of detection ($tpd = 97.7\%$, 43 of 44 instances), low false alarms ($fpd = 15.6\%$, 5 of 32 instances) and a very good precision of detection of vulnerabilities ($prd = 89.6\%$, 43 of 48 instances).

		Observed	
		Yes (FP)	No (not FP)
Predicted	Yes (FP)	27	1
	No (not FP)	5	43

Table 5: Confusion matrix of the Logistic Regression classifier applied to our original data set.

Balanced data set.

To perform a better evaluation of the models with the suggestion given by the K-NN model that the classes are not balanced, we re-evaluated all models after applying the SMOTE filter to balance the classes [41]. This filter for smaller classes doubles its instances so it creates a balance. The result of applying the filter was an increase to 108 instances. Figure 4 shows the number of occurrences in this new data set. Comparing with the Figure 3, all attributes have increased their number of occurrences.
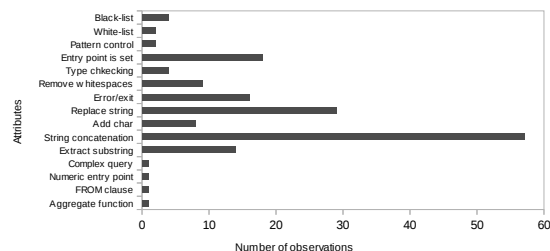

Figure 4: Number of occurrences registered by attribute in the balanced data set.

Table 6 shows the results of the re-evaluation with balanced classes. All models increased their performance and pass the Wilcoxon signed-rank test. The K-NN model has much better performance because the classes are now balanced. However, the kappa, accuracy and precision metrics show that the Bayes models continue to be the worst. The decision tree models present good results, with the Random Tree model again the best of them and the C4.5/J48 model still the worst. Observing Figure 4 there are attributes with very low occurrences that will be pruned in the C4.5/J48 model. To increase the performance of this model we remove the lowest information gain attribute (the biggest entropy attribute) and re-evaluate the model. There is an increase in its performance to 92.6% of pr, 93,7% of acc and 85.0% (excellent) of kappa, that is equal to the performance of the Random Tree model. Again the neural networks and LR models have very good performance, whereas *SVM is the best* of the three with approximately 92% of accuracy and pr, and 100% of precision in prediction of false positives ($prfp$) and detection of real vulnerabilities (tpd).

Main attributes.

To conclude the study of the best classifier we need to understand which attributes contribute most to a candidate vulnerability being a false positive. For that purpose we extracted from our data set 32 false positive instances and classified them in three sub-classes, one for each of the sets of attributes of Section 6.1: string manipulation, SQL query manipulation and validation. Then, we used WEKA to evaluate this new

Measures (%)	ID3	C4.5/J48	Random Forest	Random Tree	K-NN	Naive Bayes	Bayes Net	MLP	SVM	Logistic Regression
tpp	87.3	87.5	85.9	87.5	84.4	83.6	83.6	85.9	87.5	85.9
fpp	0.0	9.1	0.0	0.0	0.0	19.5	18.2	0.0	0.0	2.3
prfp	100.0	93.3	100.0	100.0	100.0	87.5	87.5	100.0	100.0	98.2
tpd	100.0	90.9	100.0	100.0	100.0	80.5	81.8	100.0	100.0	97.7
fpd	12.7	12.5	14.1	12.5	15.6	16.4	16.4	14.1	12.5	14.1
prd	84.6	83.3	83.0	84.6	81.5	75.0	76.6	83.0	84.6	82.7
acc	92.5	88.9	91.7	92.6	90.7	82.4	82.9	91.7	92.6	90.7
(% #)	99	96	99	100	98	89	89	99	100	98
pr	92.3	88.3	91.5	92.3	90.7	81.3	82.0	91.5	92.3	90.5
kappa	85.0 Excellent	77.0 Very Good	83.0 Excellent	85.0 Excellent	81.0 Excellent	64.0 Very Good	64.0 Very Good	83.0 Excellent	85.0 Excellent	81.0 Excellent
wilcoxon	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted	Accepted

Table 6: Evaluation of the machine learning models applied to the balanced data set.

data set with the classifiers that performed best (LR, Random Tree, and SVM), with and without balanced classes.

