MERLIN: Multi-Language Web Vulnerability Detection

Alexandra Figueiredo
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
alexandra.figueiredo@tecnico.ulisboa.pt

***Abstract*—Although there is continuous research to improve web security, web applications are constantly being attacked due to vulnerable source code. A common way used to find vulnerabilities in code is with source code static analysis tools. However, these tools have two problems: they must be coded manually to deal with all types of vulnerabilities and they only work with a specific programming language. This paper presents an approach that aims to improve security of web applications by identifying vulnerabilities in code written in different languages. Moreover, we do not hard-code the rules of detection, but instead use machine learning to configure them. The approach was implemented in a tool called MERLIN. This tool was tested with samples from the SRD database and real-world web applications written in Java and PHP.**

I. Introduction

Web applications are under constant attack. According to Symantec's monthly threat reports web applications attacks have increased to around 2 million attacks per day in 2019 [1]. Furthermore, the NTT Global Threat Intelligence Report claims that application-specific and web application attacks represented over 32% of all threats in 2019, earning the top category of attack activity [2]. Since web applications are commonly used to access services and resources, vulnerable web applications can have a deep impact on performance and reputation of an organization or a business.

For more than a decade, there has been a great amount of research aimed at improving security of web applications [3]–[20]. Source code static analysis tools that are intended to find security vulnerabilities in web applications are commonly used by software development organizations [5], [7], [12], [17]–[20]. However, these tools have two main problems: they are language-specific, and they have to be programmed, or at least configured manually, to deal with each type of vulnerabilities.

This paper presents a novel approach to detect input validation vulnerabilities in web applications. This approach aims to solve the two problems described earlier by *detecting vulnerabilities in code written in different languages* and by *learning how to detect vulnerabilities*. The approach consists of generating Java bytecode from web applications in various high-level languages. Then, the generated Java bytecode is used to produce code in an intermediate language. Java bytecode is also analyzed to construct *control-flow graphs* (CFGs) of the web application. Thereafter, we combine data flow analysis of the intermediate code with the CFGs to detect potentially vulnerable code. This approach is independent from the programming languages in which the source code is written, since potential vulnerabilities are identified in a common intermediate language.

978-1-7281-8326-8/20/$31.00 ©2020 IEEE

The techniques that are used to automatically detect vulnerabilities in code are taint analysis [4], [7], [12] and machine learning classification [21]. First, potentially vulnerable code is translated into an attribute vector. Then, a machine learning classifier processes the attribute vector extracted from the code and classifies it as vulnerable or not vulnerable. The classifier learns which instructions are associated with the presence of vulnerabilities and allows detecting different types of input validation vulnerabilities.

This paper also describes the implementation of our approach in a tool called MERLIN (Multi-language wEb vulneRabiLity detectIoN). Although MERLIN may support different programming languages, we chose to focus on code written in *PHP and Java* which are languages widely used to develop the *back-end* of web applications. While JavaScript is the most used language for the *front-end*, we do not consider it for evaluation of our tool; input validation in the JavaScript front-end should never be trusted as it can be easily tampered with by a malicious user. Therefore, in this paper we are interested in the back-end only.

During implementation of this tool we faced several challenges. The first challenge was to find an intermediate language suitable to represent different high-level languages. We ended up selecting *Jimple* [22], a three-address intermediate representation of Java bytecode that is easily obtained from Java source code by, for example, using the *javac* compiler. The second challenge was translation from PHP to Java bytecode, since the main goal of the software available for this purpose was not to produce bytecode, but to execute it. Another major challenge was to interpret functions in the intermediate code. Web applications written in languages other than Java do not preserve the original symbols in the intermediate code. Thus, it was necessary to perform an extensive analysis of the intermediate code, so that the tool is able to correctly interpret and process the symbols, and specifically function names, resulted from compilation. Yet another challenge was the need to include configuration files with sensitive sinks, sources and sanitization functions for each programming language considered.

We trained and tested several machine learning classifiers to understand which one fits best for the tool: CART, Random Forest, Naïve Bayes, KNN, Logistic Regression, Multi-Layer

Perceptron and SVM. For learning purposes, we used code samples from the SRD database [23] and code samples written by us to select the classifier. The SRD dataset includes 33,085 code samples in the two languages, of which 27,024 are written in PHP and 6,061 are in Java.

We set up the tool to detect several types of input validation vulnerabilities. So far, the types of vulnerabilities considered are: SQL injection, cross-site scripting, remote file inclusion, local file inclusion, directory/path traversal, source code disclosure, operating system command injection and PHP command injection. This set of vulnerabilities represent high risk vulnerabilities [24]. Therefore, it is important that these vulnerabilities are identified and mitigated.

For evaluation of the tool we used 12 real world web applications, code samples from the SRD database and code samples written by us. In total the tool has processed over 35 thousand files and over one million lines of code. In addition, we compared the results obtained by MERLIN with other tools that aim to detect vulnerabilities. The evaluation shows that MERLIN is capable of processing web applications written in different languages and detecting the proposed vulnerabilities.

The main contributions of this paper are: (1) an approach to improve the security of web applications written in various programming languages by analyzing common intermediate code; (2) a data mining technique that learns how to detect vulnerabilities in code; (3) the implementation of the approach in a tool that detects vulnerabilities in code written in Java and PHP and (4) an experimental evaluation to verify whether the tool is capable of detecting vulnerabilities in real word web applications written in different languages.

