TÉCNICO LISBOA

MERLIN: Multi-Language Web Vulnerability Detection

Alexandra Sofia Gaio Chaveiro Figueiredo

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Paolo Romano
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Ibéria Vitória de Sousa Medeiros

October 2020

“Never stop learning, because life never stops teaching...“

Acknowledgments

In these last past 5 years of my academical cycle, I learned valuable knowledge and I have grown on a professional level as well as on a personal level, which makes me more prepared for the next stages of my life. I want to thank my advisor, Professor Miguel Pupo Correia, for always being available to give me feedback, for helping me improve my work and for providing the opportunity to develop the thesis on a fascinating topic. I would also like to thank Tatjiana Lide that provided useful insights about my work and gave me tips to improve it.

I want to thank my family, especially my parents for supporting me and for always believing in me. Without you this would not be possible. I would like to dedicate this thesis to my brother, who unfortunately is not able to support me. But I know if he could, he would always stand by my side.

I would like to thank my colleagues that helped me throughout this academical path, especially Denis. Thank you for always listening to my concerns, for working by my side and making me laugh even in tougher times. You made this a little bit easier.

I want to thank my friends for listening to me whenever I talked about the endless projects and exams, for constantly reminding me to enjoy life and for the constantly encouragement through all the ups and downs. Thank you for always being there for me. I could not help mention the discord group that always motivated me to finish my thesis, especially through the nickname they assigned to me (*AfazerTeseDesde2010*), which I kindly ask you to replace it for something nice.

To the ones who stayed by my side from the beginning even when I pushed you away, to the ones who cared about my wellbeing, to the ones that I did not spend as much time as I should and yet were comprehensive and patient, to the ones that always believed in me even when I did not believe in myself, to each and every one of you – Thank you.

Resumo

Apesar de haver uma investigação contínua com o objetivo de melhorar a segurança na web, as aplicações web continuam a ser constantemente atacadas. Muitos ataques bem sucedidos exploram código fonte vulnerável. Uma abordagem comum para encontrar vulnerabilidades no código é utilizar ferramentas de análise estática de código fonte. Contudo, estas ferramentas têm dois problemas: têm de ser programadas manualmente para lidar com todo o tipo de vulnerabilidades e apenas trabalham com uma linguagem de programação específica.

Este trabalho apresenta uma abordagem que ambiciona melhorar a seguranças das aplicações web por identificar vulnerabilidades em código escrito em diferentes linguagens. Além disso, em vez de programarmos as regras de deteção, utilizámos *machine learning* para as configurar. A abordagem foi implementada numa ferramenta chamada MERLIN. Esta ferramenta foi testada com amostras da base de dados SRD e com aplicações web do mundo real escritas em Java e PHP. Até agora o MERLIN já processou mais de um milhão de linhas de código.

Palavras-chave: Vulnerabilidades, Aplicações Web, Machine Learning, Código Intermédio

Abstract

Although there is continuous research to improve web security, web applications are constantly being attacked. Many successful attacks exploit 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 thesis 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. So far MERLIN has processed more than one million lines of code.

Keywords: Vulnerabilities, Web Application, Machine learning, Intermediate Code

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For more than a decade, there has been a great amount of research aimed at improving security of web applications [LN01, NTGG+05, NB05, KVR05, JKK06, HVO06, GIW06, SW06, BBMV07, WS08, DCV10, CKS+11, DCKV12, VVB+09, MNC14a, MNC16, NMF+18, LZX+18]. Source code static analysis tools that are intended to find security vulnerabilities in web applications are commonly used by software development organizations [NB05, JKK06, WS08, MNC14a, MNC16, NMF+18, LZX+18]. 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 thesis 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.

The techniques that are used to automatically detect vulnerabilities in code are taint analysis [NTGG+05, JKK06, WS08] and machine learning classification [M+67]. 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 document 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 thesis 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* [PLH11], 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 [Nat] 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 [WW17]. 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 thesis 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.

Part of this work was reported in:

- Alexandra Figueiredo, Tatjana Lide, David Matos and Miguel Correia. MERLIN: Multi-Language Web Vulnerability Detection. In Proceedings of the 19th IEEE International Symposium on Network Computing and Applications (NCA), Nov. 2020

- Alexandra Figueiredo, Tatjana Lide and Miguel Correia. Multi-Language Web Vulnerability Detection (fast abstract). In Proceedings of ISSRE 2020, October 2020.

The remainder of the document is structured as follows. Section 2 presents related work and important background to develop this tool. Section 3 describes all stages of the approach used to develop the tool. Moreover, it also provides details on the implementation of MERLIN. Section 4 presents the evaluation process of the tool. Section 4.1 starts by showing MERLIN's ability to detect vulnerabilities in multiple programming languages, then in Section 4.2 it is described the evaluation of the tool's ability to detect vulnerabilities in real world web applications and Section 4.3 compares the performance of MERLIN with other tools that also aim to detect vulnerabilities. Section 5 concludes the dissertation and Section 5.1 presents ideas to be developed in the future aiming at improving the approach used.

Chapter 2

Background

In this section, we address important concepts for the development of the proposed work. We also analyze related work relevant for the context of this proposal. Section 2.1 presents the vulnerabilities that will be considered in our work. The input vulnerabilities considered are divided into four categories. Then in Section 2.2, we present several approaches to perform static analysis to improve security of web applications. In Section 2.3, we present different methods using machine learning algorithms to detect vulnerabilities and we explain how these algorithms can optimize the vulnerability detection process. Finally, in Section 2.4 we describe different techniques to analyze bytecode, an intermediate language.

2.1 Web input validation vulnerabilities

Most web application vulnerabilities are due to non-validation or incorrect validation of user input [MM14]. Taking that into account we will consider such vulnerabilities in our work. Input validation vulnerabilities are exploited as follows: a potentially malicious input enters the program through an *entry point* like $_GET in PHP and reaches a *sensitive sink* where the vulnerability can be exploited, like *mysqli_query()*. 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.

Since there is not enough time to cover every possible input validation vulnerability, we will focus on the vulnerability categories that have a higher risk and can have serious consequences such as malicious code execution, sensitive information disclosure and denial of service (DoS) [MM14]. Therefore, we will consider the following vulnerability categories: SQL injection, Cross-site Scripting, File and Path Injection and Command Injection. In the following subsections, we

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

Figure 2.1: Code sample written in PHP vulnerable to SQLi

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

Figure 2.2: Code sample written in Java vulnerable to SQLi

explain what types of attacks are associated with each category and give an example based on the examples presented by Medeiros in [MNC16].

