Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining

Ibéria Medeiros, Nuno Neves, *Member, IEEE*, and Miguel Correia, *Senior Member, IEEE*

***Abstract*—Although a large research effort on web application security has been going on for more than a decade, the security of web applications continues to be a challenging problem. An important part of that problem derives from vulnerable source code, often written in unsafe languages like PHP. Source code static analysis tools are a solution to find vulnerabilities, but they tend to generate false positives, and require considerable effort for programmers to manually fix the code. We explore the use of a combination of methods to discover vulnerabilities in source code with fewer false positives. We combine taint analysis, which finds candidate vulnerabilities, with data mining, to predict the existence of false positives. This approach brings together two approaches that are apparently orthogonal: humans coding the knowledge about vulnerabilities (for taint analysis), joined with the seemingly orthogonal approach of automatically obtaining that knowledge (with machine learning, for data mining). Given this enhanced form of detection, we propose doing automatic code correction by inserting fixes in the source code. Our approach was implemented in the WAP tool, and an experimental evaluation was performed with a large set of PHP applications. Our tool found 388 vulnerabilities in 1.4 million lines of code. Its accuracy and precision were approximately 5% better than PhpMinerII's and 45% better than Pixy's.**

***Index Terms*—Automatic protection, data mining, false positives, input validation vulnerabilities, software security, source code static analysis, web applications.**

Acronyms and Abbreviations

AST	Abstract Syntax Tree
DT-PT	Directory Traversal or Path Traversal
K-NN	K-Nearest Neighbor
LFI	Local File Inclusion
LR	Logistic Regression
MLP	Multi-Layer Perceptron
NB	Naive Bayes
OSCI	OS Command Injection
PHPCI	PHP Command Injection
RFI	Remote File Inclusion
RT	Random Tree
SCD	Source Code Disclosure
SQLI	SQL Injection
SVM	Support Vector Machine
TEPT	Tainted Execution Path Tree
TST	Tainted Symbol Table
UD	Untainted Data
Vul	Vulnerability
WAP	Web Application Protection
WAP-TA	Web Application Protection Taint Analyzer
XSS	Cross-site Scripting

Notations

acc	accuracy of classifier
fn	false negative outputted by a classifier
fp	false positive outputted by a classifier
fpp	false positive rate of prediction
$kappa$	kappa statistic
pd	probability of detection
pfd	probability of false detection
pr	precision of classifier
prd	precision of detection
$prfp$	precision of prediction
tn	true negative outputted by a classifier
tp	true positive outputted by a classifier
tpp	true positive rate of prediction
$wilcoxon$	Wilcoxon signed-rank test

Manuscript received September 05, 2014; revised December 24, 2014; accepted July 13, 2015. Date of publication August 17, 2015; date of current version March 01, 2016. This work was supported in part by the EC through project FP7-607109 (SEGRID), and in part by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LaSIGE). Associate Editor: W. E. Wong.

I. Medeiros and N. Neves are with the LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal (e-mail: ibemed@gmail.com; nuno@di.fc.ul.pt).

M. Correia is with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal (e-mail: miguel.p.correia@tecnico.ulisboa.pt).

Digital Object Identifier 10.1109/TR.2015.2457411

I. Introduction

SINCE its appearance in the early 1990s, the World Wide Web evolved from a platform to access text and other media to a framework for running complex *web applications*. These applications appear in many forms, from small home-made to large-scale commercial services (e.g., Google Docs, Twitter, Facebook). However, web applications have been plagued with security problems. For example, a recent report indicates an increase of web attacks of around 33% in 2012 [1]. Arguably, a reason for the insecurity of web applications is that many programmers lack appropriate knowledge about secure coding, so they leave applications with flaws. 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 [2]–[4]. 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 [5]–[7], and static analysis [8]–[10].

This 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 input validation 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 approach contributes directly to 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.

We explore the use of a novel combination of methods to detect this type of vulnerability: static analysis with data mining. Static analysis is an effective mechanism to find vulnerabilities in source code, but tends to report many false positives (non-vulnerabilities) due to its undecidability [11]. This problem is particularly difficult with languages such as PHP that are weakly typed, and not formally specified [12]. Therefore, we complement a form of static analysis, taint analysis, with the use of data mining to predict the existence of false positives. This solution combines two apparently disjoint approaches: humans coding the knowledge about vulnerabilities (for taint analysis), in combination with automatically obtaining that knowledge (with supervised machine learning supporting data mining).

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 combination of the three top-ranking classifiers 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. Moreover, for every vulnerability classified as false positive, we use an induction rule classifier to show which attributes are associated with it. We explore the JRip, PART, Prism, and Ridor induction rule classifiers for this goal. Classifiers are automatically configured using machine learning based on labeled vulnerability data.

Ensuring that the code correction is done correctly requires assessing that the vulnerabilities are removed, and that the correct behavior of the application is not modified by the fixes. We propose using program mutation and regression testing to confirm, respectively, that the fixes function as they are programmed to (blocking malicious inputs), and that the application remains working as expected (with benign inputs). Notice that we do not claim that our approach is able to correct any arbitrary vulnerability, or to detect it; it can only address the input validation vulnerabilities it is programmed to deal with.

The paper also describes the design of the *Web Application Protection* (WAP) tool that implements our approach [13]. WAP analyzes and removes input validation vulnerabilities from programs or scripts written in PHP 5, which according to a recent report is used by more than 77% of existing web applications [14]. WAP covers a considerable number of classes of vulnerabilities: SQL injection (SQLI), cross-site scripting (XSS), remote file inclusion, local file inclusion, directory traversal and path traversal, source code disclosure, PHP code injection, and OS command injection. The first two continue to be among the highest positions of the OWASP Top 10 in 2013 [15], whereas the rest are also known to be high risk, especially in PHP. Currently, WAP assumes that the background database is MySQL, DB2, or PostgreSQL. The tool might be extended with more flaws and databases, but this set is enough to demonstrate the concept. 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 result involves the management of several data structures, but also deals 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.

We evaluated the tool experimentally by running it with both simple synthetic code and with 45 open PHP web applications available in the internet, adding up to more than 6,700 files and 1,380,000 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; and 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.

II. Input Validation Vulnerabilities

Our approach is about input validation vulnerabilities, so this section presents briefly some of them (those handled by the WAP tool). 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. Fig. 1 shows PHP code vulnerable to SQLI. This script inserts in an SQL query (line 4) as 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 of providing

```
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);
```

Fig. 1. Login PHP script vulnerable to SQLI.

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 backslash metacharacters such as the prime), or by using prepared statements. We opted for 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']);` (similar for line 3).

Cross-site scripting (XSS) attacks execute malicious code (e.g., JavaScript) in the victim's browser. Different from the other attacks we consider, an 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 explain 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, or by encoding the output, or both. The latter consists in encoding metacharacters such as $\langle$ and $\rangle$ in a way that they are interpreted as normal characters, instead of HTML metacharacters.

We only present the other vulnerabilities briefly due to lack of space (with more information in [13]). 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 because it inserts a file from the local file system of the web application (not a remote file). A directory traversal or path traversal (DT-PT) attack consists in an attacker accessing arbitrary local 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. A PHP code injection (PHPCI) vulnerability allows an attacker to supply code that is executed by an *eval* statement.

