11-2014

Web application vulnerability prediction using hybrid program analysis and machine learning

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

Lionel BRIAND

Hee Beng Kuan TAN

Citation

SHAR, Lwin Khin; BRIAND, Lionel; and TAN, Hee Beng Kuan. Web application vulnerability prediction using hybrid program analysis and machine learning. (2014). *IEEE Transactions on Dependable and Secure Computing*. 12, (6), 688-707.
Available at: https://ink.library.smu.edu.sg/sis_research/4895

Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning

Lwin Khin Shar, *Member, IEEE*, Lionel C. Briand, *Fellow, IEEE*, and Hee Beng Kuan Tan, *Senior Member, IEEE*

Abstract—Due to limited time and resources, web software engineers need support in identifying vulnerable code. A practical approach to predicting vulnerable code would enable them to prioritize security auditing efforts. In this paper, we propose using a set of hybrid (static+dynamic) code attributes that characterize input validation and input sanitization code patterns and are expected to be significant indicators of web application vulnerabilities. Because static and dynamic program analyses complement each other, both techniques are used to extract the proposed attributes in an accurate and scalable way. Current vulnerability prediction techniques rely on the availability of data labeled with vulnerability information for training. For many real world applications, past vulnerability data is often not available or at least not complete. Hence, to address both situations where labeled past data is fully available or not, we apply both supervised and semi-supervised learning when building vulnerability predictors based on hybrid code attributes. Given that semi-supervised learning is entirely unexplored in this domain, we describe how to use this learning scheme effectively for vulnerability prediction. We performed empirical case studies on seven open source projects where we built and evaluated supervised and semi-supervised models. When cross validated with fully available labeled data, the supervised models achieve an average of 77 percent recall and 5 percent probability of false alarm for predicting SQL injection, cross site scripting, remote code execution and file inclusion vulnerabilities. With a low amount of labeled data, when compared to the supervised model, the semi-supervised model showed an average improvement of 24 percent higher recall and 3 percent lower probability of false alarm, thus suggesting semi-supervised learning may be a preferable solution for many real world applications where vulnerability data is missing.

Index Terms—Vulnerability prediction, security measures, input validation and sanitization, program analysis, empirical study

1 Introduction

Web applications play an important role in many of our daily activities such as social networking, email, banking, shopping, registrations, and so on. As web software is also highly accessible, web application vulnerabilities arguably have greater impact than vulnerabilities in other types of software. Web developers are directly responsible for the security of web applications. Unfortunately, they often have limited time to follow up with new arising security issues and are often not provided with adequate security training to become aware of state-of-the-art web security techniques.

According to OWASP's Top 10 Project [1], SQL injection (SQLI), cross site scripting (XSS), remote code execution (RCE), and file inclusion (FI) are among the most common and serious web application vulnerabilities threatening the privacy and security of both clients and applications nowadays. To address these security threats, many web vulnerability detection approaches, such as static taint analysis, dynamic taint analysis, modeling checking, symbolic and concolic testing, have been proposed. Static taint analysis approaches are scalable in general but are ineffective in practice due to high false positive rates [11], [21]. Dynamic taint analysis [11], model checking [22], symbolic [27] and concolic [21] testing techniques can be highly accurate as they are able to generate real attack values, but have scalability issues for large systems due to path explosion problem [30]. There are also scalable vulnerability prediction approaches such as Shin et al. [23]. But the granularity of current prediction approaches is coarse-grained: they identify vulnerabilities at the level of software modules or components. Hence, alternative or complementary vulnerability detection solutions that are scalable, accurate, and fine-grained would be beneficial to web developers.

From the perspective of web developers, input validation and input sanitization are two secure coding techniques that they can adopt to protect their programs from such common vulnerabilities. Input validation typically checks an input against required properties like data length, range, type, and sign. Input sanitization, in general, cleanses an input string by accepting only pre-defined characters and rejecting others, including characters with special meaning to the interpreter under consideration. Intuitively, an application is vulnerable if the developers failed to implement these techniques correctly or to a sufficient degree.

Hence, from the above observation, in this paper, we hypothesize that code attributes that characterize validation and sanitization code implemented in the program could be used to predict web application vulnerabilities. Based on this hypothesis, we propose a set of code attributes called *input validation and sanitization* (IVS) attributes from which

• *L.K. Shar and L.C. Briand are with the Interdisciplinary Centre for ICT Security, Reliability and Trust, University of Luxembourg, 4 rue Alphonse Weicker, L-2721, Luxembourg.*
E-mail: {lwinkhin.shar, lionel.briand}@uni.lu.
• *H.B.K. Tan is with the Department of Information Engineering, School of Electrical & Electronic Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798. E-mail: ibktan@ntu.edu.sg.*

Manuscript received 24 Oct. 2013; revised 24 Aug. 2014; accepted 22 Oct. 2014. Date of publication 19 Nov. 2014; date of current version 13 Nov. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2014.2373377

we build vulnerability predictors that are fine-grained, accurate, and scalable. The approach is *fine-grained* because it identifies vulnerabilities at program statement levels. We use both static and dynamic program analysis techniques to extract IVS attributes. Static analysis can help assess general properties of a program. Yet, dynamic analysis can focus on more specific code characteristics that are complementary to the information obtained with static analysis. We use dynamic analysis only to infer the possible types of input validation and sanitization code, rather than to precisely prove their correctness, and apply machine learning on these inferences for vulnerability prediction. Therefore, we mitigate the scalability issue typically associated with dynamic analysis. Thus, our proposed IVS attributes reflect relevant properties of the implementations of input validation and input sanitization methods in web programs and are expected to help predict vulnerabilities in an accurate and scalable manner. Furthermore, we use both supervised learning and semi-supervised learning methods to build vulnerability predictors from IVS attributes, such that our method can also be used in contexts where there is limited vulnerability data for training.

This work is an extension of our previous work [33], which is a pattern mining approach based on static and dynamic analyses that classify input validation and sanitization functions through the systematic extraction of their security-related properties. The extraction is based on static property inference and analysis of dynamic execution traces. The enhancements and additional contributions of this paper are as follows:

- In our previous work that only targeted SQLI and XSS vulnerabilities, we stated that the proposed method could be adapted to other, similar types of vulnerabilities. In this paper, we address two more, frequent types of vulnerabilities, which are *remote code execution* and *file inclusion* vulnerabilities. Hence, we propose additional attributes to mine the code patterns associated with these new types of vulnerabilities.
- We had only made use of data dependency graphs to identify input validation and sanitization methods. But some of these methods may be identified from control dependency graphs, e.g., input condition checks, which ensure that valid inputs are often implemented through predicates. Therefore, in this work, to better identify those methods, we leverage control dependency information.
- We propose static slicing and dynamic execution techniques that effectively mine both data dependency and control dependency information and describe the techniques in detail.
- We modified our prototype tool, *PhpMiner*, to mine the control dependency information and to extract additional attributes.
- We explore the use of semi-supervised learning schemes. To the best of our knowledge, we are the first to build vulnerability prediction models that way, which makes such models more widely applicable.
- We conducted two sets of experiments on a set of open source PHP applications of various sizes using *PhpMiner*. First, we evaluated supervised learning models built from IVS attributes. Based on cross validation, the model achieves 77 percent recall and 5 percent probability of false alarm, on average over 15 datasets, across SQLI, XSS, RCE, and FI vulnerabilities. From a practical standpoint, the results show that our approach detects many of the above common vulnerabilities at a very small cost (low false alarm rate), which is very promising considering that the existing approaches either report many false warnings or miss many vulnerabilities.
- Second, we compared supervised and semi-supervised learning models with a low sampling rate of 20 percent (i.e., only 20 percent of the available training data are labeled with vulnerability information). On average, the supervised model achieves 47 percent recall and 8 percent probability of false alarm, whereas the semi-supervised model achieves 71 percent recall and 5 percent probability of false alarm. However, when compared to the supervised model based on complete vulnerability data, on average, the semi-supervised model achieves the same probability of false alarm but a 6 percent lower recall. Therefore, our results suggest that when sufficient vulnerability data is available for training, a supervised model should be favored. On the other hand, when the available vulnerability data is limited, a semi-supervised model is probably a better alternative.

The outline of the paper is as follows. Section 2 provides background information. Section 3 presents our classification scheme that characterizes input validation and sanitization methods. Section 4 describes our vulnerability prediction framework. Section 5 evaluates our vulnerability predictors. Section 6 discusses related work. Section 7 concludes our study.

2 Background

This paper targets SQLI, XSS, RCE, and FI vulnerabilities. These security risks, if exploited, could lead to serious issues such as disclosure of confidential, sensitive information, integrity violation, denial of service, loss of commercial confidence and customer trust, and threats to the continuity of business operations. According to CVE [6], 55,504 vulnerabilities were found in web applications within 1999-2013. Among them, 34 percent belong to RCE, 13.2 percent to XSS, 10.3 percent to SQLI, and 3.8 percent to FI. Thus, these four common vulnerabilities are responsible for 61.3 percent of the total number of vulnerabilities found. All these types of vulnerabilities are caused by potential weakness in web applications regarding the way they handle user inputs. They are briefly described using PHP code examples in the following.

2.1 SQL Injection

SQLI vulnerabilities occur when user input is used in database queries without proper checks. It allows attackers to trick the query interpreter into executing unintended commands or accessing unauthorized data. Consider the following code:

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

As the validity of input parameter `$_GET` is not checked, an SQLI attack can be conducted by providing the parameter `id` with the following values:

```
/login.php?id = xxx'+OR+'1'%3D'1
```

The query becomes `SELECT * FROM user WHERE uid = 'xxx' OR '1' = '1'`. Effectively, the attack changes the semantics of the query to `SELECT * FROM user`, which provides the attacker with unauthorized access to the user table. Like `mysql_query`, any other language built-in functions such as `mysql_execute` that interact with the database can cause SQLI.

2.2 Cross Site Scripting

XSS flaws arise when the user input is used in HTML output statements without proper validation or escaping. It allows attackers to execute scripts in the victim's browser, which can hijack user sessions, deface web sites, or redirect the user to malicious sites. Consider the following code:

```
echo Welcome. $_GET['new_user'];
```

Similar to the above SQLI example, as the input parameter `$_GET` is not checked, an XSS attack can be conducted by providing the parameter `new_user` with the following values:

```
<script>alert(document.cookie);</script>
```

When the victim's browser executes the script sent by the server, it shows the new user's cookie values instead of the intended user information. Using a more malicious script, a redirection to the attacker's server is also possible and sensitive user information could be redirected. Like `echo`, any other language constructs or functions such as `print` that generate HTML output could cause XSS.

2.3 Remote Code Execution

RCE vulnerability refers to an attacker's ability to execute arbitrary program code on a target server. It is caused by user inputs in security sensitive functions such as file system calls (e.g., `fwrite`), code execution functions (e.g., `eval`), command execution functions (e.g., `system`), and directory creating functions (e.g., `mkdir`).

