Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis

Lwin Khin Shar and Hee Beng Kuan Tan
Block S2, School of Electrical and Electronic Engineering
Nanyang Technological University
Nanyang Avenue, Singapore 639798
{shar0035, ibktan}@ntu.edu.sg

Lionel C. Briand
SnT Centre, University of Luxembourg
4 rue Alphonse Weicker, L-2721, Luxembourg
lionel.briand@uni.lu

***Abstract*—In previous work, we proposed a set of static attributes that characterize input validation and input sanitization code patterns. We showed that some of the proposed static attributes are significant predictors of SQL injection and cross site scripting vulnerabilities. Static attributes have the advantage of reflecting general properties of a program. Yet, dynamic attributes collected from execution traces may reflect more specific code characteristics that are complementary to static attributes. Hence, to improve our initial work, in this paper, we propose the use of dynamic attributes to complement static attributes in vulnerability prediction. Furthermore, since existing work relies on supervised learning, it is dependent on the availability of training data labeled with known vulnerabilities. This paper presents prediction models that are based on both classification and clustering in order to predict vulnerabilities, working in the presence or absence of labeled training data, respectively. In our experiments across six applications, our new supervised vulnerability predictors based on hybrid (static and dynamic) attributes achieved, on average, 90% recall and 85% precision, that is a sharp increase in recall when compared to static analysis-based predictions. Though not nearly as accurate, our unsupervised predictors based on clustering achieved, on average, 76% recall and 39% precision, thus suggesting they can be useful in the absence of labeled training data.**

***Index Terms*—Defect prediction, vulnerability, input validation and sanitization, static and dynamic analysis, empirical study.**

I. INTRODUCTION

SQL injection (SQLI) and cross site scripting (XSS) are the two most common and serious web application vulnerabilities threatening the privacy and security of both clients and applications nowadays [1]. To mitigate the two threats, many vulnerability detection approaches such as static taint analysis and concolic testing have been proposed. These approaches have been shown to be effective in finding many security vulnerabilities. Static taint analysis approaches are generally easy to be used but are inefficient in practice due to high false positive rates [11, 21]. Concolic testing techniques are highly precise but could be impractical for large systems due to state space explosion. Alternative or complementary vulnerability detection solutions that are both practical for use and precise would be beneficial to security teams.

To guard against application-level attacks such as SQLI, XSS, path traversal, and buffer overflow, input validation and input sanitization methods are commonly implemented in web applications. Hence, intuitively, an application is vulnerable if the implementation of input validation and input sanitization methods is inadequate or incorrect. In our initial work [2, 16], we mined static code patterns that implement such methods to build vulnerability predictors based on supervised learning. We showed that those predictors provide an alternative, effective solution for SQLI and XSS vulnerabilities. Although these results were encouraging, our earlier work suffers from two major drawbacks—(1) though proposed static attributes are useful predictors, they are limited in terms of the prediction accuracy they can yield (the predictive capability of these attributes is dependent on how precise is the classification of the input validation and sanitization code patterns); (2) being a supervised learning-based approach, its effectiveness is dependent on the availability of sufficient training data labeled with manually checked security vulnerabilities.

This paper addresses the above limitations and provides a more extensive empirical study than that of our previous work. It presents a pattern mining approach based on dynamic analysis that classifies input validation and sanitization functions through the systematic execution of these functions and the analysis of their execution traces. We also use both supervised learning methods and unsupervised learning methods to build vulnerability predictors so as to determine the effectiveness of the predictors with or without labeled training data. In existing vulnerability prediction studies, supervised learning methods are generally used. We have no knowledge of vulnerability prediction models built using unsupervised learning methods. Our goal is to make vulnerability prediction both more practical and accurate than in previous work.

We evaluated our proposed approach based on experiments with a set of open source PHP-based web applications with known XSS and SQLI vulnerabilities. We implemented a tool called *PhpMinerI* for extracting relevant data from PHP programs. We trained two types of vulnerability prediction models (for predicting SQLI vulnerabilities and XSS vulnerabilities) using the extracted data. In our cross-validation experiments, supervised vulnerability predictors built from hybrid (static and dynamic) attributes achieved, on average, over 10 datasets, 90% recall and 85% precision on predicting

XSS and SQLI vulnerabilities. These new predictors improved the recall by 16% and the precision by 2% compared to predictors built from static attributes alone. Our unsupervised predictors also achieved, on average, over 5 datasets, 76% recall, 12% false alarm rate, and 39% precision.

Our study here is limited to PHP programming language due to the prevalence of SQLI and XSS vulnerabilities in PHP-based applications. However, our approach could easily be extended to other programming languages. For example, to extend it to Java, the only modification required is to classify Java-based functions and operations according to our proposed static and dynamic analysis-based classification schemes.

The outline of the paper is as follows. Section II discusses our motivation. Section III defines the static-dynamic hybrid attributes and Section IV presents the two hypotheses that we shall investigate. Section V presents our vulnerability prediction frameworks and Section VI evaluates their accuracy based on the proposed hybrid attributes. Section VII discusses related work. Section VIII concludes our study.

II. MOTIVATION

Both SQLI and XSS vulnerabilities arise from improper handling of inputs in web application programs. In a typical web program, user inputs are accessed via forms, URLs, cookies, and XML files. Those inputs are processed and propagated to various program points to accomplish the application's objectives. Some of those inputs may then be stored in the application's persistent data stores, such as databases and session objects, for further processing of the application's required functionalities. The operations carried out in those processes often include security-sensitive operations (sinks) such as HTML response outputs and database accesses. When user inputs referenced in such operations have not been sanitized or validated, vulnerabilities arise. XSS vulnerabilities arise when an unrestricted user input is used in a HTML response output statement. SQLI vulnerabilities occur when a user input is used in a SQL statement without proper checks.