Table 7 shows the confusion matrix obtained using LR without balanced classes. The 32 instances are distributed by the three classes: string manipulation, SQL query manipulation (SQL), and validation, with 17, 3, and 12 instances respectively. The LR performance was $acc = 87.5\%$, $pr = 80.5\%$, and $kappa = 76\%$ (very good). All 17 instances of the string manipulation class were correctly classified. All 3 instances from the SQL class were classified in the string manipulation class, which is justified by the presence of the *concatenation* attribute in all instances. The 11 instances of the validation class were well classified, except one that was classified as string manipulation. This mistake is explained by the presence of the *add char* attribute in this instance.

		Observed		
		String manip.	SQL	Validation
Predicted	String manip.	17	3	1
	SQL	0	0	0
	Validation	0	0	11

Table 7: Confusion matrix of Logistic Regression classifier applied to false positives data set.

This analysis lead us to the conclusion that the string manipulation class is the one that most contributes to a candidate vulnerability being a false positive.

6.4 Implementation in WAP

The main conclusion of this study is that the Logistic Regression classifier is the best for classifying false positives with our data set. LR performed best with the original data set and ranked very close to the two first with the balanced data set. Therefore the Logistic Regression classifier was the one selected for WAP. This classifier is trained with our data set and then used to classify each vulnerability detected by the taint analyzer as false positive or not.

We plan to add labeled data to our data set incrementally, as we obtain it through the process of running the taint analyzer and studying if the candidate vulnerabilities are really vulnerabilities or not. Eventually this process will lead to a large data set that we will use to re-evaluate the classifiers. However, we believe that LR will be confirmed as the best option so we will not need to modify WAP.

6.5 Extending WAP for more vulnerabilities

Most literature on static analysis to detect vulnerabilities focus on one or two classes of vulnerabilities, e.g., SQLI [39] or XSS [15]. WAP on the contrary aims to be able to detect (and correct) a wide range of input validation vulnerabilities. Currently WAP handles eight classes of vulnerabilities, but to be generic it has to be configurable to handle more.

In this section we discuss how WAP can be extended to handle more vulnerabilities. We discuss it considering WAP's

three main components: taint analyzer, data mining component, and code corrector.

The taint analyzer has three pieces of data about each class of vulnerabilities: entry points, sensitive sinks, and sanitization functions. The entry points are always a variant of the same set (functions that read input parameters, e.g., $_GET), whereas the rest tend to be simple to identify once the vulnerability class is known.

The data mining component has to be trained with new knowledge about false positives for the new class. This training may be skipped at first and be improved incrementally when more data becomes available. For the training we need data about candidate vulnerabilities of that kind detected by the taint analyzer, which have to be labeled as true or false positives. Then, the attributes associated to the false positives have to be used to configure the classifier.

The code corrector needs essentially data about what sanitization function has to be used to handle that class of vulnerability and where it shall be inserted. Again this is doable once the new class if known and understood.

7. CODE CORRECTION

WAP does code correction automatically after the detection of the vulnerabilities is performed by the taint analyzer and the data mining component. The taint analyzer returns data about the vulnerability, including its class (e.g., SQLI) and the vulnerable slice of code. The code corrector uses this data to define the fix to insert and the place to insert it. Inserting a fix involves modifying a PHP file, which is then returned to the user of the tool.

A fix is a call to a function inserted in the line of code of the sensitive sink. The functionality of such function is to sanitize or validate the data that reaches the sensitive sink. Sanitization involves modifying the data to neutralize dangerous metacharacters or metadata, if they are present. Validation involves checking the data and executing the sensitive sink or not depending on this verification. A fix is inserted in the line of the sensitive sink instead of, for example, the line of the entry point, to avoid it to interfere with other code that used the sanitized variable.

Table 8 shows the fixes, how they are inserted and other related information. The *san_sqli* fix applies PHP sanitization functions (e.g., *mysql_real_escape_string*) and lets the sensitive sink be executed with its arguments sanitized. The SQLI sanitization functions precedes any malicious meta-character with a backslash and replaces others by their literal, e.g., \n by '\n'. For the XSS vulnerability class the fixes use functions from the OWASP PHP Anti-XSS Library. All of them replace malicious metacharacters by their HTML entity, for example < by <. For stored XSS the sanitization function *addslashes* is used and the validation process verifies in runtime if an attempt of exploitation occurs, outputting an alarm message if that is the case. The fixes for the others classes of