II. Background and Related Work

This section presents relevant concepts and related work.

A. Static Analysis for Security

Static analysis tools examine source, binary or intermediate code without executing it. Static analysis can be used to detect security vulnerabilities. These tools have two main advantages when compared with manual auditing: they can be used early in the software development life cycle; and they require lower expertise in security to use.

A technique to perform static analysis is data flow analysis. Data flow analysis examines how the data flows through the code of the program, considering the code semantics. *Taint analysis* is one of the most common techniques used to perform data flow analysis [12], [17], [25]. This technique involves tainting data that enters through an entry point. If tainted data reaches a sensitive sink, it is reported as a vulnerability. However, if the data passes through a sanitization or validation function, it becomes untainted and, in this case, it is not reported as a vulnerability.

Wasserman and Su [12] combined taint analysis and string analysis to detect XSS vulnerabilities. Initially, the tool performs an adapted string analysis to track untrusted substring values. Then, the tool verifies the existence of untrusted scripts by using formal language techniques. Other approach is the one used by RIPS [25] which executes a backward-directed taint analysis capable of detecting 20 different types of vulnerabilities. RIPS first performs intra- and inter-procedural analysis to generate summaries of the data flows. Those summaries are then used to execute efficiently the taint analysis. Our tool uses a different approach. MERLIN performs data flow analysis aiming at identifying all the instructions semantically related in terms of data dependency or control dependency to the sensitive sinks. The tool generates potentially vulnerable code slices regardless of whether the data reaches the sensitive sink tainted or not. Thus, we can conclude that MERLIN does not perform taint analysis.

B. Security Analysis with Machine Learning

Machine learning techniques are used to overcome some limitations of static analysis tools. One of these limitations is that static analysis developers have to code their knowledge, which can have a negative impact on accuracy. There are several approaches to include machine learning in detection of vulnerabilities that vary from how the code is analyzed, which attributes are chosen, the method used to perform attribute extraction to the machine learning algorithms used in the tool.

PhpMinerI [26] uses data mining methods to predict the existence of SQLi and XSS vulnerabilities. The tool extracts a set of attributes from the dependence graph of each sensitive sink. Then, it uses data mining algorithms to classify the code as vulnerable or not vulnerable. WAP [17] is different, since it uses a hybrid approach to detect vulnerabilities. Initially, the tool performs taint analysis to flag candidate vulnerabilities. Then, it uses data mining to predict existence of false positives. Finally, it automatically corrects the code by inserting fixes for the actual flagged vulnerabilities. Vuldeepecker [20] uses deep learning to identify vulnerabilities, instead of machine learning classifiers. The tool uses code gadgets to represent a program. Then, the code gadgets are transformed into an attribute vector. Finally, a neural network classifies the code gadget as vulnerable or not vulnerable. Achilles [27] also uses a deep learning system, namely an array of Long-Short Term Memory Recurrent Neural Network, to detect vulnerabilities within source code. The tool uses several algorithms to preprocess the Java source code. The processed code is then provided as input to a neural network, which in turn generates a n-dimensional vulnerability prediction vector.

MERLIN uses an approach similar to the one used by PhpMinerI. Our tool, like PhpMinerI, uses machine learning classifiers to detect vulnerabilities. However, our approaches to code analysis and attribute extraction are different. Furthermore, MERLIN considers more types of vulnerabilities and analyzes performance of a larger number of machine learning classifiers.

C. Bytecode Analysis

Static analysis can also be done on bytecode, instead of source code. Bytecode is a low-level representation of a program and it is generated by compilation of the source code, e.g., Java bytecode generated from a Java program.

Genaim and Spoto created a context sensitive compositional information flow analysis for Java bytecode [28]. Although the purpose of the analysis is not to detect vulnerabilities, it can be used to verify security of a program. Phosphor [29] is a taint tracking tool for the Java Virtual Machine (JVM) that also analyzes Java bytecode. This tool analyzes the program data flow by assigning labels to the data and propagating them.

MERLIN analyzes intermediate code written in *Jimple* (Java sIMPLE), part of the Soot framework [22]. Jimple is a three-address intermediate representation of Java bytecode. The main advantage of Jimple is that it is more understandable and easier to analyze than Java bytecode. Similarly to source code, Jimple has local variables, instead of manipulating data in a stack like bytecode. Moreover, its statements are always simple, never nested, which greatly simplifies analysis. To analyze Jimple code, Soot receives as input Java source code or Java bytecode. However, it is possible to modify Soot, so it can receive other languages as input. An example of a Soot modification is a tool called Dexpler [30]. Dexpler converts Dalvik bytecode, used for Android applications, into Jimple. MERLIN does not include any modification of Soot, since it always provides Java bytecode as input.

III. Input Validation Vulnerabilities

Missing or incorrect validation of user input is the cause of most web application vulnerabilities. Input validation vulnerabilities are exploited as follows: a potentially malicious input enters the program through an entry point such as *$_GET* in PHP and reaches a sensitive sink where the vulnerability can be exploited, such as *echo()* that allows cross-site scripting. Web applications can be protected by placing sanitization functions between the entry point and the sensitive sink. Sanitization functions will verify the input inserted and if necessary transform it into trusted data by filtering or escaping suspicious characters or constructs. This section describes vulnerability types considered in evaluation.