Hence, to prevent web application vulnerabilities, developers often employ input validation/sanitization methods along the paths propagating the inputs to the sinks. These methods can be broadly categorized into escaping, meta-character matching/removal, string length truncating, and data type checking/conversion [19]. The methods typically employed are language-provided functions (e.g., `htmlentities`), trusted third-party libraries (e.g., enterprise security APIs provided by OWASP [1]), or custom functions developed for specific security or data integrity requirements by developer himself or a group of security experts.

From the analysis of many vulnerability reports in security databases such as CVE [6], we derived the following observations:

- First, many SQLI and XSS vulnerability reports show that most of these vulnerabilities arise from the misidentification of inputs. That is, developers may implement adequate input validation and sanitization methods but yet, they may fail to recognize all the data that could be manipulated by external users, thereby missing some of the inputs for validation. Therefore, in security analysis, it is important to first identify all the inputs and the sinks that use them.
- Second, when an input to be used in security-sensitive program statements is considered to be a numeric type, it is most effective to use a numeric-type check or numeric-type conversion (from string since inputs are originally strings).
- Third, we observed that individual developers often write their own piece of defensive code to protect the specific programs that they are responsible for. But as also observed by Jovanovic et al. [3] and Xie and Aiken [4], many of such customized functions are incorrect often due to insufficient expertise in security. Thus, for most cases, the use of language-provided sanitization/validation functions, widely-accepted third-party security libraries, or security functions developed by a group of security experts is typically the most effective defense method.
- Lastly, different defense methods are generally required to prevent different types of vulnerabilities. For example, to prevent SQLI vulnerabilities, escaping characters that have special meaning to SQL parser is required whereas escaping characters that have special meaning to client script interpreters is needed to prevent XSS vulnerabilities. Thus, care must be taken to use appropriate methods.

III. HYBRID CODE ATTRIBUTES

From the above observations, we could derive the following attributes to build vulnerability predictors.

Data dependence graph: Our unit of measurement is a sink. A sink is a node in a control flow graph of a web program that may cause SQLI or XSS attacks. Basically, a sink represents a program statement that interacts with database (denoted as *SQL* sink) or web client (denoted as *HTML* sink). We may use a node and a program statement interchangeably depending on the context. Given a sink k, we compute its data dependence graph (DDG_k) using data flow analysis. The graph provides reachable definitions for the variables used in the sink, that is, it contains the nodes on which the sink is data dependent [8]. As such, any input validation and sanitization operations implemented for the sink k can be found in the nodes in DDG_k.

The first step of our method is to classify the nodes in DDG_k according to their security-related properties, and then to capture these classifications in a set of attributes on which vulnerability predictors are to be built. Basically, our approach attempts to answer the following research question: "*Given the data dependence graph of a sink, from the number of inputs, and the numbers and types of input validation and sanitization functions found on the nodes in the graph, can we predict the sink's vulnerability?*".

To classify nodes in DDG_k, we use a hybrid approach that combines static and dynamic analysis techniques. The analysis method is adapted from Balzarotti et al. [11]. Basically, for each function in the nodes of a data dependence graph, Balzarotti et al. first analyzes the function's static program properties in an attempt to determine the potential sanitization effect of the function on the input. If this static analysis is likely to be imprecise, Balzarotti et al. then simulates the effect of the function on the input by exercising the code with different test inputs, containing various types of attack strings. The function return values are passed to a test oracle, which evaluates the function's sanitization effect by checking the presence of those attack strings. We adapted Balzarotti et al.'s hybrid analysis technique as explained in the following.

From the language built-in functions that have specific security purposes (e.g., `addslashes`), the language operators (e.g., string concatenation operator "."), and the predefined language parameters (e.g., `$_GET`) used in a given node *n* in DDG_k, *n* is classified *statically*. But it is classified *dynamically* if it invokes user-defined functions or some built-in functions such as string replacement and string matching functions. As a control flow node *n* may contain a variety of program operations, there may be multiple classifications for *n*. We shall address the attributes on which the classification schemes will rely as *hybrid attributes*. The attributes are listed in Table I and presented next.

Static analysis-based classification: Some of the language built-in functions and operations can be statically and precisely classified from their functional properties or specific security requirements. The classification can be carried out by simply checking the properties of the function or operation. Attributes 1-15 in Table I characterize the functions and operators to be classified statically. These attributes are similar to those proposed in our initial work [2, 16]. Hence, we shall only briefly present them.

Depending on the nature of sources, we categorize the inputs into five types as explained by attributes 1-7 in Table I. Attributes 8-13 basically involve language built-in functions and operators that could be used in input validation and sanitization procedures. Attributes 8 and 9 correspond to language-provided SQLI and XSS sanitization routines (e.g., `htmlspecialchars`), respectively. Functions that invoke stored procedures or prepared statements (e.g., `$query->prepare`) are also classified as SQLI sanitization routines. Attribute 10 involves type casting built-in functions or operations (e.g., `$a = (double) $b/$c`) that cast the input string into a numeric type data. Attribute 11 corresponds to language-provided numeric data type check functions (e.g., `is_numeric`). Attribute 12 corresponds to encoding functions. An input variable may be properly sanitized using encoding functions (e.g., `<a href= 'login.php?name='.urlencode(input)>`). Attribute 13 matches to functions or operations that return predefined information or information not extracted from the input string (e.g., `mysql_num_rows`). We include the attribute *Boolean* as a type of validation and sanitization because a Boolean value returned from a (user-defined or built-in) function is definitely safe for use in the concerned sink. And such a function can be classified statically by checking its function protocol.

Clearly, nodes in DDG_k may also include ordinary operations that may or may not serve any security purpose. They may simply *propagate* the input. Consequently, we use the attribute *Propagate* to characterize functions and operations that are not classified as any of the rest of types via either static analysis or dynamic analysis (discussed in the following).

Dynamic analysis-based classification: When a node invokes a user-defined function or a language built-in string replacement/matching function (such as `str_replace`), the type or purpose of the function cannot be easily inferred from static analysis. Since inputs to web applications are naturally strings, string replacement/matching functions are generally used to implement input validation and sanitization procedures. A good security function generally consists of a set of string functions that allow only valid strings or filter unsafe strings. A filtering action entails character removal or escaping.

