4-2013

Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Hee Beng Kuan TAN

Information and Software Technology 55 (2013) 1767–1780

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns

Lwin Khin Shar *, Hee Beng Kuan Tan

Block S2, School of Electrical and Electronic Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798, Singapore

ARTICLE INFO

Article history:
Received 27 June 2012
Received in revised form 4 April 2013
Accepted 5 April 2013
Available online 23 April 2013

Keywords:
Vulnerability prediction
Data mining
Web application vulnerability
Input sanitization
Static code attributes
Empirical study

ABSTRACT

Context: SQL injection (SQLI) and cross site scripting (XSS) are the two most common and serious web application vulnerabilities for the past decade. To mitigate these two security threats, many vulnerability detection approaches based on static and dynamic taint analysis techniques have been proposed. Alternatively, there are also vulnerability prediction approaches based on machine learning techniques, which showed that static code attributes such as code complexity measures are cheap and useful predictors. However, current prediction approaches target general vulnerabilities. And most of these approaches locate vulnerable code only at software component or file levels. Some approaches also involve process attributes that are often difficult to measure.

Objective: This paper aims to provide an alternative or complementary solution to existing taint analyzers by proposing static code attributes that can be used to predict specific program statements, rather than software components, which are likely to be vulnerable to SQLI or XSS.

Method: From the observations of input sanitization code that are commonly implemented in web applications to avoid SQLI and XSS vulnerabilities, in this paper, we propose a set of static code attributes that characterize such code patterns. We then build vulnerability prediction models from the historical information that reflect proposed static attributes and known vulnerability data to predict SQLI and XSS vulnerabilities.

Results: We developed a prototype tool called *PhpMinerI* for data collection and used it to evaluate our models on eight open source web applications. Our best model achieved an averaged result of 93% recall and 11% false alarm rate in predicting SQLI vulnerabilities, and 78% recall and 6% false alarm rate in predicting XSS vulnerabilities.

Conclusion: The experiment results show that our proposed vulnerability predictors are useful and effective at predicting SQLI and XSS vulnerabilities.

1. Introduction

According to Open Web Application Security Project (OWASP) [25], SQL injection (SQLI) and cross site scripting (XSS) are the two most common and critical web application vulnerabilities. To address these security threats, many vulnerability detection approaches based on static and dynamic taint analysis techniques [13,15,17,18,42,44] have been proposed. Though these taint analysis-based approaches have been shown to be effective at detecting SQLI and XSS vulnerabilities, static approaches [13,17,44] generally produce too many false alarms. Dynamic approaches are usually more accurate. But while dynamic analysis typically requires complex analysis frameworks such as concolic execution [15], full implementations of some of these approaches [15,18,42] are neither commercially nor publicly available. As a result, software development teams face difficulties in adopting these existing approaches. The growing numbers of vulnerability reports in security databases such as BugTraq [4] further support the need to find *alternative or complementary* solutions.

On the other hand, recently, the applications of machine learning techniques in software defect prediction and vulnerability prediction have achieved promising results. In this domain, researchers correlate software attributes with defects [40,45] or vulnerabilities [33]. In general, they build prediction models that categorize software modules/components/programs, represented by a set of software attributes, into one of the classes (e.g., defective and non-defective) by using classifiers that are trained on the same set of attributes obtained from software modules with known defect or vulnerability information [28]. The advantage of these techniques is that commonly-used *static code attributes* such as size, Halstead [12], and cyclomatic complexity [19] attributes can be easily collected. The availability of open source data mining

* Corresponding author. Tel.: +65 97423763.
E-mail addresses: shar0035@ntu.edu.sg (L.K. Shar), ibktan@ntu.edu.sg (H.B.K. Tan).

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.04.002

tools such as the *Weka* tool described in Witten and Frank [43] also allows software engineers to readily apply required machine learning methods.

Hence, prediction methods could provide a cheaper and yet efficient way of mitigating web application security issues. As mentioned by Menzies et al. [21], these data miners may not accurately identify vulnerabilities like sophisticated vulnerability detection methods. But they would be useful at providing probabilistic remarks about the vulnerabilities of code sections. Software testers could then save time and effort by focusing on the code sections predicted to be vulnerable. However, there are still developments required to improve existing vulnerability prediction approaches. Some of these approaches require process attributes (such as developer activity) to build vulnerability predictors [33]. Unlike static code attributes, process attributes are often difficult to be collected and data collected may not be consistent across projects [3,22]. And some of these approaches locate vulnerable code only at component level [11,24], which makes vulnerability auditing hard for software testers.

An application that accesses database via a SQL language is vulnerable if an unrestricted input is used to build the query string because an attacker might craft the input value to have unauthorized access to the database and perform malicious actions. This security issue is called SQLI vulnerability. An application that sends HTTP response data to a web client is vulnerable if an unrestricted input is included in the response data because an attacker might inject a malicious JavaScript code in the input value. The injected code when executed by the client's browser could perform malicious actions to the client. This security issue is called XSS vulnerability.

Web developers generally implement input sanitization schemes to prevent these two vulnerabilities [25]. An application is vulnerable if the implementation of input sanitization is inadequate or there is no such method implemented. Consequently, the characteristics of input sanitization routines implemented in a program could be useful for predicting the program's vulnerability.

The above observations serve as our main motivation for this study. In this paper, we propose *input sanitization code attributes* which can be statically collected. From these attributes, we aim to build SQLI and XSS vulnerability predictors which provide high recalls and low false alarm rates so that the predictors can be used alternatively or in combination with existing taint-based approaches. Compared to current vulnerability prediction approaches, we only use static code attributes and we target vulnerable code at statement level.

Our proposal could be easily extended to address other web application vulnerabilities such as buffer overflow, path traversal, and URL redirects/forwards. These vulnerabilities are caused by the common weakness of web applications in handling user inputs properly. But we shall limit this paper to only SQLI and XSS vulnerabilities due to the readily-available vulnerability information and the prevalence of these two vulnerabilities in many web applications.

This paper is an extension of our initial work [31], which briefly presented the idea and provided preliminary results based on the experiments on three test subjects. This paper extends and enhances our previous work in the following ways:

- We provide in details the techniques for building vulnerability predictors.
- We present the prototype tool called *PhpMinerI* which is publicly available at the first author's website [26]. The tool helps to automatically extract the data of our proposed input sanitization code attributes from PHP programs while the user has to manually tag the extracted data with vulnerability labels.
- Comprehensive experiments have been conducted on a total of eight PHP-based web applications. The test subjects are heterogeneous and their sizes vary from small to large scales.
- A data preprocessing step is introduced to ensure that our prediction framework is consistent across datasets with different data distributions.
- Statistical inference methods and additional performance measures have also been used to ensure rigorous evaluations.
- An attribute ranking technique has also been used to identify the attributes that are most informative and verify if omitting the attributes that are less informative could improve the predictive performance of our models.
- We also provide a comparison between our best predictor and a taint-based vulnerability detector and discuss strengths and weaknesses of the two related techniques based on the results.

The organization of the paper is as follows. Section 2 presents the classification schemes that characterize input sanitization methods. Section 3 presents our proposed vulnerability prediction framework. Section 4 provides our research hypotheses. Section 5 describes the prototype tool and evaluates the performance of vulnerability predictors. Section 6 compares our work with related approaches. Section 7 provides conclusions and future work.

2. Classification

Static code attributes that we propose are based on the control flow graph (CFG) of web application program. Each node in the CFG represents a single program statement. Therefore, we shall use the program statement and the node interchangeably depending on the context.

Though it is not difficult to extend the logic presented in this paper to other programming languages, our proposed approach and discussion here is limited to PHP programs because SQLI and XSS vulnerabilities are widespread in PHP applications. One could adapt our approach to languages like Java by predefining language built-in functions and operations according to our classification schemes described in this section. And since our approach only requires simple data flow analysis, data collection can be easily implemented using program analysis tools such as Soot [36].

To explain our approach, we shall use the sample vulnerable PHP code in Fig. 1 extracted from one of our test subjects—*Utopia News Pro*. The code was slightly modified for illustration purpose.

2.1. Input and sink classification

Typically, a web application program accesses user inputs and propagates them via its program variables for further processing of the application's logics. These processes may often include sensitive program operations (sensitive sinks) such as database updates, HTML outputs, and file accesses. If the program variables propagating the input data are not checked before being used in those sinks, security violations may occur if an attacker has crafted the input values. A variable is said to be *tainted* if it is assigned by a user input. Hence, in security, it is important to first identify the sources from which tainted variables are defined and the sinks that reference tainted variables.