2.1.1 SQL injection

SQL injection (SQLi) has the highest risk according to *OWASP TOP 10 – 2017* [WW17]. SQLi vulnerabilities are caused by the use of dynamically generated queries that receive unsanitized or incorrectly sanitized input which can cause unexpected actions on the database when executed. This vulnerability has a great impact in web applications since it can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. The script in Figure 2.1 written in PHP is an example of code vulnerable to SQLi.

In this case, a malicious input enters through the variable *$u* (entry point) and reaches a sensitive sink in line 3 where it is executed. The attacker can exploit this vulnerability by entering something similar to *' OR 1=1 - -* which modifies the query and gives access to all users' passwords. This vulnerability can be mitigated by sanitizing the input using standard language functions e.g. *mysqli_escape_string($_GET[$u])*, or by utilizing prepared statements.

SQLi occurs similarly in code written in Java, as shown in the code sample in Figure 2.2.

2.1.2 Cross-Site Scripting

Cross-site scripting (XSS) is also among the top 10 vulnerabilities with highest risk, since it can disclose users' sensitive information, such as cookie details or credentials. XSS consists of including untrusted data in a new web page without proper validation or escaping, or updating an existing web page with user-supplied data using browser API that can create HTML or JavaScript [WW17].There are three main classes of XSS depending on how the malicious scripts are inserted: reflected or non-persistent, stored or persistent, and DOM-based.

Reflected XSS can occur when an application returns unvalidated and unescaped user input

```
$u = $_GET['user'];
$q = "Hi".$u;
echo($q);
```

Figure 2.3: Code sample written in PHP vulnerable to XSS

```
String u = request.getParameter('user');
String q = "Hi" + u;
resp.getWriter().write(q);
```

Figure 2.4: Code sample written in Java vulnerable to XSS

in an HTML output. The sample code in Figure 2.3 written in PHP is vulnerable to Reflected XSS.

In this example, the malicious input reaches a sensitive sink when the *echo($q)* instruction is executed. An attacker just needs to build an application URL with a 'user' parameter being a payload containing a malicious script, convince a victim to click the URL and the victim's browser will automatically execute the script. Those kinds of vulnerabilities can also be present in code written in Java, as shown in Figure 2.4.

Stored XSS occurs when an application stores unsanitized user input that is viewed at a later time by another user or an administrator. These kinds of attacks can be prevented by sanitizing the input (for instance by using sanitization routines like *htmlentities* PHP function) and encoding the output.

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. This happens when a web application writes data in DOM without proper sanitization. An example of code vulnerable to DOM-based XSS is presented below in Figure 2.5.

In this case, an attacker just needs to provide a URL like the following "http://vulnerablesite.com/index.html #<script>alert(document.cookie)<script>", and the JavaScript code will be executed in the victim's browser.

```
<script>
    document.write("<p>List:</p>" + document.baseURI);
</script>
```

Figure 2.5: Code sample vulnerable to DOM-based XSS

2.1.3 File and Path Injection

This subsection considers vulnerabilities that deal with web application access to files in the local file system or remote URL locations [Med16]. In our work we are going to consider the following vulnerabilities belonging to this category: remote file inclusion (RFI), local file inclusion (LFI), directory traversal or path traversal (DT/PT) and source code disclosure (SCD).

RFI/LFI vulnerabilities allow attackers to embed code in the vulnerable application. While in LFI the code has to be in the local file system, in RFI it can be elsewhere outside the local file system. The objective of DT/PT attack is to access unexpected files, possibly outside the web site directory. The SCD attack consists in an attacker accessing web application source code and configuration files.

2.1.4 Command Injection

Command injection vulnerabilities allow attackers to execute arbitrary shell code on the application server. Within this category we will consider the following vulnerabilities: operating system command injection (OSCI) and PHP code injection (PHPCI).

An OSCI attack forces an application to execute a command defined by an attacker [Med16]. A PHPCI vulnerability allows an attacker to supply code that is executed by an eval statement.

2.2 Static Analysis for Security

Static analysis tools examine the source, binary or intermediary code without executing it. Static analysis can be used in different contexts in software (e.g., detecting software defects [NB05]), but in our work we will only consider static analysis to detect security vulnerabilities. Static analysis tools when compared with manual auditing are more advantageous for two main reasons: can be used at any moment during the software development life cycle, even if the software is not executable; and the tool operator does not need the same level of security expertise as a human auditor [CM04]. Therefore, static analysis can be used to improve the software quality throughout the development process [JSMHB13].

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. There are several approaches to perform data flow analysis, the most common being *taint analysis* (e.g. [WS08], [DH14], [MNC14a]). According to Medeiros [Med16], taint analysis does this kind of analysis by marking as tainted the data that enters through the entry point, and reporting a vulnerability if it reaches a sensitive sink without being sanitized. If the data passes through a

sanitization or validation function, it becomes untainted, and it is not reported as a vulnerability. Although there are several approaches to do static analysis using taint analysis, we present and describe five approaches relevant for our work.

2.2.1 Static Analysis to find XSS

Wasserman and Su [WS08] implemented a tool to find XSS vulnerabilities due to unchecked untrusted data or insufficiently-checked untrusted data. This tool does static analysis by combining tainted information flow and string analysis. The approach consists of two parts: 1) an adapted string analysis to track untrusted substring values, and 2) a check for untrusted scripts based on formal language techniques.

In the first part, the tool performs string-taint analysis based on Minamide's string analysis algorithm [Min05]. Its string-taint analysis uses context-free grammars to represent sets of string values that a program may generate and finite state transducers to model the semantics of string operations. So, initially the program is translated into the static single assignment (SSA) form and the output statements (e.g. *echo* statements) are transformed into assignments to an added output variable. Then, the SSA form produces *extended* context-free grammars, dropping the control structures as it is shown in Figure 2.6.a). Finally, from the extended context-free grammars, they construct context-free grammars for the arguments to the string operations and use finite state transducers (FSTs) to model the string operations' semantics. FSTs are finite state automata (FSAs) that produce output. Figure 2.6.b) shows an example of an FST that represents a function for removing backslashes ('\') used to escape quotes from a string. They perform string-taint analysis using the context-free grammars and FSTs, classifying the output context-free grammar as tainted or untainted.

To build the second part, they analyzed the W3C recommendation, the Firefox source code and online tutorials and documents to understand how an HTML document can invoke a browser's JavaScript interpreter. This part was the biggest challenge since web browsers support many ways of invoking the JavaScript interpreter. After examining the workflow of a web document in a browser, they built a policy using regular languages to identify untrusted input that can invoke the JavaScript interpreter.