In our earlier work [2, 16], we simply characterized such string functions with attributes such as *Match* (e.g., `strcmp`) and *Regex-replacement* (e.g., `preg_replace`). This is too general and thus, our earlier work could not discriminate correct and incorrect string functions (e.g., it treats all `preg_replace` functions as correct or as incorrect). Hence, to improve the accuracy of classification, in this paper, dynamic analysis is used if a node in DDG_k invokes a user-defined function or a language built-in string replacement/matching function. The dynamic attributes are defined as follows:

1) *Numeric*: functions that return only numeric, mathematic, and/or dash '-' characters (e.g., functions that validate inputs such as mathematic equations, postal code, or credit card number).
2) *LimitLength*: functions that limit the length of an input string to a specified number.
3) *URL*: functions that filter directory paths or URLs (e.g., `<a href src='www.hack.com/hack.js'`).
4) *EventHandler*: functions that filter event handlers such as `onload`.
5) *HTMLTag*: functions that filter HTML tags (e.g., strings between < and the first white space or >).
6) *Delimiter*: functions that filter delimiters that could disrupt the syntax of intended HTML documents or SQL queries (e.g., string-delimiters such as single quote and double quote; comment-delimiters such as `/*`, `#`, `//`, and `--`; and some other special characters such as parenthesis, semi-colon, backslash, null byte, and new line).
7) *AlternateEncode*: functions that filter alternate character encodings (e.g., `char(0x27)`).

We believe that the above attributes reflect the types of input validation and sanitization methods that are commonly used to prevent SQLI or XSS attacks. Clearly, a user-defined function or a string replacement/matching function may correspond to more than one attribute.

To collect the above dynamic attributes, (1) we maintain seven sets of test inputs derived from XSS and SQLI cheat sheets provided by OWASP [1] and RSnake [10]. These two security specialists provide a comprehensive coverage of XSS and SQLI attack vectors that could filter many types of input validation and sanitization routines. Each set of test inputs (denoted as test-attr-set) tests for each dynamic attribute (e.g., a test input `<p>test</p>` tests for attribute *HTMLTag* as it could

TABLE I. STATIC-DYNAMIC HYBRID ATTRIBUTES

Attribute ID	Attribute Name	Description
Static attributes		
1	Client	The number of nodes that access data from HTTP request parameters
2	File	The number of nodes that access data from files
3	Database	The number of nodes that access data from database
4	Text-database	Boolean value ‘TRUE’ if there is any text-based data accessed from database; ‘FALSE’ otherwise
5	Other-database	Boolean value ‘TRUE’ if there is any data except text-based data accessed from database; ‘FALSE’ otherwise
6	Session	The number of nodes that access data from persistent data objects
7	Uninit	The number of nodes that reference un-initialized program variable
8	SQLI-sanitization	The number of nodes that apply standard sanitization functions for preventing SQLI issues
9	XSS-sanitization	The number of nodes that apply standard sanitization functions for preventing XSS issues
10	Numeric-casting	The number of nodes that type-cast data into a numeric type data
11	Numeric-type-check	The number of nodes that perform numeric data type check
12	Encoding	The number of nodes that encode data into a certain format
13	Un-taint	The number of nodes that return predefined information or information not influenced by external users
14	Boolean	The number of nodes which invoke functions that return Boolean value
15	Propagate	The number of nodes that propagate partial or complete value of an input
Dynamic attributes		
16	Numeric	The number of nodes which invoke functions that return only numeric, mathematic, or dash characters
17	LimitLength	The number of nodes that invoke string-length limiting functions
18	URL	The number of nodes that invoke path-filtering functions
19	EventHandler	The number of nodes that invoke event-handler filtering functions
20	HTMLTag	The number of nodes that invoke HTML-tag filtering functions
21	Delimiter	The number of nodes that invoke delimiter filtering functions
22	AlternateEncode	The number of nodes that invoke alternate-character-encoding filtering functions
Target attribute		
23	Vulnerable?	Indicates a class label—Vulnerable or Not-Vulnerable

discriminate functions that accept or reject HTML tags); and (2) for a test-attr-set T that tests for an attribute A, we execute the concerned function with a test input t_1 from T and check if the function corresponds to A from the returned result. If the function cannot be classified as A, we choose a different test input t_2 and repeat the process until it is classified as A or all the test inputs from T have been used; (3) step 2 is iterated for all the seven test-attr-sets, each set testing for each dynamic attribute.

Not all function arguments are associated with user inputs. Some arguments are assigned with literal values in the program. Such *literal arguments* can be easily identified from the nodes in DDG_k. Test inputs are only assigned to arguments that are derived from user inputs and literal arguments are assigned with their own literal values extracted using data flow analysis. More than one value is also possible for a literal argument if there are conditional branches. It is logical as the same function can be used to sanitize a variable differently depending on the path along which the variable is propagated. For each possible value of a literal argument, we repeat the above dynamic classification process. As explained, we expect some functions to match multiple classifications.

Attributes 16-22 in Table I represent the classifications presented above.

Target attribute: The attribute *Vulnerable?* is the target attribute which indicates the class label to be predicted.

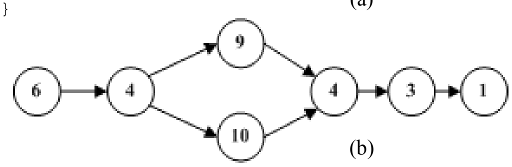
Example: Using the program in Fig. 1, the classification methods and the attribute collection process discussed above can be exemplified as follows.