It allows a remote attacker to execute arbitrary code in the system with administrator privileges. It is an extremely risky vulnerability, which can expose a web site to different attacks, ranging from malicious deletion of data to web page defacing. The following code depicts an RCE vulnerability.

```
$comments = $_POST['comments'];
$log = fopen('comments.php','a');
fwrite($log,'<br />'.'<br />'.'<center>'.
        'Comments::'.'<br />'.$comments);
```

The above code retrieves user comments and logs them without sanitization. This means that an attacker can execute malicious requests, ranging from simple information gathering using `phpinfo()` to complex attacks that obtain a shell on the vulnerable server using `shell_exec()`. Other sensitive PHP functions and operations associated with this vulnerability type include `header`, `preg_replace()` with "/e" modifier on, `fopen`, `$_GET['func_name']`, `$_GET['argument']`, `vassert`, `create_function`, and `unserialize`.

2.4 File Inclusion

FI vulnerability refers to an attacker's ability to include a file that originates from a remote (possibly an attacker's) server or access/include a local file that is not intended to be accessed without proper authorization. It is caused by user inputs being part of filenames or the use of un-initialized variables in file operations. Consider the following code:

```
include($_GET['file']);
```

An attack may conduct a file inclusion attack using the following values:

```
/include.php?file=http://evil.com/malicious.php
```

This attack causes the vulnerable PHP program to include and execute a malicious PHP file that may cause dangerous program behaviors. Similar PHP commands that may cause FI vulnerability include `include_once` and require.

Moreover, an FI vulnerability may also appear with PHP operations that involve file accesses and file operations in which the attacker may be able to view restricted files, or even execute malicious commands on the web server that can lead to a full compromise of the system. For example, consider the following code:

```
$handle = fopen($_GET['newPath'], "r");
```

In the above case, the input newPath is received from the HTTP GET parameter. An attacker could provide a value like `newPath→"../../../../../etc/passwd%00.txt"` in order to access the password file from the file system. The expression 'dot-dot-slash (../)' instructs the system to go one directory up. The attacker has to guess how many directories he has to go up to find the user confidential folder on the system, but this can be easily done by trial and error. Note that this vulnerability is known as *directory traversal*, but we group this vulnerability together with FI as it can also be seen as a local file inclusion.

3 Classification Scheme

Before presenting our proposed approach, in this section, we first describe the IVS attributes (listed in Table 1) on which vulnerability predictors shall be built. Basically, these attributes characterize various types of program functions and operations that are commonly used (collected from various sources like [1], [17], [18]) as input validation and sanitization procedures to defend against web application vulnerabilities. Using these attributes, functions and operations are classified according to their security-related properties (i.e., the type of validation- and sanitization-effects these functions and operations may enforce on the inputs being processed). For example, the PHP function `str_replace('<', ' ', $input)` removes HTML tags from the input. Since the presence of HTML tags in $input could cause XSS, the function has a security property that filters HTML tags and prevents XSS.

In Table 1, *static analysis-based attributes* are attributes to be extracted using static analysis alone. *Hybrid analysis-based attributes* are attributes to be extracted combining static analysis and dynamic analysis. The term 'filter' in Table 1 indicates a validation or sanitization process that allows only valid strings or that performs character removal, replacement, or escaping. All these attributes are *numeric* (positive integers) and are presented next.

TABLE 1
Input Validation and Sanitization Attributes

ID	Name	Description
Static analysis-based attributes		
1	Client	Input accessed from HTTP request parameters such as HTTP Get
2	File	Input accessed from files such as Cookies, XML
3	Text-database	Text-based input accessed from database
4	Numeric-database	Numeric-based input accessed from database
5	Session	Input accessed from persistent data object such as HTTP Session
6	Uninit	Un-initialized program variable
7	Un-taint	Function that returns predefined information or information not influenced by external users
8	Known-vuln-user	Custom function that has caused security issues in the past
9	Known-vuln-std	Language built-in function that has caused security issues in the past
10	Propagate	Function or operation that propagates partial or complete value of a string
Hybrid analysis-based attributes		
11	Numeric	Function or operation that converts a string into a numeric
12	DB-operator	Function that filters query operators such as (=)
13	DB-comment-delimiter	Function that filters query comment delimiters such as (–)
14	DB-special	Function that filters other database special characters different from the above, such as (\x00) and (\x1a)
15	String-delimiter	Function that filters string delimiters such as (‘) and (“)
16	Lang-comment-delimiter	Function that filters programming language comment delimiter characters such as (/*)
17	Other-delimiter	Function that filters other delimiters different from the above delimiters such as (#)
18	Script-tag	Function that filters dynamic client script tags such as (<script>)
19	HTML-tag	Function that filters static client script tags such as (<div>)
20	Event-handler	Function that disallow the use of inputs as the values of client side event handlers such as (onload =)
21	Null-byte	Function that filters null byte (%00)
22	Dot	Function that filters dot (.)
23	DotDotSlash	Function that filters dot-dot-slash (../) sequences
24	Backslash	Function that filters backslash (\)
25	Slash	Function that filters slash (/)
26	Newline	Function that filters newline (\n)
27	Colon	Function that filters colon (,) or semi-colon (;)
28	Other-special	Function that filters any other special characters different from the above special characters such as parenthesis
29	Encode	Function that encodes a string into a different format
30	Canonicalize	Function that converts a string into its most standard, simplest form
31	Path	Function that filters directory paths or URLs
32	Limit-length	Function or operation that limits a string into a specific length
Dependent attribute		
33	Vuln?	Indicates a class label—Vulnerable or Not-Vulnerable

3.1 Static Analysis-Based Classification

Attributes 1-10 in Table 1 characterize the functions and the program operations to be classified by static analysis only. The first six attributes in Table 1 characterize the classification of user inputs depending on the nature of sources. The reason for including input sources in our classification scheme is that most of the common 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 input sources.

The reason for classifying the inputs into different types is that each class of inputs causes different types of vulnerabilities and different security defense schemes may be required to secure these different classes of inputs. For example, *Client* inputs like `HTTP GET` parameters should always be sanitized before used in sinks whereas it may not be necessary to sanitize *Database* inputs if they have been sanitized prior to their storage (double sanitization might cause security problems depending on the context). *Uninit* variables are variables that are un-initialized at the point of its usage, which could cause security problems (e.g., an attacker could inject malicious values in HTTP parameters having the same name as un-initialized variables by enabling the `register_global` parameter in PHP configuration files). The reason for two types of *Database* inputs—*Text-database* (string-type data) and *Numeric-database* (numeric-type data) is to reflect the fact that string-type data retrieved from data stores can cause second order security attacks such as second order SQLI and stored XSS, while it is difficult to conduct those attacks with numeric-type data.

Un-taint refers to functions or operations that return information not extracted from the input string (e.g., `mysql_num_rows`). It also corresponds to functions or logic operations that return a Boolean value. The reason for this attribute is that since the outcome values are not obtained from an input, the taint information flow stops at those functions and operations and thus, a sink would not be vulnerable from using those values.

Known-vulnerable-user corresponds to a class of custom functions that have caused security issues in the past. *Known-vulnerable-std* characterizes a class of language built-in functions that have caused security issues in the past. For example, according to vulnerability report CVE-2013-3238 [6], `preg_replace` function with the `"/e"` modifier enabled has caused security issues. These functions are to be predefined by users based on their experiences or the information obtained from security databases (we referred to CVE [6] and PHP security [47]).

Clearly in S_k, there would also be functions and operations that do not serve any security purpose. They may simply propagate the input. Consequently, we use the attribute *Propagate* to characterize functions and operations (e.g., substring, concatenation) that do not serve any security purpose and that simply propagate (part of) the input.

Since the above functions and operations either have clear definitions with respect to security requirements or are associated with known vulnerability issues, they could be predefined in a database and classifications can be made statically. This database can be expanded as and when new vulnerability analysis information is available.

3.2 Hybrid Analysis-Based Classification

Attributes 11-32 listed in Table 1 characterize the functions to be classified by either static or dynamic analysis. This hybrid analysis-based classification is applied for validation and sanitization methods implemented using both standard security functions (i.e., language built-in or custom functions with known and tested security properties) and non-standard security functions. If there are only standard security functions to be classified, we classify them based on their security-related information (static analysis); otherwise, we use dynamic analysis.

In a program, various input validation and sanitization processes may be implemented using language built-in functions and/or custom functions. Since inputs to web applications are naturally strings, string replacement/matching functions or string manipulation procedures like escaping are generally used to implement custom input validation and sanitization procedures. A good security function generally consists of a set of string functions that accept safe strings or reject unsafe strings.

These functions are clearly important indicators of vulnerabilities, but we need to analyze the purpose of each validation and sanitization function since 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 parsers is required whereas escaping characters that have special meaning to client script interpreters is needed to prevent XSS vulnerabilities. Thus, it is important to classify these methods implemented in a program path into different types because, together with their associated vulnerability data, our vulnerability predictors can learn this information and then predict future vulnerabilities.

In Table 1, the attribute *Numeric* relates to 1) numeric-type casting built-in functions or operations (e.g., `$a = (double) $b/$c);` 2) language built-in numeric type checking functions (e.g., `is_numeric`); and 3) custom functions that return only numeric, mathematic, and/or dash '-' characters (e.g., functions that validate inputs such as mathematic equation, postal code, or credit card number). When an input to be used in a sink is supposed to be a numeric type, the sink can be made safe from this input through such functions or operations because various alphabetic characters are typically required to conduct security attacks.

DB-operator, *DB-comment-delimiter*, and *DB-special* basically reflect functions that filter sequence of characters that have special meaning to a database query parser. For example, `mysql_real_escape_string` is one such built-in function provided by PHP. Clearly, these attributes could predict SQLI vulnerability.

String-delimiter reflects functions that filter single quote (') and double quote (") characters. *Lang-comment-delimiter* reflects functions that filter comment delimiters such as (/*) that are significant to script interpreters such as JavaScript. *Other-delimiter* reflects functions that filter any other comment delimiters such as (#). All these attributes could be significant vulnerability indicators because they could disrupt the syntax of intended HTML documents, SQL queries, etc.

Script-tag reflects functions that filter sequences of characters, which could invoke dynamic script interpreters such as JavaScript, Flash, and Silverlight. *HTML-tag* reflects functions that filter sequences of *special characters* such as `<body>`, which have special meaning to the static HTML interpreter. Since *Script-tag* and *HTML-tag* filter special characters that may cause XSS, these attributes could predict XSS vulnerability. *Event-handler* reflects functions that disallow the use of inputs as values of event handlers (e.g., `onload`) or other dangerous HTML attributes (e.g., `src`). Inputs used as the values of event handlers can easily cause XSS. For example, consider the following code:

`<img src = '$user_input'>`

If a malicious value, such as `http://hackersite.org/xss.js`, is assigned to `$user_input`, XSS arises. Since the exploit does not necessarily use special characters like `<script`, filtering special characters is insufficient to prevent XSS. Instead, in such cases, only *Event-handler* type functions can safely prevent XSS. Hence, *Event-handler* attribute could predict XSS flaw.