Hence, we call a node u in a CFG at which the data submitted by an external user is accessed an *input node*. The nodes which reference data from HTTP request parameters, database, and XML files are some examples of input nodes. According to different natures of *input sources*, we classify inputs into the following types:

(1) *Client*: Data submitted via HTML forms and URLs (e.g., `$_GET`, `$_POST`).

```
1   <html><head><title><?php echo $sitetitle; //vuln HTML sink ?>
      News - Powered by Utopia News Pro</title></head><body>

    <?php
2   if ($_POST['action'] == 'submitip') {
3        $ipaddress = addslashes($_POST['ipaddress']);
4        $description = addslashes($_POST['description']);
5        mysql_query("INSERT INTO unp_bannedip ('ip','reason') VALUES
              ('$ipaddress','$description')");   //non-vuln SQL sink

6   } else if ($_POST['action'] == 'removeban') {

7        if (isset($_POST['unp_user'])) {
8             $username = escape_string($_POST['unp_user']);
         } else {
9             $username = trim($_COOKIE['unp_user']);
         }

10       echo '<form action=\"commentsadmin.php\" method=\"post\">
          <input type=\"hidden\" value=\"'.$username.'\" name=\"username\" />'; //vuln HTML sink

11       if (isset($_POST['password'])) {
12            $htmlPassword = escape_string($_POST['password']);
13            echo '<input type=\"hidden\" value=\"'.$htmlPassword.
                                  '\"name=\"password\"/>'; //non-vuln HTML sink

         }
14       $ipid = $_POST['ipid'];
15       $getip= mysql_query("SELECT * FROM unp_bannedip
                                  WHERE id=".addslashes($ipid)); //non-vuln SQL sink

16       echo 'Are you sure you want to remove the ban on IP'.
                        $getip['ip'].'?<br />';  //vuln HTML sink
17       echo '<a href=\"commentsadmin.php?action=unban&ipid=';
                        urlencode($ipid).'\">Yes</a><br /></form>';  //non-vuln HTML sink
18       echo '<a href=\"commentsadmin.php?action=bannedips\">No</a><br />';//non-vuln HTML sink

    }

    function escape_string($string) {
19      $string = addslashes($string); // 'SQLI-sanitization'
20      $string = str_replace('<','&lt;', $string); // 'Replacement'
21      $string = str_replace('>','&gt;', $string); // 'Replacement'
22      return $string;
    }
    ?> </body></html>
```

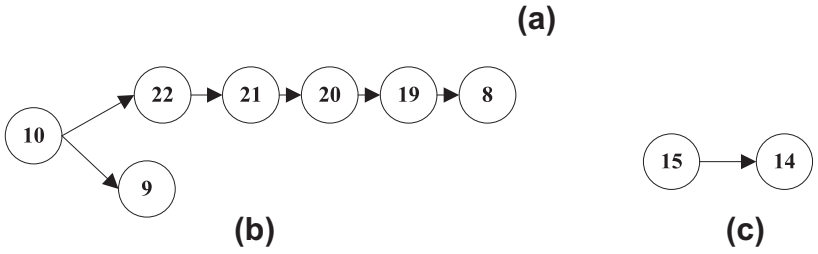

Fig. 1. (a) Sample web application program that contains vulnerable and non-vulnerable sinks (modified code snippet from utopia/users.php). (b) Data dependence graph of statement 10. (c) Data dependence graph of statement 15.

(2) *File*: Data accessed from external files such as cookies and XML files (e.g., `$_COOKIE`, `fgets`). The contents in these files may have been tempered by malicious users.
(3) *Database*: Data retrieved from database that is updated by programs with unknown vulnerabilities (e.g., `mysql_fetch_array`). As the data retrieved from database could be of different data types, we further classify the database inputs into two sub-types—*Text* and *Other*. This is to reflect the fact that string data from database are often exploited to cause second order security attacks such as second order SQLI [2] and stored XSS [15]. Therefore, *Text-database* inputs represent 'String' type data (e.g., `varchar`, `text`, and `blob`). *Other-database* inputs represent data of any other data types such as 'Numeric' (e.g., `int`) and 'Date and Time' (e.g., `timestamp`).
(4) *Session*: Data accessed from persistent data objects that may have been defined by programs with unknown vulnerabilities (e.g., `$_SESSION`).
(5) *Uninit*: Variables which may not have been initialized in the program before referenced in sensitive sinks. In programming languages like PHP, '*Uninit*' variables are often referenced in program operations because it is possible to configure settings such as register_global so that any uninitialized variable is automatically assigned by data from HTTP request parameters.

It is clear that '*Client*', '*File*', and '*Uninit*' are direct user inputs that would definitely cause SQLI and XSS vulnerabilities if used immediately without any checks. '*Database*' and '*Session*' represent data originated from usually different programs (or different user

sessions) and hence, their tainted-ness is usually unknown or hard to be tracked. A naïve taint analysis-based vulnerability detection approach (such as [13,17]) might result in many false negatives if users assume that such inputs are safe and in false positives if users assume otherwise. This is the area where our proposed vulnerability predictors are expected to perform better because the prediction is not based on user assumptions and instead based on historical information.

We call a node *k* in the CFG of a web program a *sensitive sink* if the execution of *k* may lead to harmful operations. Throughout the paper, we shall address *k* as a sensitive sink. For this study, we use two types of sensitive sinks:

(1) *SQL:* Database operations that are susceptible to SQLI attacks (e.g., `mysql_query`).
(2) *HTML:* HTML output operations that are susceptible to XSS attacks (e.g., `print and echo`).

For example, in the program shown in Fig. 1a, variable `$site-title` referenced in statement 1 is an *Uninit* input. Statements 3, 4, 8, 12, and 14 are *Client* inputs. Statement 9 is a *File* input. Statements 1, 10, and 16 are vulnerable *HTML* sinks and statements 13, 17, and 18 are non-vulnerable *HTML* sinks. Statements 5 and 15 are non-vulnerable *SQL* sinks. Statement 15 is also a *Database* input.

2.2. Input sanitization classification

To prevent SQLI and XSS vulnerabilities, web applications generally adopt various input sanitization routines. Input sanitization removes, replaces or escapes malicious characters from user inputs according to the context of sensitive program operation so that those characters may not cause the program to perform unintended operations. As such, whether or not a sensitive program operation is vulnerable is dependent on the effectiveness of sanitization methods applied with respect to the context of that particular operation. In consequence, static code attributes that reflect the characteristics of these methods could be used to predict vulnerability.

By default, inputs to web application programs are strings. As such, input sanitization operations performed in a program are mainly string operations. Our concept is to classify the natures of string operations applied according to their potential effects on the tainted-ness of the variables referenced in sensitive sink *k*. Intuitively, such operations can be found in the data dependence graph DDG_k of *k*. A data dependence graph of a node in the CFG provides reaching definitions, that is, it contains all the nodes in the CFG that define the values of the variables used in the node [7]. For instance, the data dependence graphs of *HTML* sink node 10 and *SQL* sink node 15 are shown in Fig. 1b and c respectively.

A variety of input sanitization operations may be found in the nodes in DDG_k. For example, a node (or a set of nodes) in DDG_k may ensure that an input representing a client's age contains only numeric values and another node may remove some predefined characters from an input. Different operations may serve different purposes and may have different effects on the tainted-ness of an input. Hence, we shall classify the nodes in DDG_k such that the classifications reflect the potential input sanitization operations performed on *k*. For each node *n* in DDG_k, classification is carried out by analyzing the types of the language-built-in functions invoked, and program parameters and operators used in *n*. The classification of input sanitization methods is as follows:

(1) *SQLI-sanitization*: standard (language-provided) functions designed to prevent SQL injection issues (e.g., `addslashes`, `mysql_real_escape_string`). Functions that implement parameterized queries (e.g., `$dbh->prepare`) are also included in this type.
(2) *XSS-sanitization*: standard (language-provided) functions designed to prevent XSS issues (e.g., `htmlspecialchars`).
(3) *Encoding*: functions that encode an argument according to a specific encoding format (e.g., `urlencode`).
(4) *Encryption*: encryption or hashing functions designed to ensure secure data transfer (e.g., `crypt`).
(5) *Replacement*: string-based substring replacement functions (e.g., `substr_replace`).
(6) *Regex-replacement*: regular expression-based substring replacement functions (e.g., `preg_replace`).
(7) *Numeric-conversion*: functions that process one or more arguments and return a numeric value (e.g., `floatval`) or numeric type casting operations (e.g., `$x = (int) $_GET['id']`).
(8) *Un-taint*: functions or operations that return predefined information (e.g., `$a = 'text'`), information derived from configuration settings (e.g., `localeconv`), or numeric information derived from program operations (e.g., `mysql_field_len`). These functions or operations may not be intended as input sanitization but are considered as one because the resulting data is generally benign.

It is apparent that as '*SQLI-sanitization*' and '*XSS-sanitization*' functions are standard methods, they would be commonly used in web applications to defend against SQLI and XSS. But as shown by Anley [2] and Fogie et al. [9], there are many ways to use inputs in SQL and HTML output statements (contexts). For some contexts, language-provided sanitization methods are inadequate, or they are not suitable. Hence, it is also common that developers implement (additional) custom sanitization methods using string functions like '*Replacement*' functions.