SQL injection (SQLi) has the highest risk according to OWASP Top 10 2017 [24]. SQLi vulnerabilities are caused by the use of dynamically generated queries that receive unsanitized or incorrectly sanitized input [8]. When executed, these queries can cause unexpected actions on the database and have a great impact on a web application. Figure 1 shows code samples written in PHP and Java vulnerable to SQLi. In these code samples, a malicious input enters through the variable $u/u (entry point) and reaches a sensitive sink on line 3, where it is executed. The attacker can exploit this vulnerability by entering something similar to *' OR 1=1 --* which modifies the query and gives access to all users' passwords. This vulnerability can be mitigated by sanitizing the input using standard language functions, e.g., *mysqli_escape_string()* in PHP or *StringEscapeUtils.escapeSql()* in Java, or by utilizing prepared statements.

Cross-site scripting (XSS) is also among the top 10 vulnerabilities with the highest risk [24]. XSS occurs when unvalidated data from an untrusted source is included into dynamic content of a web page, e.g., as HTML or JavaScript [12].

```
$u = $_GET['user'];
$q = "SELECT pass FROM users where user='".$u."'";
$query = mysqli_query($link, $q);
```

(a) Code sample in PHP

```
String u = request.getParameter('user');
String q = "SELECT pass FROM  users where user='" + u + "'";
ResultSet query = conn.createStatement().executeQuery(q);
```

(b) Code sample in Java

Fig. 1. Examples of vulnerabilities detected by the tool

There are three main XSS classes depending on the source of the data: reflected or non-persistent, stored or persistent, and DOM-based. In case of reflected XSS, data enters from a web request. In case of stored XSS, data enters from the back-end storage. DOM-based XSS occurs when an attacker modifies the DOM in the victim's browser and consequently, causes client side code to run differently than expected.

The other six vulnerabilities supported by MERLIN are presented briefly. Remote File Inclusion (RFI) and Local File Inclusion (LFI) vulnerabilities allow attackers to embed user-supplied files, stored locally or remotely, into a vulnerable web page leading to sensitive information disclosure or arbitrary code execution. Directory traversal or path traversal (DT/PT) vulnerability allows access to files that otherwise would not have been accessible by manipulating the paths using "dot-dot-slash" sequences. The source code disclosure (SCD) vulnerability allows accessing web application source code and configuration files. An operating system command injection (OSCI) allows executing an arbitrary command crafted by an attacker. A PHP command injection vulnerability allows an attacker to supply PHP code that is executed by an eval() statement.

IV. The MERLIN Approach

The proposed approach aims to detect security vulnerabilities in code by analyzing data flow of an intermediate code representation.

Regardless of the programming language, source code is translated into a common intermediate code representation: *Jimple*. Analysis of the intermediate code representation results in a language-independent tool, making it possible to use it for processing web applications written in different languages, such as Java, PHP, JavaScript, and Python. For now, we chose to focus on code written in PHP and Java, which are languages widely used to develop the back-end of web applications.

Our approach does not require explicit coding for each vulnerability. Machine learning classifiers are trained with code samples properly identified as vulnerable or non-vulnerable. With this training, the classifiers learn which categories of instructions are associated with the presence of vulnerabilities.

Our approach includes the following stages, implemented by the modules represented in Figure 3:

1) Conversion to intermediate code: compile source code into Java bytecode; convert Java bytecode into Jimple; generate control-flow graphs (CFGs) for all code.

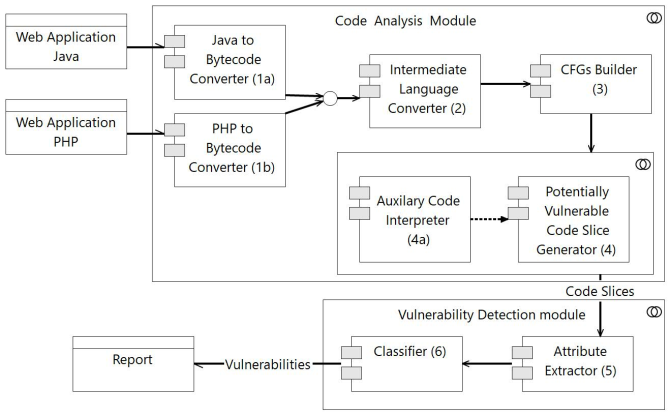
```
$r45 = jphp.runtime.invoke.InvokeHelper.call(r0, $r44,
       "mysqli_query", "mysqli_query", $r42, $r43, 0)
```

(a) Jimple code generated from PHP code

```
$r12 = r15.createStatement()
r5 = $r12.executeQuery(r4)
```

(b) Jimple code generated from Java code

Fig. 2. Translation to Jimple of the 3rd line of the code in Fig. 1



Code Analysis Module
Web Application Java
Web Application PHP
Java to Bytecode Converter (1a)
PHP to Bytecode Converter (1b)
Intermediate Language Converter (2)
CFGs Builder (3)
Auxilary Code Interpreter (4a)
Potentially Vulnerable Code Slice Generator (4)
Code Slices
Vulnerability Detection module
Attribute Extractor (5)
Classifier (6)
Vulnerabilities
Report

Fig. 3. Architecture of the MERLIN tool (for PHP and Java code)

2) Analysis of intermediate code: extract the different control flow paths from the CFGs; analyze each path to search for potentially vulnerable code slices;
3) Vulnerability detection: extract attributes from the code slices; classify them using machine learning algorithms as vulnerable or non-vulnerable.