Null-byte, *Dot*, *DotDotSlash*, *Backslash*, *Slash*, *Newline*, *Colon*, and *Other-special* reflect functions that filter different types of *meta-characters*. Filtering *Dot* (.) character is important to handle unintended file extensions or double file extension cases, which may cause file inclusion attacks (see real world example at CVE-2013-3239). *NullByte* `(%00)` characters can be used to bypass sanitization routines and trick underlying systems into interpreting a given value incorrectly. For example, a file value like `script.php%00.txt` can trick a PHP program to see it as a non-malicious text file but the underlying web server or the

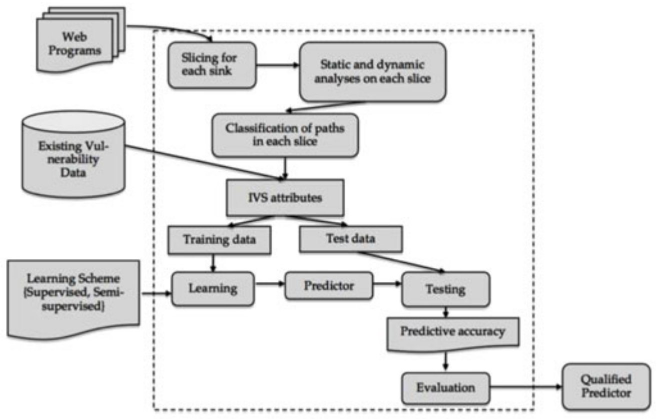
Web Programs
Slicing for each sink
Static and dynamic analyses on each slice
Existing Vulnerability Data
Classification of paths in each slice
IVS attributes
Training data
Test data
Learning Scheme (Supervised, Semi-supervised)
Learning
Predictor
Testing
Predictive accuracy
Evaluation
Qualified Predictor

Fig. 1 Proposed vulnerability prediction framework.

operating system may interpret it as a PHP program. Hence, it can be used to perform many security attacks such as file inclusion and remote code execution.

DotDotSlash include "dot-dot-slash (../)" sequences. These sequences can be used to conduct local file inclusion attacks. *Backslash* (\) is typically used as escape character in most escaping processes and therefore, if an input actually contains this character, it has to be escaped first to avoid confusion for the entire escaping process.

Like the above special characters, *Slash*, *Newline*, *Colon*, and *Other-special* characters could also force an interpreter to misinterpret the input data. *Other-special* characters include characters such as leading and trailing spaces, parenthesis, (|), (%), (_), (^), and ([). For example, the newline character (\n) could break a string into two parts where the second part could become unintended code. The characters (^) and ([) could cause a regular-expression function to misinterpret a regular expression. The character (%) used in an 'SQL-LIKE' clause could cause unintended database-record matches.

Hence, since the above meta-characters could cause unintended program behaviors and security issues, the presence or absence of functions that escape or remove those characters from the inputs could indicate vulnerabilities.

Encode reflects functions that encode an input string into a different format. An input may be properly sanitized using encoding functions. For example, in `<a href = 'login.php?name = '.urlencode($input)>`, the variable `$input` is properly sanitized to be safely included in a sink that generates a URL reference. Inversely, *Canonicalize* reflects functions that transform an input string, which may have more than one possible representation into a standard, normal form so that malicious data disguised in a different, possibly encoded, form can be detected. For example, given a disguised malicious input.`/../../etc/passwd`, PHP's `realpath` function returns the canonicalized path `/etc/passwd` removing symbolic links and extra (/) characters from the input. *Path* reflects functions that filter directory paths or URLs (e.g., `<a href='www.hack.com/hack.js'`). These functions can detect the inclusion of external or illegitimate URLs in sensitive program locations, preventing potential XSS, remote code execution, and file inclusion attacks. *Limit-length* reflects functions that limit the length of an input string. Such functions can limit the possibilities of attacks to a certain extent since the number of malicious characters that can be used is limited.

We believe that the above attributes reflect the types of input validation and sanitization methods that are commonly used to prevent SQLI, XSS, RCE, and FI attacks. We note that our list of attributes may not be exhaustive. Users should refine and update them on a regular basis to reflect latest vulnerability reports. As our vulnerability detection approach is based on machine learning, it is not difficult to re-train vulnerability predictor to learn new vulnerability information.

4 Vulnerability Prediction Framework

Our vulnerability modeling is based on the observations from the analysis of many vulnerability reports in security databases such as CVE [6] and from the study of typical security defense methods. Our vulnerability prediction framework is depicted in Fig. 1. It comprises two main activities:

1) *Hybrid program analysis*. For each sink, a backward static program slice is computed with respect to the sink statement and the variables used in the sinks. Each path in the slice is analyzed using hybrid (static and dynamic) analysis to extract its validation and sanitization effects on those variables. The path is then classified according to its input validation and sanitization effects inferred by the hybrid analysis. Classifications are captured with IVS attributes described in Section 3.
2) *Building vulnerability prediction models*. We then build vulnerability prediction models from those attributes based on supervised or semi-supervised learning schemes and evaluate them using robust accuracy measures.

The details of these activities are described in the following sections.

4.1 Hybrid Program Analysis

4.1.1 Terms and Definitions Used

Our analysis is based on the control flow graph (CFG), the program dependence graph (PDG), and the system dependence graph (SDG) of a web application program. Each node in the graphs represents one source code statement. We may therefore use program statement and node interchangeably depending on the context.

A *sink* is a node in a CFG that uses variables defined from input sources and thus, may be vulnerable to input manipulation attacks. This allows us to predict vulnerabilities at statement levels. *Input nodes* are the nodes at which data from the external environment are accessed. A variable is *tainted* if it is defined from input nodes.

As described earlier, the first step of our approach is to compute a backward static program slice for each sink k and the set of tainted variables used in k. According to the original definition given by Weiser [31], backward static slice S_k with respect to *slicing criterion* $<k, V>$ consists of all nodes (including predicates) in the CFG that may affect the values of V at node k, where V is a subset of variables used in k. We compute S_k using Horwitz et al.'s interprocedural

```
<?php
1  $errMsg = 'userID must be provided!';
2  $id = $_POST['id'];
3  $pwd = $_POST['password'];
4  $name = $_POST['name'];

   //language built-in validation function
5  if(is_numeric($id)) {
      //language built-in sanitization function
6     $pwd = mysql_real_escape_string($pwd);
7     $name = mysql_query("SELECT name FROM user
WHERE id=$id AND pass='$pwd'");//sink
8     $name = 'Welcome '. $name;
   } else {
9     $name = sanitize($name). $errMsg;
   }

10 echo $name;//sink

   //custom sanitization function
11 function sanitize($data) {
12    $data = preg_replace('/[^A-Za-z0-9_.-]/',
                                          '_', $data);
                 . . .
13    return $data;
   }
?>
```

Fig. 2. Sample PHP program with custom and language built-in validation and sanitization functions.

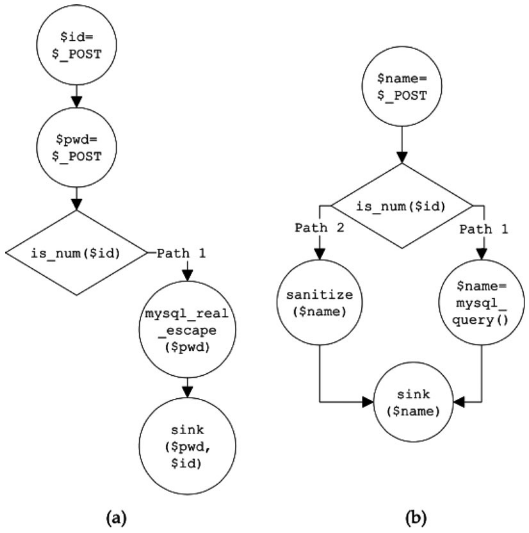

Fig. 3. CFG of a program slice on tainted variables (a) $id and $pwd at sink 7 and (b) $name at sink 10.

slicing algorithm based on the SDG [32]. We first construct the PDG for the main method of a web application program and also construct PDGs for the methods called from the main method according to the algorithm given by Ferrante et al. [8]. We then construct the SDG. A PDG models a program procedure as a graph in which the nodes represent program statements and the edges represent data or control dependences between statements. SDG extends PDG by modeling interprocedural relations between the main program and its subprograms.

To illustrate, Fig. 2 shows an interprocedural slice of the sink at line 10 (denoted as S_{10}) with respect to variable `$name`. Fig. 3a shows the CFG for the slice of the sink at line 7 (denoted as S_7) and Fig. 3b shows the CFG for the slice of S_{10}.

4.1.2 Hybrid Analysis

Typically, a web application program accesses inputs and propagates them via tainted variables for further processing of the application's logics. These processes may often include sensitive program operations such as database updates, HTML outputs, and file accesses. If the program variables propagating the input data are not properly checked before being used in those sinks, vulnerabilities arise. Therefore, to prevent web application vulnerabilities, developers typically employ input validation and input sanitization methods *along the paths* propagating the inputs to the sinks. By default, inputs to web application programs are strings. As such, input validation checks and sanitization operations performed in a program are mainly based on *string operations*. These operations typically include language built-in validation and sanitization functions (e.g., `mysql_real_escape_string`), string replacement and string matching functions (e.g., `str_match`), and regular-expression-based string replacement and matching functions (e.g., `preg_replace`).

Basically, our approach attempts to answer the following research question: "Given a slice of sink, from the types and numbers of inputs, and the types and numbers of input validation and sanitization functions identified from each path in the slice, can we predict the sink's vulnerability?"

Therefore, our objective is to infer the potential effects of validation checks and sanitization operations on tainted variables using static and dynamic analyses, and classify those operations based on these inferences. For every path in S_k that propagates the values of tainted variables into k, we carry out this analysis.

Our hybrid (static and dynamic) analysis includes the techniques proposed by Balzarotti et al. [11]. Basically, for each function in a data flow graph, Balzarotti et al. first analyze 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, then they simulate the effect of the sanitization functions on the input by executing the code with different test inputs, containing various types of attack strings. The execution results are then passed to a test oracle, which evaluates the functions' sanitization effect by checking the presence of those attack strings.

Building on Balzarotti et al.'s work, we model the same information using IVS attributes to enable machine learning and vulnerability prediction. Another difference is that our analysis is performed on program slices rather than data flow graphs. As we discussed earlier, since input validation and sanitization can be performed using predicates, the analysis of data flow graphs may be insufficient. A detailed comparison of our work with Balzarotti et al.'s work is provided in the related work section. In the following, we explain how we made use of Balzarotti et al.'s analysis technique in our context.

Step 1. We first extract all possible paths from S_k. To avoid infinite paths, we use Balzarotti et al.'s solution that is to traverse each loop only once. For example, as shown in Fig. 3, S_7 has only one path and S_{10} has two paths.

Step 2. Each extracted path P_i is classified according to the IVS attributes (Table 1). As described next, classification is performed with compulsory static analysis first followed by optional dynamic analysis.