As shown in statements 20–21 of the program in Fig. 1a, '*Replacement*' functions or '*Regex-replacement*' functions can be used to filter potentially dangerous characters. '*Encryption*' and '*Encoding*' functions can also play a role in input sanitization (in this paper, input sanitization is generally referred to as sanitization of inputs to prevent SQLI and XSS issues). For example, `$urlencode(ipid)` in statement 17 in Fig. 1a is sanitized enough to be safely used as a URL parameter. '*Numeric-conversion*' method is the most effective way to prevent SQLI and XSS when the input is intended to be numeric. For example, `$getip['ip']` in statement 16 in Fig. 1a which causes the XSS vulnerability can be properly sanitized by applying the '*Numeric-conversion*'-type function as follows: intval `$(getip['ip'])`. '*Un-taint*' functions or operations are those that may not be intended as input sanitization, but since those functions or operations only propagate benign information, they are also important indicators of vulnerabilities.

Some nodes contained in DDG_k may also perform ordinary operations that do not serve any security purpose. They may simply *propagate* the input data. From the empirical studies, we observed that the above input sanitization types are commonly implemented to prevent web vulnerabilities. However, there may also be *other* types of preventive measures that we did not observe. Furthermore, functions invoked at *n* could also be library functions or user-defined functions as input sanitization is often *customized* to users' needs. Consequently, we classify the remaining nodes in DDG_k that are not classified as any of the above types into one or more of the following types:

(1) *Propagate:* functions or operations that may convert arguments into different representations but return part or whole of the original arguments (e.g., substr, trim, explode, `$$str = str.'abc'`); functions that unquote or decode arguments (e.g., `html_entity_decode`, `urldecode`, `stripslashes`).

(2) *Custom:* library or user-defined functions with unknown vulnerabilities.
(3) *Other:* functions or operations that are not classified as any of the above types.

Since a node may include different program parameters, operators, and functions, there may be multiple classifications for a node. For example, in Fig. 1c, from the properties of node 15, the string function `addslashes` can be extracted, hence, it would be classified as the type '*SQLI-sanitization*'. As the node also invokes the function `mysql_query` function, it would also be classified as the type '*SQL*' sink.

3. Proposed vulnerability prediction framework

Based on the classification scheme presented in Section 2, in this section, we present the static code attributes that reflect the classifications. We then provide the vulnerability prediction modeling techniques.

3.1. Static code attributes

As shown in Table 1, we propose 21 static code attributes that reflect the characteristics of common input sanitization methods. Our unit of measurement is a sensitive sink. The last attribute *Vulnerable?* in Table 1 is the target attribute which indicates the vulnerability of the sink. Each sink is to be represented by a 21-dimensional attribute vector. Hence, attribute vectors collected for *HTML* sinks and *SQL* sinks in a web program are instances that can be used to learn XSS and SQLI vulnerability predictors respectively if their vulnerability information is available, or instances for which their vulnerability can be predicted by the learned predictors.

To illustrate, for the program in Fig. 1a, we have computed the data dependence graphs for *HTML* sink 10 and *SQL* sink 15 as shown in Fig. 1b and c. Similarly, if computed, the data dependence graphs of other *HTML* sinks 1, 13, 16, 17, and 18 shall be formed with the sets of nodes {1}, {12, 13, 19, 20, 21, 22}, {14, 15, 16}, {14, 17}, and {18} respectively. Whereas, the data dependence graphs of *SQL* sink 5 shall be formed with the set of nodes {3, 4, 5}. Table 2 shows the sample attribute vectors of *HTML* sinks 1, 10, 13, 16, 17, and 18 extracted from their respective data dependence graphs.

3.2. Prediction models

3.2.1. Data preprocessing

In Section 3.1, we proposed 21 attributes. Eighteen attributes are numeric. These numeric attributes could have arbitrary and different data distributions (see Fig. 4 in Section 5). Depending on program requirements and developer's style of programming, our attributes that characterize input sanitization functions may be defined on different numeric scales. Hence, a data preprocessing step is required so that classifiers are not biased towards some attributes. We use *min–max data normalization* technique to standardize our data. Normalization maps a given numeric value to a value within a specified range. A range of zero to one is used for our prediction framework. All numeric attributes of both training and test data are to be normalized.

3.2.2. Data reduction

When proposing a set of attributes that characterize the quality of software, it is important to identify the attributes that best capture it. This process is known as *feature selection* or *attribute ranking*. It can also be used to identify irrelevant or redundant attributes. Various attribute ranking techniques, such as information gain, gain ratio, chi-square, and symmetrical uncertainty, have been investigated in literature [10]. These techniques independently rank attributes regardless of the classifier used. In this study, we shall use gain ratio method to find *best* attributes (i.e., attributes which contribute most to a classifier's performance) and check if *data reduction* (i.e., using only the best attributes) could improve performances.

3.2.3. Classifiers

We use classifiers as the base data miners for building vulnerability prediction models. Based on different characteristics of classification algorithms, classifiers can be grouped into different categories such as tree-based approaches, neural networks, support vector machines, nearest-neighbor approaches, statistical procedures, and ensembles [16]. Literature studies [3,21] have shown that different classification algorithms may produce different performances. Therefore, in this study, we use three different classifiers, C4.5, Naïve Bayes (NB), and Multi-Layer Perceptron (MLP), for predicting SQLI and XSS vulnerabilities. This allows us to use the algorithm that performs best in our context.

C4.5 is a decision tree-based classifier that recursively partitions the training data by means of attribute splits at each node of the tree. The splitting criterion is the information gain that results from choosing an attribute for splitting the data. For example, C4.5 executing on the attribute vectors of six *HTML* sinks from Table 2 (presented in Section 3.1) results in a tree:

```
SQLI-sanitization ⩽ 0: Non-Vulnerable.
SQLI-sanitization > 0: Vulnerable.
```

which can be interpreted as "a sensitive sink is vulnerable if the number of nodes classified as *SQLI-sanitization* is more than 0". Such a predictor could be especially useful when a programmer misuses SQL injection-specific escaping procedures to protect *HTML* sinks. The use of inadequate escaping procedures causing XSS vulnerabilities is fairly common in practice.

NB is a simple statistical classifier that estimates the posterior probability of each class of the target attribute (in our case, *Vulnerable* or *Non-Vulnerable*) based on the values of the training data so that a given module (in our case, a module is a sensitive sink) can be assigned to the class label with the highest probability.

MLP is a sophisticated classifier which depicts a neural network structure of the human brain. It consists of an input layer, one or more hidden layers, and an output layer. The data of training instances are fed to the units in the input layer. Weighting, aggregation, and thresholding functions are then iteratively applied to the data propagated along the units in the layers to predict the class label of an instance which is presented in the output layer.

Equations and detailed information of these classification algorithms can be found in data mining books such as Witten and Frank [43].

3.2.4. Model training and testing

We shall use 10×10 cross-validation method for training and testing the three classifiers. This test design has been used by many software defect prediction studies [10,20,21]. The dataset is randomly divided into 10 folds. The classifier is trained on the 9 folds and then tested on the remaining fold, rotating each fold as the test fold. This entire process is iterated ten times.

Fig. 2 provides the pseudocode of our model evaluation procedure. To control threats to validity of the results, model training and testing method has to conform to *holdout* test design and needs to be immune to *order effects* [21]. Order effects are exhibited when a certain ordering of training and test data results in a

Table 1
Proposed static code attributes.

Attribute	Description
Client	The number of nodes which access data from external users
File	The number of nodes which access data from files
Database	The number of nodes which access data from database
Text-database	Boolean value 'TRUE' if there is any text-based data accessed from database; 'FALSE' otherwise
Other-database	Boolean value 'TRUE' if there is any data except text-based data accessed from database; 'FALSE' otherwise
Session	The number of nodes which access data from persistent data objects
Uninit	The number of nodes which reference un-initialized program variable
SQL	The number of SQL sink nodes
HTML	The number of HTML sink nodes
SQLI-sanitization	The number of nodes that apply language-provided sanitization functions for preventing SQLI issues
XSS-sanitization	The number of nodes that apply language-provided sanitization functions for preventing XSS issues
Encoding	The number of nodes that encode data
Encryption	The number of nodes that encrypt data
Replacement	The number of nodes that perform string-based substring replacement
Regex-replacement	The number of nodes that perform regular-expression-based substring replacement
Numeric-conversion	The number of nodes that convert data into a numerical format
Un-taint	The number of nodes that return predefined information or information not influenced by external users
Propagate	The number of nodes that propagate the tainted-ness of an input string
Custom	The number of user-defined functions
Other	The number of nodes that are not classified as any of the above types
Vulnerable?	Target attribute which indicates a class label—Vulnerable or Non-Vulnerable

Table 2
Sample attribute vectors.

HTML sink	Attribute								
	Client	File	Other-database	Uninit	SQLI-sanitization	Encoding	Replacement	Custom	Vulnerable?
1	0	0	0	1	0	0	0	0	Vulnerable
10	1	1	0	0	1	0	2	1	Vulnerable
13	1	0	0	0	1	0	2	1	Non-vulnerable
16	1	0	1	0	1	0	0	0	Vulnerable
17	1	0	0	0	0	1	0	0	Non-vulnerable
18	0	0	0	0	0	0	0	0	Non-vulnerable

significant change in performance. Holdout test design is important for assessing the classifiers' capability to predict unknown vulnerabilities. As shown in Fig. 2, our procedure is immune to order effects due to randomization [8] and it also conforms to holdout test design because it only uses certain amount of available dataset for training, and best attribute selection is only performed on training data [35], and it sets aside some for testing.