III. Approach and Architecture

A. The Approach

The notion of detecting and correcting vulnerabilities in the source code that we propose is tightly related to *information flows*: detecting problematic information flows in the source code, and modifying the source code to block these flows. The notion of information flow is central to two of the three main

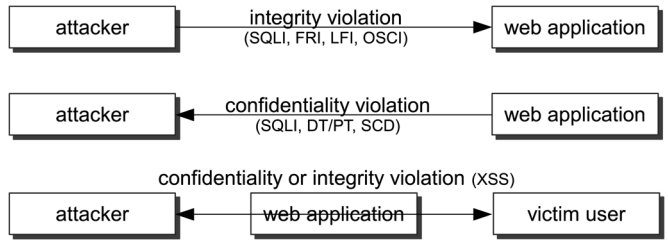

Fig. 2. Information flows that exploit web vulnerabilities.

security properties: confidentiality, and integrity [16]. 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 is, therefore, about *information-flow security* in the context of web applications. We are mostly concerned with the server-side of these applications, which is normally written in a language such as PHP, Java, or Perl. Therefore, the problem is a case of language-based information-flow security, a topic much investigated in recent years [8], [17], [18]. Attacks against web vulnerabilities can be expressed in terms of violations of information-flow security. Fig. 2 shows the information flows that exploit each of the vulnerabilities of Section II. The information flows are labeled with the vulnerabilities that usually permit them (a few rarer cases are not represented). XSS is different from other vulnerabilities because the victim is not the web application itself, but a user. Our approach is a way of enforcing information-flow security at the language-level. The tool detects the possibility of the existence of the information flows represented in the figure, and modifies the source code to prevent them.

The approach can be implemented as a sequence of steps.

1) Taint analysis: parsing the source code, generating an abstract syntax tree (AST), doing taint analysis based on the AST, and generating trees describing candidate vulnerable control-flow paths (from an entry point to a sensitive sink).
2) Data mining: obtaining attributes from the candidate vulnerable control-flow paths, and using the top 3 classifiers to predict if each candidate vulnerability is a false positive or not. In the presence of a false positive, use induction rules to present the relation between the attributes that classified it.
3) Code correction: given the control-flow path trees of vulnerabilities (predicted not to be false positives), identifying the vulnerabilities, the fixes to insert, and the places where they have to be inserted; assessing the probabilities of the vulnerabilities being false positives; and modifying the source code with the fixes.
4) Feedback: providing feedback to the programmer based on the data collected in the previous steps (vulnerable paths, vulnerabilities, fixes, false positive probability, and the attributes that classified it as a false positive).
5) Testing: higher assurance can be obtained with two forms of testing, specifically program mutation to verify if the fixes do their function, and regression testing to verify if the behavior of the application remains the same with benign inputs.

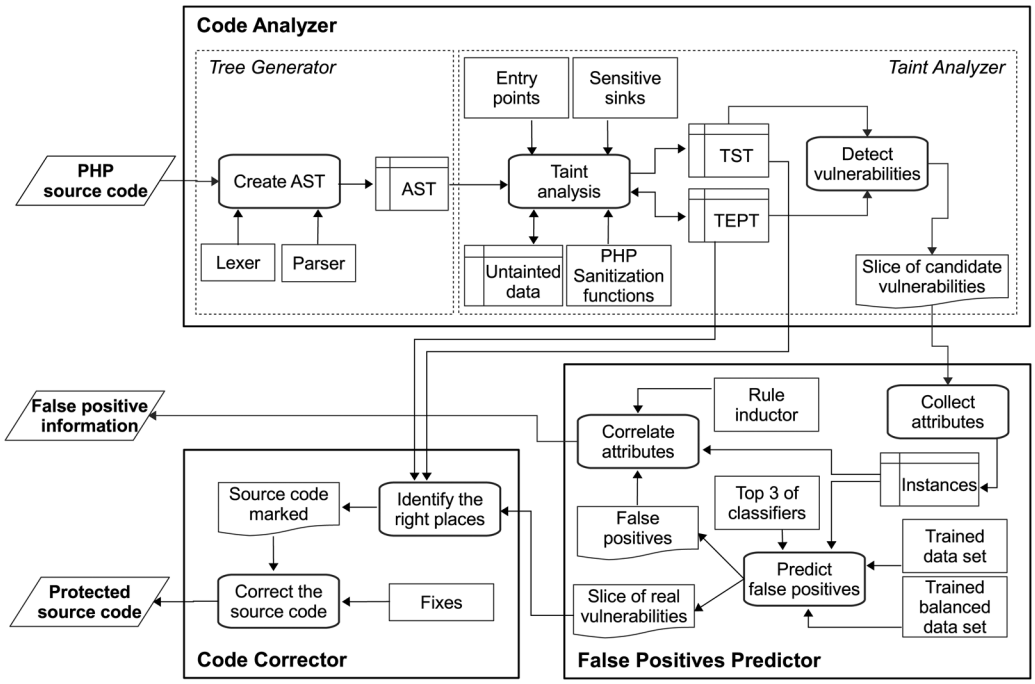

Fig. 3. Architecture including main modules, and data structures.

B. Architecture

Fig. 3 shows the architecture that implements steps 1 to 4 of the approach (testing is not represented). It is composed of three modules: code analyzer, false positives predictor, and code corrector. The *code analyzer* first parses the PHP source code, and generates an AST. Then, it uses tree walkers to do *taint analysis*, i.e., to track if data supplied by users through the entry points reaches sensitive sinks without sanitization. While doing this analysis, the code analyzer generates tainted symbol tables and tainted execution path trees for those paths that link entry points to sensitive sinks without proper sanitization. The *false positives predictor* continues where the code analyzer stops. For every sensitive sink that was found to be reached by tainted input, it tracks the path from that sink to the entry point using the tables and trees just mentioned. Along the track paths (slice candidate vulnerabilities in the figure), the vectors of attributes (instances) are collected and classified by the data mining algorithm as true positive (a real vulnerability), or false positive (not a real vulnerability). Note that we use the terms true positive and false positive to express that an alarm raised by the taint analyzer is correct (a real vulnerability) or incorrect (not a real vulnerability). These terms do not mean the true and false positive rates resulting from the data mining algorithm, which measure its precision and accuracy.

The *code corrector* picks the paths classified as true positives to signal the tainted inputs to be sanitized using the tables and trees mentioned above. The source code is corrected by inserting fixes, e.g., calls to sanitization functions. The architecture describes the approach, but represents also the architecture of the WAP tool.

IV. Taint Analysis

Taint analysis for vulnerability detection has been investigated for more than a decade [19]. However, papers in the area do not present the process in detail, and usually do not do interprocedural, global, and context-sensitive analysis, so we present how we do it. The taint analyzer is a static analysis tool that operates over an AST created by a lexer and a parser, for PHP 5 in our case (in WAP we implemented it using ANTLR [20]). In the beginning of the analysis, all *symbols* (variables, functions) are *untainted* unless they are an entry point (e.g., $a in $a = $_GET['u']). The tree walkers (also implemented using the ANTLR [20]) build a *tainted symbol table* (TST) in which every cell is a program statement from which we want to collect data (see Fig. 4). Each cell contains a subtree of the AST plus some data. For instance, for statement $x = $b+$c; the TST cell contains the subtree of the AST that represents the dependency of $x on $b and $c. For each symbol, several data items are stored, e.g., the symbol name, the line number of the statement, and the taintedness.

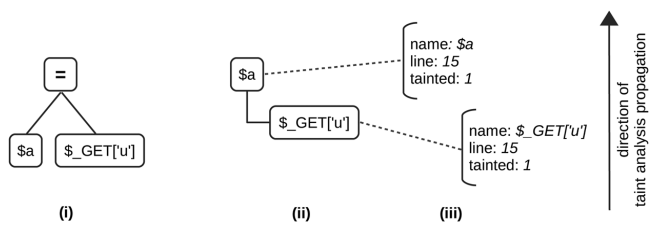

Fig. 4. Example (i) AST, (ii) TST, and (iii) taint analysis.