Compulsory static analysis. We classify each path P_i according to our classification scheme (Section 3) using static analysis first. Standard security functions and some of the language built-in functions/operations can be statically and precisely classified based on their known specific security requirements or their functional properties. We classify such functions and operations into different types according to their security-related properties and store that classification in a database. If a node n in P_i processes a tainted variable that is also used by sink k, we analyze its static properties such as the language parameters and operators used by n, and also the functions with known, specific security purposes that are invoked by n. Then, if there is any match to our predefined classifications, n is classified accordingly.

To illustrate, recall the code snippet in Fig. 2:

```
4 $name = $_POST['name'];
8 $name = 'Welcome '. $name;
```

In statement 4, $_POST is a language parameter which we can predefine as a type of input. In statement 8, the language operator that performs string concatenation (.) is used. As this operation only propagates the values of tainted variables to the next operation, we could classify (.) as a taint propagation type.

Likewise, as shown in Fig. 3, standard security functions can be identified from P_1 of both sink 7 and sink 10. As sink 7 has only one path, it would only require static analysis for the whole classification process. For P_1 of sink 7, we would identify two standard validation and sanitization functions that process `$id`, which is also used in sink 7. These functions are:

```
5 if(is_numeric($id)) {
6   $pwd = mysql_real_escape_string($pwd);
7   $name = mysql_query("SELECT name FROM user
            WHERE id = $id AND pass = '$pwd'");
```

In statement 5, `is_numeric()` is used to validate that `$id` is a numeric. In statement 6, `mysql_real_escape_string` is used to escape MySQL database special characters. As these functions are language built-in validation and sanitization functions, we could classify them statically. We classify them into different validation and sanitization types according to their specifications.

Optional dynamic analysis: If P_i contains non-standard security functions or language built-in functions involving complex string manipulations such as `preg_replace`, the type or purpose of the function cannot be easily inferred using static analysis alone. In this case, we perform dynamic analysis on P_i.

We maintain a database containing different test suites. Test cases are made of various types of attack strings containing malicious characters and benign strings. Attack strings are derived from the security attack vectors provided by OWASP [1] and RSnake [10]. These two security specialists provide a comprehensive coverage of security attack vectors that could bypass various types of input validation and sanitization routines. Each test suite T is designed to test each hybrid analysis-based attribute (discussed in Section 3.2). For example, a test case: `<script>alert();</script>` could discriminate functions that accept or reject JavaScript tags and we would use it to test the attribute *Script-tag*. Our test suite contains such a test case and its variants generated using different combinations of special characters or

```
1  $errMsg='userID must be provided!';
2  $id = '<script>alert();</script>';
3  $pwd = '<script>alert();</script>';
4  $name = '<script>alert();</script>';

5  if(false) {

   } else
9       $name=sanitize($name). $errMsg;

10 echo oracle($name, 'Script-tag'); //sink
```

Fig. 4. Code to be exercised to test for attribute *Script-tag.*

different encoding schemes. We acknowledge that our test suite may be incomplete. However, our test-suite database can be updated and extended as and when new, sophisticated attack vectors are available.

For dynamic execution and analysis, we first extract the code according to the sequence of instructions in P_i and then generate the test code C_i by instrumenting the extracted code. Each input source is replaced with a desired test input. For each test execution, the same test input is used for every input source in P_i. We also handle predicates like statement 5, in which the predicate checks a variable not used in sink 10. We want to ensure that the path under test is exercised until the sink is reached, in the presence of such irrelevant predicate checks. For example, for P_2 of S_{10}, predicate 5 does not validate the variable `$name`. Instead, it validates another variable used in another sink. To ensure that P_2 is exactly followed, the standard solution is to solve a path condition involving the constraint of `$id` and find its appropriate value. But as this solution is not scalable, we simply set the predicate to be false (see Fig. 4). Or we set it to be true if P_1 of S_{10} is to be tested. Note that our solution which forces a predicate to be true or false could cause our classification of P_i to be inaccurate if P_i is an infeasible path. But this is a necessary trade-off between scalability and accuracy.

However, we do not perform the above instrumentation if the predicate validates the variable used in the sink. Instead, we generate a piece of code as an alternative branch of the predicate that P_i follows. The code invokes the test oracle function with an empty string indicating that the validation method successfully found and rejected the invalid test input. An oracle function accepts two arguments. The first argument is the final values at the sink and the second one specifies the type of test suite used. An example of such a case is provided in the following:

```
$id = 'xx OR '1' = '1;
if(is_numeric($id)) {
    oracle("...id = $id...", 'String-delimiter');
} else {
    oracle('','String-delimiter');
    exit;
```

Finally, we instrument the sink such that the final values reached into the sink can be analyzed by a test oracle. The oracle function evaluates whether the malicious values contained in test input variables have been filtered in the final values of the variables. If so, P_i is classified according to the type of test case used.

For each test suite T that is designed to test a hybrid attribute a, we execute the code C_i with a test input t_1 from T

```
function oracle($data, $type) {
  if($type=='Script-tag') {
     if(isFiltered($data, '<script'))
        incrementAttrValue('Script-tag');
     else if …

  } else if($type=='String-delimiter'){
       if(isEscaped($data, "'")
          incrementAttrValue('String-delimiter');
       else if …

  } else if …

  return $data;
}
```

Fig. 5. A test oracle function.

and check if P_i can be classified as a from the execution result. If P_i cannot be classified as a, we choose a different test input t_2 and repeat the process until it is classified as a (i.e., increase the value of a by one) or all the test inputs from T have been used. This whole process is iterated for all test suites, excluding those that are irrelevant to the type of sink. For example, if the sink is a class of HTML outputs such as `echo`, the test suites for attributes such as *DB-operator*, *DB-comment-delimiter*, and *DB-special* (see Section 3.2) are irrelevant.

For our running example in Fig. 2, we have identified that P_1 of S_7 and P_1 of S_{10} require only static analysis for classification. Only P_2 of S_{10} needs to be classified using dynamic analysis. Fig. 4 shows the instrumented code snippet to test P_2 of S_{10}. Fig. 5 shows a sample test oracle function that evaluates if an execution result relates to attribute *Script-tag*. In the example, the oracle verifies if the input string contains the value `<script`. After executing the code in Fig. 4, P_2 of S_{10} would be classified as *Script-tag* since the value `<script` has been filtered from `$name` before being used in the sink.

4.2 Building Vulnerability Prediction Model

Many machine learning techniques can be used to build vulnerability predictors. Regardless of the specific technique used, the goal is to learn and generalize patterns in the data associated with sinks, which can then be efficiently used for predicting vulnerability for new sinks. As more sophisticated security attacks are being discovered, it is important for a vulnerability analysis approach to be able to adapt. With machine learning, it is possible to adapt to new vulnerability patterns via re-training.

4.2.1 Data Representation

Our unit of measurement, an instance in machine learning terminology, is a path in the slice of a sink and we characterize each path with IVS attributes. The attribute values may range from zero to an upper bound that depends on the number of classified program operations or functions. Since we propose 33 IVS attributes (Table 1), each path would be represented by a 33-dimensional attribute vector. To illustrate, Fig. 6 shows the attributes for sink 7 and sink 10 extracted from the paths in their respective slices. The last column is the class attribute to be predicted, that is whether a sink is vulnerable or not in a given path. In our case studies, this comes from existing vulnerability data.

4.2.2 Data Preprocessing

Data balancing. As shown in Table 3, in most of our datasets, the proportion of vulnerable sinks to non-vulnerable ones is small. This is an imbalanced data problem and should be expected in many such vulnerability datasets. Prior studies have shown that imbalanced data can significantly affect the performance of machine learning classifiers [19], [49] because some of the data might go unlearned by the classifier due to their lack of representation, thus leading to induction rules which tend to explain the majority class data and favoring its predictive accuracy. Since for our problem, the minority class data capture the 'vulnerable' instances, we need a high predictive accuracy for this class as missing a vulnerability is far more critical than reporting a false alarm. To address this problem, we use a sampling method called adaptive synthetic oversampling [48]. It balances the (unbalanced) data by generating synthetic, artificial data for the minority class instances, thus reducing the bias introduced by the class imbalance problem. It does not require modification of standard classifiers and thus, can be conveniently added as an additional data preprocessing step [49].

Given an imbalanced data ds with majority class data ds_{maj} and minority class data ds_{min}, the algorithm to generate synthetic data, given by He et al. [48], can be summarized as follows:

1) Compute the total number of instances to be generated for the minority class data: $G = (ds_{maj} - ds_{min}) * \beta$, where $\beta \in (0, 1]$ is the desired balance level after generating synthetic data. We use $\beta = 1$ to achieve a fully balanced dataset.
2) For each instance x_i in ds_{min}, K nearest neighbors are searched in ds based on the Euclidean distance in the attribute space and the ratio γ_i is calculated as: $\gamma_i = K_{maj}/K$ where K_{maj} is the number of instances from K that belong to the majority class. A high ratio value indicates that x_i is mostly surrounded with majority class instances and thus, has a high risk of misclassification.
3) Normalize γ_i according to $\hat{\gamma}_i = \gamma_i / \sum_{i=1}^{ds_{min}} \gamma_i$ so that $\hat{\gamma}_i$ is a density distribution ($\sum \hat{\gamma}_i = 1$).

Attribute / **<*Path*$_{id}$, *Sink*$_{id}$>**	Client	Text-database	Propagate	Numeric	DB-special	String-delimiter	Script-tag	Vuln? (tagged by user)
<P_1, Sink$_7$>	1	0	0	1	1	1	0	No
<P_1, Sink$_{10}$>	0	1	1	0	0	0	0	Yes
<P_2, Sink$_{10}$>	1	0	1	0	0	1	1	No

Fig. 6. Attribute vectors (instances).

4) Compute the number of synthetic instances that need to be generated for each minority instance x_i: $g_i = \widehat{\gamma}_i * G$.
5) Finally, g_i instances for each minority instance x_i are generated using the formula: $x_{new} = x_i + (\hat{x}_i - x_i) * \delta$ where $\hat{x}_i$ is one of the K nearest neighbors of x_i and $\delta \in (0, 1]$ is a random number.

Hence, the idea of adaptive synthetic oversampling is to focus on generating more synthetic data for borderline minority class instances in the attribute space that have a high risk of misclassification, rather than blindly generating new minority class instances to balance the data, which, for some minority class instances, could result in over-fitting while still under-representing the borderline instances. It ensures the adequate representation of minority class data by systematically generating synthetic data where learning is expected to be more difficult.

Attribute selection. Some of the IVS attributes are only relevant for a specific type of vulnerability (for example, *DotDotSlash* is only relevant for detecting FI vulnerability) and some attributes may be correlated. We use an attribute selection technique called correlation-based feature subset selection with a greedy stepwise backward search algorithm [50] to filter the irrelevant or redundant attributes and thus, to reduce the potential negative impact they may have on the learning process. This technique selects the best subset of attributes by performing a greedy backward search through the space of attribute subsets. It starts with a subset of attributes and deletes each attribute one by one. It then evaluates the worth of a subset of attributes by considering the individual predictive ability of each attribute along with the degree of redundancy between them. Subsets of features that are highly correlated with the class while having low inter-correlation are preferred. The algorithm stops when the deletion of any remaining attributes results in a decrease in predictive accuracy.