Statement 1 is a class of *input* because it accesses an HTTP session parameter. It can be statically classified via checking the accessed, predefined parameter (`$_SESSION`). Statement 2 can be classified as *XSS-sanitization* because it invokes a standard escaping routine. Again, it can be statically classified via checking the invoked function name; that is, we predefine the function `htmlspecialchars` as a XSS sanitization type. Statement 3 is a *Un-taint* type. Figure 1b shows the data dependence graph of *HTML* sink 6 in Fig. 1a. Node 4 invokes a user-defined function and it is clear that it could not be precisely classified by just looking up the predefined classifications. We classify such nodes via dynamic analysis.

```
1  $trg_db = $_SESSION['trg_db'];
2  echo ‘<table><tr><th>Target database: ’ .
         htmlspecialchars($trg_db);  //HTML sink
                               . . .
3  $sqlEscape = ‘``’;
4  $query = “UPDATE ” . PMA_backquote($trg_db,
                              $sqlEscape) . “ SET …”;
5  if ($display == true) {
6     echo “<p>” . $query . “</p>”; //HTML sink
7  $rs = mysql_query($query);  //SQL sink

    function PMA_backquote($a_name, $replace) {
8     if (strlen($a_name) && $a_name !== ‘*’) {
9        return ‘`’ .
           str_replace(‘`’, $replace, $a_name) . ‘`’;
      } else {
10       return $a_name;
      }
   }
```

(a)



(b)

Fig. 1. (a) Sample vulnerable PHP code extracted from PhpMyadmin\server_synchronize.php (slightly modified for illustration purpose). The code cleanses an input using standard and customized sanitization functions. (b) Data dependence graph of sink statement 6.

In node 4, a customized security function `PMA_backquote` is invoked with two arguments `$trg_db` and `$sqlEscape`. By a data flow analysis, the literal value '`' for `$sqlEscape` is extracted from node 3. From node 1, `$trg_db` is identified as an input variable. It is then assigned with a value obtained from test-attr-sets. And the function is executed multiple times (each time selecting a different value from test-attr-sets) to determine if it can be classified with one or more dynamic attributes (see *Dynamic analysis-based classification*). Classifications are carried out based on the types of input values used and the contents of the resulting outputs. When a test input such as `` `1 or 1=1 `` is used, the returned result is ``` ``1 or 1=1 ``` and the function is classified as *Delimiter* as it escapes a string-delimiter '`'. Nodes 9 and 10 are not classified as the nodes are contained in the user-defined function that has already been classified.

Based on the above classifications, the attribute vectors for *HTML* sinks 2 and 6 can be extracted from their respective data dependence graphs as (1, 1, 0, 0, 1,…, Not-Vulnerable) and (1, 0, 1, 1, 1,…, Vulnerable), respectively, according to attribute vector (*Session*, *XSS-sanitization*, *Un-taint*, *Delimiter*, *Propagate*,…, *Vulnerable?*). As we propose 23 hybrid attributes, each sink would be represented by a 23-dimensional attribute vector.

IV. Research Hypotheses

This paper aims to investigate the two hypotheses presented in the following.

Our first hypothesis (*H1*) is that the hybrid attributes proposed above could be used to build accurate vulnerability prediction models based on supervised machine learning algorithms. That is, given a sufficient sample of vulnerability data, classifiers learnt from such data are accurate at vulnerability predictions.

If *H1* is true, we should get high recall and precision from a classifier learnt from hybrid attributes. We should also expect that the classifier built from hybrid attributes generally performs better than the one built from static attributes alone.

Although classifiers can be effective, a sufficient number of instances with known vulnerability information is required to train a classifier (supervised training). It is usually tedious and labor-intensive to tag many instances with vulnerability labels. Sometimes, the vulnerability information is not even yet known. In such situations, supervised training is simply not feasible.

Cluster analysis, on the other hand, is a type of unsupervised learning methods in which no class labels are required for training with instances. Intrusion detection studies [17, 18] have shown that cluster analysis could identify numerous anomalies (intrusions in their context) based on the two assumptions that (1) normal instances are much more frequent than anomalies and (2) anomalies have different characteristics from normal instances. If, in our context, the same two assumptions hold, cluster analysis could be used for identifying vulnerable sinks as well. This leads us to our second hypothesis (*H2*): Vulnerable sinks can be distinguished from non-vulnerable sinks based on the proposed hybrid attributes.

If *H2* is true, we should observe that cluster analysis on unlabeled instances containing the data of hybrid attributes can predict vulnerabilities. Hence, when classification-based models are not a feasible option, our approach also includes making use of clustering for building vulnerability prediction models from our hybrid attributes, when the above assumptions are met.

V. Vulnerability Prediction Framework

A. Data Preprocessing

Normalization: To generalize the results, our vulnerability predictors must be able to handle data of arbitrary distributions. Excluding the target attribute, we have 22 hybrid attributes. Twenty attributes take on numeric values and two attributes are binary. From our preliminary tests, we observed that different numeric attributes are defined on different scales and most of the attributes' distributions are highly skewed. This may cause bias toward some attributes (e.g., attributes with large scale values), especially in the context of clustering where similarity measurement combines multiple attribute scales. We use a data standardization technique called *min-max normalization* to avoid this problem, as described in Witten and Frank [9].

Min-max normalization enables our predictors to work in a standardized data space instead of a raw data space. An attribute is normalized when its value is scaled so as to fall within a small specified range (we shall use the range of zero to one). As the normalized value is a linear transformation from the original data value, the relationships among the original data values are preserved. The min-max normalization is to be made for all the instances of every numeric attribute. This shall result in a set of values within the range of zero to one. The binary attributes do not need to be transformed.

Principal component analysis: Principal component analysis (PCA) is a useful technique to identify linearly uncorrelated dimensions in a large datasets with possibly many inter-correlated attributes. Multivariate data mining and statistical techniques used to build classifiers, such as logistic regression, see their performance negatively impacted in the presence of numerous inter-correlated attributes. PCA results in a new set of attributes (principal components), each of which is a linear combination of some of the original attributes. The number of principal components is usually much smaller.