Taint analysis involves 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. Fig. 4 (iii) shows the propagation of the taintedness of the symbol $_GET['u'] to the symbol $a, where the attribute *tainted* of $a receives the value of the attribute *tainted* from $_GET['u']. 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

```
1: $a = $_GET['user'];
2: $b = $_POST['pass'];
3: $c = "SELECT * FROM users WHERE u = 'mysql_real_escape_string($a)' ";
4: $b = "wap";
5: $d = "SELECT * FROM users WHERE u = '$b' ";
6: $r = mysql_query($c);
7: $r = mysql_query($d);
8: $b = $_POST['pass'];
9: $query = "SELECT * FROM users WHERE u = '$a' AND p = '$b"';
10: $r = mysql_query($query);
```

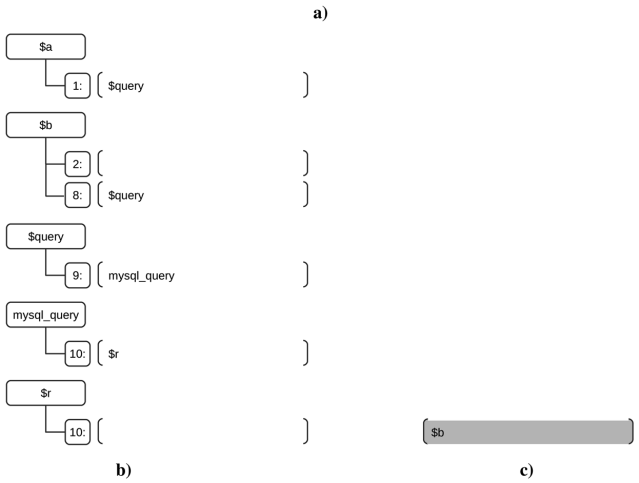

Fig. 5. Script with SQLI vulnerability, its TEPT, and untaint data structures. a) Sample script vulnerable to SQLI; b) TEPT of a); c) untainted data of a).

(a list of such functions is in [13]). The process finishes when all symbols are analyzed this way.

While the tree walkers are building the TST, they also build a *tainted execution path tree* (TEPT; example in Fig. 5(b)). Each branch of the TEPT corresponds to a tainted variable, and contains a sub-branch for each line of code where the variable becomes tainted (a square in the figure). The entries in the sub-branches (curly parentheses in the figure) are the variables that the tainted variable propagated its state into (dependent variables). Taint analysis involves updating the TEPT with the variables that become tainted.

Fig. 5 shows a sample script vulnerable to SQLI, its TEPT, and *untainted data* (UD) structures. The analysis understands that $\$a$ and $\$b$ are tainted because they get non-sanitized values from an entry point (lines 1–2). When analyzing line 3, it finds out that $\$c$ is not tainted because $\$a$ is sanitized. Analyzing line 5, $\$d$ is not tainted because $\$b$ becomes untainted in line 4. In line 8, $\$b$ is tainted again; and in line 9, $\$query$ becomes tainted due to $\$a$ and $\$b$. A vulnerability is flagged in line 10 because tainted data reaches a sensitive sink (*mysql_query*). When $\$a$ becomes tainted, a new branch is created (Fig. 5(b)). Also, a sub-branch is created to represent the line of code where $\$a$ became tainted. The same procedure occurs to $\$b$ in line 2. The state of $\$b$ in line 4 becomes untainted. An entry of it is added to UD (Fig. 5(c)) to avoid its taintedeness propagation from TEPT. So, in line 5, the statement is untainted because $\$b$ belongs to UD, and its taintedness propagation is blocked. When, in line 8, $\$b$ becomes tainted again, a new sub-branch is created in $\$b$ to line 8, and its entry is removed from UD. For $query, a branch with a sub-branch representing line 9 is created. Here, $query is tainted because $\$a$ and $\$b$ propagated their taintedness, so an entry of $query is added in the last sub-branch created in $\$a$ and $\$b$ (1: to $\$a$; 8: to $\$b$). Analyzing line 10, *mysql_query* and $\$r$ become tainted because $query taintedness is propagated. The procedure is repeated for the creation of the branch and insertion of the dependency in the sub-branch. As we can see, the process of taint analysis is a symbiosis of exploring the TST, TEPT, and UD structures. A symbol from a statement of TST propagates its taintedness to its root node iff it belongs to TEPT but not to UD. At the end of the analysis of a statement, the TEPT or UD or both are updated: TEPT with new tainted variables and tainted dependent variables, and UD with the addition or the removal of variables.

To summarize, the taint analysis model has the following steps. 1) Create the TST by collecting data from the AST, and flagging as tainted the entry points. 2) Propagate taintedness by setting variables as tainted in the TST iff the variable that propagates its taintdeness belongs to the TEPT and not to the UD. 3) Block taintedness propagation by inserting in the UD any tainted variable that belongs to the TEPT and is sanitized in the TST; conversely, remove a variable from the UD if it becomes tainted. 4) Create the TEPT: (i) a new branch is created for each new tainted variable resulting from the TST; (ii) a sub-branch is created for each line of code where the variable becomes tainted; and (iii) an entry in a sub-branch is made with a variable that becomes tainted by the taintedness propagation from the branch variable. 5) Flag a vulnerability whenever a TST cell representing a sensitive sink is reached by a tainted variable in the same conditions as in step 2.

During the analysis, whenever a variable that is passed to a sensitive sink becomes tainted, the false positives predictor is activated to collect the vector of attributes, creating thus an instance, and classify the instance as being a false positive or a real vulnerability. In the last case, the code corrector is triggered to prepare the correction of the code. 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.

V. Predicting False Positives

The static analysis problem is known to be related to Turing's halting problem, and therefore is undecidable for non-trivial languages [11]. In practice, this difficulty is solved by making only a partial analysis of some language constructs, leading static analysis tools to be unsound. In our approach, this problem can appear, for example, with string manipulation operations. For instance, it is unclear what to do to the state of a tainted string that is processed by operations that return a substring or concatenate it with another string. Both operations can untaint the string, but we cannot decide with complete certainty. We opted to let the string be tainted, 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 [21]. However, coding explicitly more knowledge in a static analysis tool is hard, and typically has to be done for each class of vulnerabilities ([21] follows this direction, but considers a single class of vulnerabilities, SQLI). Moreover, the humans who code the knowledge have first to obtain it, which can be complex.

TABLE I
ATTRIBUTES AND CLASS FOR SOME VULNERABILITIES

Potential vulnerability		**String manipulation**					**Validation**						**SQL query manipulation**				
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	**Class**
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	SAMATE	N	N	N	Y	N	N	Y	N	N	N	N	NA	NA	NA	NA	No
RFI	SAMATE	N	N	N	Y	N	N	Y	Y	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.	OWASP Vicnum	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

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 knowledge acquired from the labelled samples. 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 (not a vulnerability). The assessment has mainly two steps, as follows.

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, and define the parameters of the classifier.
2) *classification of vulnerabilities:* given the classifier, for every vulnerability found by our approach, determine if it is a false positive or not.

A. Classification of Vulnerabilities

Any process of classification involves two aspects: the *attributes* that allow classifying an instance, and the *classes* in which these instances are classified. We identified the *attributes* by analyzing manually a set of vulnerabilities found by WAP's taint analyzer. We studied these vulnerabilities to understand if they were false positives. This study 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 V-C.

From this analysis, we found three main sets of attributes that led to false positives, as outlined next.

String Manipulation: Attributes that represent PHP functions or operators that manipulate strings. These attributes 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 and operators that untaint a data flow are hard to establish, so this knowledge is not simple to retrofit into the taint analyzer.

Validation: A set of attributes related to the validation of user inputs, often involving an if-then-else construct. We define several attributes: data type (calls to is_int(), is_string()), is value set (isset()), control pattern (preg_match()), a test of belonging to a white-list, a test of belonging to a black-list, and 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 related to insertion of data in SQL queries (SQL injection only). We define attributes: string inserted in a SQL aggregate function (AVG, SUM, MAX, MIN, etc.), string inserted in a FROM clause, a test if the data are 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 the 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 I shows some examples of candidate vulnerabilities flagged by the taint analyzer, one per line. For each candidate vulnerability, the table shows the values of the attributes (Y or N), and the class, which has to be assessed manually (supervised machine learning). In each line, the set of attributes forms an instance which is classified in the class. The data mining component is configured using data like this.