4.2.3 Supervised Learning

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 RandomForest (RF) models from the proposed attributes. There are two reasons for choosing these two types of classifiers: 1) These classifiers were benchmarked as among the top classifiers in the literature [14], 2) LR-based predictor achieved the best result in our initial work [33] and yields results that are easy to interpret in terms of the impact of attributes on vulnerability predictions.

LR [38] is a type of statistical classification model. It can be used for predicting the outcome (class label) of a dependent attribute based on one or more predictor attributes. The probabilities describing the possible outcomes of a given instance are modeled, as a function of the predictor attributes, using a logistic function:

$$\pi(a_i, \ldots, a_n) = \frac{1}{1 + e^{-A}},$$

where π is a conditional probability: the probability that a sink in a path is vulnerable as a function of the path's security-related properties reflected through predictor attributes. $A(= \beta_0 + \beta_1 a_i + \cdots + \beta_n a_n)$ is a linear combination of n predictor attributes that are statistically significant in terms of their association with the dependent attribute and thus, are selected by the LR modeling process. β_0 is a constant. β_i is the regression coefficient estimated using a maximum likelihood estimation method for attribute a_i.

The curve between π and any attribute a_i, assuming that all other attributes are constant, takes a flexible 'S' shape which ranges between two extreme cases:

a) When a_i is not a significant predictor of vulnerability, then the curve approximates a horizontal line, that is, π does not depend on a_i
b) When a_i strongly indicates vulnerability, then the curve approximates a step function.

As such, logistic regression analysis is flexible in terms of the types of monotonic relationships it can model between the probability of vulnerability and predictor attributes.

RF [37] is an ensemble learning method for classification that consists of a collection of tree-structured classifiers. In many cases the predictive accuracy is greatly enhanced as the final prediction output comes from an ensemble of learners, rather than a single learner. Given an input sample, each tree casts a vote (classification) and the forest outputs the classification having the majority vote from the trees. At an intuitive level, the forest construction procedure is as follows:

1) Select K bootstrap samples from the training set. Bootstrapping, i.e., sampling with replacements, ensures that about one-third of the training set is left out, which can be used as a test set.
2) Fit a classification tree to each bootstrap sample, resulting in K trees. Each tree is grown to the largest extent possible without pruning.
3) Each instance i left out in the construction of the $k\text{th}$ tree is classified by the $k\text{th}$ tree. Due to bootstrapping, i can be classified by about one-third of the trees. Taking c to be the class that got most of the votes across these classifications, the proportion of times that c is not equal to the true class of i averaged over all instances is the so-called *out-of-bag* error estimate. This estimate can be used as an estimate of the generalization error and is used to guide the forest construction process.

4.2.4 Semi-Supervised Learning

As discussed above, for supervised learning, we use LR and RF, the latter being a type of ensemble learning method that has achieved high accuracy in the literature [14]. However, as ensemble learning works by combining individual classifiers, it typically requires significant amounts of labeled data for training. In certain industrial contexts, relevant and labeled data available for learning may be limited.

Semi-supervised methods [39] use, for training, a small amount of labeled data together with a much larger amount of unlabeled data. This method that exploits unlabeled data can enable ensemble learning when there are very few labeled data. As explained by Zhou [43], combining semi-supervised learning with ensembles has many advantages. Unlabeled data is exploited to help enrich labeled training samples allowing ensemble learning: Each individual learner

TABLE 2
Test Subjects

Test Subject	Description	LOC	Security Advisories
SchoolMate 1.5.4	School administration system	8,145	Vulnerability information in [21]
FaqForge 1.3.2	Document creation and management system	2,238	Bugtraq-43897
Utopia News Pro 1.1.4	News management system	5,737	Bugtraq-15027
Phorum 5.2.18	Message board system	12,324	CVE-2008-1486 CVE-2011-4561
CuteSITE 1.2.3	Content management system	11,441	CVE-2010-5024 CVE-2010-5025
PhpMyAdmin 3.4.4	MySQL database management system	44,628	From PMASA-2011-13 to PMASA-2013-4
PhpMyAdmin 3.5.0	MySQL database management system	102,491	From PMASA-2011-13 to PMASA-2013-4

is improved with unlabeled data labeled by the ensemble consisting of all other learners. As listed in Lu et al. [41], a few different types of semi-supervised methods, such as EM-based, clustering-based, and disagreement-based learning, have been proposed in literature. But none of these techniques has been explored for vulnerability prediction so far.

Hence, based on these motivations, we explore the use of an algorithm called CoForest, Co-trained Random Forest, (CF), which applies semi-supervised learning on RF. It is a disagreement-based, semi-supervised learner initially proposed by Li and Zhou [42]. CF uses multiple, diverse learners, and combines them to exploit unlabeled data (semi-supervised learning), and maintains a large disagreement between the learners to promote the learning process.

CF is based on RF and its procedure is as follows:

1) Construct a random forest H with K trees with the available labeled data L.
2) For each tree k in H, repeat the following steps $3 \sim 6$.
3) Construct a new random forest H_{-k} by removing k from H.
4) Use H_{-k} to label all the unlabeled data U and estimate the labeling confidence based on the degree of agreements on the labeling, i.e., the number of classifiers that vote for the label assigned by H_{-k}.
5) Generate a new labeled dataset L' by combining L with the unlabeled data labeled with the confidence levels above a preset confidence threshold.
6) Refine k with L'.
7) Repeat the above steps $2 \sim 6$ until none of the trees in H changes.

For detail information on CF, please refer to [40] and [42].

5 Experimental Evaluation

5.1 Research Questions

This paper aims to investigate the following two research questions:

Question 1 (Q1). Can our proposed IVS attributes, when fed to a machine learner, accurately predict SQLI, XSS, RCE, and FI vulnerabilities?

High accuracy is expected to translate into high recall and low probability of false alarm when predicting vulnerabilities. Although classifiers can be effective, as discussed above, a sufficient number of instances with known vulnerability information is required to train a classifier (supervised learning). As a result, in certain situations, supervised learning is either infeasible or ineffective. In the context of defect prediction, some studies [40], [41] have endorsed the use of semi-supervised learning instead of supervised learning if there are few defects reported. But no performance comparison between semi-supervised learning and supervised learning has yet been investigated in the context of vulnerability prediction. This leads us to our next research question.

Question 2 (Q2). Even if the availability of vulnerability data is limited, can vulnerabilities be predicted using semi-supervised learning? Further, will the performance of a semi-supervised learner be superior to that of a supervised learner when the availability of vulnerability data is limited?

5.2 Experiment Subjects

To evaluate the effectiveness of our vulnerability prediction framework, we perform experiments on seven, real-world PHP web applications, with known vulnerabilities and benchmarked for the evaluation of many vulnerability detection approaches [3], [4], [21], [28]. These applications can be obtained from SourceForge [5]. Table 2 shows relevant statistics for these applications. The vulnerability information can be found in security advisories such as CVE [6]. Securities advisories typically report only vulnerable web pages, which is too coarse-grained for our purpose. And its vulnerability information can typically be traced to multiple vulnerabilities appearing in different program statements. Therefore, we still had to manually inspect the reported vulnerable web pages and analyze the server programs to locate the vulnerable program statements.

For data collection, we enhanced the prototype tool *PhpMiner* used in our previous work [33]. *PhpMiner* basically implements the steps shown in Fig. 1. It is a fully automated data collection tool. Given a PHP program, it generates control flow graphs, program dependence graphs, and system dependence graphs of the program. It then computes backward static program slices of the sinks found in the program, according to the interprocedural slicing algorithm given by Horwitz et al. [32]. Then, it uses a depth-first search strategy to extract the paths in the slices. We also implement the techniques discussed in Section 4 to automate the static and dynamic analysis-based classifications of the paths. For static-based classification, we classify over 330 PHP built-in functions and 30 PHP operators into various input validation and sanitization types and store them in a database. As output, *PhpMiner* produces the attribute vectors like the ones shown in Fig. 6, without the vulnerability labels, which were manually tagged by us for the experiment. Our tool also implements the evaluation procedures (Fig. 7) for supervised and semi-supervised learners. For learning

```
Procedure cross_validate (Dataset ds, Learner C) {
   Prediction model M
   Accuracy acc ← 0
   repeat 10 times {
      B ← Randomly divide ds into 5 equal bins
      for each bin $b_i$ in B {
         $ds_{test}$      ← $b_i$
         $ds_{train}$     ← B - {$b_i$}
         $ds_{train\_bal}$ ← adaptive synthetic
                              oversampling of $ds_{train}$
         bestAttr ← correlation-based feature
                         subset selection from $ds_{train\_bal}$
         M         ← Train C on $ds_{train\_bal}$
                                        and bestAttr
         Test M on $ds_{test}$
         acc       ← acc + predictive accuracy of M
      }
   }
   return acc ← acc / 50
}
```

Fig. 7. Prediction model evaluation procedure.

supervised learners, it relies on the *Weka 3.7* Java package with default options provided by Witten et al. [9]. For learning CoForest, we use the Java package from Li et al. [40].

Table 3 shows the datasets extracted from the test subjects by *PhpMiner*. The dataset name *myadmin1* refers to PhpMyAdmin 3.4.4 and *myadmin2* refers to PhpMyAdmin 3.5.0. The rest is self-explanatory. Because we have the RCE and FI vulnerability data available for only PhpMyAdmin systems, we used two versions of PhpMyAdmin to avoid having only one dataset for these two types of vulnerabilities. As shown in Table 3, we extracted four different sets of datasets, each corresponding to a different type of vulnerabilities. In total, we collected 15 datasets. Table 4 shows descriptive statistics for the values of IVS attributes extracted from those datasets. On our web site [7], we provide the implementation of *PhpMiner* and the datasets.