3.2.5. Performance measures

As used by many research works [3,10,16,20,21,40] in software defect prediction, a typical and common confusion matrix [1] can be used to assess the performance of learned predictors. Fig. 3 shows the matrix. The following performance measures can be derived from this matrix:

- Probability of detection (pd) = $tp/(tp + fn)$.
- Probability of false alarm (pf) = $fp/(fp + tn)$.
- Precision (pr) = $tp/(tp + fp)$.
- Accuracy (acc) = $(tp + tn)/(tp + fp + fn + tn)$.

Pd measures how good our prediction model is in finding actual vulnerable sinks. *Pr* measures the actual vulnerable sinks that are correctly predicted in terms of a percentage of total number of sinks predicted as vulnerable. *Pf* is generally used to measure the cost of the model, that is, increasing *pd* or *pr* by tuning the prediction model may, on the other hand, cause more false alarms. In an ideal situation, *pd* should be close to 1 and *pf* should be close to 0. That is, the vulnerability prediction model neither misses actual vulnerabilities nor throws false alarms. *Acc* measures the number of times the model predicted correctly in terms of a percentage of total number of sinks. In this study, we shall assess the performance of our learned predictors based on all these measures.

4. Research hypotheses

In this section, we provide the three hypotheses that we shall formally investigate to evaluate the usefulness and effectiveness of our proposed static code attributes and prediction method presented in Section 3.

4.1. Hypothesis for discriminative power

Our first hypothesis (*H1*) is that each of our proposed static code attributes has discriminative power (the ability to distinguish vulnerable sinks and non-vulnerable sinks) in general. We use Welch *t*-test [6] to prove the statistical difference between the means of the attribute values for vulnerable sinks and non-vulnerable sinks in each test subject. That is, we test the discriminative power of each attribute for each test subject. Since different web applications have different programming characteristics, it is unrealistic to perform Welch *t*-test on all the test subjects together. And we should expect different attributes having discriminative powers for different test subjects.

For example, if a programmer tends to use regular expressions to sanitize user inputs, we should expect the attribute '*Regex-replacement*' to have significant discriminative power for his particular application. Similarly, another programmer may only use language-provided sanitization functions to sanitize user inputs. In that case, we expect the attribute '*SQLI-sanitization*' or '*XSS-sanitization*' to have significant discriminative power.

```
Procedure model_evaluation (dataset, classifier)
  Input: dataset – historical data
  Output: performance – the averaged results over randomized tests
  M = 10
  N = 10
  repeat M times {
    D = Preprocess(dataset)
    D = Randomize(dataset)
    Divide D into N partitions
    for n = 1 to N do {
      D_test = n^th partition of D
      D_train = D - D_test
      best_attr = select best attributes from D_train using GainRatio method
      D'_train = D_train with only best_attr
      D'_test = D_test with only best_attr
      model_all = training(D_train, classifier)
      model_best = training(D'_train, classifier)
      performance_all = performance_all + testing(model_all, D_test)
      performance_best = performance_best + testing(model_best, D'_test)
    }
  }
  performance_all = performance_all / (M*N)
  performance_best = performance_best / (M*N)
  return performance = <performance_all, performance_best>
```

Fig. 2. Pseudocode for model evaluation.

		Actual	
		Vulnerable	Non-Vulnerable
Predicted by classifier	Vulnerable	True positive (*tp*)	False positive (*fp*)
	Non-Vulnerable	False negative (*fn*)	True negative (*tn*)

Fig. 3. A typical confusion matrix.

Welch *t*-test compares one variable (independent attribute) between two groups (vulnerable sinks and non-vulnerable sinks). This test is suitable since the attribute values of vulnerable sinks and non-vulnerable sinks have unequal variances (see Fig. 4 in Section 5). Our null hypothesis is that the means of the attribute values for vulnerable sinks and non-vulnerable sinks are equal. We consider that an attribute has discriminative power for a test subject if the null hypothesis is rejected at 95% confidence level.

For completeness, we also compared the means to determine the correlation direction between the independent attribute and the vulnerability (positively or negatively correlated). However, we should expect the correlation direction to be inconsistent across test subjects. Therefore, in our work, the correlation 'direction' is of no interest. We should only pay attention to the correlation. The reason is explained below.

For example, if the above programmer uses regular expressions correctly, the inputs would be properly sanitized and the sinks which use those inputs would be non-vulnerable. Hence, the correlation direction of the attribute '*Regex-replacement*' with the vulnerability would be "negative". Conversely, the correlation direction would be 'positive' if he uses regular expressions incorrectly resulting in vulnerable sinks.

4.2. Hypothesis for predictability

Our next hypothesis (*H2*) is that the proposed static code attributes can predict vulnerable sinks, that is, the prediction models presented in Section 3 can achieve good prediction accuracy. There is no universal standard for the threshold of good prediction accuracy. In recent software defect prediction [16,20–22,40] and vulnerability prediction [33] studies, the prediction accuracy (*pd* = 70% and *pf* = 25%) has generally been benchmarked. Therefore, to investigate *H2*, we shall compare our result (achieved by our best prediction model) with this benchmark result. As suggested by Demšar [5], we apply Wilcoxon signed-rank test to evaluate the hypothesis. This statistical method is non-parametric and is suitable for pair-wise comparison of prediction models.

4.3. Hypothesis for usefulness

We claimed that our predictors could be an alternative or complementary solution to vulnerability detection approaches. This leads us to our last hypothesis (*H3*): Our proposed prediction approach can complement existing static analysis-based vulnerability detection approaches.

To investigate *H3*, we shall compare our predictors with Jovanovic et al. [13], which is a static analysis-based vulnerability detection approach based on taint tracking. We shall again apply Wilcoxon signed-rank test to compare the vulnerability detection results from the two approaches. Jovanovic et al.'s approach identifies tainted data accessed from various input sources, tracks the propagation of these inputs along program paths, and determines if an input is subjected to any form of input sanitization (defined as adequate by user) before it is used in security-sensitive program statements such as SQL statements and HTML output statements.

5. Evaluation

This section evaluates our vulnerability prediction framework. Section 5.1 describes our prototype tool and the datasets. Section 5.2 presents the results of discriminative power tests for the proposed attributes. Section 5.3 presents the results of our predictors built from all the proposed attributes. Section 5.4 compares our predictors with a taint-based vulnerability prediction approach. Section 5.5 discusses threats to validity of our results.

5.1. Prototype tool and data collection

The data of our proposed attributes are collected from eight open source PHP-based web applications of different sizes ranging

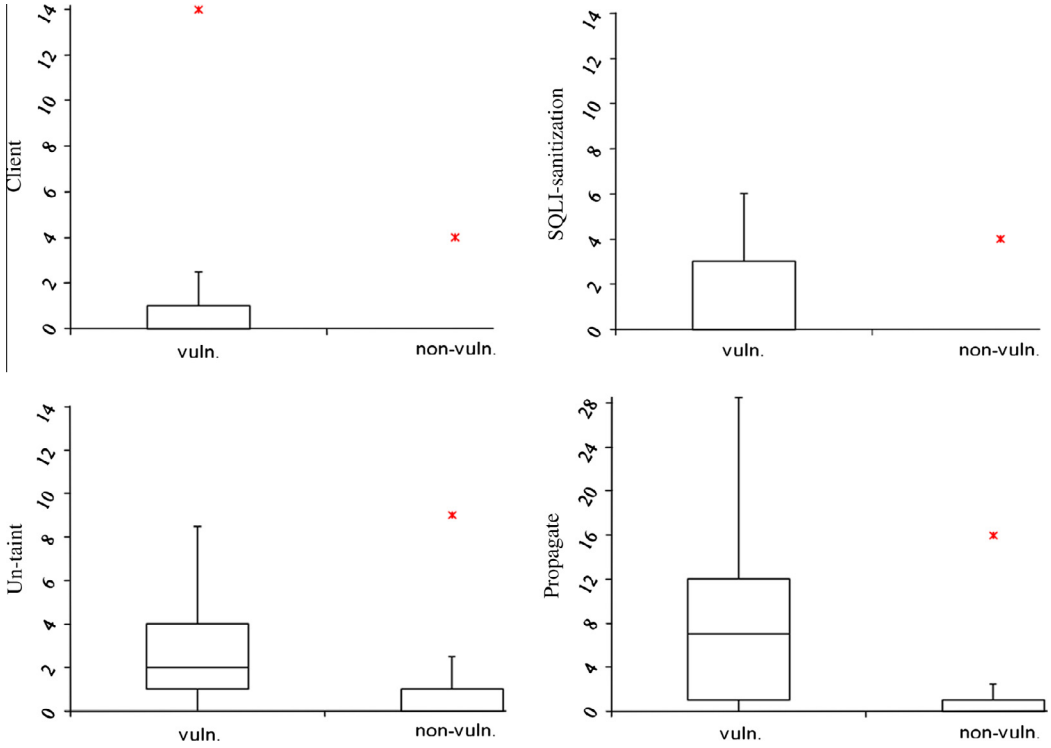

Fig. 4. Comparison of attribute values for vulnerable and non-vulnerable sinks collected from *CuteSite1.2.3*.

from 1 k LOC to 44 k LOC. Table 3 shows the information of these test subjects. It shows the PHP files from which the data of our proposed attributes are collected. It also shows the lines of code (LOC) of each test file (LOC count includes PHP include files). As listed in the last column of Table 3, their vulnerability information is known to public and accessible in various security advisories such as BugTraq [4]. The test subjects are obtained from SourceForge [37]. All these benchmark applications have also been used in evaluating some vulnerability detection approaches [13,15,42,44].