B. Building Supervised Vulnerability Predictors

Classifiers: Classification is a type of supervised learning methods because the class label of each training instance has to be provided. In this study, we build Logistic Regression (LR) and Multi-Layer Perceptron (MLP) models from the proposed attributes. These classifiers were benchmarked as among the top classifiers in recent studies [14]. Two very different techniques are used in an attempt to optimize accuracy. MLP is a type of neural networks. LR is a type of statistical regression models. Details about these classification techniques are provided by Witten and Frank [9].

Training and testing: We use a standard sampling method called 10-fold cross validation setup. The data is divided into ten sets. A classifier is trained on nine sets and then tested on the remaining set. This process is repeated ten times; each time testing on a different set. The order of training and test set is randomized. This test design overcomes the ordering effects due to randomization. This is important to avoid biased performance results due to a certain ordering of training and test data. Isolating a test set from the training set also conforms to the hold-out test design, which is important to evaluate the classifier' capability to predict new vulnerabilities [9].

C. Building Unsupervised Vulnerability Predictors

Cluster analysis: Unlike classification methods, cluster analysis works in the absence of class labels for training instances. But its predictive capability is expected to be inherently lower due to the absence of supervision. Like Portnoy et al.'s unsupervised intrusion detection study [17], the performance of our cluster analysis here should depend on the following two assumptions: (1) non-vulnerable sinks are much more frequent than vulnerable sinks and (2) vulnerable sinks have different characteristics from non-vulnerable sinks. If these two assumptions are met and *H2* is true, vulnerable sinks would be clustered together as outliers in terms of hybrid attribute values, which could then be detected by cluster analysis.

To build an unsupervised vulnerability prediction model, we apply *k*-means clustering algorithm to the proposed hybrid attributes. *k*-means is a simple and often effective partitioning algorithm. Given an input *k*, it partitions a set of instances into *k* clusters in such a way that similarity among instances is maximized within the same clusters and minimized across the different clusters. For similarity measurement, a standard *Euclidean distance function* is used. Further details about the algorithm are provided in [9].

Parameter estimation: As clustering only groups instances based on their similarities, some parameters must be defined to label the clusters as Vulnerable or Not-Vulnerable. The problem here is "*Given a set of clusters produced by a clustering algorithm, what are the best rules (parameters) to single out clusters that contain a large proportion of vulnerable sinks?*". In Portnoy et al.'s clustering-based intrusion detection study [17], a parameter *N*=15% was used as the percentage of the largest clusters that would be labeled as normal as it was found to optimize their results.

For our clustering-based vulnerability prediction study, we use a parameter *%Normal*. It defines the minimum size (in terms of percentage of total instances) of clusters that would be labeled as Not-Vulnerable. For example, if *%Normal*=10, the clusters containing more than 10% of data would be labeled as Not-Vulnerable. As required by the *k*-means algorithm, we also need to determine a parameter *k* that indicates the number of clusters to be produced by *k-means*.

We determined the two parameters by performing experiments that optimize results on the test subjects used in our initial work [2, 16]. The resulting parameters were *k*=4 and *%Normal*=12.

D. Performance Measures

To assess the above models, we shall use recall or probability of detection (*pd*), probability of false alarm (*pf*), and precision (*pr*). We can use the following contingency table to define these standard measures.

		Actual	
		Vulnerable	Not-Vulnerable
Predicted	Vulnerable	True positive (*tp*)	False positive (*fp*)
	Not-Vulnerable	False negative (*fn*)	True negative (*tn*)

Recall (*pd*=*tp*/(*tp*+*fn*)) measures how good our prediction model is in finding actual vulnerable sinks. Precision (*pr*=*tp*/(*tp*+*fp*)) measures the actual vulnerable sinks that are correctly predicted in terms of a percentage of total number of sinks predicted as vulnerable. Probability of false alarm (*pf*=*fp*/(*fp*+*tn*)) is generally used to measure the cost of using the model. Increasing *pd* by tuning the prediction model may, in turn, cause more false alarms or reduce precision. Ideally, the model should neither miss actual vulnerabilities (*pd*~1) nor throw false alarms (*pf*~0, *pr*~1).

VI. EVALUATION

A. Data

To collect data for experiments, we modified the tool *PhpMinerI*, which was used in our earlier work [2]. *PhpMinerI* is based on an open source PHP program analysis tool called *Pixy* [3]. More than 300 PHP built-in functions and 30 PHP operators are predefined in *PhpMinerI* for classification via static analysis. It computes data dependence graph for each sink and collects static attributes. In this work, we modified the tool to incorporate dynamic analysis-based classification. Dynamic analysis is used when a node in DDG_k invokes user-defined functions, or language built-in string replacement or matching functions. No classification is made for nodes in DDG_k that are contained in dynamically classified user-defined functions to avoid unnecessary or overlapping classifications. To identify function arguments (i.e., literals or inputs), static data flow analysis is used. Test inputs are generated from our predefined test suite which reflects the dynamic classification scheme proposed in Section III. Functions are executed using the APIs from a PHP/Java Bridge Java package (provided in http://php-java-bridge.sourceforge.net/pjb/). Function return results are then analyzed to determine the intended validation and sanitization scheme and classify the function accordingly.

Experiments were conducted on six real-world PHP-based web applications obtained from SourceForge [5]. Table II shows relevant statistics for these test subjects. The last column in Table II shows the security advisories, such as CVE [6], from which the test subjects' vulnerability information is extracted. Some of these test subjects have also been benchmarked for the evaluation of some vulnerability detection approaches [3, 4, 21, 28]. Table III shows the datasets collected by *PhpMinerI*. As shown in Table III, we extracted two types of datasets—one corresponds to *HTML* sinks and another corresponds to *SQL* sinks. In total, we collected 10 datasets (only 4 sets of *SQL* sinks were used as we have not tagged the vulnerability labels for *SQL* sinks in *PhpMyAdmin* and *Utopia* systems yet). Column 3 in Table III shows the number and