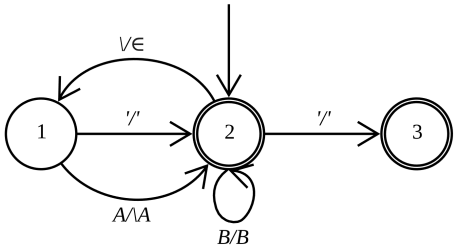
They analyzed seven web applications with the developed tool and XSS vulnerabilities were found in all applications. For instance, in the Claroline application, the tool found 32 vulnerabilities, whereas CVE-2005-1374 lists only 10 XSS vulnerabilities. The tool failed to analyze some of the web applications because the tool could not deal with the use of alias relationships between variables whose values are used for dynamic features. Therefore, they demonstrated that the

```
$data1 = $_REQUEST['module'];
$data2 = preg_replace(
    '/([\s"\']+on\w+)\s*=/i',
    'HordeCleaned=', $data1);
$module = $data2;

if ($use) {
    $output1 = '<br>';
} else {
    $hiddenfields = "<input value='$module'/>";
    $output2 = ' </div> '.$hiddenfields;
}
$output3 =   ($output1, $output2);
```

$$
\begin{aligned}
&REQUESTmodule^T \rightarrow \textstyle\sum^* \\
&data1 \rightarrow REQUESTmodule \\
&data2 \rightarrow preg\_replace('/([\backslash s"\backslash'] + on\backslash w+)\backslash s* = /i', \\
&\qquad\qquad 'HordeCleaned =', \$data1); \\
&module \rightarrow data2 \\
&output1 \rightarrow
 \\
&hiddenfields \rightarrow <inputvalue =' module'/ \\
&output2 \rightarrow </div> hiddenfields \\
&output3 \rightarrow output1|output2
\end{aligned}
$$

(a) Translation of the code in SSA form to extended context-free grammars



(b) Example of a FST

Figure 2.6: Data representations used in String-taint analysis based on [WS08]

tool can scale to large web applications and can detect known and unknown XSS vulnerabilities in real world applications.

2.2.2 Precise static analysis

Dahse and Holz [DH14] implemented an innovative tool called RIPS that performs precise static analysis of PHP code. This tool performs a comprehensive analysis and simulation of over 900 built-in features. RIPS also performs an intra and inter-procedural data flow analysis to create summaries of the data flow within the application. Those summaries allow the tool to execute efficiently a backward-directed taint analysis for 20 different types of vulnerabilities. Furthermore, they perform context-sensitive string analysis to refine the taint analysis results based on the current markup context, source type, and PHP configuration.

For each PHP file in the project, the tool starts by building an *Abstract Syntax Tree* (AST) based on PHP's open source internals. The body of user-defined functions are saved as separate ASTs and are removed from the main AST of the parsed file. Then, they transform each main AST into a *Control Flow Graph* (CFG) using the CFGBuilder (which creates the basic blocks). Whenever it detects a jump statement, it creates a new basic block and connects it to the previous basic block with a block edge. And the jump condition is added to the block edge. As soon as it creates a new basic block, the data flow of this block is simulated. The purpose of the simulation is to create a summary of the data flow within one basic block of the CFG. The nodes of the basic block can perform an assignment whose value is stored in the block summary as a symbol. A symbol is the language set that represents memory locations. In the block summary,

the symbols allow efficient backwards analysis of upcoming basic blocks in the CFG. If during the simulation it encounters a call to a previously unknown user-defined function, the CFG is built from the function AST, and a function summary is created once with an intra-procedural analysis. After analyzing the user-defined function, the effects of a call of this function can be evaluated and changes to the current scope can be processed.

During the basic block simulation, upon finding user-defined functions or sensitive sinks, it conducts taint analysis of their parameters to look for potential vulnerabilities. Therefore, they identified 288 sensitive sinks which they configured by function name, sensitive parameter, and vulnerability type. To find all possible values of a sensitive sink's argument, the argument is traced backwards through all basic blocks linked as an entry edge to the current block. In order to optimize this process, the results of each symbol are saved in a basic block cache. Moreover, they configured a maximum limit of traversed edges to optimize the performance. The string is analyzed in a context-sensitive way. This means that for each vulnerability type, a different analyzer that identifies the context within the markup is invoked. Depending on the context, the string is associated with specific vulnerability tags. If the taint symbols are not sanitized against the current vulnerability tag, they are marked as a tainted symbol and a vulnerability is reported.

RIPS was evaluated with five popular open source applications with over 185 000 lines of code. The tool detected 73 previously unknown vulnerabilities with a true positive rate of 72%. The reasons behind the false positives were path-insensitive data flow analysis, sanitization through database whitelist and wrong content-type. They concluded that other tools are not able to detect so many vulnerabilities because they incompletely model the PHP language.

2.2.3 Accurate and Scalable Security Analysis

ANDROMEDA is a static taint analysis tool [TPC+13] that computes data flow propagations on demand, ensuring accuracy and scalabality. This tool supports the processing of applications written in Java, .NET and JavaScript.

ANDROMEDA receives as input a web application together with its supporting libraries, and a configuration file with sources, sanitization functions and sensitive sinks. The tool builds a call graph of the web application and at the same time it performs tracking of vulnerable information flows. The call graph is built using an intra-procedural type-inference algorithm that determines whether the call site should be expanded or not. As a result, ANDROMEDA tracks vulnerable information flows in a demand-driven manner, instead of building an eager complete representation of the whole program. Moreover, ANDROMEDA also computes an

aliasing relationship when an untrusted value flows into an object field. By using this approach, the algorithm used to detect vulnerabilities can achieve soundness, accuracy and scalability. They also developed framework and library support which allows to perform a more complete analysis of real world web applications, which are often developed upon frameworks.

ANDROMEDA also performs incremental security analysis, i.e., it performs efficient rescanning of the web application when there are code changes. This is possible because the tool uses change impact analysis (CIA) algorithm that compares the old code version with the new, and it marks the differences as modified, deleted or added. Then, it locates the changes to determine to which layer of data structures they affect. In this manner, the tool tracks data flows in a "local", on-demand fashion.

ANDROMEDA was tested with a set of 16 benchmarks. During the evaluation, the tool achieved high accuracy. Furthermore, the tool obtained better results when compared with other taint analysis tools. This can be explained by the fact that ANDROMEDA uses a combination of soundness and framework modeling that allows to find more entry points and follow data flows through more parts of the application.