The following sections present each of these stages.

V. Conversion to intermediate code

First, source code received as input is compiled into Java bytecode. A tool for performing this translation depends on the source language in which the web application is written: *javac* when processing Java code, JPHP [31] when processing PHP code. Since PHP and Java are languages with different characteristics, the translated code for PHP and Java is also different. Java is an object-oriented programming language. Therefore, *javac* produces a file with bytecode for each class. PHP, on the other hand, is a scripting language that supports object-oriented and procedural programming. So, JPHP generates a class and produces a file with bytecode for each class, function and script code. In addition to compiling PHP into Java bytecode, JPHP is also able to execute the resulting bytecode. To correctly interpret and execute the resulting Java bytecode, JPHP modifies the format of some instructions. For instance, assignment instructions are sometimes transformed in a call to the function *jphp.runtime.Memory.assignRight($a,$b)*, where the value of $a is assigned to the variable $b. These changes have to be handled by our tool.

Then, the Soot framework [22] analyzes and converts Java bytecode into Jimple, a typed 3-address intermediate code

```
Instruction tag: SOURCE
Source Name: $_POST("user")
Instruction tag: FUNCTION
Function Name: mysqli_query
```

(a) Simplified instructions generated from PHP code

```
Instruction tag: SOURCE
Source Name: javax.servlet.http.HttpServletRequest.getParameter(
             "user")
Instruction tag: FUNCTION
Function Name: java.sql.Statement.executeQuery
```

(b) Simplified instructions generated from Java code

Fig. 4. Example instructions in simplified representation (from code in Fig. 1)

representation. Each source code instruction is broken down into several separate Jimple instructions; variable names are changed and temporary symbols are generated. A translation example of the third line of code from Figure 1 into Jimple code is shown in Figure 2. Thus, when MERLIN reports a vulnerability, it is specified the class of vulnerability and the file and function/method where the vulnerability is located. However, it is not possible to specify with precision the code line where the vulnerability is found, as that information is lost during intermediate code translation. This is an example that shows that it is not possible to achieve the same precision when analyzing intermediate code that is achieved when analyzing source code. Furthermore, by using intermediate code, the memory complexity increases, since a source code instruction is transformed into a three-address representation, which in turns increases the amount of memory used.

The Soot framework itself also generates control-flow graphs (CFGs) for each method declared within a class defined in the generated Java bytecode.

VI. Analysis of intermediate code

Jimple code and the generated CFGs are analyzed with the objective of identifying potentially vulnerable code slices. MERLIN starts analysis by processing a method that corresponds to the class constructor. Then it proceeds to processing all the remaining methods of the class. The tool repeats this process for all source classes until it has processed all methods in all files.

To correctly process parallel branches in source code, we analyze data flow of the code together with CFGs. The tool extracts CFG subtrees that correspond to control flow paths and process them independently. The tool creates a context for each extracted subtree. A context is a copy of the symbol table that contains variables and assignment instructions. Maintaining separate contexts for different control flow paths ensure that when there are parallel branches independent from each other, instructions of one branch do not interfere with instructions of other parallel branches. If the branches merge, variables processed in each branch are merged as well and stored in a symbol table outside of the contexts.

Another vital part of the tool is correct interpretation of Jimple instructions. MERLIN performs lexical analysis of all instructions. The tool includes a *Tokenizer* that breaks each instruction into a sequence of tokens. Then, the tokens are

TABLE I
EXAMPLES OF ATTRIBUTES

Attribute Category	PHP Examples	Java Examples
Sources	$_GET	getParameter
Sanitization	escapeshellarg	escapeSql
Extract Substring	substr	split
Concatenate String	concat	append
Replace String	str_replace	replace
Remove Whitespace	trim	deleteWhitespace
Type Checking	gettype	instanceof
IsSet Source	isset($source)	isNull
Pattern Control	preg_match	matches
Whitelist	filter_var	isValid
Error	throw	throw
Encoding	utf8_encode	encode
Encryption	crypt	Cipher.doFinal
Numeric conversion	intval	Integer.intValue
Add Char	addslashes	-
SQL Query: Agg Function	AVG	
SQL Query: From clause	FROM	
SQL Query: Numeric Entry	REGEXP	
SQL Query: Complex Query	INNER JOIN	

parsed by MERLIN. Finally, the parsed tokens are converted into a simplified representation, that, in turn, is translated into an attribute. When the tool processes an assignment statement, it also stores the variable name and the simplified representation of the assignment instruction in a symbol table. Examples of the simplified representation of instructions in Figure 1 are presented in Figure 4.

In order to detect potential vulnerabilities, MERLIN needs a configuration file with variables and functions that correspond to entry points, sanitization functions and sensitive sinks for the source language. When the tool finds a sensitive sink, MERLIN collects the potentially vulnerable code slice. The code slice includes instructions in the simplified representation relative to the sensitive sink, all the parameters and all statements semantically related in terms of data dependency or control dependency with those parameters.