B. Classifiers and Metrics

As already mentioned, our data mining component uses machine learning algorithms to extract knowledge from a set of

TABLE II
CONFUSION MATRIX (GENERIC)

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

labeled data. This section presents the machine learning algorithms that were studied to identify the best approach to classify candidate vulnerabilities. We also discuss the metrics used to evaluate the merit of the classifiers.

Machine Learning Classifiers: We studied machine learning classifiers from three classes.

Graphical and Symbolic Algorithms: This class includes algorithms that represent knowledge using a graphical model. In the ID3, C4.5/J48, Random Tree, and Random Forest classifiers, 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 (a leaf of the tree). An attribute with a small information gain has 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 an 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 (KNN), and Logistic Regression (LR). They classify an instance in the class that has the highest probability. NB is a simple probabilistic classifier based on Bayes' theorem, based on the assumption of conditional independence of the probability distributions of the attributes. K-NN 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 an 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 II). This matrix is a cross reference table where its columns are the observed instances, and its rows are the predicted results (instances classified by a classifier). Note that through all the paper we use the terms *false positive* (FP) and *true positive* (not FP) to express that an alarm raised by the taint analyzer is incorrect (not a real vulnerability) or correct (a real vulnerability). In this section, we use the same terms, *false positive (fp)*, and *true positive (tp)*, as well as *false negative (fn), and true negative (tn)*, for the output of the next stage, the FP classifier. To reduce the possibility of confusion, we use uppercase FP and lowercase *fp, tp, fn, tn* consistently as indicated.

True positive rate of prediction (tpp) measures how good the classifier is: $tpp = tp/(tp + fn)$.

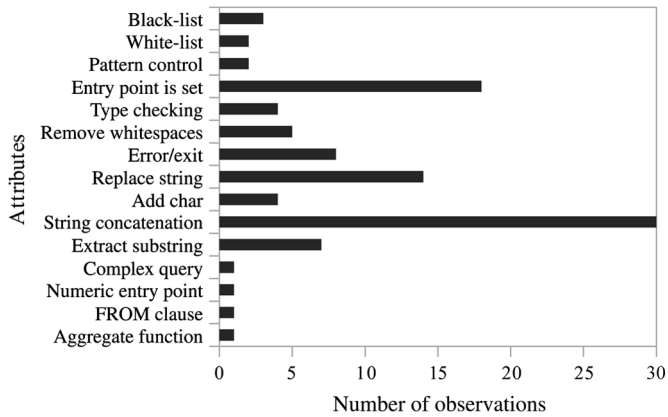

Fig. 6. Number of attribute occurrences in the original data set.

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

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

Probability of detection (pd) measures how the classifier is good at detecting real vulnerabilities: $pd = tn/(tn + fp)$.

Probability of false detection (pfd) measures how the classifier deviates from the correct classification of a candidate vulnerability that was a real vulnerability: $pfd = fn/(fn + tp)$.

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

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

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

Kappa statistic (kappa) measures the concordance between the classes predicted and observed. It can be stratified into six categories: worst, bad, reasonable, good, very good, excellent. $kappa = (po - pe)/(1 - pe)$, 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) compares classifier results with pairwise comparisons of the metrics tpp and fpp, or pd and pfd, with a benchmark result of $tpp, pd > 70\%$, and $fpp, pfd < 25\%$ [22].

Some of these metrics are statistical, such as rates and $kappa$, while acc and pr are probabilistic, and the last is a test.

C. Evaluation of Classifiers

Here we use the metrics to select the best classifiers for our case. Our current data set has 76 vulnerabilities labeled with 16 attributes: 15 to characterize the candidate's vulnerabilities, and 1 to classify it as being false positive (Yes) or a real vulnerability (No). For each candidate vulnerability, we used a version of WAP to collect the values of the 15 attributes, and we manually classified them as false positives or not. Needless to say, understanding if a vulnerability was real or a false positive

TABLE III
EVALUATION OF THE MACHINE LEARNING MODELS APPLIED TO 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
pd	100.0	86.4	95.5	100.0	100.0	86.4	86.4	100.0	95.5	97.7
pfd	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

was a tedious process. The 76 potential vulnerabilities were distributed by the classes Yes, and No, with 32, and 44 instances, respectively. Fig. 6 shows the number of occurrences of each attribute.

The 10 classifiers are available in WEKA, an open source data mining tool [23]. We used it for training and testing the ten candidate classifiers with a standard *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 III 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 fewer 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 small number of observations of certain attributes.

In 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 Fig. 6). 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 conclusion because this model is the J48 model without tree pruning. We can observe that ID3 has better accuracy and precision results when compared 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% which 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 pd tells us that all instances classified in class No were well classified. The Bayes Net classifier is the third worst model

TABLE IV
CONFUSION MATRIX OF THE TOP 3 CLASSIFIERS (FIRST TWO WITH ORIGINAL DATA, THIRD WITH A BALANCED DATA SET)

		Observed					
		Logistic Regression		Random Tree		SVM	
Predicted		Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)
Yes (FP)		27	1	25	0	56	0
No (not FP)		5	43	7	44	8	44

in terms of kappa, which is justified by the random selection of attributes to be used as the nodes of its acyclic graphical model. Some selected attributes have high entropy, so they insert noise in the model that results in bad performance.

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

Balanced Data Set: To try to improve the evaluation, we applied the SMOTE filter to balance the classes [23]. This filter doubles instances of smaller classes, creating a better balance. Fig. 7 shows the number of occurrences in this new data set. Table V shows the results of the re-evaluation with balanced classes. All models increased their performance, and passed 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 Fig. 7, there are attributes with very low occurrences that are

TABLE V
EVALUATION OF THE MACHINE LEARNING MODELS APPLIED TO THE BALANCED DATA SET

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
pd	100.0	90.9	100.0	100.0	100.0	80.5	81.8	100.0	100.0	97.7
pfd	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

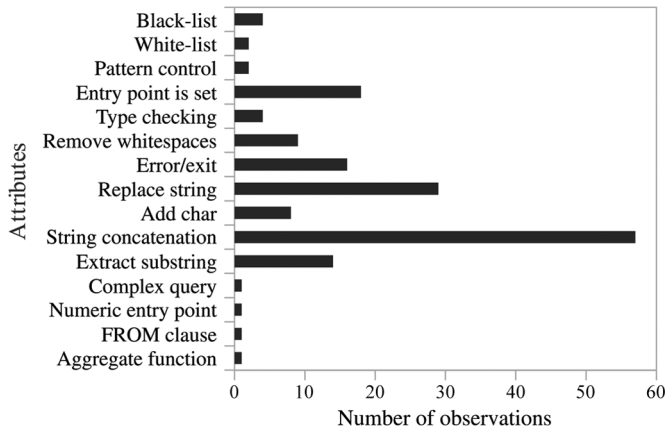

Fig. 7. Number of attribute occurrences in the balanced data set.

TABLE VI
CONFUSION MATRIX OF LOGISTIC REGRESSION CLASSIFIER APPLIED TO A FALSE POSITIVES DATA SET

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

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, in such a way that it is equal to the performance of the Random Tree model. Again, the neural networks and LR models have very good performance, but *SVM is the best* of the three (accuracy of 92.6%, precision of 92.3%, $prfp$ and pd of 100%).

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 V-A: string manipulation, SQL query manipulation, and validation. Then, we used WEKA to evaluate this new data set with the classifiers that performed best (LR, Random Tree, and SVM), with and without balanced classes. Table VI shows the confusion matrix obtained using LR without balanced classes. The 32 instances are distributed by the three classes with 17, 3, and 12 instances. 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. 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.