2.2.4 Scalable and precise points-to analysis

Livshits and Lam [LL05] developed a static analysis tool to find vulnerabilities caused by unchecked input in Java web applications. The tool is able to detect SQLi, XSS, HTTP response splitting, path traversal and command injection attacks. The approach used requires providing vulnerability patterns containing sources, sensitive sinks and derivation descriptors. Derivation descriptors specify the behavior of string manipulation routines, which is important to propagate correctly the taintdness in the program. The vulnerability patterns are written in PQL, which is an easy-to-use program query language with a Java-like syntax. The vulnerability patterns are then translated into Datalog queries.

The tool analyzes Java bytecode to search for security violations that correspond to the security patterns specified. The tool performs context-sensitive Java points-to analysis. This analysis is based in an algorithm, developed by Whaley and Lam, that uses binary decision diagrams (BDD) to exploit similarities across the calling contexts. The results of this algorithm can be easily accessed using Datalog queries, which are used to look for security vulnerabilities. As a result the approach used is both precise and scalable. Furthermore, the tool also provides an interactive interface built on top of Eclipse. This way, the developer can evaluate the produced code without leaving the development environment.

The tool was tested with a set of nine popular open-source applications and, was able to find

29 vulnerabilities. Moreover, the tool obtained few false positives in the experiments. These results demonstrate that is an effective practical tool to find security vulnerabilities.

2.2.5 Static analysis for OOP Web Application Plugins

phpSafe [NFV15] is a source code analyzer that identifies vulnerabilities in PHP plugins developed using object oriented programming (OOP). The tool is capable of detecting XSS and SQLi vulnerabilities and supports the processing of plugins developed for WordPress.

phpSafe provides a web interface that allows the user to perform vulnerability scanning in PHP applications and plugins. Moreover, it also allows to specify search and output options. Therefore, to run the tool it is necessary a local web server with the PHP interpreter enabled and a web browser. phpSafe can also be integrated into the software development process, by including the tool in a PHP project like an API.

Initially, phpSafe loads the configuration data, containing the list of vulnerabilities correlated with the PHP language functions, and the target Content Management System (CMS) framework specific functions that may affect the presence of vulnerabilities. The functions included in the configuration data belong to the following categories: potentially malicious sources, sanitization and filtering functions, revert functions (that revert the actions of the sanitization and filtering functions) and, sensitive output functions. The tool builds an abstract syntax tree (AST) for each file of the plugin. Then, it cleans the AST by remmoving comments and extra whitespaces. Next, it analyzes the data flow of the tainted variables. phpSafe performs data flow analysis of all code. It is capable of covering all code, by performing an interprocedural analysis starting from the "main function" and following the program flow from there. Furthermore, phpSafe is also able to process correctly OOP concepts like objects, properties and methods. Finally, the tool processes the results of the data flow analysis and it reports the vulnerabilities found.

To evaluate the tool, they compared its performance with the performance of other two tools - RIPS and Pixy. To perform the evaluation, they used a set of 35 WordPress plugins with differents sizes and complexities. phpSafe was able to detect more vulnerabilities than the other tools, with fewer false alarms.

2.3 Security Analysis with Machine Learning

Although static analysis tools are a useful mechanism for detecting vulnerabilities, they have some drawbacks. Static analysis tools developers must code their knowledge about vulnerabilities, which is a complex and tedious work. Furthermore, this knowledge can be incomplete or

```
1   $MAX = 999;
2   if ($MAX > 0) {
3       $sz_orig = getimagesize('photos/id.jpg');
4       $ratio = $sz_orig[1]/$sz_orig[0];
5       if ($sz_orig[0] > $MAX) {
6           if($ratio > 1) {
7               $height = $MAX;
8               $width = (int) ($MAX/$ratio);
9           }
10          else {
11              $width = $MAX;
12              $height = (int) ($MAX*$ratio);
13          }
14          $img_size="style='width:$width;height:$height'";
15      }
16  }
17  echo "<div $img_size>";
```

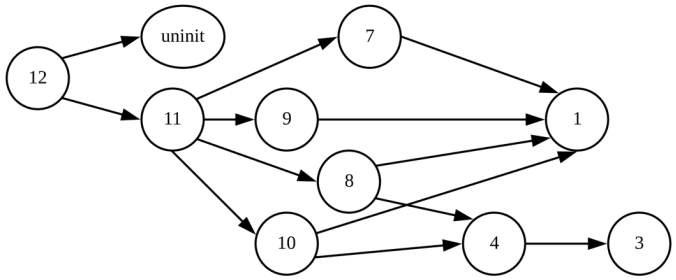


Figure 2.7: Data Dependence Graph of sensitive sink node 12 [ST12a]

wrong, which affects the tool accuracy [MNC16]. To overcome these disadvantages, some detection systems use Machine Learning techniques. There are several approaches using machine learning to detect vulnerabilities in web applications which differ from the way they analyze the source code, attributes they chose, methods they use to perform feature extraction to types of machine learning algorithms used in the tool. Some approaches are simpler than others. Therefore, we will consider some examples of how to use machine learning to improve vulnerability detection systems.

2.3.1 Taint Analysis with data mining

PhpMinerI is a tool developed by *Shar and Tan* [ST12a] which uses data mining methods to predict SQL injection and cross site scripting vulnerabilities. To train and build the vulnerability prediction model, they chose a set of attributes that represent different types of input sanitization methods.

PhpMinerI is based on an open source PHP code analysis tool called *Pixy* [JKK06]. Initially, the tool builds a control flow graph (CFG) of the web application. For each sensitive sink k in a CFG, the tool extracts its data dependence graph DDG_k (generated by Pixy) as illustrated in Figure 2.7.

Each node in DDG_k is classified with a type that represents an attribute. The classification types can be divided into different categories: nature of input sources, type of sensitive sink, input sanitization methods and other operations that may or may not serve any security

purpose. Therefore, PhpMinerI counts the number of nodes in DDG_k that correspond to each classification type and assigns the number to the attribute which represents that classification type. Thus, each sensitive sink has an attribute vector. They mined the attribute vectors, using three different classifiers implemented in *WEKA* [HFH+09] C4.5/J48 which is a decision tree-classifier, Naïve Bayes which is a simple statistic-based classifier and a Multi-Layer Perceptron which is an artificial neural network-based classifier. To validate the training set, they used (M=10)*(N=10)-way cross validation. This method consists of dividing the data in 10 subsamples. Then, 9 subsamples are used to train the classifier and the remaining subsample is used to test the classifier. This procedure is repeated 10 times without testing the same subsample twice.