TABLE 3
Datasets

Dataset	#Instances	#Vuln. instances
(a) Datasets with SQL injection vulnerabilities		
schmate-sqli	189	152
faqforge-sqli	42	17
phorum-sqli	122	5
cutesite-sqli	63	35
(b) Datasets with cross-site scripting vulnerabilities		
schmate-xss	172	138
faqforge-xss	115	53
utopia-xss	86	17
phorum-xss	237	9
cutesite-xss	239	40
myadmin1-xss	305	20
myadmin2-xss	425	14
(c) Datasets with remote code execution vulnerabilities		
myadmin1-rce	221	3
myadmin2-rce	297	5
(d) Datasets with file inclusion vulnerabilities		
myadmin1-fi	139	5
myadmin2-fi	121	2

TABLE 4
Data Distributions of Attributes Across Instances Across Datasets

Attribute	Mean	StdDev	Min	Max
Client	0.43	0.80	0	17
File	0.11	0.21	0	3
Text-database	0.30	0.46	0	11
Numeric-database	0.02	0.08	0	3
Session	0.36	0.69	0	16
Uninit	0.10	0.23	0	5
Un-taint	1.08	1.35	0	30
Known-vuln-user	0.05	0.16	0	10
Known-vuln-std	0.08	0.14	0	3
Propagate	3.23	4.19	0	99
Numeric	0.10	0.32	0	8
DB-operator	0.00	0.01	0	1
DB-comment-delimiter	0.20	0.22	0	8
DB-special	0.20	0.23	0	8
String-delimiter	0.05	0.23	0	6
Lang-comment-delimiter	0.00	0.01	0	1
Other-delimiter	0.00	0.02	0	2
Script-tag	0.03	0.14	0	6
HTML-tag	0.03	0.14	0	6
Event-handler	0.00	0.01	0	2
Null-byte	0.01	0.07	0	5
Dot	0.01	0.05	0	2
DotDotSlash	0.01	0.05	0	2
Backslash	0.00	0.04	0	2
Slash	0.01	0.05	0	4
Newline	0.01	0.04	0	2
Colon	0.00	0.02	0	3
Other-special	0.01	0.06	0	2
Encode	0.02	0.12	0	5
Canonicalize	0.10	0.22	0	4
Path	0.00	0.04	0	2
Limit-length	0.02	0.10	0	2

5.3 Accuracy

We assess the predictive accuracy of our models in terms of probability of detection or recall, probability of false alarm, and precision. We can use the following contingency table to define these standard measures.

		Actual	
		Vulnerable	Non-Vulnerable
Prediction	Vulnerable	True positive (*tp*)	False positive (*fp*)
	Non-Vulnerable	False negative (*fn*)	True negative (*tn*)

Recall ($pd = {}^{tp}/_{(tp+fn)}$) measures how complete our model is in correctly predicting vulnerable sinks. Probability of false alarm ($pf = {}^{fp}/_{(fp+tn)}$) is generally used to measure the cost of using the model. Precision ($pr = {}^{tp}/_{(tp+fp)}$) measures the extent to which vulnerable sinks are correctly predicted. Ideally, the model should neither miss actual vulnerabilities ($pd \sim 1$) nor throw false alarms ($pf \sim 0, pr \sim 1$) As this is, however, difficult to achieve in practice, our aim is to achieve the highest possible recall with a very low probability of false alarm. The model would then be very useful in our context as it would detect many vulnerabilities at a very low cost. We prefer to focus on the

Dataset	Attributes (IDs)
schmate-sqli	1, 2, 4, 5, 7, 10, 11, 13, 14, 15, 18, 29, 30, 32
faqforge-sqli	3, 4, 6, 7, 10, 30
phorum-sqli	1, 7, 10, 11, 13
cutesite-sqli	1, 3, 6, 7, 13
schmate-xss	1, 3, 5, 7, 10, 11, 16, 29, 32
faqforge-xss	6, 7, 10, 30
utopia-xss	1, 2, 3, 4, 5, 6, 7, 10, 13, 15, 18, 32
phorum-xss	1, 3, 5, 6, 7, 10, 11, 15, 30
cutesite-xss	1, 2, 3, 4, 6, 7, 9, 10, 11, 15, 18, 19, 25, 29, 30, 32
myadmin1-xss	1, 5, 6, 7, 10, 18, 30
myadmin2-xss	1, 5, 6, 7, 8, 10, 11, 18, 19, 30
myadmin1-rce	1, 5, 7, 8, 10, 11, 15, 23, 24, 25, 26, 29, 30
myadmin2-rce	1, 2, 9, 10, 15, 21, 23
myadmin1-fi	1, 2, 5, 7, 9, 10, 15, 21, 23, 29
myadmin2-fi	1, 2, 5, 7, 9, 10, 21, 29, 30

Fig. 8 Attributes (IDs) selected in logistic regression models.

probability of false alarm rather than precision because, in security, there are typically few vulnerable sinks in a dataset and thus, even a small number of false alarms could result in low precision, though the model would actually still be useful.

5.4 Supervised Learning Experiments

To investigate our first research question (*Q*1), supervised learning experiments were conducted on the 15 datasets described above. We use two types of supervised prediction models, LR and RF (Section 4.2.3), to evaluate if our proposed IVS attributes can accurately predict SQLI, XSS, RCE, and FI vulnerabilities. Using logistic regression analysis, we also discuss the relative importance of each proposed attribute in vulnerability prediction.

5.4.1 Experimental Design

We evaluate the supervised models using the procedure shown in Fig. 7. Each model is cross validated on each dataset. We follow a fivefold, standard cross validation procedure, repeated ten times (i.e., training and testing 50 times for each model) [9]. As discussed in Section 4.2.2, oversampling is included in the procedure to address the imbalanced data problem. Attributes selection is also included to filter irrelevant, redundant, or correlated attributes. But, to prevent data sampling bias, oversampling and attribute selection is only applied to training instances. Repeating the procedure ten times reduces possible sampling bias due to random splits in cross validation. The randomization also defends against ordering effects [36].

5.4.2 Attribute Relevancy Analysis

One major advantage of using machine learning approaches is that it can select the most informative and significant attributes in such a way as to optimize prediction. It would not be straightforward to assess vulnerability by just inspecting attribute values in the presence of highly complex relationships. It is also expected that different attributes will exhibit widely varying levels of importance in vulnerability prediction. Machine learning algorithms aim at identifying such relationships and assessing the effect of attributes on vulnerability prediction.

ID	Attribute	Frequency
1	Client	450
2	File	182
3	Text-database	140
4	Numeric-database	110
5	Session	173
6	Uninit	258
7	Un-taint	482
8	Known-vuln-user	63
9	Known-vuln-std	168
10	Propagate	508
11	Numeric	91
12	DB-operator	0
13	DB-comment-delimiter	81
14	DB-special	47
15	String-delimiter	159
16	Lang-comment-delimiter	11
17	Other-delimiter	0
18	Script-tag	148
19	HTML-tag	90
20	Event-handler	0
21	Null-byte	70
22	Dot	0
23	DotDotSlash	10
24	Backslash	5
25	Slash	21
26	Newline	18
27	Colon	0
28	Other-special	0
29	Canonicalize	62
30	Encode	173
31	Path	0
32	Limit-length	91

Fig. 9. Frequencies of the attributes selected in logistic regression models.

Logistic regression selects attributes for vulnerability classification based on their statistical significance and uses regression coefficient values to weigh the effect of attributes on vulnerability prediction. For example, the following shows a logistic regression model obtained through maximum likelihood estimation obtained during cross validation on the *phorum_xss* dataset:

$$Vuln? = \frac{1}{1 + e^{-(-5.3+9.6Client+7.7Uninit-37.6Un-taint-40String-delimiter)}}.$$

Such an equation can be informally interpreted as "a path in a sink is highly likely to be vulnerable if it accesses user inputs from input sources of *Client* and *Uninit*, but the odds of being vulnerable decrease significantly if the path also contains *String-delimiter* and *Un-taint* -type input validation and sanitization functions." Quantitatively, the above logistic regression equation can account for non-linear relationships between the probability of vulnerability and the predictor attributes.

In total, 750 LR models (50 cross validations $\times$ 15 datasets) were built. Fig. 8 shows the union of the attributes selected by the LR models during cross validation of each dataset. Fig. 9 shows the frequency with which they were selected during cross validation.

Dataset	Learner	*pd*	*pf*	*pr*
schmate-sqli	LR	91	4	98
	RF	93	4	98
faqforge-sqli	LR	82	21	51
	RF	87	3	89
phorum-sqli	LR	100	2	42
	RF	100	1	46
cutesite-sqli	LR	85	6	95
	RF	89	8	94
Mean results on	**LR**	**90**	**8**	**72**
SQLI prediction	**RF**	**92**	**4**	**82**

Dataset	Learner	*pd*	*pf*	*pr*
schmate-xss	LR	72	19	94
	RF	84	18	95
faqforge-xss	LR	76	21	79
	RF	89	10	90
utopia-xss	LR	67	14	61
	RF	69	21	53
phorum-xss	LR	79	3	53
	RF	69	2	56
cutesite-xss	LR	83	4	78
	RF	76	5	75
myadmin1-xss	LR	77	11	35
	RF	68	4	56
myadmin2-xss	LR	56	5	29
	RF	48	1	62
Mean results on	**LR**	**73**	**11**	**61**
XSS prediction	**RF**	**72**	**9**	**70**

Dataset	Learner	*pd*	*pf*	*pr*
myadmin1-rce	LR	67	1	43
	RF	67	1	53
myadmin2-rce	LR	60	1	47
	RF	60	1	56
Mean results on	**LR**	**64**	**1**	**45**
RCE prediction	**RF**	**64**	**1**	**55**

Dataset	Learner	*pd*	*pf*	*pr*
myadmin1-fi	LR	62	1	79
	RF	52	2	60
myadmin2-fi	LR	100	1	65
	RF	100	0	94
Mean results on	**LR**	**81**	**1**	**72**
FI prediction	**RF**	**76**	**1**	**77**

Fig. 10. Cross validation results of the supervised learners—Logistic Regression (LR) and RandomForest (RF).

First, we can observe, as we expected, that frequencies vary significantly across attributes, thus showing their widely different importance in terms of predicting vulnerability. The most selected attributes include *Client*, *Uninit*, *Un-taint*, and *Propagate*. We note that a few attributes, namely *DB-operator*, *Other-delimiter*, *Event-handler*, *Dot*, *Colon*, *Other-special*, and *Path*, were not selected at all, although those attributes reflect functions that could sanitize potentially dangerous meta-characters like (.) and (,). This is not surprising since, as observed in Table 4, the data distributions of those attributes are sparse, indicating that they are not present in most instances. The attributes like *Dot* and *Path* are not relevant for most datasets since they are designed for detecting FI and RCE vulnerabilities and our experiment only contains two datasets that correspond to each of these vulnerabilities. We also manually checked that some of the rarely-selected attributes are actually present in some of our datasets, but they were not selected by logistic regression as they were found to be not statistically significant. Lastly, the overall key observation is that most of the proposed attributes are selected by different models with varying frequencies, suggesting that the set of proposed attributes reflects the various vulnerability patterns in the selected datasets.

5.4.3 Prediction Results

Fig. 10 shows the predictive accuracy of LR and RF models learnt from IVS attributes, in terms of recall, precision, and probability of false alarm, based on cross validation. On average, RF performed slightly better than LR. Thus, this study allows us to recommend a better supervised learning scheme (RandomForest) than the one used (Logistic Regression) in our previous work [33], for our vulnerability prediction context. We focus our discussion below on predictive accuracy based on RF's results.

Averaging over all 15 datasets, the RF models achieved the result ($pd = 77\%$, $pf = 5\%$), which is better than the result generally benchmarked ($pd > 70\%$, $pf < 25\%$) by many prediction studies [23], [34]. This implies that our prediction approach detects 77 percent of the top four web application vulnerabilities at the cost of filtering a few false positives. Given that, in practice, web application projects typically have many software modules containing many sinks, and undergo many versions over a long lifespan, such models can be very useful in practice to predict vulnerabilities in new versions based on vulnerability data from past versions.