MERLIN also supports the processing of a few relevant built-in functions from Java and PHP. For instance, the tool is capable of correctly processing functions that create and manipulate arrays. As a result the tool performs a more complete and accurate analysis. As previously mentioned, JPHP changes the format of some instructions. Therefore, we had to include an auxiliary submodule for interpreting functions and instructions generated by JPHP. This required extensive reverse engineering to understand how JPHP transforms instructions and symbols. Similar submodules may be needed for other programming languages that usually do not compile into Java bytecode, because tools that translate them into Java bytecode may also make adaptations to the instructions, as performed by JPHP.

When MERLIN finds a call to an unrecognized function/method, it checks whether the method/function is defined by a user by checking if the method/function is defined in one of the modules. If it is found, the tool verifies whether the method was previously called. If the method/function was not previously called, the tool analyzes the Jimple code and CFG of the method/function and, builds a summary. A summary stores potentially vulnerable code slices and if applicable code slices associated with the return value of the function. If the function has been previously called, MERLIN processes the existing summary. In both cases, the values of the parameters are propagated as the method/function summary is processed. Thus, MERLIN is capable of performing inter-procedural analysis.

VII. Vulnerability Detection

In addition to standard input validation and sanitization functions (e.g., *mysqli_escape_string()*), there are some other operations that can also untaint data. For instance, adding characters to a string or extracting a substring may untaint a string. However, these operations may in some cases untaint instructions and, in others, they may not. Therefore, this problem is considered to be undecidable and it is known to be related to Turing's Halting problem [32]. As a result, when using static analysis to detect vulnerabilities, these operations cannot be analyzed precisely. The problem is even more complicated when the aim is to detect vulnerabilities in multiple programming languages.

Static analysis tools usually require to explicitly code knowledge for each vulnerability type considered, which can affect accuracy. To eliminate the need for additional coding, MERLIN uses machine learning classifiers to detect existence of vulnerabilities. All potentially vulnerable code slices are transformed into an attribute vector. Then, they are classified as vulnerable or non-vulnerable. The development of this module required a three-stage process:

1) Configuration stage: where we define the set of attributes and the classifier to use;
2) Learning stage: where we train the classifier with a set of vulnerable and non-vulnerable code slices;
3) Classification stage: where we classify the code slices as vulnerable or not; this stage unlike the previous stages which are part of the configuration of the tool, it is performed by the tool.

The following sections explain these stages.

A. Attribute Extraction

Potentially vulnerable code slices identified during the intermediate code analysis are transformed into an attribute vector. Each code slice instruction in the simplified representation is processed separately to check if it matches any attribute. If it matches an attribute, it gets reflected in the attribute vector. The attributes are binary. They can only have two values: 0, which indicates that there is no instruction in the code slice that matches the attribute, and 1, otherwise.

We started with a list of attributes selected for the WAP tool [33]. We extended this list by analyzing code manually and identifying other operations that could taint or untaint data. We could identify seven main sets of attributes that influence the presence of vulnerabilities:

- Sources: represent places where potential malicious input can enter the program. This set of attributes is essential since a vulnerability only exists if the code slice has an entry point.

TABLE II
CLASSIFIERS' RESULTS WITH UNBALANCED DATASET

Classifier	Precision (%)	Recall (%)	F-Score (%)	Acc (%)
CART	81.79	79.23	80.42	91.09
Random Forest	81.14	79.67	80.48	90.99
Naïve Bayes	70.04	87.91	71.95	79.63
KNN	76.70	68.28	71.29	88.47
LR	81.29	76.09	78.34	90.57
MLP	81.87	78.78	80.20	91.07
SVM	81.87	79.18	80.43	91.11

TABLE III
CLASSIFIERS' RESULTS WITH BALANCED DATASET

Classifier	Precision (%)	Recall (%)	F-Score (%)	Acc (%)
CART	85.79	98.59	91.74	91.13
Random Forest	85.76	98.63	91.75	91.13
Naïve Bayes	80.75	99.29	89.06	87.80
KNN	83.03	94.41	88.36	89.60
LR	84.73	98.75	91.20	90.47
MLP	85.78	98.33	91.63	91.01
SVM	85.74	98.68	91.76	91.14

- Sanitization functions: represent functions that are able to transform untrusted data into trusted data by filtering or escaping characters. This set of attributes is also important, because these functions can mitigate a vulnerability.
- String manipulation: represent functions that manipulate strings. We consider functions that extract substrings, concatenate and replace strings, add a character and remove spaces. These functions can untaint strings depending on how they are used and the vulnerability considered.
- Validation: represent functions and operations that validate data. In this category we consider attributes that verify data types, check if the value is set, or if it matches a pattern, belongs to a white-list or an error function.
- SQL query manipulation: these attributes are only relevant when the tool is dealing with SQL injection. The tool checks if an SQL query contains: data inserted in the SQL aggregate function, a FROM clause, a complex SQL query and a test to verify if the data is numeric.
- String conversion: represent functions that transform a string into another data format. In this category, we consider functions that encode a string, return numeric values and hash or encrypt a string in order to ensure secure data transfer. When the format is converted, the input may no longer pose a threat to a web application.
- Others: this category includes functions capable of untainting data by conditioning data flow (using ifs), by using operators that perform automatic type conversions or by performing type casting.

We provide a configuration file to MERLIN with functions and variables that match each attribute. For each programming language, it is necessary to provide a configuration file. Examples of attributes considered in the configuration file are presented in the Table I. We use two class labels to classify code slices: 0 that indicates that there is no vulnerability and 1 that reports the existence of a vulnerability.