For each sensitive sink in each test PHP file, one attribute vector was collected. Every vector contains the data of our proposed attributes shown in Table 1. For collecting the data from PHP programs, we implemented an automated data collection tool called *PhpMinerI*. In the following, we describe our prototype tool.

PhpMinerI is based on *Pixy* [13], which is a static taint analysis tool for PHP programs. Using *Pixy*'s APIs, *PhpMinerI* generates control flow graph and data dependence graphs of sensitive program points in a given PHP program. It then classifies the nodes in the data dependence graph of each sensitive sink by checking the functions invoked and the operators used. Currently, we have configured over 300 PHP built-in functions and 30 PHP operators in *PhpMinerI* for classification.

Since *Pixy* identifies inputs and sensitive sinks based on the configuration files, in *Pixy*'s configuration, we defined the input sources of *Client*, *File*, *Database*, *Session*, and *Uninit* (un-initialized variables) so that *Pixy* could identify the input types classified by us. However, in order to differentiate the two sub-types of *Database* inputs—*Text* and *Other*, we needed to implement an independent database schema analysis and a data flow analysis of *Database* inputs. Database schema analysis (database schema of the test subject is required) is carried out first to provide *PhpMinerI* with the classifications of *Text* and *Other* database table columns. Thereafter, whenever *PhpMinerI* finds *Database* inputs in the PHP program under test, it uses data flow analysis to verify the column name or the

Table 3
Statistics of the test subjects used in this study.

Test subject	Description	Tested PHP files	LOC	Security advisories
SchoolMate 1.5.4	A tool for school administration	Index	8145	No formal advisory available but its vulnerability information can be found in *groups.csail.mit.edu/pag/ardilla/*
FaqForge 1.3.2	Document creation and management tool	Index, admin\index	790, 1448	Bugtraq-43897
WebChess 0.9.0	Online chess game	Mainmenu, chess	2205, 3236	Bugtraq-43895
Utopia News Pro 1.1.4	News management system	Index, login, postnews, users,	1294, 1174, 1570, 1699	BugTraq-15028
Yapig 0.95b	Image gallery	View	4748	BugTraq-413255
PhpMyAdmin 2.6.0-pl2	A tool for handling MySQL database operations	select_server.lib, left	89, 952	PMASA-2005-01, PMASA-2005-05
PhpMyAdmin 3.4.4	MySQL database management	display_export.lib, db_import, server_synchronize	44628	PMASA-2011-14 – PMASA-2011-20
CuteSite 1.2.3	Content management framework	All the files	11441	CVE-2010-5024 CVE-2010-5025

Table 4
Datasets.

Dataset	#*HTML* sinks	#Vuln. sinks (%Vuln.)	#*SQL* sinks	#Vuln. sinks (%Vuln.)
Schoolmate	172	138 (80%)	189	152 (80%)
Faqforge	115	53 (46%)	42	17 (40%)
Webchess	73	22 (30%)	53	24 (45%)
Cutesite	239	40 (17%)	63	35 (56%)
Utopia	74	17 (23%)	–	–
Yapig	21	6 (29%)	–	–
Myadmin2.6	58	16 (28%)	–	–
Myadmin3.4	305	20 (7%)	–	–

column index of the database table being accessed. It then uses information provided by database schema analysis to further classify the input type. We also configured in *Pixy* the two types of sensitive sinks: *SQL* sink and *HTML* sink. This configurable nature of *Pixy* allows us to *define various types of sinks* that are sensitive to different types of security vulnerabilities; and thus, *PhpMinerI* could support a variety of vulnerability predictors.

Hence, like size and code complexity attributes, our proposed attributes can also be easily collected through an automated tool. As our technique only requires definition/use analysis on CFG, any other program analysis tool could also be used. Table 4 shows the 12 datasets collected by *PhpMinerI* for this study. Eight datasets are collected from data dependence graphs of *HTML* sinks and four datasets are collected from data dependence graphs of *SQL* sinks. For each sink collected, we manually inspected the source code to examine the vulnerability. To ease our workload, we also made use of the data dependence graphs and the information provided by security advisories. We did not use the datasets of *SQL* sinks for some applications as we have not checked the vulnerability regarding to SQLI. Fig. 4 shows the boxplots of comparisons of (non-normalized) values between vulnerable and non-vulnerable sinks for a few selected attributes collected from *CuteSite1.2.3*.

In our website [26], we provide the complete package of collected datasets and the detailed workings of *PhpMinerI*. The tool can also be downloaded at the site. We also use the *Weka* Java package provided by Witten and Frank [43] to implement our model evaluation algorithm in Fig. 2. This implementation is incorporated into *PhpMinerI* as well.

5.2. Discriminative power test results

As discussed in Section 4.1, to evaluate our first hypothesis, *H1*, we analyzed the discriminative power of each proposed static code attribute using *Welch t-test*. Table 5 shows the results. The plus sign indicates positive correlation, that is, vulnerable sinks have higher attribute values than non-vulnerable sinks. As expected (discussed in Section 4.1), different sets of attributes are significant for different applications. Although some attributes appear to have discriminative power for most applications and some do not, a very different set of applications could have produced different results. Considering that an attribute has discriminative power if it is significant for at least one application under test, our results shown in Table 5 support *H1*.

Regarding correlation direction, as we predicted, we observed inconsistent directions for some attributes across applications. The correlation directions were dependent on the nature of the applications and the developers' coding styles. The task of our predictors is to learn on these attributes and the available vulnerability information such that the predictors are able to predict the new instances (possibly from different programs or modules) from the same application.

5.3. Prediction results

Using the APIs provided by *Weka* [43], *PhpMinerI* was configured to run each of our three chosen classifiers on each dataset. The model training and testing was carried out according to the procedure described in Fig. 2. Our tool was run on a Pentium 3.4 GHz 4GBRAM Windows XP machine. Both C4.5 and NB took less than a second to complete each run (i.e., process a dataset) whereas MLP took nearly a minute to complete each run.

Table 6 shows the performance of XSS vulnerability predictors and Table 7 shows the performance of SQLI vulnerability predictors. The average and the standard deviation of each cross-validation process are shown in the tables. Overall results are also averaged. We focus on analyzing the performances of the predictors built from all attributes rather than predictors built from best attributes selected by gain ratio method because we did not observe the performance improvement for MLP and C4.5 after data reduction is applied (overall performance of MLP and C4.5 de-

Table 5
Results of discriminative power test using Welch *t*-test.

Attribute	Test subject							
	Schoolmate	Faqforge	Webchess	Cutesite	Utopia	Yapig	Myadmin2.6	Myadmin3.4
Client	√ (+)*	√ (+)	√ (+)	√ (+)				
File						√ (+)		
Database		√ (+)		√ (+)	√ (+)			
Text-database	√ (−)	√ (−)		√ (+)				
Other-database	√ (+)	√ (+)			√ (+)			
Session				√ (+)				√ (+)
Uninit		√ (+)	√ (−)	√ (+)	√ (+)	√ (+)	√ (+)	
SQL				√ (+)				
HTML				√ (−)				
SQLI-sanitization				√ (+)				
XSS-sanitization	√ (−)				√ (−)			√ (−)
Encoding							√ (+)	√ (+)
Encryption				√ (+)				
Replacement								√ (−)
Regex-replacement				√ (+)				
Numeric-conversion				√ (+)				
Un-taint	√ (+)	√ (+)	√ (−)	√ (+)		√ (+)		
Propagate	√ (+)	√ (+)		√ (+)				√ (+)
Custom				√ (+)				
Other		√ (+)	√ (+)	√ (+)				√ (+)

* The sign √ indicates statistical significance at $p < 0.05$. The sign (+) or (−) indicates the correlation direction between the attribute and the vulnerability.

graded slightly after data reduction although *pd* of NB increased by 3%).

Overall, our vulnerability predictors achieved very good results. MLP was our best predictor based on the average results. For predicting XSS vulnerabilities, on average, MLP achieved *pd* = 78% and *pf* = 6%. For predicting SQLI vulnerabilities, on average, MLP achieved *pd* = 93% and *pf* = 11%. We also observed that SQLI vulnerability predictors generally performed better in terms of *pd* but produced higher false alarm rates than XSS vulnerability predictors did. C4.5's results also closely followed MLP's. Although NB classifier performed the worst among the three classifiers, it still detected 92% of SQLI vulnerabilities and 65% of XSS vulnerabilities. Its relatively high false alarm rates could be due to some redundant attributes or some relations that might exist among the proposed static code attributes. Both of these issues are known to hurt NB classifier's performance because of its strong independence assumptions [16].