For all the datasets, the RF models achieved low *pf* results. And for most datasets, the RF models also achieved high *pd* results. But we also note that, for a few datasets, the models achieved *pd* results lower than our benchmark *pd* result (*pd* > 70 percent). If we take *myadmin2-xss* as a representative example, our model only achieved $pd = 48\%$, but still, achieving a very low *pf* (1 percent) makes such a model useful in practice. Looking more closely at the numbers, *myadmin2-xss* contains a total of 425 instances, including 14 vulnerable instances (Table 3). Thus, the model catches nearly half of the vulnerabilities at the expense of only four false warnings, which are not costly for developers to filter.

Hence, to answer *Q1*, the supervised prediction models built from IVS attributes can predict SQLI, XSS, RCE, and FI vulnerabilities in most datasets, with a sufficient level of accuracy to be useful. And even in the few cases where the

Dataset	Learner	*pd*	*pf*	*pr*
schmate-sqli	CF	98	5	98
	RF	94	12	94
faqforge-sqli	CF	80	5	81
	RF	67	6	73
phorum-sqli	CF	48	1	32
	RF	26	1	19
cutesite-sqli	CF	89	12	91
	RF	76	17	87
Mean results on	**CF**	**79**	**6**	**76**
SQLI prediction	**RF**	**66**	**9**	**68**

Dataset	Learner	*pd*	*pf*	*pr*
schmate-xss	CF	98	23	95
	RF	79	23	94
faqforge-xss	CF	94	10	91
	RF	73	21	80
utopia-xss	CF	77	8	77
	RF	40	12	56
phorum-xss	CF	68	1	70
	RF	38	3	38
cutesite-xss	CF	88	3	83
	RF	57	10	54
myadmin1-xss	CF	83	1	85
	RF	42	8	34
myadmin2-xss	CF	51	1	63
	RF	23	2	33
Mean results on	**CF**	**80**	**7**	**81**
XSS prediction	**RF**	**50**	**11**	**56**

Dataset	Learner	*pd*	*pf*	*pr*
myadmin1-rce	CF	53	0	70
	RF	7	1	20
myadmin2-rce	CF	62	0	73
	RF	36	1	38
Mean results on	**CF**	**58**	**0**	**72**
RCE prediction	**RF**	**22**	**1**	**29**

Dataset	Learner	*pd*	*pf*	*pr*
myadmin1-fi	CF	58	1	85
	RF	18	2	25
myadmin2-fi	CF	25	0	32
	RF	25	1	27
Mean results on	**CF**	**42**	**1**	**59**
FI prediction	**RF**	**22**	**2**	**26**

Fig. 11. Cross validation results of the semi-supervised learner—CoForest (CF) and the supervised learner—RandomForest (RF) at a sampling rate $\mu = 20\%$.

learnt classifiers cannot effectively detect vulnerabilities, given that our approach consistently achieved low *pf* results for all the datasets, we can confidently claim that detected vulnerabilities always come at an acceptable cost.

5.5 Semi-Supervised Learning Experiments

In this second case study, based on the same 15 datasets, we compare the accuracy of semi-supervised and supervised prediction models (CF and RF) in the presence of low amounts of labeled data. We wish to determine if the semi-supervised model should be preferred for vulnerability prediction when there is limited vulnerability data available (*Q2*).

5.5.1 Experimental Design

The model evaluation procedure is similar to the one used in the above supervised learning experiments (Fig. 7), except that the training data is now split into two—the labeled training set *L* and the unlabeled training set *U*. From the training set, a small percentage (denoted as data sampling rate μ) of training data is randomly sampled as *L*. Like case study 1, adaptive synthetic sampling and correlation-based feature subset selection is then applied to *L* before training. The remaining training data is used as unlabeled training set *U* for the semi-supervised learner. For example, given a dataset containing 100 instances and $\mu = 20\%$, for each trial during fivefold cross validation, the test set contains 20 instances, *L* contains 16 instances (20 percent of available training samples), and *U* contains 64 instances. The supervised learner RF is trained on *L* and tested on *T* whereas the semi-supervised learner CoForest (CF) is trained on *L* and *U*, and tested on *T*.

5.5.2 Prediction Results

We compared the predictive accuracy of RF and CF at different sampling rates and observed that CF clearly outperforms RF with data sampling rates below 40 percent in terms of recall and precision. Here, we discuss the results based on accuracy with a sampling rate $\mu = 20\%$. Fig. 11 shows the results.

On average, over all 15 datasets, even though only a small amount of labeled data is used, the CF model showed good accuracy ($pd = 71\%$, $pf = 5\%$, $pr = 75\%$), thus outperforming the RF model ($pd = 47\%$, $pf = 8\%$, $pr = 51\%$). To test for statistical significance of the difference between CF and RF, as suggested by Demšar [20], we conducted one-tailed Wilcoxon signed-ranks tests on the results. With a significance level equal to 0.01, the tests show that CF performs better than RF in terms of all the accuracy measures we used.

Comparing with the average result ($pd = 77\%$, $pf = 5\%$, $pr = 72\%$) of the RF models trained with fully available labeled data (Section 5.4.3), it is interesting to note that the CF models achieved comparable predictive accuracy. However, as can be observed in Figs. 10 and 11, the semi-supervised learner shows larger variations in accuracy across the 15 datasets than the supervised learner. Across the 15 datasets, the CF models' accuracy (Fig. 11) shows a larger standard deviation of ($pd = 22\%$, $pf = 6\%$, $pr = 20\%$) than the RF models' accuracy (Fig. 10) with a standard deviation of ($pd = 17\%$, $pf = 6\%$, $pr = 19\%$). This implies that supervised learning with sufficient labeled data performs more consistently compared to semi-supervised learning and thus, should be preferred when there is sufficient labeled data available for training. On the other hand, to address *Q2*, when labeled data is rare,

TABLE 5
Runtime Performance of PhpMiner

Test Subject	Static Analysis Time (s)	Dynamic Analysis Time (s)	Average Learning Time (s)	Total time (s)
SchoolMate 1.5.4	8,211	792	99	9,102
FaqForge 1.3.2	6,789	511	84	7,384
Utopia News Pro 1.1.4	7,699	1,250	87	9,036
Phorum 5.2.18	10,592	2,134	132	12,858
CuteSITE 1.2.3	9,205	1,977	105	11,287
PhpMyAdmin 3.4.4	17,549	3,104	141	20,794
PhpMyAdmin 3.5.0	28,700	4,570	160	33,430

semi-supervised learning should be favored to supervised learning.

5.6 Discussion on Data Collection and Model Learning Performance

We showed above that both static analysis- and dynamic analysis-based attributes contribute to achieving sufficient predictive accuracy for the models to be useful in practice, that is, for vulnerabilities to be detected at reasonable cost. Still, it is required that we also analyze the scalability of these analyses. Table 5 shows the runtime performance of *PhpMiner*. Since our hybrid analysis technique is based on the work of Balzarotti et al. [11], the runtime performance of our tool also showed similar results. That is, *PhpMiner* actually spent most of the time on static analysis in extracting slices and their paths. The time spent on running the test suites (dynamic analysis) was considerably less. Although the total time taken was up to a maximum of nine hours, we believe that it is reasonable considering that some of the test subjects are widely-used, real world applications and thus, our performance results suggest that our tool can be applicable in practice. Also, while implementing our tool, performance optimization was not as much a focus as would be expected in an industry strength tool and there is probably significant room for improvement. Average learning time in Table 5 refers to time spent on training and testing a learner with one specific setting. We did not differentiate the time spent on supervised learning and semi-supervised learning because the time difference between these machine learning processes is insignificant. It took a maximum of three minutes for training and testing a learner (including 50 trials for each setting).

5.7 Threats to Validity

Our current work targets PHP web applications because the vulnerabilities we address are very common and serious for PHP applications [51]. However, though this is a practical limitation, it is possible to extend the logic presented in this paper to other programming languages. For example, to adapt our approach to Java, the same classification schemes described in this work could be used. One could predefine Java built-in functions and operations to perform static analysis-based classification. And to perform static and dynamic analyses, there are readily available Java program analysis tools such as Chord [44]. Furthermore, despite these necessary adaptations to other languages, it is important to note that the overall approach would be similar.

Data sampling bias is one of our threats to validity. Our results here may not generalize well to other types of applications in commercial sectors since all our test applications are open source. But it is difficult to conduct experiments on commercial applications since their vulnerability data is not publicly accessible. Also, the implicit assumption of our approach that all the application code is available for analysis is clearly a limitation in some application contexts. Some (especially commercial) applications might use plug-ins or third-party software components, which may be only known at runtime or for which the source code is unavailable. We also consider that a security-sensitive program operation is vulnerable if it uses an input read from an external environment with unknown security controls. Hence, our result would be incorrect if the application is run inside a framework that provides a layer of safeguards that properly validate all the incoming inputs.

Our data only reflect the vulnerability patterns of those that are reported in vulnerability databases. Hence, our vulnerability predictions may not detect vulnerabilities having different characteristics in terms of our proposed attributes. But, considering the wide variability in characteristics of the test subjects (see Table 2), our results should be widely applicable. It is also noteworthy that our underlying hybrid analysis may produce classification errors affecting the prediction results. For example, dynamic analysis may incorrectly flag a function as JavaScript tag filtering function. But since our predictors are learnt on past data, if the same function is causing a number of sinks to be vulnerable, machine learning algorithms learn from it and the presence of such function in the program slices will indicate vulnerabilities.

The use of additional or different machine learning techniques might alter our results. For data balancing, we also tried other sampling techniques like undersampling (remove majority class data) [49], but adaptive synthetic oversampling provided better results. Regarding attribute selection, we also evaluated learners without any attribute selection and with different attribute selection methods such as gain ratio [9]. But correlation-based method provided slightly better results. For supervised learning, we used two very different classification algorithms which are statistical-based and ensemble-based, respectively. We also tried other types of classifiers like multi-layer perceptron and C4.5 that are neural network-based and tree-based, respectively. But RandomForest's results were superior. We have not tried other algorithms for semi-supervised learning. We did not focus our attention on fine-tuning the prediction models and therefore, better results might be obtained.

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

6 Related Work

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

Defect prediction. In defect prediction studies, defect predictors are generally built from static code attributes such as object-oriented design attributes [12], LOC counts, and code complexity attributes [14], [34], [35] because static attributes can be cheaply and consistently collected across many systems [34]. 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 could still be effective if predictors are tuned to user-specific goals.

In many real world applications, defect data is often limited, which makes supervised learning infeasible or ineffective. Li et al. [40] and Lu et al. [41] showed that semi-supervised learning can be used to address this problem and that semi-supervised learners could also perform well in software defect prediction. Li et al. [40] used the CoForest method, which is also used by our work.