They evaluated the tool with three open source PHP-based web applications. predicted over 85% of SQLi and XSS vulnerabilities in those web applications. The tool missed some vulnerabilities because it did not consider input validation through predicates. Thus, they concluded that PhpMinerI is practical and a good alternative to detect security vulnerabilities in web applications.

Aiming at reducing false positive cases, they developed a tool called *PhpMinerII* [ST12b]. This tool is able to handle input validation code patterns, unlike PhpMinerI which ignores those cases. Thus, PhpMinerII uses 28 attributes to represent a sensitive sink in contrast to the 19 attributes used by PhpMinerI. Nevertheless, PhpMinerII had worse results than PhpMinerI, which can be explained by the fact that PhpMinerII was tested with seven web applications whereas PhpMinerI was tested only with three web applications.

2.3.2 Removing false positives with data mining

Web Application Protection (WAP) [MNC14a] is a tool that uses a hybrid approach to detect vulnerabilities. In the beginning, it performs taint analysis to flag candidate vulnerabilities, then uses data mining to predict the existence of false positives. Thus, this approach combines two methods: knowledge coded by the developers (taint analysis) and knowledge obtained automatically (data mining). Finally, the tool corrects the source code automatically by inserting fixes while keeping the programmer in the loop. WAP detects a wide range of input validation vulnerabilities: SQLi, XSS, remote file inclusion, local file inclusion, directory traversal/path traversal, source code disclosure, PHP code injection and command injection.

WAP performs taint analysis using tree walkers to navigate through the AST. The AST and the tree walkers are generated by a lexer and parser created using ANTLR [Par09]. The tree walkers build a tainted symbol table (TST) in which every cell is a program statement from

which we want to collect data, and a tainted execution path (TEPT) in which each branch corresponds to a tainted variable. Therefore, the taint analysis consists in travelling through the TST. Initially, all symbols are untainted except for the entry points. If a variable is tainted, the taintedness is propagated to the symbols that depend on it and the TEPT is updated with variables that become tainted. Furthermore, WAP performs global, inter-procedural, and context-sensitive analysis, which means that they follow data flows even when they enter new functions and other modules.

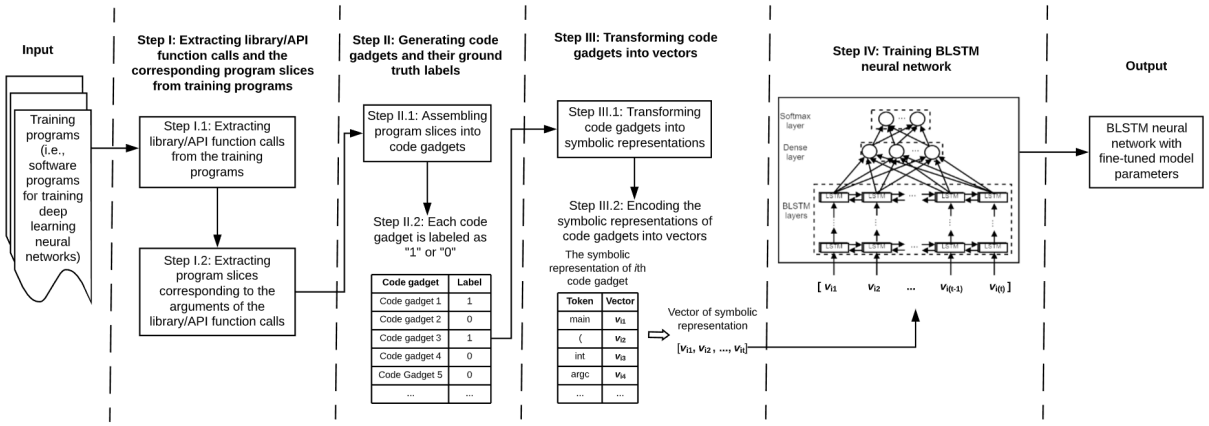
In the second step, WAP extracts the attributes of the candidate vulnerabilities and uses a classifier to determine if the candidate vulnerability is a false positive or not. To configure the tool for this step, they manually identified a set of attributes that they verified to be associated with the presence of false positives. These attributes fit into three categories: String manipulation (represent PHP functions or operations that manipulate strings), Validation (related with validations of user input), and SQL query manipulation (related with the insertion of data in SQL queries). Moreover, they evaluated a set of machine learning algorithms: graphical/symbolic algorithms – ID3, C4.5/J48, Random Tree and Random Forest; probabilistic algorithms – Naïve Bayes, K-NN and Logistic Regression; neural network algorithms – MLP and SVM. The classifier with the best performance was LR and therefore it was the one implemented in WAP.

Once the tool has confirmed that the vulnerability is not a false positive, WAP corrects the code by inserting the suitable fix. Finally, it provides feedback to the programmer which includes where the vulnerability was found and how it was corrected.

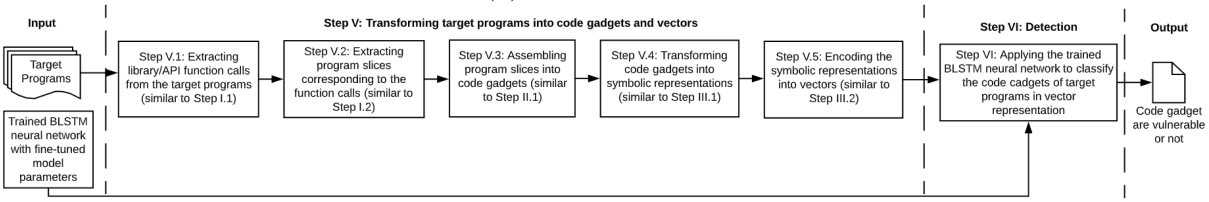
To evaluate WAP, they run the tool with 35 PHP applications with more than 2,800 files and 470,000 lines of code. WAP found 294 vulnerabilities with at least 28 false positives. These results suggest that this tool can detect the proposed types of vulnerabilities and can also scale to large web applications.