Vulnerability	Fix				Applied to	Function	Output	
	Sanitization		Validation				Alarm	Stop
	Addition	Substitution	Black-list	White-list			message	execution
SQLI	X	X			sensitive sink	san_sqli	-	No
Reflected XSS		X			sensitive sink	san_out	-	No
Stored XSS	X	X		X	sensitive sink	san_wdata	X	No
Stored XSS		X		X	sensitive sink	san_rdata	X	No
RFI			X		sensitive sink	san_mix	X	Yes
LFI			X		sensitive sink	san_mix	X	Yes
DT /PT			X		sensitive sink	san_mix	X	Yes
SCD			X		sensitive sink	san_mix	X	Yes
OSCI			X		sensitive sink	san_osci	X	Yes
Eval Injection	The fix and the output depend of the type of the vulnerability above							

Table 8: Functioning of the fixes and its output when executed.

vulnerabilities were developed by us and perform validation of the arguments that reach the sensitive sink, using black lists and emitting an alarm in the presence of an attack. The last two columns of the table indicate if the fixes output an alarm message when an attack is detected and what happens to the execution of the web application when that action is made. For SQLI and reflected XSS nothing is outputted and the execution of the application proceeds. For the stored XSS an alarm message is emitted, but the application proceeds with its execution. For the others where the fixes perform validation, when an attack is detected an alarm is raised and the execution of the web application stops.

8. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation was to answer the following questions: (1) Is WAP able to process a large set of PHP applications? (Section 8.1) (2) Is it more accurate and precise than other tools that do not combine taint analysis and data mining? (Sections 8.2–8.3) (3) Does it correct the vulnerabilities it detects? (Section 8.4) (4) Does the tool detect the vulnerabilities that it was programmed to detect? (Section 8.4)

WAP was implemented in Java, using the ANTLR parser generator. It has around 95,000 lines of code, around 78,500 of which generated by ANTLR.

8.1 Large scale evaluation

To show the ability of running WAP with a large set of PHP applications, we run it with 35 open source packages. Table 9 shows the packages that were analyzed and summarizes the results. The table shows that more than 2,800 files and 470,000 lines of code were analyzed, with 294 vulnerabilities found (at least 28 of which false positives). The largest packages analyzed were phpMyAdmin version 2.6.3-pl1 with 143,171 lines of code and Mutillidae version 2.3.5 with 102,567 lines of code. The code analyzed varied from well-known applications such as phpMyAdmin itself, to small applications in their initial versions like PHP-Diary. The functionality is equally varied, with a small content management application like phpCMS, an event console for the iPhone (ZiPEC), and an application to show the weather (PHP Weather). The vulnerabilities in ZiPEC were in the last version so we informed the programmers that acknowledged their existence and fixed them.

8.2 Taint analysis comparative evaluation

To answer the second question we compare WAP with Pixy and PhpMinerII. To the best of our knowledge, Pixy is the most cited PHP static analysis tool in the literature and PhpMinerII is the only one that does data mining. Other open PHP verification tools are available, but they are mostly simple prototypes designed in a more or less ad hoc manner. The full comparison of WAP with the two tools can be found in the next section. This one has the simpler goal of comparing

Web application	Files	Lines of code	Analysis time (s)	Vuln. files	Vulner. found
adminer-1.11.0	45	5,434	27	3	3
Butterfly insecure	16	2,364	3	5	10
Butterfly secure	15	2,678	3	3	4
currentcost	3	270	1	2	4
dmoz2mysql	6	1,000	2	0	0
DVWA 1.0.7	310	31,407	15	12	15
emoncms	76	6,876	6	6	15
Ghost	16	398	2	2	3
gilbitron-PIP	14	328	1	0	0
GTD-PHP	62	4,853	10	33	111
Hexjector 1.0.6	11	1,640	3	0	0
Lithuanian-7.02.05-v1.6	132	3,790	24	0	0
Measureit 1.14	2	967	2	1	12
Mfm 0.13	7	5,859	6	1	8
Mutillidae 1.3	18	1,623	6	10	19
Mutillidae 2.3.5	578	102,567	63	7	10
ocsvg-0.2	4	243	1	0	0
OWASP Vicnum	22	814	2	7	18
paCRUD 0.7	100	11,079	11	0	0
Peruggia	10	988	2	6	22
PHP X Template 0.4	10	3,009	5	0	0
PhpBB 1.4.4	62	20,743	25	0	0
Phpcms 1.2.2	6	227	2	3	5
PhpCrud	6	612	3	0	0
PhpDiary-0.1	9	618	2	0	0
PHPFusion	633	27,000	40	0	0
phpldapadmin-1.2.3	97	28,601	9	0	0
PHPLib 7.4	73	13,383	35	3	14
PHPMyAdmin 2.0.5	40	4,730	18	0	0
PHPMyAdmin 2.2.0	34	9,430	12	0	0
PHPMyAdmin 2.6.3-pl1	287	143,171	105	0	0
Phpweather 1.52	13	2,465	9	0	0
WebCalendar	122	30,173	12	0	0
WebScripts	5	391	4	2	14
ZiPEC 0.32	10	765	2	1	7
Total	2854	470,496	473	107	294

Table 9: Summary of the results of running WAP with open source projects.