Lessmann et al.'s study [16] reported that additional data mining activities such as data reduction could be used to improve simple classifiers like NB. In our experiments with gain ratio attribute

Table 6
Results on XSS vulnerability prediction.

Data and classifier		Measure (%)							
		Mean				Standard deviation			
		Pd	*Pf*	*Pr*	*Acc*	*Pd*	*Pf*	*Pr*	*Acc*
Schoolmate	NB	76	5	99	79	1.2	2.4	0.7	1.2
	C4.5	99	4	99	99	0.0	2.4	0.6	0.4
	MLP	99	2	100	99	0.0	2.4	0.6	0.4
Faqforge	NB	87	14	85	87	2.9	1.1	0.9	1.2
	C4.5	91	2	97	95	2.9	1.8	2.1	1.7
	MLP	90	5	94	92	3.4	1.4	1.6	1.7
Webchess	NB	46	22	48	69	5.3	1.0	2.8	1.4
	C4.5	82	5	87	91	7.0	1.0	2.7	2.4
	MLP	77	7	83	89	7.6	1.8	4.5	3.0
Cutesite	NB	47	1	92	90	4.2	0.5	4.4	0.7
	C4.5	71	6	72	91	1.3	0.9	3.4	0.8
	MLP	84	6	74	92	6.2	0.6	1.5	0.8
Utopia	NB	94	5	84	95	0.0	0.0	0.0	0.0
	C4.5	85	5	84	93	4.7	1.1	3.2	1.5
	MLP	88	5	85	94	5.8	1.4	4.3	1.9
Yapig	NB	45	15	55	74	7.7	4.0	6.1	3.2
	C4.5	35	19	42	68	11.7	2.7	9.5	3.7
	MLP	53	13	61	77	10.0	0.0	4.7	2.9
Myadmin2.6	NB	77	10	76	87	2.8	1.1	2.2	1.1
	C4.5	76	11	73	86	2.5	1.6	2.9	1.3
	MLP	77	10	75	86	2.8	1.4	2.5	0.9
Myadmin3.4	NB	51	1	82	96	2.0	0.2	4.0	0.3
	C4.5	60	1	89	97	3.5	0.2	4.6	0.3
	MLP	55	1	80	96	0.0	0.2	2.4	0.1
Average	NB	65	9	78	85	3	1	3	1
	C4.5	75	7	80	90	4	1	4	2
	MLP	78	6	82	91	4	1	3	1

Table 7
Results on SQLI vulnerability prediction.

Data and classifier		Measure (%)							
		Mean				Standard deviation			
		Pd	*Pf*	*Pr*	*Acc*	*Pd*	*Pf*	*Pr*	*Acc*
Schoolmate	NB	89	31	92	85	1.4	6.8	1.6	1.8
	C4.5	95	24	94	91	1.3	4.4	1.0	1.2
	MLP	90	15	96	89	0.5	4.8	1.2	1.0
Faqforge	NB	97	3	95	97	3.0	1.6	2.3	1.5
	C4.5	88	3	96	94	0.0	1.8	2.8	1.1
	MLP	94	3	96	96	1.8	1.8	2.7	1.4
Webchess	NB	96	23	77	86	1.3	1.4	1.0	0.9
	C4.5	100	24	77	87	0.0	0.0	0.0	0.0
	MLP	100	19	82	90	0.0	1.7	1.4	0.9
Cutesite	NB	84	6	95	88	2.2	2.4	2.0	1.8
	C4.5	94	7	94	94	0.0	0.0	0.0	0.0
	MLP	89	7	94	91	2.8	0.0	0.2	1.6
Average	NB	92	16	90	89	2	3	2	2
	C4.5	94	15	90	92	0	2	1	1
	MLP	93	11	92	92	1	2	1	1

selection, NB improved its *pd* by 3% on average over 12 datasets and hence, this result supports Lessmann et al.'s findings. However, as our focus is to report how useful the proposed input sanitization attributes are for predicting web application vulnerabilities, we did not compare the performance among classifiers using statistical tests such as Friedman test [5].

As discussed in Section 4.2, to evaluate our second hypothesis, *H2*, we compared the results achieved by MLP with a benchmark result of ($pd = 70\%$ and $pf = 25\%$). The test showed that both in terms of *pd* and *pf*, our best predictor achieved statistically better results than the benchmark results (significant at 98%). Its 10 out of 12 *pd* results were better than the benchmark *pd* of 70% and all the *pf* results were better than the benchmark *pf* of 25%.

In terms of accuracy and precision, on average, MLP and C4.5 achieved $acc \geqslant 90\%$ and $pr \geqslant 80\%$. These results can be interpreted as at least 9 out of 10 predictions are correct and at least 8 out of 10 vulnerable cases reported by our predictors are worth investigating for security audits.

Although the results are promising, our predictors are not without flaws. There are cases that our predictors may not appropriately handle. In web applications, it is common that regular-expression-based string replacement routines are used to sanitize inputs. But as observed by Jovanovic et al. [13] and Xie and Aiken [44], these custom sanitization routines are often incorrect. For example, consider the following sanitization code:

```
$sanitize = preg_replace("/\ < script(.*?)\ > (.*?)\
                < \/script(.*?)\ > /i", "",$input);
```

The regular expression looks for the closing script tags (`/script>`) to detect malicious JavaScript code in a user input. However, as some web browsers accept malformed documents, a malicious input such as `<script>hack();<` would still circumvent this sanitization. We did not encounter any such incorrect string replacement routines in our experiments. But if the application under test contains such similar mistakes all over its programs, our predictors would still predict well because, say other attributes are irrelevant, the predictor would classify an instance as vulnerable if there is regular-expression-based-replacement. But when such mistakes are inconsistent, there is a danger of high false alarm rates. This is because, as predictors are trained on past data, they inherently struggle when the instance to be predicted is represented with inconsistent data. Intuitively, this problem can be alleviated by more precise classification methods. For example, we could refine the abstracted classification of regular-expression-based replacement ('*Regex-replacement*' attribute) with more in-depth classifications such as *regular-expression-based allHTML-tags-replacement*, *regular-expression-based OnlyClosingTags-replacement*, etc. However, this task shall require complex pattern matching techniques.

In summary, our Wilcoxon test result supports *H2*, proving that the proposed static code attributes can predict XSS and SQLI vulnerabilities. To use a simple classifier like NB, data reduction could be applied to improve the predictive power. But, based on our results here, we advise to the use of all the proposed attributes as different set of attributes could have discriminative power for different applications (discussed in Section 5.2) and the use of an advanced classifier like MLP which could effectively learn from all available information.

5.4. Comparison with a static analysis-based vulnerability detection approach

One drawback of our prediction method compared to taint-based vulnerability detection methods is that sufficient amount of historical information including vulnerability data is required to train the predictors. We discussed above that our method generally worked for the datasets used in this experiment. And we can observe that the amount of information required to use our predictors is not a lot. As shown in Table 4, the size (number of sinks) of our smallest dataset is twenty-one in which the number of vulnerable sinks is six. This vulnerability data required is reasonable for many vulnerable applications because, once an application is vulnerable due to a weak input sanitization routine, it typically contains a number of vulnerabilities as applications tend to deploy a common sanitization routine to sanitize the sinks.

To evaluate the usefulness of our approach, we investigated our third hypothesis, *H3*. As explained in Section 4.3, we evaluated Jovanovic et al.'s approach [13]. Their vulnerability detection tool, *Pixy*, is open source and available to public. We ran *Pixy* on the same datasets and computed *pd* and *pf* from its detection results. Results are shown in Table 8. We then used Wilcoxon signed-rank test to compare our MLP's results with *Pixy*'s, based on all the 12 datasets.

On average, *Pixy* detected 95% of both XSS and SQLI vulnerabilities whereas our best predictor, MLP, detected 78% of XSS and 93% of SQLI vulnerabilities respectively (Tables 6 and 7). Wilcoxon test also shows that *Pixy*'s *pd* results were significant (at 99%). But, this result is not surprising because, in principal, static analysis-based approach can detect all vulnerabilities by simply reporting any suspicious cases as vulnerable. *Pixy* did not achieve $pd = 100\%$ in our experiments because it lost track of some of the tainted data stored into complex data structures like arrays. It may perform worse in practice in terms of *pd* because, as we explained in Section 4.3, to use *Pixy*, a user has to pre-define adequate input sanitization routines. We were able to define this information properly because of our expert knowledge in XSS and SQLI issues. As our approach does not require such knowledge, it could be alternatively used by developers or auditors who are not security-expert.

On the other hand, it should also be expected that an approach reporting any suspicious cases as vulnerable would generate many false positives. On average, *Pixy* produced high false alarm rates of 30% and 24% in predicting XSS and SQLI vulnerabilities respectively. Wilcoxon test shows that this is significantly high (at 99%) compared to our MLP predictor's achievement (6% and 11% respectively). Noticeably, *Pixy*'s *pf* for *PhpMyadmin3.4* was 86%.

From these results, we can conclude that an overlapped use of a static analysis-based detector and our vulnerability predictor could achieve high detection rates and low false alarms, thus supporting

Table 8
Results of a taint-based vulnerability detection approach (a) on XSS vulnerability detection (b) on SQLI vulnerability detection.