D. Selection of Classifiers

After the evaluation of classifiers, we need to select the classifier that is best at classifying candidate vulnerabilities as false positives or real vulnerabilities. For that purpose, we need a classifier with great accuracy and precision, but with a rate of fpp as low as possible, because this rate measures the false negatives of the classifier, which is when a candidate vulnerability is misclassified as being a false positive. We want also a classifier with a low rate of pfd, which is when a candidate vulnerability is misclassified as being a real vulnerability. This pfd rate being different from zero means that source code with a false positive may be corrected, but it will not break the behavior of the application because the fixes are designed to avoid affecting the behavior of the application. Finally, we want to justify why a candidate vulnerability is classified as a false positive, i.e., which attributes lead to this classification.

Meta-Models: To optimize the classification performed by classifiers, our first attempt was to combine machine learning algorithms. WEKA allows us to do this using meta-models. In the evaluation made in the previous section, the Random Tree (RT) and LR were two of the best classifiers. We used the Bagging, Stacking, and Boosting algorithms with RT; and Boosting with LR (LogitBoost). The Stacking model had the worst performance with an $acc = 58\%$, and thus we removed it from the evaluation. The others meta-models had in average $acc = 86.2\%$, $pr = 87.7\%$, $fpp = 3.8\%$, and 66 instances well classified. Given these results, we concluded that the meta-models had no benefit, as they showed worst performance than RT and LR separately (see Tables III, and V for these two classifiers).

TABLE VII
EVALUATION OF THE INDUCTION RULE CLASSIFIERS APPLIED TO OUR ORIGINAL DATA SET

Measures (%)	JRip	PART	Prism	Ridor
acc	88.2	88.2	86.8	86.8
(% #)	67	67	66	66
pr	90.0	88.5	88.4	87.5
fpp	0.0	6.8	13.6	4.5
pfd	28.1	18.6	9.7	25.0

TABLE VIII
SET OF INDUCTION RULES FROM THE JRIP CLASSIFIER

String concatenation	Replace string	Error / exit	Extract substring	IsSet entry point	While list	Class	Cover
Y	Y					Yes	9, 0
Y		Y				Yes	7, 0
			Y			Yes	7, 0
					Y	Yes	2, 0
	N			Y		Yes	2, 0
						No	49, 5

Top 3 Classifiers: LR was the best classifier with our original data set, but had $fpp = 2.3\%$ so it can misclassify candidate vulnerabilities as false positives. With the balanced data set, it was one of the best classifiers, despite fpp remaining unchanged. On the other hand, RT was the best decision tree classifier in both evaluations with $fpp = 0\%$, i.e., no false negatives. Also, the SVM classifier was one of the best with the original data set, and the best with the balanced data set, with $fpp = 0\%$ unlike the $fpp = 4.5\%$ in the first evaluation. It was visible that SVM with the balanced data set classified correctly the two false negative instances that it classified wrongly with the original data set. Table IV shows the confusion matrix for RT (4th and 5th columns), and SVM (last two columns) with no false negatives; and for LR (2nd and 3rd columns) with the number of false positives (a false positive classified as a vulnerability) lower than the other two classifiers.

Rules of Induction: Data mining is typically about correlation, but the classifiers presented so far do not show this correlation. For that purpose, our machine learning approach allows us to identify combinations of attributes that are correlated with the presence of false positives, i.e., what attributes justify the classification of false positives. To show this correlation, we use induction or coverage rules for classifying the instances, and for presenting the attributes combination to that classification. For this effect, we evaluated the JRip, PART, Prism, and Ridor induction classifiers. The results are presented in Table VII. Clearly, JRip was the best induction classifier, with higher pr and acc, and the only one without false negatives ($fpp = 0\%$). It correctly classified 67 out of 76 instances. The instances wrongly classified are expressed by $pfd = 28.1\%$. As explained, this statistic reports the number of instances that are false positives but were classified as real vulnerabilities. In our approach, these instances will be corrected with unnecessary fixes, but a fix does not interfere with the functionality of the code. So, although JRip has a higher pfd than the other classifiers, this is preferable to a fpp different from zero.

Table VIII shows the set of rules defined by JRip to classify our data set. The first six columns are the attributes involved in the rules, the seventh is the classification, and the last is the total number of instances covered by the rule, and the number of instances wrongly covered by the rule (the two numbers are separated by a comma). For example, the first rule (second line) classifies an instance as being false positive (Class Yes) when the *String concatenation* and *Replace string* attributes are present. The rule covers 9 instances in these conditions, from the 32 false positives instances from our data set, none were wrongly classified (9, 0). The last rule classifies as real vulnerability (Class No) all instances that are not covered by the previous five rules. The 44 real vulnerabilities from our data set were correctly classified by this rule. The rule classified five instances in class No that are false positives. These instances are related with *Black list* and SQL attributes, which are not cover by the other rules. This classification justifies the pfd value in Table VII. Notice that the attributes involved in this set of rules confirms the study of main attributes presented in Section V-C, where the SQL attributes are not relevant, and the string manipulation and validation attributes (string manipulation first) are those that most contribute to the presence of false positives.

E. Final Selection and Implementation

The main conclusion of our study is that there is no single classifier that is the best for classifying false positives with our data set. Therefore, we opted to use the top 3 classifiers to increase the confidence in the false positive classification. The top 3 classifiers include Logistic Regression and Random Tree trained with the original data set, and SVM trained with the balanced data set. Also, the JRip induction rule is used to present the correlation between the attributes to justify the false positives classification. The combination of 3 classifiers is applied in sequence: first LR; if LR classifies the vulnerability as false positive, RT is applied; if false positive, SVM is applied. Only if SVM considers it a false positive is the final result determined to be a false positive. These classifiers were implemented in WAP, and trained with the original and balanced data sets as indicated.

VI. FIXING AND TESTING THE VULNERABLE CODE

A. Code Correction

Our approach involves doing 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 these data to define the fix to insert, and the place to insert it. Inserting a fix involves modifying a PHP file.

A fix is a call to a function that sanitizes or validates 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. Most fixes are inserted in the line of the sensitive sink instead of, for example, the line of the entry point, to avoid interference with other code that sanitizes the variable. Table IX shows the fixes, how they are inserted, and other related information.

For SQLI, the fix is inserted into the last line where the query is composed, and before it reaches the sensitive sink. However, the fix can be inserted in the line of the sensitive sink, if the

TABLE IX
ACTION AND OUTPUT OF THE FIXES

Vulnerability	Fix						Output	
	Sanitization		Validation		Applied to	Function	Alarm message	Stop execution
	Addition	Substitution	Black-list	White-list				
SQLI	X	X			query	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
PHPCI	X	X	X		sensitive sink	san_eval	X, –	Yes, No

query is composed there. 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 function precedes any malicious metacharacter with a backslash, and replaces others by their literal, e.g., \n by '\n'. The sanitization function applied by the *san_sqli* fix depends on the DBMS, and the sensitive sink. For example, for MySQL, the *mysql_real_escape_string* is selected if the sensitive sink *mysql_query* is reached; but for PostgreSQL, the *pg_escape_string* is used if the sensitive sink is *pg_query*. For XSS, the fixes use functions from the OWASP PHP Anti-XSS library that replace dangerous metacharacters by their HTML entity (e.g., $<$ becomes <). For stored XSS, the sanitization function *addslashes* is used, and the validation process verifies in runtime if an attempt of exploitation occurs, raising an alarm if that is the case. For these two classes of vulnerabilities, a fix is inserted for each malicious input that reaches a sensitive sink. For example, if three malicious inputs appear in an *echo* sensitive sink (for reflected XSS), then the *san_out* fix will be inserted three times (one per each malicious input).