2.3.3 Vulnerability detection with deep learning

Li *et al.* [LZX+18] developed a deep learning-base vulnerability detection system called VulDeepecker. They chose to use deep learning because in this way human experts do not have to manually define the features, which is a tedious task. Since deep learning was not created to perform vulnerability detection, they needed some guiding principles to apply deep learning in this kind of applications. Therefore, they used code gadgets to represent programs. A code gadget is a number of (not necessarily consecutive) lines of code that are semantically related with each other, i.e. there is a data or control dependency on the selected lines of code. Thus, a code gadget achieves a fine granularity representation. Then, the code gadgets are transformed



(a) Learning phase



(b) Detection phase

Figure 2.8: Overview of VulDeePecker [LZX+18]

in vectors. These vectors are the input for the neural networks. Since a vulnerability depends on the context, they chose to implement a Bidirectional Long Short-Term Memory (BLSTM) neural network.

VulDeePecker has two phases: the learning phase and the detection phase as it is represented in Figure 2.8.

In Step I.1 – Extracting library/API function calls and the corresponding program slices from training programs, the tool extracts two categories of library/API function calls: forward library/API function calls where the function receives input directly from the external input and backward library/API function where the function does not receive external input. In the next step - Extracting program slices corresponding to the arguments of the library/API function calls, they also divided the slices into two types: forward slices which correspond to statements that are affected by the argument and backward slices which correspond to the statements that can affect the argument. To extract these two types of slices they use a *data dependency graph* produced by the tool Checkmarx [che].

In Step II.1 – Assembling program slices into code gadgets, they start by assembling the statements which belong to the same, user-defined function into a single peace according to the order of the statements' appearance in the user-defined function. Then, they combine the statements belonging to different, user-defined functions into a single code gadget. In the following step - Labeling the ground truth of code gadgets, the code is labeled with 1 if it

contains a vulnerability, and with 0 otherwise.

Step III.1 – Transforming code gadgets into symbolic representations, can be divided in three substeps: first, remove the non-ASCII character and comments; second, map user-defined variables to symbolic-names; third, map user-defined functions to symbolic names. In the next step - Encoding the symbolic representations of code gadgets into vectors, at the beginning they divide the code gadget in the symbolic representation into a sequence of tokens using lexical analysis. Then, the tool transforms the tokens into vectors. Since the produced vectors can have different lengths and BLSTM only takes equal-length vectors as input, they introduce a parameter τ as the fixed length of the vectors. When the vector is shorter than τ, they pad the vector with zeros and when the vector is longer than τ they delete part of the vector. Steps IV – Training BLSTM neural network and VI - Detection are standard and step V – Transforming target programs into code gadgets and vectors is similar to Steps I-III.

They ran the tool against three software products: Xen, Seamonkey and Libav. The tool detected four vulnerabilities that were not reported in the National Vulnerability Database, but were "silently" patched in the following versions. Furthermore, VulDeepecker detected more vulnerabilities than other tools.

2.3.4 Detection using a Recurrent Neural Network

Project Achilles [SDD+19] is a prototype tool that uses an array of Long-Short Term Memory (LSTM) Recurrent Neural Network (RNN) implemented in Python using Keras [ker] to detect vulnerabilities in Java source code.

For the neural network to achieve good results, it is necessary to perform an appropriate pre-processing of Java source code. Therefore, regular expressions were used to remove comments and an algorithm was developed to extract the methods from Java source code. Furthermore, they also developed algorithms to tokenize and label the extracted Java methods. Initially, the extracted Java methods are separated in Java tokens by the Javalang tokenizer. Then, the tool joins the list of tokens with a space between them and uses them as input for a tokenizer and an embedder, which are implemented in the Keras library. The data resulting from this pre-processing is used as input for the LSTMs models.

For each category of the CWE vulnerabilities, several LSTM models were trained. In a dataset containing n categories of vulnerabilities, the tool generates a n-dimensional vector of predictions which indicates the probability of risk for each method against each of the vulnerability categories. The produced n-dimensional vector is used as input for the softmax function.

To train and test the project Achilles, they used a Juliet Test Suite which contains 81000

C/C++ and Java programs with known vulnerabilities designed to test the efficiency of static analysis tools. In the training phase, besides adjusting the weights in each memory cell, they also tuned the hyperparameters, such as the dropout, the epoch, the loss function and the activation function. The test phase shows that the tool performs effectively and accurately.

2.3.5 Deep learning model for vulnerability detection

The developed tool uses Deep Learning (DP) and Natural Language Processing (NLP) to detect vulnerabilities in PHP slices [aIMAN20].

The tool processes PHP slices in an intermediate language, composed of their respective opcode. To obtain the opcodes from PHP, they used a tool called Vulcan Logic Dumper (VLD). VLD intercepts the opcodes processing before they are executed and saves them into a file. By using an intermediate language, the tool has a higher perception of the internal structure of the language, which in turn can improve the classification of vulnerabilities. Next, the embedding layer maps tokens to embedding vectors. Then, the model uses a Long Shot-Term Memory (LSTM), Dropout, and Dense layers to output the probability, between 0 and 1, of the sample being vulnerable to SQLi. In order to implement the model they used the Python package Keras that has an easy-to-use interface.

To train and test the tool, they used a dataset retrieved from the Software Assurance Reference Database (SARD). The dataset contains 858 samples vulnerable to SQLi and 504 non-vulnerable. They also evaluated various hyperparameter configurations for different DL optimizers to verify which model produces the best results. The optimizer that achieved the best results was RMSProp and it is the one used in the tool. The results obtained during the evaluation show that the tool is capable of detecting SQLi. Moreover, it also shows that it can help programmers to avoid attacks that could cause a lot of damage.

2.3.6 Sequence classification to detect web application vulnerabilities

DEKANT [MNC16] is a tool used to detect web application vulnerabilities inspired in NLP. The tool uses machine learning techniques that take the order of code instructions into account - sequence models. The tool accomplishes that by using a Hidden Markov Model (HMM). A HMM is composed of nodes that represent states and edges that represent transitions between states. In this case, the HMM finds the sequence of states that best represents the sequence of code instructions. The possible final states are vulnerable (Taint) or not vulnerable (N-taint). As a result, DEKANT is able to perform detection and identification of vulnerabilities in the code.