Data	Measure (%)	
	Pd	*Pf*
(a)		
schoolmate	97	4
faqforge	87	13
webchess	95	43
cutesite	100	26
utopia	100	23
yapig	100	25
myadmin2.6	97	21
myadmin3.4	87	86
Average	95	30
(b)		
schoolmate	94	4
faqforge	100	15
webchess	100	47
cutesite	86	30
Average	95	24

H3. And as both methods are static-based, this solution would be easy to use.

5.5. Threats to validity

For every empirical study, it is important to be aware of the potential threats to the validity of the obtained results. In this section, we discuss two types of threats to validity—internal validity and external validity [34], which might affect empirical studies like ours. Threats to internal validity refer to factors that may influence the results without our knowledge. Threats to external validity refer to factors which might prevent the generalization of our results.

5.5.1. Internal validity

Training and testing procedure influences the outcomes of the experiments. We used cross-validation method which is a well-established approach for data mining experiments. *Weka* also provides split-sample set up which is another data sampling method commonly used in many software defect prediction studies [3,16,20,40,45]. We ran our dataset with this setup as well. However, as the differences between the results are minimal, we resorted to using only one data sampling method.

5.5.2. External validity

We discuss issues regarding generalizability across samples and settings. Data sampling bias could affect the generalization of our results. In our experiments, classifiers are trained using data from a set of applications which are all vulnerable. However, these applications have been tested and verified for required functionalities before their releases and it is also hard to find web-based applications that are without SQL and/or XSS vulnerabilities. Furthermore, all applications except *SchoolMate* are real-world projects developed by professionals not by students. As such, we believe that the proposed predictors are practical and would be useful for real-world applications.

In our data preprocessing step, we used min–max normalization approach so as to standardize the distributions of our attributes. Although we did not encounter in our experiment, this approach in practice may throw an "out-of-bounds" error if a future instance to be predicted has a value that falls outside the range of the values in the training instances. A solution is to map values in future instances exceeding the largest value and the smallest value in training instances to the upper limit (one) and the lower limit (zero) respectively.

There are many types of data preprocessing methods. We used a data normalization method. Other methods such as logarithmic data filtering might produce different results. Similarly, there are also many types of data reduction methods. We used gain ratio-based data reduction method, but it was effective only for Naïve Bayes classifier. We might obtain different results if other data reduction methods are applied. The choice of classifiers might also affect our empirical study because classification algorithms we used may over-fit or under-fit the datasets. Therefore, we have used 3 classifiers with different learning models. Interested researchers may try data mining methods different from our current settings or use more data mining activities in order to validate or improve our results.

Our classification schemes may not always be adequate for all cases. For example, functions such as similar_text are classified as *un-taint* based on our observations that such functions do not cause security attacks. But there may be exceptions (which we are not aware of) that some sophisticated attackers make use of the information returned from such functions to generate successful security attacks. Still, the advantage of using data miners is that when such cases become significant, they can be easily re-trained with newly reported vulnerability information to become aware of similar cases in future.

For all the above threats, the best way to prove or refute our results is to replicate the experiments. Researchers could do so since we have clearly defined our experiment methods and we also provide both the dataset used in the experiment and the tool used to collect the data in our web site [26].

6. Related work

This study explores the use of machine learning techniques in finding SQLI and XSS vulnerabilities in web applications. In comparison, our work is related to defect prediction, vulnerability prediction, and vulnerability detection approaches.

6.1. Defect prediction

In this domain, researchers [3,16,20,21,39] have investigated the performance of several classification algorithms, such as C4.5, neural networks, Naïve Bayes, support vector machine, rain forest, and logistic regression, for predicting defects in software modules. Most of these approaches used static code attributes which include LOC counts, Halstead [12], code complexity [19], and other miscellaneous attributes, and applied performance measures such as recall, probability of false alarm, precision, and accuracy.

Although results are encouraging, researchers such as Menzies et al. [23] observed that information contents available from size and code complexity attributes are limited. Zimmermann and Nagappan [45] proposed network attributes that measure dependencies between binaries. Tosun et al. [40] further validated and endorsed their results [45] based on the experiments on public datasets [27]. Arisholm et al. [3] also reported that the use of process attributes (such as developer experience) combining with code complexity attributes could significantly improve the performance of prediction models. However, they also reported that despite the improvement, it is very expensive to collect process measures. Menzies et al. [22] also showed that tuning the classifier according to a user-specific goal (e.g., finding the fewest modules that contain the most errors) improves the classifier's performance without the help of process or other types of attributes. They stated that static code attributes are one of the few measures that can be consistently collected across systems. Tosun and Bener [39] also reported that up to 71% of the software engineering effort could be saved by first identifying possibly-defective software components with data miners before performing manual code audits for debugging and software maintenance purposes.

In summary, their works showed that static code attributes are useful in practice and predictors with $pd > 70\%$ and $pf < 25\%$ could save much software engineering effort. Like these defect prediction approaches, our approach is also built on similar data mining models. Motivated by their findings, we only use static code attributes so that our models are practical. But we use static attributes that characterize input sanitization code patterns.

6.2. Vulnerability prediction

Neuhaus et al. [24] predicted vulnerabilities in software components of Mozilla open source project. Their vulnerability predictors are built from file imports and function calls attributes. Basically, they mine header files and function calls in known vulnerable software components to predict the vulnerability of new components by analyzing their file imports and function calls information. They achieved $pd > 45$ and $pr > 70$.

Gegick et al. [11] used static analysis-based fault alerts, code churn, LOC counts to predict vulnerable software components. Like

our approach, they also used static program analysis to collect attributes. However, their static analysis only generates programming-fault alerts and manual audition is required to determine whether those alerts could be warnings of security vulnerabilities. This task requires a security expert and might be error-prone. By contrast, our attribute collection process is fully automated.

Walden et al. [41] investigated the association between the security resource indicator (SRI) and the vulnerabilities in PHP web applications. SRI is derived from publicly available security information such as documentation of security implications regarding system configurations, past vulnerabilities, and secure development guidelines. In contrast to our work, their method does not focus on locating vulnerable code at any level as their target is in the vulnerability of the application.

Shin et al. [33] used code complexity, code churn, and developer activity attributes to predict vulnerable files in Firefox web browser and Enterprise Linux kernel. They achieved $pd > 80$ and $pf < 25$. Their concept of vulnerability prediction is based on the hypotheses such as the more complex the code, the higher chances that the code is vulnerable. But from our observations, many of the vulnerabilities arise from simple code such

```
$mysql_query(''SELECT * FROM user WHERE id='' ._GET
['user_id']);
```

If a program file does not employ any input sanitization routines, the program code could become simpler but would contain many vulnerabilities. Furthermore, Shin et al. [33] requires process attributes whose measurements may not be consistent across projects [3,22]. By contrast, our predictors are built from only static code attributes.

The major and general differences between the above vulnerability prediction approaches and ours are (1) we focus on SQLI and XSS vulnerabilities instead of general vulnerabilities; (2) hence, we propose and use static code attributes that reflect the patterns of defensive code against SQLI and XSS; and (3) most importantly, we direct auditors to specific program statements rather than software components or program files in reporting vulnerability. Although their results cannot be generalized to ours due to different prediction levels and different benchmarks used, because of our specialization, we achieved significantly low level of false alarm rates in our experiments (6% and 11% false alarm rates on predicting XSS and SQLI vulnerabilities respectively).

In our preliminary evaluation [31], data analysis was performed on all the test subjects together (i.e., cross-validated on all datasets combined). When the test subjects with more data can be easily predicted, this may result in better performance due to over-fitting of data. Such data analysis may not be ideal for heterogeneous set of web applications. Furthermore, previous study only used 5 datasets for experiments. By contrast, this study uses a total of 12 datasets. Data analysis is performed on each individual dataset to avoid over-fitting. Still, our current predictors have displayed good predictive performances. Shar et al. [32] also proposed vulnerability prediction models based on hybrid program analysis. Similar to this work, their work also analyzes input sanitization code patterns. But the major difference is that their work includes dynamic program analysis and focuses on code patterns extracted from dynamic execution traces. They aim to achieve higher accuracy through dynamic analysis whereas the aim of this study is to achieve both good accuracy and usability by using only static analysis.

6.3. Vulnerability detection

These approaches identify the locations of vulnerabilities in program source code using taint analysis methods. Jovanovic et al. [13], Livshits and Lam [17], Xie and Aiken [44] used prominent static analysis techniques such as flow-(in)sensitive, interprocedural, and context-(in)sensitive analyses to implement tainted-information-flow tracking. Their approaches track the flow of user inputs and check if any of input data reaches sensitive sinks without passing through input sanitization routines. Such approaches provide a quick detection of potential vulnerabilities in web programs. But these approaches also tend to produce many false alarms as they do not check the correctness of input sanitization routines. By contrast, our vulnerability predictors learn from available vulnerability information and associated input sanitization code patterns. And, vulnerability predictions on new instances are based on the probabilities that the implemented input sanitization routines are correct or incorrect.