The fixes for the other classes of vulnerabilities were developed by us from scratch, 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 *san_eval* fix also performs sanitization, replacing malicious metacharacters by their HTML representation, for example backtick by &#96.

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, reflected XSS, and PHPCI, nothing is outputted, and the execution of the application proceeds. For 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.

B. Testing Fixed Code

Our fixes were designed to avoid modifying the (correct) behavior of the applications. So far, we witnessed no cases in which an application fixed by WAP started to function incorrectly, or that the fixes themselves worked incorrectly. However, to increase the confidence in this observation, we propose using software testing techniques. Testing is probably the most widely adopted approach for ensuring software correctness. The idea is to apply a set of test cases (i.e., inputs) to a program to determine for instance if the program in general contains errors, or if modifications to the program introduced errors. This verification is done by checking if these test cases produce incorrect or unexpected behavior or outputs. We use two software testing techniques for doing these two verifications, respectively: 1) program mutation, and 2) regression testing.

1) Program mutation: We use a technique based on program mutation to confirm that the inserted program fixes prevent the attacks as expected. Program mutation is a form of code-based testing, as it involves using the source code of the program [24]. This technique consists in generating variations of the program (mutants), which are afterwards used to verify if the outputs they produce differ from those produced by the unmodified program. The main idea is that, although understanding if the behavior of a program is incorrect or not is not trivial, on the contrary comparing the results of two tests of similar programs is quite feasible.

A mutant of a program P is defined as a program P' derived from P by making a single change to P [25], [26]. Given programs P and P', and a test-case T, (A1) T *differentiates* P from P' if executions of P and P' with T produce different results; and (A2) if T fails to differentiate P from P', either P is functionally equivalent to P', or T is ineffective in revealing the changes introduced into P'. For each vulnerability it detects, WAP returns the vulnerable slice of code, and the same slice with the fix inserted, both starting in an entry point, and ending in a sensitive sink. Consider that P is the original program (that contains the vulnerable slice), and P' is the fixed program (with the fix inserted). Consider that both P and P' are executed with a test case T.

T Differentiates P from P' (A1): If T is a malicious input that exploits the vulnerability in P, then P executed with T produces an incorrect behavior. P' is the fixed version of P. Therefore, if the fix works correctly, the result of the execution of P' with T differs from the result of the execution of P with T. As explained above, comparing the results of the two tests is quite feasible.

T does not differentiate P from P' (A2): If T is a benign input, and P and P' are executed with T, a correct behavior is obtained in both cases, and the result produced by both programs is equal. Input sanitization and validation do not interfere with benign inputs, so the fixes only act on malicious inputs, leaving the benign inputs untouched, and remaining the correct behavior.

Applying this approach with a large set of test cases, we can gain confidence that a fix indeed corrects a vulnerability.

2) Regression testing: A concern that may be raised about the use of WAP for correcting web applications is that the applications may start to function incorrectly due to the modifications made by the tool. As mentioned, we have some experience with the tool, and we never observed this problem. Nevertheless, we propose using regression testing to verify if the (correct) behavior of an application was modified by WAP. Regression testing consists in running the same tests before and after the program modifications [24]. The objective is to check if the functionality that was working correctly before the changes still continues to work correctly.

We consider that the result of running an application test can be either *pass* or *fail*, respectively if the application worked as expected with that test case or not. We are not concerned about how the test cases are obtained. If WAP is used by the application developers, then they can simply do their own regression testing process. If WAP is employed by others, they can write their own suite of tests, or use the tests that come with the application (something that happens with many open source applications). Regression testing is successful if all the test cases that resulted in pass before the WAP modification also result in pass after inserting the fixes.

VII. Experimental Evaluation

WAP was implemented in Java, using the ANTLR parser generator. It has around 95,000 lines of code, with 78,500 of which generated by ANTLR. The implementation followed the architecture of Fig. 3, and the approach of the previous sections.

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 VII-A)
2) Is it more accurate and precise than other tools that do not combine taint analysis and data mining? (Section VII-B through VII-C)
3) Does it correct the vulnerabilities it detects? (Section VII-D)
4) Does the tool detect the vulnerabilities that it was programmed to detect? (Section VII-D)
5) Do its corrections interfere with the normal behavior of applications? (Section VII-E)

A. Large Scale Evaluation

To show the ability of using WAP with a large set of PHP applications, we run it with 45 open source packages. Table X shows the packages that were analyzed, and summarizes the results. The table shows that more than 6,700 files and 1,380,000 lines of code were analyzed, with 431 vulnerabilities found (at least 43 of which were false positives (FP)). The largest packages analyzed were Tikiwiki version 1.6 with 499,315 lines of code, and phpMyAdmin version 2.6.3-pl1 with 143,171 lines of code. We used a range of packages from well-known applications (e.g., Tikiwiki) to small applications in their initial versions (like PHP-Diary). The functionality was equally diverse, including for instance a small content management application like phpCMS, an event console for the iPhone (ZiPEC), and a weather application (PHP Weather). The vulnerabilities found in ZiPEC were in the last version, so we informed the programmers, who then acknowledged their existence and fixed them.

B. 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 tool that does data mining. Other open PHP verification tools are available, but they are mostly simple prototypes. The full comparison of WAP with the two tools can be found in the next section. This one has the simpler goal of comparing 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 XI shows the results of the execution of the two tools with a randomly selected subset of the applications of Table X: 9 open source applications, and all PHP samples of NIST's SAMATE [41]. 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 (FP). Pixy detected 73 vulnerabilities (20 SQLI, and 53 XSS), with 41 false positives, and 5 false negatives (FN, i.e., it did not detect 5 vulnerabilities that WAP-TA did).

Pixy reported 30 false positives that were not raised by WAP-TA. This difference is explained in part by the interprocedural, global, and context-sensitive analyses performed by WAP-TA, but not by Pixy. Another part of the justification is the bottom-up taint analysis carried out by Pixy (AST navigated from the leafs to the root of the tree), whereas the WAP-TA analysis is top-down (starts from the entry points, and verifies if they reach a sensitive sink).

Overall, WAP-TA was more accurate than Pixy: it had an accuracy of 69%, whereas Pixy had only 44%.

C. Full Comparative Evaluation

This section compares the complete WAP with Pixy and PhpMinerII.

PhpMinerII does data mining of program slices that end at a sensitive sink, regardless of data being propagated through them starting at an entry point or not. PhpMinerII does this analysis to predict 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 the same classifiers as PhpMinerII's authors [27], [28] (a subset of the classifiers of Section V-B). The results of this evaluation are in Table XII. 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 ($fpp = 20\%$).

The confusion matrix of the LR model for PhpMinerII (Table XIII) shows that it correctly classified 68 instances, with 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%.