B. Classifiers

A machine learning classifier receives as an input an attribute vector that corresponds to a code slice and classifies it as vulnerable or non-vulnerable. There is a wide range of machine learning algorithms that are able to map input into a specific class. In order to select the most appropriate classifier for our problem, we studied the following classes of machine learning classifiers:

- Decision Tree algorithms: these algorithms use decision trees to predict a value of class label. A decision tree consists of nodes that correspond to attribute values to compare to; branches that correspond to results of the comparison; and leaves that represent classes. CART (Classification And Regression Tree) and Random Forest (RF) are two examples of algorithms that use this method. CART is one of the most used methods to generate decision trees. This algorithm generates only binary trees. Whereas RF generates multiple trees using a random selection of attributes.
- Probabilistic algorithms: these algorithms assign the class with highest probability. In this category, we consider Naïve Bayes (NB), K-Nearest Neighbor (KNN) and Logistic Regression (LR). NB is a classifier based on Bayes' theorem with the ”naive” assumption of independence between attributes. KNN assigns the most common class among its k neighbors. Finally, LR is a statistical model that uses a logistic function to classify an instance.
- Network algorithms: this category includes the Multilayer Perceptron (MLP) and the Support Vector Machine (SVM). MLP is an artificial neural network that uses artificial neurons to map input data into an output. Whereas, SVM classifier constructs a hyperplane to classify data.

C. Evaluation Metrics

The metrics chosen to evaluate the classifiers were precision, recall, f-score and accuracy. To compute these metrics, it is necessary to know the number of vulnerabilities correctly detected (true positives), the number of false vulnerabilities detected (false positives), the number of true vulnerabilities undetected (false negatives) and the number of no vulnerabilities correctly undetected (true negatives). *Precision* measures the ratio of vulnerabilities correctly identified among all vulnerabilities that were discovered and, it is measured according to the following formula $P = TP/(TP+FP)$ where P is the precision, TP the true positives and FP the false positives.

Recall measures the ratio of vulnerabilities correctly identified among the number of total known vulnerabilities and is given by the following formula $R = TP/(TP+FN)$ where R represents the recall and FN the false negatives.

F-score is a harmonic mean of the precision and recall metrics, and it is computed according to the following formula $F\text{-}score = 2 \times P \times R/(P + R)$.

Accuracy is the ratio of number of correct predictions to the total number of predictions made. Accuracy is computed by using the following formula $A = (TP+TN)/(TP+TN+FP+FN)$ where A represents the accuracy and TN the true negatives.

Ideally, the classifier will have a high positive rate and a low false negative rate, i.e., precision and recall will have a high value, and consequently, the F-score will also have a high value. In addition, it must correctly classify the largest number of vulnerabilities, which means it should have a high accuracy.

D. *Selection of the classifier*

In order to select the classifier that best fits our problem, we considered several machine learning classifiers. To evaluate performance of each classifier, we used the metrics previously presented. The classifiers were trained and tested with code samples from the SRD database [23]. MERLIN processed 33085 files from the SRD database; of which 6061 files were written in Java and 27024 were written in PHP. It was possible to train and test the tool with this large volume of files because the code samples were already properly classified as vulnerable or non-vulnerable, and this information could be added to the generated attribute vectors. We also created our own code samples to evaluate the tool. In total, the tool generated 65552 vectors. The vectors contained 22 attributes that characterized code and one class that classified it as vulnerable or non-vulnerable.

The classifiers were implemented in *scikit-learn*, which is a machine learning library for Python. We also used scikit-learn functions to calculate the proposed evaluation metrics. To validate the models, we used the *k-fold cross validation* technique also implemented in scikit-learn. K-fold cross validation is one of the most used techniques to test the effectiveness of a machine learning model. This method consists of splitting the training set into k folds. The classifier is trained with k-1 folds and the remaining one is used to test the model. This procedure is repeated k times. We chose to split the data set into 10 subsets (k=10).

Initially, the classifiers returned acceptable results as it is shown in the Table II. The obtained results can be explained by the data set being highly unbalanced, as it contained 56547 non-vulnerable data samples and 9005 vulnerable data samples. NB showed the worst performance. It makes a strong assumption that the attributes are independent from each other, which may not be true in our case. KNN was the second worst classifier. KNN had an unbalanced number of neighbors, and this may have lead to classifying more data samples as non vulnerable. The remaining models returned similar results. The accuracy was about 91%, the precision was around 82%, the recall was around 79% and the F-Score was 80%.

Next, we tried to improve the results by balancing the data set. The two most used techniques to balance the data are oversampling and undersampling. Oversampling involves replicating the number of instances in the minority class, while undersampling requires deleting data samples from the majority class. Since we are working with a large data set, we chose to use the undersampling technique. Evaluation of the classifiers trained with the balanced data set is presented in Table III. In terms of selection of the best classifier, the results are similar to the results previously obtained for the unbalanced data set. However, results show remarkable improvement in precision of the models, which is fundamental for good performance of the tool. As discussed earlier, there are some operations that do not always ensure proper sanitization of input data. This generates uncertainty in the data set, which also explains the obtained evaluation results.