To improve the accuracy, recent approaches incorporated dynamic analysis techniques. Martin and Lam [18] used model checking, and Kieżun et al. [15] and Wassermann et al. [42] used concolic (concrete + symbolic) execution to generate concrete test inputs that are likely to result in genuine security attacks. As their approaches generate concrete attack vectors, there is no false positive. However, the performance of these approaches depends on their underlying model checkers or string constraint solvers such as [14] and [29] because program operations generally involve many string operations which are often complex (e.g., character manipulation, numeric-string interaction). To the best of our knowledge, there is no open source concolic execution tool that could sufficiently handle such string operations. By contrast, our work applies only static analysis methods to collect attribute vectors and uses light-weight modeling methods to predict vulnerabilities. In theory, our static vulnerability predictors may never identify vulnerabilities with the same precision as those dynamic approaches. But adoption of those approaches would require dynamic analysis frameworks which may be computationally expensive. By contrast, our approach could easily be used. And being a data mining-based approach, it also has the advantage of being able to process many data instances, and thus, it provides an alternative, cheaper, and efficient mode of finding many vulnerabilities.

Thomas et al. [38] and Shar and Tan [30] proposed approaches that automatically remove SQLI and XSS vulnerabilities respectively by inserting secured code in place of identified vulnerabilities. However, their approaches do not obviate the need of our approach because their approaches rely on static analysis-based vulnerability detectors to identify vulnerabilities. Therefore, our work complements their approaches.

7. Conclusion

The goal of this work is to aid web security testing by providing vulnerability prediction models that are easy to use and accurate. We proposed a set of static code attributes that characterize input sanitization code patterns and analyzed if these attributes can indicate program statements that are vulnerable to SQLI or XSS. We showed that the attributes can be easily collected via simple static analysis techniques.

In the experiments across 8 test subjects, each of the proposed attributes showed discriminative power between vulnerable and non-vulnerable program statements for at least one test subject. Our best prediction model (MLP) built from all the proposed attributes achieved, on average, ($pd = 93$, $pf = 11$) and ($pd = 78$, $pf = 6$) for predicting SQLI and XSS vulnerabilities respectively. These results show that our proposed prediction method is useful and effective. We do not claim that vulnerability prediction method is the complete replacement of existing vulnerability detection methods because predictors could only provide probabilistic conclusions based on past data. In fact, our experiments with an

existing vulnerability detection approach show that these methods complement each other. But, much work still needs to be done in integrating these different approaches effectively.

As many prediction studies have observed that predictors built with static code attributes do not produce consistent performances across studies, we intend to conduct further experiments on different set of systems and re-evaluate current findings. We also hope that researchers repeat our experiments possibly with more data mining activities and unearth better vulnerability predictors. We aim to identify and propose attributes that reflect defensive code patterns against other types of vulnerabilities such as path traversal and URL redirects to further enhance our predictors. Furthermore, as our current predictors are not designed to predict across applications, we shall also explore the use of other types of attributes such as process attributes to address this problem.

References

[1] E. Alpaydin, Introduction to Machine Learning, MIT Press, Massachusetts, 2004.
[2] Anley, C., 2002. Advanced SQL Injection in SQL Server Applications. Next Generation Security Software Ltd., White Paper.
[3] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, Journal of Systems and Software 83 (1) (2010) 2–17.
[4] BugTraq. <http://www.securityfocus.com/archive/1> (accessed March 2011).
[5] J. Demšar, Statistical comparisons of classifiers over multiple data sets, Journal of Machine Learning Research 7 (2006) 1–30.
[6] M.W. Fagerland, L. Sandvik, Performance of five two-sample location tests for skewed distributions with unequal variances, Contemporary Clinical Trials 30 (5) (2009) 490–496.
[7] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.
[8] D. Fisher, L. Xu, N. Zard, Ordering effects in clustering, in: Proceedings of the 9th International Workshop on Machine Learning, Aberdeen, Scotland, 1992, pp. 163–168.
[9] S. Fogie, J. Grossman, R. Hansen, A. Rager, XSS Exploits: Cross Site Scripting Attacks and Defense, Syngress, 2007, pp. 395–406.
[10] K. Gao, T.M. Khoshgoftaar, H. Wang, N. Seliya, Choosing software metrics for defect prediction: an investigation on feature selection techniques, Software Practice and Experience 41 (5) (2011) 579–606.
[11] M. Gegick, L. Williams, J. Osborne, M. Vouk, Prioritizing software security fortification through code-level metrics, in: Proceedings of the 4th ACM Workshop on Quality of Protection, Alexandria, Virginia, 2008, pp. 31–38.
[12] M. Halstead, Elements of Software Science, Elsevier, New York, 1977.
[13] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: a static analysis tool for detecting web application vulnerabilities, in: Proceedings of the IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, 2006, pp. 258–263.
[14] A. Kieżun, V. Ganesh, P.J. Guo, P. Hooimeijer, M.D. Ernst, HAMPI: a solver for string constraints, in: Proceedings of the 18th International Symposium on Testing and Analysis, Chicago, IL, 2009, pp. 105–116.
[15] A. Kieżun, P.J. Guo, K. Jayaraman, M.D. Ernst, Automatic creation of SQL injection and cross-site scripting attacks, in: Proceedings of the 31st International Conference on Software Engineering, Vancouver, BC, 2009, pp. 199–209.
[16] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496.
[17] V.B. Livshits, M.S. Lam, Finding security errors in Java programs with static analysis, in: Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, 2005, pp. 271–286.
[18] M. Martin, M.S. Lam, Automatic generation of XSS and SQL injection attacks with goal-directed model checking, in: Proceedings of the 17th USENIX Security Symposium, San Jose, CA, 2008, pp. 31–43.
[19] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.
[20] T. Mende, Replication of defect prediction studies: problems, pitfalls and recommendations, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, Timisoara, Romania, 2010, pp. 1–10.
[21] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (1) (2007) 2–13.
[22] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Automated Software Engineering 17 (4) (2010) 375–407.
[23] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, Y. Jiang, Implications of ceiling effects in defect predictors, in: Proceedings of the 30th International Conference on Software Engineering), Leipzig, Germany, 2008, pp. 47–54.
[24] S. Neuhaus, T. Zimmermann, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, Virginia, 2007, pp. 529–540.
[25] OWASP. Top Ten Project 2010. <http://www.owasp.org> (accessed January 2012).
[26] PhpMinerI. <http://sharlwinkhin.com/phpminer.html>.
[27] PROMISE. Software Engineering Repository. <http://promise.site.uottawa.ca/SERepository/> (accessed November 2011).
[28] N.F. Schneidewind, Methodology for validating software metrics, IEEE Transactions on Software Engineering 18 (5) (1992) 410–422.
[29] K. Sen, G. Agha, CUTE and jCUTE: concolic unit testing and explicit path model-checking tools, Lecture Notes in Computer Science 4144 (2006) 419–423.
[30] L.K. Shar, H.B.K. Tan, Automated removal of cross site scripting vulnerabilities in web applications, Information and Software Technology 54 (5) (2012) 467–478.
[31] L.K. Shar, H.B.K. Tan, Mining input sanitization patterns for predicting SQLI and XSS vulnerabilities, in: Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, 2012, pp. 1293–1296.
[32] L.K. Shar, H.B.K. Tan, L.C. Briand, Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis, in: Proceedings of the 35th International Conference on Software Engineering, San Francisco, USA, in press.
[33] Y. Shin, A. Meneely, L. Williams, J.A. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Transactions on Software Engineering 37 (6) (2011) 772–787.
[34] D.I.K. Sjøberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanović, N.-K. Liborg, A.C. Rekdal, A survey of controlled experiments in software engineering, IEEE Transactions on Software Engineering 31 (9) (2005) 733–753.
[35] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect-proneness prediction framework, IEEE Transactions on Software Engineering 37 (3) (2011) 356–370.
[36] Soot. A Java Optimization Framework. <http://www.sable.mcgill.ca/soot/> (accessed October 2012).
[37] Sourceforge. <http://www.sourceforge.net> (accessed March 2011).
[38] S. Thomas, L. Williams, T. Xie, On automated prepared statement generation to remove SQL injection vulnerabilities, Information and Software Technology 51 (3) (2009) 589–598.
[39] A. Tosun, A. Bener, Ai-based software defect predictors: applications and benefits in a case study, in: Proceedings of the 22nd Innovative Applications of Artificial Intelligence Conference, Atlanta, Georgia, 2010.
[40] A. Tosun, B. Turhan, A. Bener, Validation of network measures as indicators of defective modules in software systems, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, Vancouver, BC, 2009, pp. 1–9.
[41] J. Walden, M. Doyle, G.A. Welch, M. Whelan, Security of open source web applications, in: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, 2009, pp. 545–553.
[42] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su, Dynamic test input generation for web applications, in: Proceedings of the International Symposium on Software Testing and Analysis, Seattle, WA, 2008, pp. 249–260.
[43] I.H. Witten, E. Frank, Data Mining, second ed., Morgan Kaufmann, 2005.
[44] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, 2006, pp. 179–192.
[45] T. Zimmermann, N. Nagappan, Predicting defect using network analysis on dependency graphs, in: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 2008, pp. 531–540.