WAP's taint analyzer with Pixy, which does this same kind of analysis. We consider only SQLI and reflected XSS vulnerabilities, as Pixy only detects these two (recall that WAP detects vulnerabilities of eight classes).

Table 10 shows the results of the execution of the two tools with 9 open source applications and all PHP samples of NIST's SAMATE Reference Dataset (http://samate.nist.gov/SRD/). Pixy did not manage to process *mutilidae* and *WackoPicko* because they use the object-oriented features of PHP 5.0, whereas Pixy supports only those in PHP 4.0. WAP's taint analyzer (WAP-TA) detected 68 vulnerabilities (22 SQLI and 46 XSS), with 21 false positives. Pixy detected 73 vulnerabilities (20 SQLI and 53 XSS), with 41 false positives and 5 false negatives taking WAP-TA as reference (i.e., it did not detect 5 vulnerabilities that WAP-TA did). We postpone the discussion on the full version of WAP to the next section.

Webapp	WAP-TA			Pixy				WAP (complete)		
	SQLI	XSS	FP	SQLI	XSS	FP	FN	SQLI	XSS	Corrected
CurrentCost	3	4	2	3	5	3	0	1	4	5
DVWA 1.0.7	4	2	2	4	0	2	2	2	2	4
emoncms	2	6	3	2	3	0	0	2	3	5
Measureit 1.14	1	7	7	1	16	16	0	1	0	1
Mfm-0.13	0	8	3	0	10	8	3	0	5	5
Multilidae 2.3.5	0	2	0	-	-	-	-	0	2	2
SAMATE	3	11	0	4	11	1	0	3	11	14
Vicnum15	3	1	3	3	1	3	0	0	1	1
Wackopicko	3	5	0	-	-	-	-	3	5	8
ZiPEC 0.32	3	0	1	3	7	8	0	2	0	2
Total	22	46	21	20	53	41	5	14	33	47

Table 10: Results of running WAP's taint analyzer (WAP-TA), Pixy and the complete WAP (WAP-TA plus data mining).

Pixy detected the same real vulnerabilities than WAP-TA, except 5 (1 SQLI and 4 XSS). The 10 false positives that WAP-TA reported more than Pixy are related to the $_SERVER entry point (in *emoncms*) and to the *error* sensitive sink (in *measureit*) that Pixy does not handle. However Pixy reported 30 false positives more than WAP-TA. This big difference can be explained in part by the interprocedural/ global/ context-sensitive analysis performed by WAP-TA, but not by Pixy. Another part of the justification is the bottom-up taint analysis performed by Pixy (AST navigated from the leafs to the root of the tree), whereas WAP-TA's is top-down (starts from the entry points and verifies if they reach a sensitive sink).

WAP-TA has shown to be more accurate than Pixy: it had an accuracy of 69%, whereas Pixy had only 44%.

8.3 Full comparative evaluation

This section compares the complete WAP with Pixy and PhpMinerII. The comparison with Pixy can be extracted from Table 10. The accuracy of WAP was 92.1%, whereas WAP-TA's was 69% and Pixy's only 44%. We can not show the results of PhpMinerII in the table because it does not really identify vulnerabilities.

PhpMinerII does data mining of program slices that end at a sensitive sink, independently of data being propagated through them starting at an entry point or not. PhpMinerII does this analysis to identify vulnerabilities, whereas WAP uses data mining to predict false positives in vulnerabilities detected by the taint analyzer.

We evaluated PhpMinerII with our data set using some of the classifiers of Section 6.2. We used the same classifiers as PhpMinerII's authors [29, 30]. The results of this evaluation are in Table 11. In the table it is possible to observe that the best classifier is LR, which is the only one that passed the Wilcoxon signed-rank test. It had also the highest precision (pr) and accuracy (acc), and the lowest false alarm rate (*fpp* = 20%).