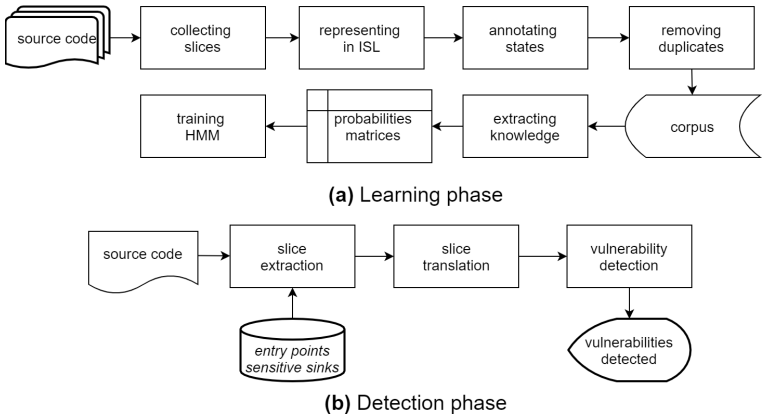


Figure 2.9: Overview of the approach used in DEKANT [MNC16]

The approach used in the tool has two phases: learning and detection. Figure 2.9 presents in more detail the approach used in the tool. In the learning phase, the tool acquires knowledge about vulnerabilities. First, DEKANT collects slices from the source code. The slices start with a source and end with a sensitive sink. Then, the slices are translated into an intermediate slice language (ISL). ISL is an abstraction of the original slice, which is simpler to process. Next, each slice is annotated as vulnerable or not vulnerable and, the duplicates are removed. From the processed ISL code slices, it is build the corpus. Then, the corpus is used to extract knowledge and it is represented with probability matrices. Finally, the HMM is trained to identify vulnerabilities.

In the detection phase, the vulnerabilities are detected using the HMM. Initially, the slices are extracted from the source code. Then, the slices are translated into ISL format and, it is build their variable map, which allows to track how input data propagates to different variables. Finally, HMM classifies the code slice as vulnerable or not vulnerable.

DEKANT is capable of detecting the same classes of vulnerabilities as the WAP tool (SQLi, XSS, remote file inclusion, local file inclusion, directory traversal/path traversal, source code disclosure, PHP code injection and command injection). In order to prove the DEKANT's ability to detect those vulnerabilities, the tool was tested with 10 WordPress plugins and 10 open source real world web applications. DEKANT was capable of identifying correctly 16 vulnerabilities in plugins and 211 vulnerabilities in web applications, demonstrating that is able to correctly characterize vulnerabilities.

2.4 Bytecode analysis

The methods considered so far perform static analysis of the source code. However, it is also possible to perform static analysis of *bytecode*. Bytecode is a low-level representation of a program

and it is generated by the compilation of the source code, e.g. Java bytecode generated from a Java program. Bytecode is executed by a virtual machine rather than by dedicated hardware [E. 08]. In the case of Java language, the compiled bytecode are interpreted by Java Virtual Machine (JVM). Bytecode analysis is particularly useful when the source code of a web application is not available, which is the case with most of real-world web applications. Furthermore, bytecode analysis has some advantages when compared with source code analysis, since it analyzes only the code that is actually executed and it avoids redundant work done by the compiler, such as name resolution, type checking, template/generics instantiation [LF08]. Nevertheless, performing an accurate bytecode analysis can be a difficult task. Thus, in subsection 2.4.1 we will inspect a method to perform information flow analysis for Java bytecode, in subsection 2.4.2 we describe an approach to perform dynamic taint tracking for Java bytecode, in subsection 2.4.3 we present a framework that converts Java bytecode and Java source code into intermediate representations to facilitate the analysis process and, in subsection 2.4.4 we describe a modification of the previously mentioned framework to convert Dalvik bytecode - a variant of Java bytecode used to distribute Android applications - to an intermediate representation for static analysis.

2.4.1 Information flow analysis for Java bytecode

Genaim and Spoto created a context sensitive compositional information flow analysis for full Java bytecode [GS05]. It is an interesting analysis to verify security of a program since it can infer dependencies between program variables, which allows identification of undesired information flows. However, their objective was not to do information flow analysis to detect vulnerabilities.

Java bytecode is an object-oriented low-level language. Since it uses an operand stack to hold intermediate representations, it lacks an explicit structure. So, in order to recover the structure, the tool transforms Java bytecode into *basic blocks* [HP11], which are a straight-line code sequence with no branches in except to the entry and no branches out except to the end. Allowing transfers of control only at the end of a block. They also connect these blocks with directed edges which represent the transfers of control in the program. Thus, by following this method they build a control-flow graph of the program.

Flows can be *direct* or *indirect*. Additionally, direct flows can be *explicit* when they arise from assignments, or *implicit* when they arise from conditionals. Implicit flows are represented by a one-to-one correspondence between sources of implicit flows and nodes of the control-graph with at least two immediate successors. Furthermore, to analyze implicit flows they also need to compute the sets of bytecodes that are affected by the implicit flow arising at the sources, i.e., their scope.

Additionally, the tool computes information flows for single bytecodes. This analysis is modular, since the tool analyzes each component independently from its context. To perform bytecode sequence analysis, the tool needs to propagate the implicit flows from each basic block, so they modified the denotation of a basic block to accept or ignore incoming implicit flows. Moreover, they also transformed the denotation of a basic block to deal with outgoing implicit flows. Therefore, with this donation for each basic block in the control-graph, they generate an equation system whose least solution approximates the information flows of the corresponding Java bytecode program.

In order to approximate statically the dynamic classes and determine a superset of the methods that might be called at run-time, they used class hierarchy analysis. Therefore for each method *k.m*, they created a basic block which statically calls *k.m*. Furthermore, they also perform denotational static analysis by using a table from method names *k.m.*.

To represent exceptions, they placed each bytecode that might raise an exception at the end of a basic block with at least two successors, for normal or exception continuation.

Finally, to implement this analysis they created a tool called *Julia*. They used Boolean functions to represent information flows, and in turn Boolean functions were implemented as binary decision diagrams by using Buddy Library [LN]. Julia was evaluated with six non-trivial applications and managed to successfully analyze them all.

2.4.2 Dynamic taint tracking

Bell and Kaiser designed a tool called *Phosphor* [BK14]. This tool uses a different approach to analyze the data flow of Java bytecode. Despite being a dynamic analysis tool, it is interesting to understand how the taintedness is efficiently propagated through the bytecode variables.

Phosphor provides dynamic taint tracking within the Java Virtual Machine (JVM). Taint tracking is used to analyze the data flow of a program. This technique consists in assigning labels to data and propagate them through the data flow. Phosphor performs taint analysis by instrumenting all code in such a way that every variable maps to a "shadow" variable, that stores the taint tag for that variable. This approach ensures the following properties:

- Portability: the approach used in the tool is compromised between an interpreter level approach and a source code level approach, since it does not need any modification to the interpreter, and it uses the strong specification of the intermediate language that runs in the interpreter. Thus, Phosphor presents portability as it analyzes the data flow of the code without requiring any changes to the language interpreter, Virtual Machine (VM), operating system or requiring access to the source code.