Even though the results obtained during evaluation of the classifiers were very similar, we noticed that the values obtained by the SVM algorithm were slightly better. Therefore, we chose to use the SVM classifier in our tool. At this moment, MERLIN detects eight types of vulnerabilities. However, it is possible to configure the tool to detect other types of vulnerabilities. To handle a new vulnerability type, we need to update the configuration file with related information that includes sensitive sinks, entry points and sanitization functions. Then, the classifier should be retrained for the model to obtain new knowledge. After this, MERLIN is capable of detecting vulnerabilities that belong to the new vulnerability type.

VIII. Experimental Evaluation

The experimental evaluation aims to answer if the tool is capable of: 1) detecting vulnerabilities in multiple languages? 2) detecting vulnerabilities in real world web applications? 3) identifying the same vulnerabilities as other tools that analyze source code?

A. *Multiple Language Vulnerability Detection*

As mentioned before, MERLIN is able to detect vulnerabilities in multiple languages. For now we decided to focus on web applications written in Java and PHP. One of the objectives of the evaluation was to verify MERLIN's ability to correctly process code written in Java and PHP in the same manner. To test this, we ran the tool with web applications written in Java and PHP containing the same types of vulnerabilities and similar sanitization. An example of two code samples vulnerable to SQLi is shown in Figure 1. The tool was able to correctly identify the vulnerabilities in both code samples, so the answer to the first question is positive.

B. *Vulnerability Detection in Real World Web Applications*

In order to understand whether the tool is capable of detecting vulnerabilities in real world web applications, we evaluated the tool using two data sets that contained 12 web applications: a data set that included 6 real world web applications with vulnerabilities created on purpose (e.g., Mutillidae) and a data set that included 6 widely used web applications (e.g., phpMyAdmin and Spring Security).

First we tested the tool using the data set that contained six web applications with seeded vulnerabilities. In order to better evaluate the tool's performance, we computed the evaluation metrics used to select the machine learning classifier. To compute these metrics, we needed to calculate the number of vulnerabilities contained in the data set by source file name. Each file was individually analyzed to identify the vulnerabilities. This task was challenging and time consuming, as each web application contained a large number of source

TABLE IV
ANALYSIS OF REAL WORLD WEB APPLICATIONS WITH SEEDED VULNERABILITIES

webapp	language	#loc	#files	#TP	#FP	#FN	P (%)	R (%)	F-Score (%)
DVWAP	PHP	14,895	353	20	3	7	86.96	74.07	80
Mutillidae	PHP	142,515	919	50	20	38	71.43	56.82	63.29
bWAPP	PHP	24,070	198	337	0	263	100	56.17	71.93
WackoPicko	PHP	1,916	48	19	13	0	59.38	100	74.51
Java Vulnerable Lab	Java	1,795	60	73	29	5	71.57	93.59	81.11
HackMe	Java	824	17	31	0	14	100	68.89	81.58
Total/Avg	Java+PHP	186,015	1,595	530	65	327	81.56	74.92	75.40

TABLE V
ANALYSIS OF OPEN SOURCE WEB APPLICATIONS

webapp	language	year	#loc	#files	#identified vuln
MantisBT	PHP	2017	54,876	449	Yes
phpMyAdmin	PHP	2014	143,219	755	Yes
DokuWiki	PHP	2010	79,397	709	Yes
MISP	PHP	2016	24,006	157	No
Pinpoint	Java	2016	28,927	584	No
Spring OAuth2	Java	2015	18,332	230	No
Total	Java+PHP	-	513,807	2,884	3

files and lines of code. In addition, it was also necessary to verify if the vulnerabilities identified by the tool were real or not. We did not consider accuracy to evaluate the tool because the number of code slices correctly identified as non vulnerable will always be far more superior to the remaining values. Hence, this metric will not give any relevant information regarding the tool's performance.

The results of the analysis and the processing done are presented in the Table IV. MERLIN obtained the worst results when processing Multillidae. These results are mainly due to the following reasons: how the data flow in CFG is analyzed, which can be incomplete when the file to be analyzed is long and contains a lot of conditional paths; incorrect propagation of interprocedural data flow; and, because Multillidae includes files containing vulnerable code with *inc* format that are not processed by MERLIN, since they do not have a PHP extension. The first two reasons also influenced the results obtained with other web applications. bWAPP was the second web application with the worst results and, with the worst result regarding the metric recall. This is explained by a large number of false negative results obtained for bWAPP. MERLIN was not capable of detecting a few vulnerabilities that were replicated in nearly every file of the bWAPP web application. This inability to detect a few vulnerabilities had a major impact on the calculated value of the recall. It should also be noted that the vast majority of false positives obtained in the case of Java Vulnerable Lab web application are found in jsp files. This is because most jsp files include a jsp header that contains vulnerable code. When compiling these files, maven automatically includes the *header.jsp* bytecode into each of the processed files. Thus, whenever MERLIN processes these files, it reports the vulnerability regarding header.jsp, instead of reporting only when processing header.jsp. In total, MERLIN was able to correctly detect 530 vulnerabilities.

We also tested the tool with a set of widely used open source web applications. These web applications belong to a database called *Secbench* [34]. Secbench is a data set of real security vulnerabilities from open source web applications. The vulnerabilities were mined from Github which hosts millions of open source web applications. We chose six web application from this data set. Then we verified whether MERLIN was capable of detecting the vulnerability identified in the data set. The obtained results are presented in the Table V. MERLIN was able to identify vulnerabilities in three out of six analyzed web applications.