TABLE II. STATISTICS OF THE TEST SUBJECTS

Test Subject	Description	LOC	Security Advisories
SchoolMate 1.5.4	A tool for school administration	8145	Vulnerability information in [21]
FaqForge 1.3.2	Document creation and management	2238	Bugtraq-43897
Utopia News Pro 1.1.4	News management system	5737	Bugtraq-15027
Phorum 5.2.18	Message board software	12324	CVE-2008-1486 CVE-2011-4561
CuteSITE 1.2.3	Content management framework	11441	CVE-2010-5024 CVE-2010-5025
PhpMyAdmin 3.4.4	MySQL database management	44628	PMASA-2011-14 – PMASA-2011-20

TABLE III. DATASETS

Dataset	#*HTML* sinks	#Vuln. sinks (%Vuln.)	Principal components
schmate-html	172	138 (80%)	7
faqforge-html	115	53 (46%)	7
utopia-html	86	17 (20%)	9
phorum-html	237	9 (4%)	9
cutesite-html	239	40 (17%)	10
myadmin-html	305	20 (7%)	9
Dataset	**#*SQL* sinks**	**#Vuln. sinks (%Vuln.)**	**Principal components**
schmate-sql	189	152 (80%)	7
faqforge-sql	42	17 (40%)	3
phorum-sql	122	5 (4%)	6
cutesite-sql	63	35 (56%)	7

percentage of vulnerable sinks in each dataset (manually inspected and tagged by the first author). On our web site [7], we provide the implementations of *PhpMinerI* and the datasets.

B. Experiments

As *Weka* [9] provides the implementations of required machine learning algorithms, we used it to build the models described in Section V. We first applied min-max normalization and then PCA to every dataset collected. We used a subset of principal components as attributes such that it explained at least 95% of the data variance. The last column in Table III shows the numbers of principal components selected.

To investigate the two research hypotheses presented in Section IV, we conducted two types of experiments. In the first experiment (Supervised vulnerability prediction), we built classifiers and evaluated them, by using the two machine learning techniques discussed above on the datasets with class labels. In the second experiment (Unsupervised vulnerability prediction), we evaluated the abovementioned clustering algorithm on the datasets without class labels.

Supervised vulnerability prediction: In *Weka*, we configured the 10-fold cross validation mode and ran each classifier (LR and MLP) on each dataset. The results of the two classifiers learnt from hybrid attributes are shown in Fig. 2. On average, both models showed good performances with high vulnerability detection rates (≥74%) and low false alarm rates (≤8%). But on some datasets such as *phorum-html* and *phorum-sql*, MLP could not discriminate vulnerabilities whereas LR is able to. Therefore, based on current results, we advise to the use of LR for supervised vulnerability prediction.

The significantly low false alarm rates achieved by our new models indicate that the models' precision has improved from our initial work [2, 16]. Yet, to provide an exact comparison baseline, we also built LR models from static attributes alone and evaluated them in the same way as the above models. Results are shown in Fig. 3. On average, our proposed LR models built from hybrid attributes achieved (*pd*=16%, *pf*=3%, *pr*=2%) improvements over the LR models built from static attributes only. As suggested by Demšar [20], we also used one-tailed Wilcoxon signed-ranks tests to perform pairwise comparisons of the measures achieved by the two types of LR models over the 10 datasets. The tests show that the improvements of recall and precision were statistically significant at a 95% level, though only the increase in recall is interesting from a practical standpoint.

Data & Classifier	Measure (%)	*Pd*	*Pf*	*Pr*
schmate-html	LR	99	3	98
	MLP	99	0	100
faqforge-html	LR	89	5	94
	MLP	91	5	94
utopia-html	LR	94	1	94
	MLP	94	2	89
phorum-html	LR	78	1	70
	MLP	33	0	100
cutesite-html	LR	68	9	61
	MLP	78	8	67
myadmin-html	LR	85	1	89
	MLP	75	1	83
Average results on XSS prediction	**LR**	**86**	**3**	**84**
	MLP	**78**	**3**	**89**
schmate-sql	LR	97	8	98
	MLP	96	35	92
faqforge-sql	LR	88	4	94
	MLP	88	4	94
phorum-sql	LR	100	3	63
	MLP	0	1	0
cutesite-sql	LR	91	14	89
	MLP	89	18	86
Average results on SQLIV prediction	**LR**	**94**	**7**	**86**
	MLP	**68**	**15**	**68**
Overall average	**LR**	**90**	**5**	**85**
	MLP	**74**	**8**	**81**

Fig. 2. Classification results of XSS and SQLI vulnerability predictors built from hybrid attributes.

Data & Classifier	Measure (%)	*Pd*	*Pf*	*Pr*
schmate-html	LR	99	9	98
faqforge-html	LR	91	6	92
utopia-html	LR	88	3	88
phorum-html	LR	44	1	67
cutesite-html	LR	35	6	54
myadmin-html	LR	80	1	89
schmate-sql	LR	93	30	93
faqforge-sql	LR	88	4	94
phorum-sql	LR	40	1	67
cutesite-sql	LR	86	18	86
Overall average	**LR**	**74**	**8**	**83**

Fig. 3. Classification results of XSS and SQLI vulnerability predictors built from static attributes.

We can conclude that dynamic attributes contribute to significantly improving the accuracy of vulnerability predictors. As these attributes are designed to reflect the information about potentially correct and incorrect input validation and sanitization procedures implemented in the program, these results support *H1*.

Unsupervised vulnerability prediction: In *Weka*, we applied *k*-means cluster analysis with Euclidean distance function on each dataset. For identifying vulnerable clusters, the two parameter values (see Section V), *k*=4 and *%Normal*=12, were consistently used for all the datasets. Because there is no need to label instances, unsupervised learning like *k*-means clustering is expected to be much less expensive than building classifiers for vulnerability prediction. But we should also expect it to be less accurate.

More than 40% of sinks in *schmate-html*, *faqforge-html*, *schmate-sql*, *faqforge-sql*, and *cutesite-sql* are vulnerable sinks (see %Vuln. in Table III). These datasets clearly violate our first assumption (stated in Section V) as they contain many vulnerabilities. We expect low predictive power from our clustering models for such datasets. Consequently, we separated the datasets that meet our assumptions from the ones that violate the assumptions, and performed separate evaluations. The results on the former datasets are shown in Fig. 4 and the results on the latter sets are shown in Fig. 5.