TABLE X
SUMMARY OF THE RESULTS OF RUNNING WAP WITH OPEN SOURCE PACKAGES

Web application	Files	Lines of code	Analysis time (s)	Vul files	Vul found	FP	Real vul
adminer-1.11.0	45	5,434	27	3	3	0	3
Butterfly insecure	16	2,364	3	5	10	0	10
Butterfly secure	15	2,678	3	3	4	0	4
currentcost	3	270	1	2	4	2	2
dmoz2mysql	6	1,000	2	0	0	0	0
DVWA 1.0.7	310	31,407	15	12	15	8	7
emoncms	76	6,876	6	6	15	3	12
gallery2	644	124,414	27	0	0	0	0
getboo	199	42,123	17	30	64	9	55
Ghost	16	398	2	2	3	0	3
gilbitron-PIP	14	328	1	0	0	0	0
GTD-PHP	62	4,853	10	33	111	0	111
Hexjector 1.0.6	11	1,640	3	0	0	0	0
Hotelmis 0.7	447	76,754	9	2	7	5	2
Lithuanian-7.02.05-v1.6	132	3,790	24	0	0	0	0
Measureit 1.14	2	967	2	1	12	7	5
Mfm 0.13	7	5,859	6	1	8	3	5
Mutillidae 1.3	18	1,623	6	10	19	0	19
Mutillidae 2.3.5	578	102,567	63	7	10	0	10
NeoBill0.9-alpha	620	100,139	6	5	19	0	19
ocsvg-0.2	4	243	1	0	0	0	0
OWASP Vicnum	22	814	2	7	4	3	1
paCRUD 0.7	100	11,079	11	0	0	0	0
Peruggia	10	988	2	6	22	0	22
PHP X Template 0.4	10	3,009	5	0	0	0	0
PhpBB 1.4.4	62	20,743	25	0	0	0	0
Phpcms 1.2.2	6	227	2	3	5	0	5
PhpCrud	6	612	3	0	0	0	0
PhpDiary-0.1	9	618	2	0	0	0	0
PHPFusion	633	27,000	40	0	0	0	0
phpldapadmin-1.2.3	97	28,601	9	0	0	0	0
PHPLib 7.4	73	13,383	35	3	14	0	14
PHPMyAdmin 2.0.5	40	4,730	18	0	0	0	0
PHPMyAdmin 2.2.0	34	9,430	12	0	0	0	0
PHPMyAdmin 2.6.3-pl1	287	143,171	105	0	0	0	0
Phpweather 1.52	13	2,465	9	0	0	0	0
SAMATE	22	353	1	10	20	1	19
Tikiwiki 1.6	1,563	499,315	1	4	4	0	4
volkszaehler	43	5,883	1	0	0	0	0
WackoPicko	57	4,156	3	4	11	0	11
WebCalendar	129	36,525	20	0	0	0	0
Webchess 1.0	37	7,704	1	5	13	0	13
WebScripts	5	391	4	2	14	0	14
Wordpress 2.0	215	44,254	10	7	13	1	12
ZiPEC 0.32	10	765	2	1	7	1	6
Total	6,708	1,381,943	557	174	431	43	388

TABLE XI
RESULTS OF RUNNING WAP'S TAINT ANALYZER (WAP-TA), PIXY, AND WAP COMPLETE (WITH DATA MINING)

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

TABLE XII
EVALUATION OF THE MACHINE LEARNING MODELS APPLIED TO THE DATA SET RESULTING FROM PHPMINERII

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
pd	68.0	40.0	68.0	80.0
pfd	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

We now compare the three tools. The comparison with Pixy can be extracted from Table XI; however, we cannot show the results of PhpMinerII in the table because it does not really identify vulnerabilities. The accuracy of WAP was 92.1%, whereas

TABLE XIII
CONFUSION MATRIX OF PHPMINERII WITH LR

		Observed	
		Yes (Vul)	No (not Vul)
Predicted	Yes (Vul)	48	5
	No (not Vul)	5	20

TABLE XIV
SUMMARY FOR WAP, PIXY AND PHPMINERII

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

TABLE XV
RESULTS OF THE EXECUTION OF WAP WITH ALL VULNERABILITIES IT DETECTS AND CORRECTS

Webapp	Detected taint analysis							Detected data mining	Fixed
	SQLI	RFI, LFI DT/PT	SCD	OSCI	XSS	Total	FP		
currentcost	3	0	0	0	4	7	2	5	5
DVWA 1.0.7	4	3	0	6	4	17	8	9	9
emoncms	2	0	0	0	13	15	3	12	12
Measureit 1.14	1	0	0	0	11	12	7	5	5
Mfm 0.13	0	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	0	1	4	3	1	1
SAMATE	3	6	0	0	11	20	1	19	19
WackoPicko	3	2	0	1	5	11	0	11	11
ZiPEC 0.32	3	0	0	0	4	7	1	6	6
Total	22	11	0	9	69	111	28	83	83

the accuracy of WAP-TA was 69%, and of Pixy was only 44%. The PHPminerII results (Tables XII and XIII) 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 III) with the same classifier.

Table XIV summarizes the comparison between WAP, Pixy, and PhpMinerII. We refined these values for a more detailed comparison. We obtained the intersection between the 53 slices classified as vulnerable by PHPminerII and the 68 vulnerabilities found by WAP. Removing from the 68 those found in applications that PHPminerII could not process, 37 remain, 11 of which are false positives. All the 22 real vulnerabilities detected by PHPminerII were also detected by WAP, and PHPminerII did not detect 4 vulnerabilities that WAP identified. The 11 false positives from WAP are among the 31 false positives of PHPminerII.

D. Fixing Vulnerabilities

WAP uses data mining to discover false positives among the vulnerabilities detected by its taint analyzer. Table XI shows that in the set of 10 packages WAP detected 47 SQLI, and reflected XSS vulnerabilities. The taint analyzer 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 besides SQLI and reflected XSS. Table XV expands the data of Table XI for all the vulnerabilities discovered by WAP. The 69 XSS vulnerabilities detected include reflected and stored XSS vulnerabilities, which explains the difference to the 46 reflected XSS of Table XI. Again, all vulnerabilities were corrected by the tool (last column).

E. Testing Fixed Applications

WAP returns new application files with the vulnerabilities removed by the insertion of fixes in the source code. As explained in Section VI-B, regression testing can be used to check if the code corrections made by WAP compromise the previously correct behavior of the application.

For this purpose, we did regression testing using Selenium [42], a framework for testing web applications. Selenium automates browsing, and verifies the results of the requests sent to web applications. The DVWA 1.0.7 application and the samples in SAMATE were tested because they contain a variety of vulnerabilities detected and corrected by the WAP tool (see Table XV). Specifically, WAP corrected 6 files of DVWA 1.0.7, and 10 of SAMATE.

The regression testing was carried out in the following way. First, we created in Selenium a set of test cases with benign inputs. Then, we ran these test cases with the original DVWA and SAMATE files, and observed that they passed all tests. Next, we replaced the 16 vulnerable files by the 16 files returned by WAP, and reran the tests to verify the changes introduced by the tool. The applications passed again all the tests.

VIII. DISCUSSION

The WAP tool, like any other static analysis approach, can only detect vulnerabilities it is programmed to. WAP can, however, be extended to handle more classes of input validation 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 improved incrementally when more data become available. For the training, we need data about candidate vulnerabilities of that kind found 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 data about what sanitization function has to be used to handle that class of vulnerability, and where it shall be inserted. Again, getting this information is doable once the new class is known and understood.

A limitation of WAP derives from the lack of formal specification of PHP. During the experimentation of the tool with many open source applications (Section VII-A), several times WAP was unable to parse the source code for lack of a grammar rule to deal with strange constructions. With time, these rules were added, and these problems stopped appearing.

IX. 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 to conserve space.

Detecting Vulnerabilities with Static Analysis: Static analysis tools automate the auditing of code, either source, binary, or intermediate. In this paper, we use the term *static analysis* in a narrow sense to designate static analysis of source code to detect vulnerabilities [8]–[10], [29]. 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.

Taint analysis tools like CQUAL [10] and Splint [19] (both for C code) use two qualifiers to annotate source code: the *untainted* qualifier indicates either that a function or parameter returns trustworthy data (e.g., a sanitization function), or a parameter of a function requires trustworthy data (e.g., *mysql_query*). The *tainted* qualifier means that a function or a parameter returns non-trustworthy data (e.g., functions that read user input).

Pixy [9] 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 [29]. 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 or file, but can involve several).

Vulnerabilities and Data Mining: Data mining has been used to predict the presence of software defects [30]–[32]. 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 [33]–[35]. They used attributes such as past vulnerabilities and function calls [33], or code complexity and developer activities [34]. Contrary to our work, these other works did not aim to detect bugs and identify their location, but to assess the quality of the software in terms of the prevalence of defects and vulnerabilities.