Measures (%)	C4.5/J48	Naive Bayes	MLP	Logistic Regression
tpp	94.3	88.7	94.3	90.6
fpp	32.0	60.0	32.0	20.0
prfp	86.2	75.8	86.2	90.6
tpd	68.0	40.0	68.0	80.0
fpd	5.7	11.3	5.7	9.4
prd	85.0	62.5	85.0	80.0
acc (% #)	85.9	73.1	85.9	87.2
	67	57	67	68
pr	85.6	69.2	85.6	85.3
kappa	65.8 Very Good	31.7 Reasonable	65.8 Very Good	70.6 Very Good
wilcoxon	Rejected	Rejected	Rejected	Accepted

Table 11: Results of data mining models applied to Php-MinerII.

The confusion matrix of the LR model for PhpMinerII (Table 12) shows that it correctly classified 68 instances, 48 as vulnerabilities and 20 as non-vulnerabilities. We can conclude that LR is a good classifier for PhpMinerII, with an accuracy of 87.2% and a precision of 85.3%. These results are much better than Pixy's, but not as good as WAP's, which has an accuracy of 92.1% and a precision of 92.5% (see Table 4) with the same classifier.

Table 13 summarizes the comparison between WAP, Pixy, and PhpMinerII.

8.4 Fixing vulnerabilities

WAP uses data mining to discover false positives among the vulnerabilities detected by its taint analyzer. Table 10 in Section 8.2 shows that in the set of 10 packages WAP detected 47 SQLI and reflected XSS vulnerabilities. The taint analyzer

		Observed	
		Yes (Vuln.)	No (not Vuln)
Predicted	Yes (Vuln.)	48	5
	No (not Vuln)	5	20

Table 12: Confusion matrix of Logistic Regression in PhpMinerII dataset.

Metric	WAP	Pixy	PhpMinerII
accuracy	92.1%	44.0%	87.2%
precision	92.5%	50.0%	85.2%

Table 13: Accuracy and precision of WAP, Pixy and PhpMinerII.

raised 21 false positives that were detected by the data mining component. All the vulnerabilities detected were corrected (right-hand column of the table).

WAP detects several other classes of vulnerabilities that not SQLI and reflected XSS. Table 14 expands the data of Table 10 for all the vulnerabilities detected by WAP. The 69 XSS vulnerabilities detected include reflected and stored XSS vulnerabilities, which explains the difference to the 46 reflected XSS of Table 10. Again all vulnerabilities were corrected by the tool (last column).

Webapp	Detected taint analysis							Detected data mining	Corrected
	SQLI	RFI, LFI DT/PT	SCD	OCSI	XSS	Total	FP		
currentcost	3	0	0	4	7	2		5	5
DVWA 1.0.7	4	3	0	6	4	17	8	9	9
emoncms	2	0	0	13	15	3		12	12
Measureit 1.14	1	0	0	11	12	7		5	5
Mfm 0.13	0	0	0	8	8	3		5	5
Mutillidae 2.3.5	0	0	0	2	8	10	0	10	10
OWASP Vicnum	3	0	0	1	4	3		1	1
SRD[(1)]	3	6	0	11	20	1		19	19
Wackopico	3	2	0	1	5	11	0	11	11
ZiPEC 0.32	3	0	0	4	7	1		6	6
Total	22	11	0	9	69	111	28	83	83

Table 14: Results of the execution of WAP considering all vulnerabilities it detects and corrects.

9. CONCLUSION

The paper explores a new point in the design space of approaches and tools for web application security. It presents an approach for finding and correcting vulnerabilities in web applications and a tool that implements the approach for PHP programs and input validation vulnerabilities. The approach and the tool search for vulnerabilities using a combination of two techniques: static source code analysis and data mining. Data mining is used to identify false positives using a machine learning classifier selected after a thorough comparison of several alternatives. It is important to note that this combination of detection techniques can not provide entirely correct results. The static analysis problem is undecidable and the resort to data mining can not circumvent this undecidability, only provide probabilistic results. The tool corrects the code by inserting fixes, currently sanitization and validation functions. The tool was experimented both with synthetic code with vulnerabilities inserted on purpose and with a considerable number of open source PHP applications. This evaluation suggests that the tool can detect and correct the vulnerabilities of the classes it is programmed to handle.