As shown in Fig. 4, the *k*-mean's detection rate is very good, especially on *utopia-html* and *phorum-sql* datasets. But its average precision is half that of the supervised models above. This is directly caused by the inherent weakness of unsupervised learning scheme. It is also affected by different trade-offs between detection rates and false alarms. The trade-offs mainly result from the parameter *%Normal*. With a high value of *%Normal* we label more clusters as Vulnerable and reduce precision. Tuning such a parameter must be done in context based on available resources for vulnerability detections.

Data \ Measure (%)	*Pd*	*Pf*	*Pr*
utopia-html	100	13	65
phorum-html	56	11	16
cutesite-html	70	20	41
myadmin-html	55	8	33
phorum-sql	100	7	38
Average	**76**	**12**	**39**

Fig. 4. *k*-means clustering analysis results on the datasets which meet the assumptions.

Data \ Measure (%)	*Pd*	*Pf*	*Pr*
schmate-html	9	0	100
faqforge-html	26	0	100
schmate-sql	3	32	29
faqforge-sql	0	0	undefined
cutesite-sql	0	0	undefined
Average	**8**	**6**	**undefined**

Fig. 5. *k*-means clustering analysis results on the datasets which violate the assumptions.

As expected, as shown in Fig. 5, cluster analyses on datasets which violate our first assumption result in very low detection rates because many or all of the vulnerable sinks did not appear as outliers (in terms of hybrid attribute values) to our clustering model. *Pr* was also undefined for some datasets as both *pd* and *pf* were null.

From the results in Fig. 4, we can conclude that, if certain assumptions are met, cluster analysis on unlabeled instances using hybrid attributes can help accurately predict vulnerabilities, thus supporting *H2*.

C. Threats to Validity

Our data only reflects the known vulnerabilities that are reported in vulnerability databases. Hence, our vulnerability predictions based on classifiers do not account for undiscovered vulnerabilities.

The application of cluster analysis is limited by the two assumptions stated above. In our experiments, clustering-based prediction models could accurately isolate vulnerabilities in the datasets which satisfy those assumptions. However, it is unclear how frequently these assumptions hold in practice across systems and types of vulnerabilities. Further, we estimated two parameters *(k* and *%Normal*) driving the accuracy of cluster analysis based on our experience with preliminary experiments. We used the same two parameters for all the datasets. The parameters worked well for our context but may not generalize well elsewhere. But as most of our test subjects such as *PhpMyAdmin* are widely-used, real-world applications, we believe that the above threats do not significantly affect our results although tuning the parameters may be required for some applications.

The use of different or more data preprocessing activities may also alter our results. For example, during our preliminary experiments, we tested the datasets with and without PCA (see Section V). Results without PCA were significantly inferior to results with PCA for the majority of datasets though no significant differences were observed for some.

Different classification and clustering algorithms could result in different results. In our experiments, we used two very different classification algorithms which are statistical-based and network-based, respectively. We also tried other classifiers like C4.5 and naïve bayes, but the average results were similar. We have not tried another algorithm for clustering-based prediction, but we expect similar results if similar parameters (i.e., *k* and *%Normal*) are used.

Like all other empirical studies, our results are limited to the applied data mining processes, the test subjects, and the experimental setup used. One good solution to refute, prove, or improve our results is to replicate the experiments with new test subjects and probably with further data mining strategies. This can be easily done since we have clearly defined our methods and setup, and we also provide the data used in the experiments and the data collection tool on our web site [7].

VII. Related Work

Our work applies data mining for the prediction of vulnerabilities in web applications. Hence, its related work falls into three categories: defect prediction, vulnerability prediction, and vulnerability detection.

Defect prediction: Data mining models used by our approach are similar to those used in many defect prediction studies [12, 13, 14, 15, 25]. In these studies, defect predictors are generally built from static code attributes such as object-oriented design attributes [12], LOC counts, and code complexity attributes [14, 15] because static attributes can be cheaply and consistently collected across many systems [15]. However, it was quickly realized that such attributes can only provide limited accuracy [13, 15, 25]. Arisholm et al. [13] and Nagappan et al. [25] reported that process attributes (e.g., developer experience and fault history) could significantly improve prediction models. On the other hand, as process attributes are difficult to measure and measurements are often inconsistent, Menzies et al. [15] showed that static code attributes can still be effective if predictors are tuned to user-specific goals.

In contrast to defect prediction studies, our study targets security vulnerabilities in web applications. Since these studies show that there is no universal set of attributes, we define specific attributes targeted at predicting vulnerabilities based on automated and scalable static and dynamic analysis.

Vulnerability prediction: Shin et al. [23] used code complexity, code churn, and developer activity attributes to predict vulnerable programs. They achieved 80% recall and 25% false alarm rate. Their assumption was that, the more complex the code, the higher the chances of vulnerability. But from our observations, many of the vulnerabilities arise from simple code and, if a program does not employ any input validation and sanitization routines, it would be simpler but nevertheless contain many vulnerabilities.

Walden et al. [24] investigated the correlations between security resource indicator (SRI) and numbers of vulnerabilities in PHP web applications. SRI is derived from publicly available security information such as past vulnerabilities, secure development guidelines, and security implications regarding system configurations. Neuhaus et al. [26] also predicted vulnerabilities in software components from the past vulnerability information, and the imports and function calls attributes. Their work is based on the concept that components which contain similar imports and function calls as known vulnerable components are likely to be vulnerable as well. They achieved 45% recall and 70% precision.

These existing vulnerability prediction approaches generally target software components. By contrast, our method targets specific program statements for vulnerability prediction. The other difference is that we use code attributes that characterize input validation and sanitization routines.