Shar and Tan presented PhpMinerI, and PhpMinerII, which are two tools that use data mining to assess the presence of vulnerabilities in PHP programs [27], [28]. 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 [23]. More recently, the authors evolved this idea to use also traces or program execution [36]. 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 whether the vulnerabilities found by taint analysis are really vulnerabilities or 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 it checks if the call being made satisfies the format defined by the programmer [37]. 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 [38]. WebSSARI also does static analysis, and inserts runtime guards, but no details are available about what the guards are, or how they are inserted [8]. 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 [39]. saferXSS does static analysis for finding XSS vulnerabilities, then removes them using functions provided by OWASP's ESAPI [43] to wrap user inputs [40]. None of these works use data mining or machine learning.

X. Conclusion

This paper 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 the top 3 machine learning classifiers, and to justify their presence using an induction rule classifier. All classifiers were selected after a thorough comparison of several alternatives. It is important to note that this combination of detection techniques cannot provide entirely correct results. The static analysis problem is undecidable, and resorting to data mining cannot circumvent this undecidability, but only provide probabilistic results. The tool corrects the code by inserting fixes, i.e., sanitization and validation functions. Testing is used to verify if the fixes actually remove the vulnerabilities and do not compromise the (correct) behavior of the applications. The tool was experimented with using synthetic code with vulnerabilities inserted on purpose, and with a considerable number of open source PHP applications. It was also compared with two source code analysis tools: Pixy, and PhpMinerII. This evaluation suggests that the tool can detect and correct the vulnerabilities of the classes it is programmed to handle. It was able to find 388 vulnerabilities in 1.4 million lines of code. Its accuracy and precision were approximately 5% better than PhpMinerII's, and 45% better than Pixy's.

References

[1] Symantec, Internet threat report. 2012 trends, vol. 18, Apr. 2013.
[2] W. Halfond, A. Orso, and P. Manolios, "WASP: protecting web applications using positive tainting and syntax aware evaluation," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 65–81, 2008.
[3] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Proc. 8th Int. Conf. Recent Advances in Intrusion Detection*, 2005, pp. 124–145.
[4] X. Wang, C. Pan, P. Liu, and S. Zhu, "SigFree: A signature-free buffer overflow attack blocker," in *Proc. 15th USENIX Security Symp.*, Aug. 2006, pp. 225–240.
[5] J. Antunes, N. F. Neves, M. Correia, P. Verissimo, and R. Neves, "Vulnerability removal with attack injection," *IEEE Trans. Softw. Eng.*, vol. 36, no. 3, pp. 357–370, 2010.

[6] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proc. 7th ACM Eur. Conf. Computer Systems*, 2012, pp. 281–294.
[7] Y.-W. Huang *et al.*, "Web application security assessment by fault injection and behavior monitoring," in *Proc. 12th Int. Conf. World Wide Web*, 2003, pp. 148–159.
[8] Y.-W. Huang *et al.*, "Securing web application code by static analysis and runtime protection," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 40–52.
[9] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proc. 2006 Workshop Programming Languages and Analysis for Security*, Jun. 2006, pp. 27–36.
[10] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proc. 10th USENIX Security Symp.*, Aug. 2001, vol. 10, pp. 16–16.
[11] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, 1992.
[12] N. L. de Poel, "Automated security review of PHP web applications with static code analysis," M.S. thesis, State Univ. Groningen, Groningen, The Netherlands, May 2010.
[13] WAP tool website [Online]. Available: http://awap.sourceforge.net/
[14] Imperva, Hacker intelligence initiative, monthly trend report #8, Apr. 2012.
[15] J. Williams and D. Wichers, OWASP Top 10 - 2013 rcl - the ten most critical web application security risks, OWASP Foundation, 2013, Tech. Rep.
[16] R. S. Sandhu, "Lattice-based access control models," *IEEE Comput.*, vol. 26, no. 11, pp. 9–19, 1993.
[17] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003.
[18] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing*, 2005, pp. 295–307.
[19] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, pp. 42–51, Jan./Feb. 2002.
[20] T. Parr*, Language Implementation Patterns: Create Your Own Domain Specific and General Programming Languages.*. Frisco, TX, USA: Pragmatic Bookshelf, 2009.
[21] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. 28th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2007, pp. 32–41.
[22] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.
[23] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.
[24] J. C. Huang*, Software Error Detection through Testing and Analysis.* . New York, NY, USA: Wiley, 2009.
[25] T. Budd *et al.*, "The design of a prototype mutation system for program testing," in *Proc. AFIPS National Computer Conf.*, 1978, pp. 623–627.
[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.
[27] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proc. 34th Int. Conf. Software Engineering*, 2012, pp. 1293–1296.
[28] L. K. Shar *et al.*, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering*, 2012, pp. 310–313.
[29] S. Son and V. Shmatikov, "SAFERPHP: Finding semantic vulnerabilities in PHP applications," in *Proc. ACM SIGPLAN 6th Workshop Programming Languages and Analysis for Security*, 2011.
[30] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.
[31] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in objectoriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, 2000.
[32] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.
[33] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Computer and Communications Security*, 2007, pp. 529–540.
[34] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, 2011.
[35] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," *Proc. 3rd Int. Symp. Empirical Software Engineering and Measurement*, pp. 545–553, 2009.
[36] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proc. 35th Int. Conf. Software Engineering*, 2013, pp. 642–651.
[37] W. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *Proc. 20th IEEE/ACM Int. Conf. Automated Software Engineering*, Nov. 2005, pp. 174–183.
[38] G. T. Buehrer, B. W. Weide, and P. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," in *Proc. 5th Int. Workshop Software Engineering and Middleware*, Sep. 2005, pp. 106–113.
[39] E. Merlo, D. Letarte, and G. Antoniol, "Automated Protection of PHP Applications Against SQL Injection Attacks," in *Proc. 11th Eur. Conf. Software Maintenance and Reengineering*, Mar. 2007, pp. 191–202.
[40] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Inf. Softw. Technol.*, vol. 54, no. 5, pp. 467–478, 2012.
[41] NIST's SAMATE Reference Dataset (SRD) [Online]. Available: http://samate.nist.gov/SRD/
[42] Selenium IDE [Online]. Available: http://docs.seleniumhq.org
[43] OWASP ESAPI [Online]. Available: http://www.owasp.org/index.php/ESAPI

Ibéria Medeiros is a Ph.D. student at the Department of Informatics, Faculty of Sciences, University of Lisboa. She is a member of the Large-Scale Informatics Systems (LaSIGE) Laboratory, and the Navigators research group. She is also an Assistant Professor of the University of Azores, teaching courses of the graduation in Informatics, Computer Networks, and Multimedia. Her research interests are concerned with software security, source code static analysis, data mining and machine learning, and security. More information about her can be found at https://sites.google.com/site/ibemed/.

Nuno Neves is Associate Professor with Habilitation at the Faculty of Sciences of the University of Lisboa. He is also Director of the LaSIGE Lab, and he leads the Navigators group. His main research interests are in security and dependability aspects of distributed systems. Currently, he is principal investigator of the SUPERCLOUD and SEGRID European projects, and he is involved in projects BiobankClouds and Erasmus+ ParIS. His work has been recognized in several occasions, for example with the IBM Scientific Prize, and the William C. Carter award. He is on the editorial board of the *International Journal of Critical Computer-Based Systems*. More information about him can be found at http://www.di.fc.ul.pt/~nuno.

Miguel Correia is an Associate Professor at Instituto Superior Técnico of the Universidade de Lisboa, and a researcher at INESC-ID, in Lisboa, Portugal. He has been involved in several international and national research projects related to security, including the PCAS, TCLOUDS, ReSIST, MAFTIA, and CRUTIAL European projects. He has more than 100 publications. His main research interests are security, intrusion tolerance, distributed systems, distributed algorithms, software security, cloud computing, and critical infrastructure protection. More information about him can be found at http://www.gsd.inesc-id.pt/~mpc/.