The similarity with these defect prediction studies is that our work also uses machine learning techniques in building vulnerability predictors. However, the major difference is that our study targets security vulnerabilities in web applications. Since these studies show that existing set of attributes do not work everywhere, 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 $pd = 80\%$ and $pf = 25\%$. 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 software components similar to known vulnerable ones, in terms of imports and function calls, are likely to be vulnerable as well. They achieved $pd = 45\%$ and $pr = 70\%$.

Yamaguchi et al. [45] and [46] use natural language processing techniques to identify and extract API usage patterns from abstract syntax trees [45] or dependency graphs [46], which are then represented as attributes for machine learning. The numbers of attributes are not bounded. Whereas, we propose 32 code attributes, each of which is specifically designed to reflect a specific type of input validation and sanitization code pattern and thus, is an important indicator of vulnerability. Also, we use program analysis techniques—both static and dynamic analyses to accurately extract those attributes for machine learning.

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

Shar and Tan [2], [16] predicted vulnerabilities using static analysis. Similar to this extension work, they classify the types of validation and sanitization functions implemented for the sinks and reflect those classifications on static code attributes. Although their supervised learners built from static attributes achieved good accuracies, they observed that static analysis could not precisely classify the types of some of the validation and sanitization functions. Later, Shar et al. [33] predicted vulnerabilities using hybrid code attributes. Dynamic analysis was incorporated into static analysis to improve the classification accuracy. Although these earlier works only targeted SQLI and XSS vulnerabilities, they stressed that the work should be extended to address other types of vulnerabilities as well. This work extends the prior ones by addressing two additional types of common vulnerabilities. We propose new attributes and analyze code patterns related to these additional vulnerabilities. More importantly, this work also introduces semi-supervised learning in the domain of vulnerability prediction.

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. Such static taint tracking approaches often generate too many false alarms as these approaches cannot reason about the correctness and the adequacy of those sanity checks. Thus, these approaches are not precise in general.

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 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 path explosion problem [30]. 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. Therefore, these approaches typically suffer from scalability issues.

Our static and dynamic analysis technique builds on Balzarotti et al. [11]. But, similar to the above techniques, Balzarotti et al. apply static and dynamic analysis to determine the correctness of custom sanitization functions identified on data flow graphs, thus leading to scalability issues as well. The difference or the contribution of our work is that we leverage machine learning techniques to mitigate this scalability problem. That is, a predictor can learn correct and incorrect custom functions based on historical data. Though we apply Balzarotti et al.'s static and dynamic analysis technique, we do not do so to precisely compute the correctness of custom functions, but rather to infer their security purposes and apply these inferences in machine learning. As a result, our approach also 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 vulnerability detection accuracy, which may never be matched by machine learning-based methods. Thus, our objective is not to provide a replacement for such techniques but rather to provide a complementary approach to combine with them and to use when they are not applicable. One could, for example, first gather vulnerability predictions on code sections using machine learning and then focus on code sections with predicted vulnerabilities using the more accurate techniques mentioned above. Thereafter, ideally, the confirmed vulnerabilities should be removed by manual audits or by using automated vulnerability removal techniques such as Shar and Tan [29].

7 Discussions and Concluding Remarks

The main goal of this paper is to achieve both high accuracy and good scalability in detecting web application vulnerabilities. In principle, our proposed approach leverages all the advantages provided by existing static and dynamic taint analysis approaches and further enhances accuracy by using prediction models developed with machine learning techniques and based on available vulnerability information. Static analysis is generally sound but tends to generate many false alarms. Dynamic analysis is precise but could miss vulnerabilities as it is difficult or impossible to exercise every test case scenario. Our strategy consisted in building predictors using machine learners trained with the information provided by both static and dynamic analyses and available vulnerability information, in order to achieve good accuracy while meeting scalability requirements.

Our static analysis only involves computing program slices. Dynamic analysis is only used to infer security-checking types of validation and sanitization functions and we use this inferred information for prediction rather than correctness analysis. This approach is scalable since it does not require constraint solving and model checking to reason about correctness as in existing dynamic techniques, e.g., concolic execution. Our analysis is also fine-grained since it identifies vulnerabilities at the program statement level as opposed to the component level, as in existing vulnerability prediction approaches.

In our experiments on seven PHP web applications, we first showed that the proposed IVS attributes can be used to detect several types of vulnerabilities. On average, the RandomForest models, built on IVS attributes, achieved $(pd = 92\%,\ pf = 4\%)$, $(pd = 72\%,\ pf = 9\%)$, $(pd = 64\%,\ pf = 1\%)$, $(pd = 76\%, pf = 1\%)$ when predicting SQL injection, cross site scripting, remote code execution, and file inclusion vulnerabilities, respectively. We also showed that, when a limited number of sinks with known vulnerabilities are available for training the prediction model, semi-supervised learning is a good alternative to supervised learning. We compared RandomForest (supervised) and CoForest (semi-supervised) models with a low data sampling rate of 20 percent, that determine the amount of labeled training data. The CoForest model achieved $(pd = 71\%,\ pf = 5\%)$, on average over 15 datasets, outperforming the RandomForest model that achieved $(pd = 47\%, pf = 8\%)$.

To generalize our current results, our experiment can be easily replicated and extended as we made our tool and data available online [7]. 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.

Acknowledgments

The authors would like to thank Hongyu Zhang [40] for providing us with the Java implementation of CoForest algorithm. This work was partially supported by the National Research Fund, Luxembourg (FNR/P10/03). Lwin Khin Shar is the corresponding author.

References

[1] OWASP. (2012, Jan.). The open web application security project [Online]. Avaialble: http://www.owasp.org
[2] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1767–1780, 2013.
[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. IEEE Symp. Security Privacy*, 2006, pp. 258–263.
[4] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proc. USENIX Security Symp.*, 2006, pp. 179–192.
[5] (2012, Mar.). SourceForge. [Online]. Available: http://www.sourceforge.net
[6] (2013, May). CVE: Distributions of vulnerabilities by types [Online]. Available: http://www.cvedetails.com/vulnerabilities-by-types.php
[7] PhpMiner [Online]. Availble: http://sharlwinkhin.com/phpminer.html, 2013.
[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programm. Languages Syst.*, vol. 9, pp. 319–349, 1987.
[9] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining*, 3rd ed. San Mateo, CA, USA: Morgan Kaufmann, 2011.

[10] (2012, Mar.). RSnake [Online]. Available: http://ha.ckers.org

[11] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proc. IEEE Symp. Security 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," *J. Syst. Softw.*, vol. 51, no. 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," *J. Syst. Softw.*, vol. 83, no. 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 Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 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 Softw. Eng.*, vol. 17, no. 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 *Proc. Int. Conf. Automated Softw. Eng.*, 2012, pp. 310–313.

[17] C. Anley, *Advanced SQL Injection in SQL Server Applications*, Next Generation Security Software Ltd., White Paper, 2002.

[18] S. Palmer, Web application vulnerabilities: Detect, exploit, prevent, Syngress, 2007.

[19] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2007, pp. 196–204.

[20] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learning Res.*, 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 *Proc. Int. Conf. Softw. Eng.*, 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 *Proc. USENIX Security Symp.*, 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 Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.

[24] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 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 *Proc. Int. Symp. Softw. Rel. Eng.*, 2006, pp. 62–74.

[26] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. ACM Conf. Comput. Commun. Security*, 2007, pp. 529–540.

[27] X. Fu and C.-C. Li, "A string constraint solver for detecting web application vulnerability," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, 2010, pp. 535–542.

[28] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2007, pp. 32–41.

[29] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Inf. Softw. Technol.*, vol. 54, no. 5, pp. 467–478, 2012.

[30] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. Int. Conf. Static Anal.*, 2011, pp. 95–111.

[31] M. Weiser, "Program slicing," in *Proc. Int. Conf. Softw. Eng.*, 1981, pp. 439–449.

[32] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Languages Syst.*, vol. 12, no. 1, pp. 26–61, 1990.

[33] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 642–651.

[34] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[35] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, May/Jun. 2011.

[36] D. Fisher, L. Xu, and N. Zard, "Ordering effects in clustering," in *Proc. Int. Workshop Mach. Learning*, 1992, pp. 163–168.

[37] L. Breiman, "Random forests," *Mach. Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[38] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. New York, NY, USA: Wiley, 2013.

[39] O. Chapelle, B. Schölkopf, and A. Zien, Eds., *Semi-Supervised Learning*. Cambridge, MA, USA: MIT Press, 2006.

[40] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Softw. Eng.*, vol. 19, pp. 201–230, 2012.

[41] H. Lu, B. Cukic, and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction," in *Proc. Int. Conf. Automated Softw. Eng.*, 2012, pp. 314–317.

[42] M. Li and Z.-H. Zhou, "Improve computer-aided diagnosis with machine learning techniques using undiagnosed samples," *IEEE Trans. Syst., Man Cyberne., Part A: Syst. Humans*, vol. 37, no. 6, pp. 1088–1098, Nov. 2007.

[43] Z.-H. Zhou, "When semi-supervised learning meets ensemble learning," in *Proc. Int. Workshop Multiple Classifier Syst.*, 2009, pp. 529–538.

[44] Chord: A versatile platform for program analysis. (2011). *Proc. Tutorial ACM Conf. Program. Language Des. Implementation* [Online]. Available: http://pag.gatech.edu/chord

[45] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. Annu. Comput. Security Appl. Conf.*, 2012, pp. 359–368.

[46] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2013, pp. 499–510.

[47] PHP Security [Online]. Available: http://www.php.net/manual/en/security.php, 2013.

[48] H. He, Y. Bai, E. A. Garcia, and S. Li, "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," in *Proc. Int. Joint Conf. Neural Netw.*, 2008, pp. 1322–1328.

[49] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[50] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. thesis, Dept. Comput. Sci., Univ. Waikato, Hamilton, New Zealand, 1998.

[51] PHP Top 5 [Online]. Available: https://www.owasp.org/index.php/PHP_Top_5, 2014.

Lwin Khin Shar received the PhD degree in electrical and electronic engineering from the Nanyang Technological University of Singapore. He is a research associate in software verification and validation at the SnT centre for Security, Reliability, and Trust, University of Luxembourg. His research interests include software security and privacy analysis using program analysis and machine learning techniques. He is a member of the IEEE.

Lionel C. Briand is a full professor and a vice-director of the Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT), University of Luxembourg. He was granted the IEEE Computer Society Harlan Mills award in 2012 for contributions to Model-based Verification and Testing, and elected Reliability Engineer of the year (2013) by the IEEE Reliability Society. His research interests include software testing and verification, model-driven engineering, quality assurance and control, and applications of machine learning and evolutionary computation to software engineering. He is a fellow of the IEEE (2010) and a Canadian professional engineer (P. Eng.) registered in Ontario, Canada.

Hee Beng Kuan Tan received the PhD degree in computer science from the National University of Singapore. He is an associate professor in the Division of Information Engineering in the School of Electrical and Electronic Engineering, Nanyang Technological University. He has 13 years of experience in IT industry before moving to academic. He was also a lecturer in the Department of Information Systems and Computer Science in the National University of Singapore. His research interests include software testing and analysis, software security, and software size estimation. He is a senior member of IEEE and a member of the ACM.