Vulnerability detection: Jovanovic et al. [3] and Xie and Aiken [4] showed that many XSS and SQLI vulnerabilities can be detected by static program analysis techniques. They identify various input sources and sensitive sinks, and determine whether any input data is used in those sinks without passing through sanity checks. In general, such static taint tracking approaches are effective but not efficient as they generate many false alarms.

To improve precision, Fu and Li [27] and Wassermann and Su [28] approximated the string values that may appear at sensitive sinks by using symbolic execution and string analysis techniques. More recent approaches incorporate dynamic analysis techniques such as testing based on input sanitization graphs [11], concolic execution [21], and model checking [22]. These approaches reason about various paths in the program that lead to sensitive sinks and attempt to generate test cases that are likely to be attack vectors. All these approaches reduce false alarm rates. But symbolic, concolic, and model checking techniques often lead to a path explosion problem. It is difficult to reason about all the paths in the program when the program contains many branches and loops. Further, the performance of these approaches also depends very much on the capabilities of their underlying string constraint solvers in handling a myriad of string operations offered by programming languages.

By contrast, although our approach also requires dynamic analysis, this is done at the function level. It does not require string solving and reasoning of (potentially infinite) program paths like concolic execution and model checking techniques.

However, symbolic, concolic, and model checking approaches could possibly yield high precision rates which may never be matched by data mining methods. Thus, our objective is not to provide a replacement for such techniques but rather to provide a complementary approach to use when they are not applicable or in combination with them. One could, for example, first gather vulnerability predictions on code sections using data mining and then focus on the code sections with predicted vulnerabilities using the more precise techniques mentioned above. Thereafter, ideally, the confirmed vulnerabilities should be removed by manual audits or by using automated vulnerability removal techniques such as [29].

VIII. CONCLUSION

The goal of this paper is to aid security auditing and testing by providing probabilistic alerts about potentially vulnerable code statements. We propose attributes, based on hybrid static and dynamic code analysis, which characterize input validation and sanitization code patterns for predicting SQL injection and XSS vulnerabilities. Given a security-sensitive program statement, we collect the hybrid attributes by classifying the nodes from its data dependency graph. Static analysis is used to classify nodes that have unambiguous security-related purposes. Dynamic analysis is used to classify nodes that invoke user-defined or language built-in string replacement/matching functions since classification of such nodes by static analysis could be imprecise.

We evaluated if these hybrid attributes can be used to build effective vulnerability predictors, using both supervised and unsupervised learning methods. The latter have, in practice, the advantage of not requiring labeled training data (with known vulnerabilities) but may be significantly less accurate. In the experiments on six PHP web applications, we first showed that the hybrid attributes can accurately predict vulnerabilities. We also observed that dynamic analysis helped achieve much better accuracy than static analysis alone, thus justifying its application. Last but not least, when meeting certain assumptions, cluster analysis showed to be a reasonably accurate, unsupervised learning method when no labeled data is available for training. But since it is not nearly as accurate as supervised learning, it should be considered as a trade-off between data collection cost and accuracy.

To generalize our current results, we hope that researchers will replicate our experiment, possibly using the data and tool we posted online. We also intend to conduct more experiments with industrial applications. While we believe that the proposed approach can be a useful and complementary solution to existing approaches, studies need to be carried out to determine the feasibility and usefulness of integrating multiple approaches (e.g., prediction+detection+removal).

Acknowledgment

Lionel C. Briand was supported by a PEARL grant from the Fonds National de la Recherche, Luxembourg (FNR/P10/03 – Validation and Verification Laboratory).

References

[1] OWASP. "The open web application security project," http://www.owasp.org, accessed January 2012.

[2] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in International Conference on Software Engineering, 2012, pp. 1293-1296.

[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in IEEE Symposium on Security and Privacy, 2006, pp. 258-263.

[4] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in USENIX Security Symposium, 2006, pp. 179-192.

[5] SourceForge. http://www.sourceforge.net, accessed March 2012.

[6] CVE. http://cve.mitre.org, accessed March 2012.

[7] PhpMiner. http://sharlwinkhin.com/phpminer.html.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems, vol. 9, pp. 319-349, 1987.

[9] I. H. Witten and E. Frank, Data Mining, 2nd ed., Morgan Kaufmann, 2005.

[10] RSnake. http://ha.ckers.org, accessed March 2012.

[11] D. Balzarotti et al., "Saner: composing static and dynamic analysis to validate sanitization in web applications," in IEEE Symposium on Security and Privacy, 2008, pp. 387-401.

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

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

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

[15] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," Automated Software Engineering, vol. 17 (4), pp. 375-407, 2010.

[16] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 310-313.

[17] L. Portnoy, E. Eskin, and S. Stolfo, "Intrusion detection with unlabeled data using clustering," in ACM CSS Workshop on Data Mining Applied to Security, 2001.

[18] S. Thamaraiselvi, R. Srivathsan, J. Imayavendhan, R. Muthuregunathan, and S. Siddharth, "Combining naive-bayesian classifier and genetic clustering for effective anomaly based intrusion detection," Lecture Notes in Computer Science, vol. 5908, pp. 455-462, 2009.

[19] S. Palmer, Web Application Vulnerabilities: Detect, Exploit, Prevent, Syngress, 2007.

[20] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," Journal of Machine Learning Research, vol. 7, pp. 1-30, 2006.

[21] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in International Conference on Software Engineering, 2009, pp. 199-209.

[22] M. Martin and M. S. Lam, "Automatic generation of XSS and SQL injection attacks with goal-directed model checking," in USENIX Security Symposium, 2008, pp. 31-43.

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

[24] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 545-553.

[25] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," in International Symposium on Software Reliability Engineering, 2006, pp. 62-74.

[26] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in ACM Conference on Computer and Communications Security, 2007, pp. 529-540.

[27] X. Fu and C.-C. Li, "A string constraint solver for detecting web application vulnerability," in International Conference on Software Engineering and Knowledge Engineering, 2010, pp. 535–542.

[28] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 32-41.

[29] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," Information and Software Technology, vol. 54 (5), pp. 467-478, 2012.