10. REFERENCES

[1] Antunes, J., Neves, N.F., Correia, M., Verissimo, P., Neves, R.: Vulnerability removal with attack injection. IEEE Transactions on Software Engineering 36(3), 357–370 (2010)

[2] Arisholm, E., Briand, L.C., Johannessen, E.B.: A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83(1), 2–17 (2010)

[3] Banabic, R., Candea, G.: Fast black-box testing of system recovery code. In: Proceedings of the 7th ACM European Conference on Computer Systems. pp. 281–294 (2012)

[4] Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 12–24 (Oct 2007)

[5] Briand, L.C., Wüst, J., Daly, J.W., Victor Porter, D.: Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software 51(3), 245–273 (2000)

[6] Buehrer, G.T., Weide, B.W., Sivilotti, P.: Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware. pp. 106–113 (Sep 2005)

[7] Davidson, M.A.: The supply chain problem (Apr 2008), http://blogs.oracle.com/maryanndavidson/2008/04/the_supply_chain_problem.html

[8] Demšar, J.: Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research 7, 1–30 (Dec 2006)

[9] Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software pp. 42–51 (Jan/Feb 2002)

[10] Halfond, W., Orso, A.: AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 174–183 (Nov 2005)

[11] Halfond, W., Orso, A., Manolios, P.: WASP: protecting web applications using positive tainting and syntax-aware evaluation. IEEE Transactions on Software Engineering 34(1), 65–81 (2008)

[12] Huang, Y.W. et al.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International Conference on World Wide Web. pp. 148–159 (2003)

[13] Huang, Y.W. et al.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th International World Wide Web Conference. pp. 40–52 (2004)

[14] Imperva: Hacker intelligence initiative, monthly trend report #8 (Apr 2012)

[15] Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security. pp. 27–36 (Jun 2006)

[16] Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems 1(4), 323–337 (1992)

[17] Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34(4), 485–496 (2008)

[18] Medeiros, I., Neves, N.F., Correia, M.: Website of WAP tool (Jan 2014), http://awap.sourceforge.net/

[19] Merlo, E., Letarte, D., Antoniol, G.: Automated Protection of PHP Applications Against SQL Injection Attacks. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. pp. 191–202 (Mar 2007)

[20] Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 529–540 (2007)

[21] Nguyen-Tuong, A. et al.: Automatically hardening web applications using precise tainting. Security and Privacy in the Age of Ubiquitous Computing pp. 295–307 (2005)

[22] Papagiannis, I., Migliavacca, M., Pietzuch, P.: PHP Aspis: using partial taint tracking to protect against injection attacks. In: Proceedings of the 2nd USENIX Conference on Web Application Development (2011)

[23] Parr, T.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf (2009)

[24] Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection. pp. 124–145 (2005)

[25] de Poel, N.L.: Automated Security Review of PHP Web Applications with Static Code Analysis. Master's thesis, State University of Groningen (May 2010)

[26] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)

[27] Sandhu, R.S.: Lattice-based access control models. IEEE Computer 26(11), 9–19 (1993)

[28] Shankar, U. et al.: Detecting format-string vulnerabilities with type qualifiers. In: Proceedings of the 10th USENIX Security Symposium. vol. 10, pp. 16–16 (Aug 2001)

[29] Shar, L.K., Tan, H.B.K.: Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In: Proceedings of the 34th International Conference on Software Engineering. pp. 1293–1296 (2012)

[30] Shar, L.K., Tan, H.B.K.: Predicting common web application vulnerabilities from input validation and sanitization code patterns. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 310–313 (2012)

[31] Shar, L.K., Tan, H.B.K., Briand, L.C.: Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In: Proceedings of the 35th International Conference on Software Engineering. pp. 642–651 (2013)

[32] Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering 37(6), 772–787 (2011)

[33] Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: Proceedings of the 30th IEEE Symposium on Security and Privacy. pp. 305–316. IEEE (2010)

[34] Son, S., Shmatikov, V.: SAFERPHP: Finding semantic vulnerabilities in PHP applications. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (2011)

[35] Symantec: Internet threat report. 2012 trends, volume 18 (Apr 2013)

[36] Vieira, M., Antunes, N., Madeira, H.: Using web security scanners to detect vulnerabilities in web services. In: Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (Jul 2009)

[37] Walden, J., Doyle, M., Welch, G.A., Whelan, M.: Security of open source web applications. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement. pp. 545–553 (2009)

[38] Wang, X., Pan, C., Liu, P., Zhu, S.: SigFree: A signature-free buffer overflow attack blocker. In: Proceedings of the 15th USENIX Security Symposium. pp. 225–240 (Aug 2006)

[39] Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 32–41 (2007)

[40] Williams, J., Wichers, D.: OWASP Top 10 - 2013 rc1 - the ten most critical web application security risks. Tech. rep., OWASP Foundation (2013)

[41] Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, 3rd edn. (2011)