During this evaluation, the tool analyzed 4,479 files and it processed over 699,822 lines of code. The web application analyzed with the largest number of files was Mutillidae and with the most lines of code was phpMyAdmin.

C. Comparison with other tools

We selected two tools – WAP [17] and Achilles [27] – to compare with MERLIN. As previously mentioned, these two tools also use machine learning for detecting vulnerabilities. WAP uses classifiers to predict the existence of false positives in the identified vulnerabilities. Achilles uses a neural network to detect vulnerabilities. In order to evaluate the tools, we chose to use code samples from the SRD database.

Achilles produces as output an n-dimensional vector of predictions, ranging from 0 to 1 indicating the probability of risk for each method against each type of vulnerability [27]. The value we defined as threshold to consider the existence of a vulnerability was 0.95. After running Achilles with a set of the most basic samples which contained different types of vulnerabilities, we verified that the tool was unable to correctly detect any vulnerability. Furthermore, it detected wrongly other vulnerabilities, and therefore it was not possible to lower the threshold. When we ran MERLIN with the same set of code samples, we found that MERLIN was able to correctly identify all vulnerabilities. Thus, taking into account the discrepancy between the results obtained with the simplest samples, we did not carry out any further evaluation.

In order to compare MERLIN to WAP, we randomly selected 100 code samples written in PHP from the SRD database. We ensured that the selected samples contained a balanced number of non-vulnerable and vulnerable samples, but took no special care to select samples processable by the tools. To compare the tools, we used the evaluation metrics previously presented. The results obtained from the tools are shown in Table VI. From the results, we can conclude that MERLIN had better performance than WAP. The accuracy of MERLIN was 74%, whereas the accuracy of WAP was 60%.

TABLE VI
EVALUATION OF WAP AND MERLIN

Tool	Precision (%)	Recall (%)	F-Score (%)	Acc (%)
MERLIN	65.88	94.92	77.78	73.55
WAP	76.47	22.81	35.14	60.33

However, it should be noted that the selected code samples contained sources that were not considered tainted by WAP, leading to much worse results than those originally reported for that tool. For instance, WAP does not consider files to be sources, while the SRD database considers that that form of input can be malicious.

IX. Conclusion

This paper presents an approach to improve web application security by detecting vulnerabilities in code written in different languages using machine learning. The approach was implemented and evaluated with code samples from the SRD database and real world web applications written in Java and PHP. The evaluation shows that the tool is capable of detecting different types of vulnerabilities in both languages and in real world web applications.

Acknowledgements We warmly thank Ibéria Medeiros for feedback on an earlier version of this work. Research supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference PTDC/CCI-INF/29058/2017 (SEAL) and UIDB/50021/2020 (INESC-ID).

References

[1] Symantec, "Symantec ISTR 24 – Internet security threat report," Feb. 2019.

[2] WhiteHat Security, "2019 application security statistics report," White-Hat, Tech. Rep., 2019.

[3] Q. Luo and J. F. Naughton, "Form-based proxy caching for database-backed web sites," in *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001, pp. 191–200.

[4] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *IFIP International Information Security Conference*, 2005, pp. 295–307.

[5] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[6] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, 2005.

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

[8] W. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proc. of the International Symposium on Secure Software Engineering*, Mar. 2006.

[9] M. Gegick, E. Isakson, and L. Williams, "An early testing and defense web application framework for malicious input attacks," in *ISSRE Supplementary Conference Proceedings*, 2006.

[10] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2006, pp. 372–382.

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

[12] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[13] A. Doupé, M. Cova, and G. Vigna, "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010, pp. 111–131.

[14] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, "Intrusion recovery for database-backed web applications," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 101–114.

[15] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner." in *USENIX Security Symposium*, 2012.

[16] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda, "Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries," *Journal of Computer Security*, vol. 17, no. 3, pp. 305–329, 2009.

[17] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the International World Wide Web Conference*, Apr. 2014, pp. 63–74.

[18] I. Medeiros, N. Neves, and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[19] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159–1175, 2018.

[20] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[21] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, 1967, pp. 281–297.

[22] P. Lam, E. Bodden, O. Lhoták and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[23] National Institute of Standards and Technology (NIST), "Srd database," `https://samate.nist.gov/SRD/`.

[24] J. Williams and D. Wichers, "OWASP Top 10 - 2017 - the ten most critical web application security risks," OWASP Foundation, Tech. Rep., 2017.

[25] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

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

[27] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong, "Project Achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network," in *34th IEEE/ACM International Conference on Automated Software Engineering Workshop*, 2019, pp. 114–121.

[28] S. Genaim and F. Spoto, "Information flow analysis for java bytecode," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2005.

[29] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity JVMs," in *ACM SIGPLAN Notices 49.10*, 2014.

[30] J. K. A. Bertel and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," in *arXiv preprint arXiv:1205.3576*, 2012.

[31] "JPHP," `https://github.com/jphp-group/jphp`.

[32] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, 1992.

[33] I. Medeiros, N. F. Neves, and M. Correia, "Website of WAP tool," Jan. 2014, http://awap.sourceforge.net/.

[34] S. Reis and R. Abreu, "SECBENCH: A database of real security vulnerabilities," in *International Workshop on Secure Software Engineering in DevOps and Agile Development*, 2017, pp. 69–85.