- Precision and Soundness: Phosphor performs taint tracking using variable-level tracking. Since variables are clear units of data, it is guaranteed that every time a variable is accessed, the variable is associated with a taint tag. By using this approach, the tool increases its accuracy and soundness.

- Performance: an obstacle for the good performance of the tool was the efficient storage of value mapping for taint tags within the confines of a memory-managed environment. To overcome this challenge, Phosphor instruments the code by using the ASM byte code manipulation library, which intercepts all classes as they are loaded and adds variables and instructions for taint tracking. Therefore, the tool to track the tags adds a field to classes and a shadow variable to variables. When it is not possible to add a shadow variable, Phosphor combines the taint tag with the value in a class container. Thus, Phosphor considers five different categories of variables that may have different representations: primitives (Boolean, byte, character, integer, short, long, float and double), primitive arrays, multi-dimension primitive arrays, general references (objects and arrays), and arrays of other references. Just like in JVM, the tool also considers four different shadow variable storage areas: fields, local variables, operand stack and method return values.

The approach used by Phosphor, presents some drawbacks: it does not perform taint tracking of implicit operations, i.e., it does not consider control flows and, since this method involves modifying the application's byte code to propagate taintedness, it can change the normal behavior of the application.

The tool was evaluated in terms of performance, precision, soundness and portability. To test the performance, they executed Phosphor with a series of micro and macro benchmarks, and the tool presented a good performance when compared with other taint tracking tools. To evaluate the precision and soundness, they designed a set of tests specific to test these properties and the tool passed all the tests that did not include implicit flows as expected. Finally, they demonstrated the portability of the tool by running the tool with two completely different VMs: Apache Harmony VM and Kaffe VM (in addition to the VMS they had already worked with: Oracle's HotSpot and OpenJDK's IcedTea).

2.4.3 Soot framework

Soot is a framework used to develop static analysis tools for Java programs [PLH11]. At its core Soot is a compiler, since it receives Java source code or Java Virtual Machine bytecode as input and produces executable Java bytecode as output. The key features of Soot include a simplified

three-address intermediate representation of Java bytecode and, pointer analysis and call graph construction algorithms.

Soot's main intermediate representation is called Jimple (Java sIMPLE). It is a simplified three-address intermediate representation of Java bytecode. Jimple was created because it is difficult to directly analyze Java bytecode since the implicit stack masks the flow of data, instead of manipulating data in a stack like bytecode. Therefore, Jimple stores data in named local variables which makes the local flow of data much more obvious. Moreover, Jimple is more readable to humans since it can be considered as a cleaned-version of Java bytecode. On the other hand, this representation, to be readable, is not complete (e.g. omits full names of fields). In addition to Jimple, Soot has other intermediate representations:

- Shimple is an SSA-based version of Jimple [Uma06];
- Baf is a streamlined representation of bytecode;
- Grimple is an aggregated version of Jimple suitable for optimization;
- Dava is an abstract syntax tree-based intermediate representation produced via decompilation of the Jimple Intermediate Representation.

Since Java bytecode includes the class and method structure of the original source code, Soot can provide class and method name information. This framework can also provide line and variable name information for the method it is analyzing, and make original variable names available to analyses on a best-effort basis. Furthermore, Soot has the key feature of supporting the implementation of intra-procedural data flow analysis. The framework also allows the creation of a data flow analysis by specifying the abstraction and implementing transfer functions for that analysis.

In order to perform inter-procedural analysis, Soot includes the Spark pointer analysis toolkit [Lho02] and the PADDLE pointer analysis framework implemented using Binary Decision Diagram (BDD) which adds context-sensitivity. Spark builds call graphs which provide very useful information to perform static analyses. The call graph construction algorithms compute an over-approximation (i.e., any call that could occur in any execution of the program must appear in the call graph) of the set of calls that may occur at runtime. Call graph edges connect sources and are represented as (method, statement) pairs. Spark implements several different call graph construction methods. The call graphs can be queried by call site (the location where the function was called), calling method or target method ("backwards"). Spark also provides pointer information (i.e. determines if two variables p and q refer to the same heap object at runtime), by implementing a context-sensitive subset-based points-to analysis. Moreover, Soot performs

side effect analysis, i.e. determines if a statement s has a possible dependence on statement s'. This analysis is implemented on top of points-to and call-graph analysis. Furthermore, Soot provides in addition to Spark other basic call graph and pointer analysis producers, such as a demand-driven pointer analysis [MSB05] or a BDD-based context-sensitive pointer analysis [Lho06].

Soot makes analysis results available by outputting transformed class files; printing error messages; generating HTML or graphs containing analysis results; or creating class files annotated with results obtained from program analysis. The framework can be executed on the command line or through its Eclipse plugin [JLH04]. The latest allows the user to view Jimple CFGs, static analysis results and flow analysis results as they are being computed.

Soot has been used by many researchers and students and it has been able to perform sophisticated analyses of Java programs, starting from the bytecode for these programs.

2.4.4 Dexpler

Dexpler is a tool to convert Dalvik bytecode to Jimple [ABM12]. This tool allows Soot [PLH11] to directly read Dalvik bytecode and perform analysis and/or transformation on it's internal Jimple representation. Therefore, this Soot modification is able to analyze Android applications, since they are distributed as Dalvik bytecode.

Dexpler starts by mapping Dalvik bytecode instructions and registers to Jimple statements and Jimple local variables, respectively. Dexpler leverages the information provided by *dedexer* Dalvik bytecode disassembler to generate Jimple classes, methods and statements. Then, it uses Soot fast typing Jimple component to infer the type of the local variables. However, the Soot component sometimes is not capable of inferring the type for local variables because some instructions do not provide enough information. This is the case with null initialization instructions (cannot determine if it is zero or null) and numeric constant initialization instructions (in 32 bits cannot determine if it is integer or float and in 64 bits it cannot determine if it is long or double). Therefore, to infer the type for these ambiguous instructions they implemented an algorithm that performs depth first search in the control flow graph of Jimple statements to find out how the declared local variable is used. This algorithm exposes the type of the local variable when it parses the following statements: comparison with a known type, instructions that operate only in specific types, non void return instructions and method invocation.

When the tool comes across with a Dalvik branch instruction, a Jimple jump instruction is generated and its target is received by fetching the Jimple statement mapped to the Dalvik branch instruction target's address. On every Jimple method, they insert a nop instruction as

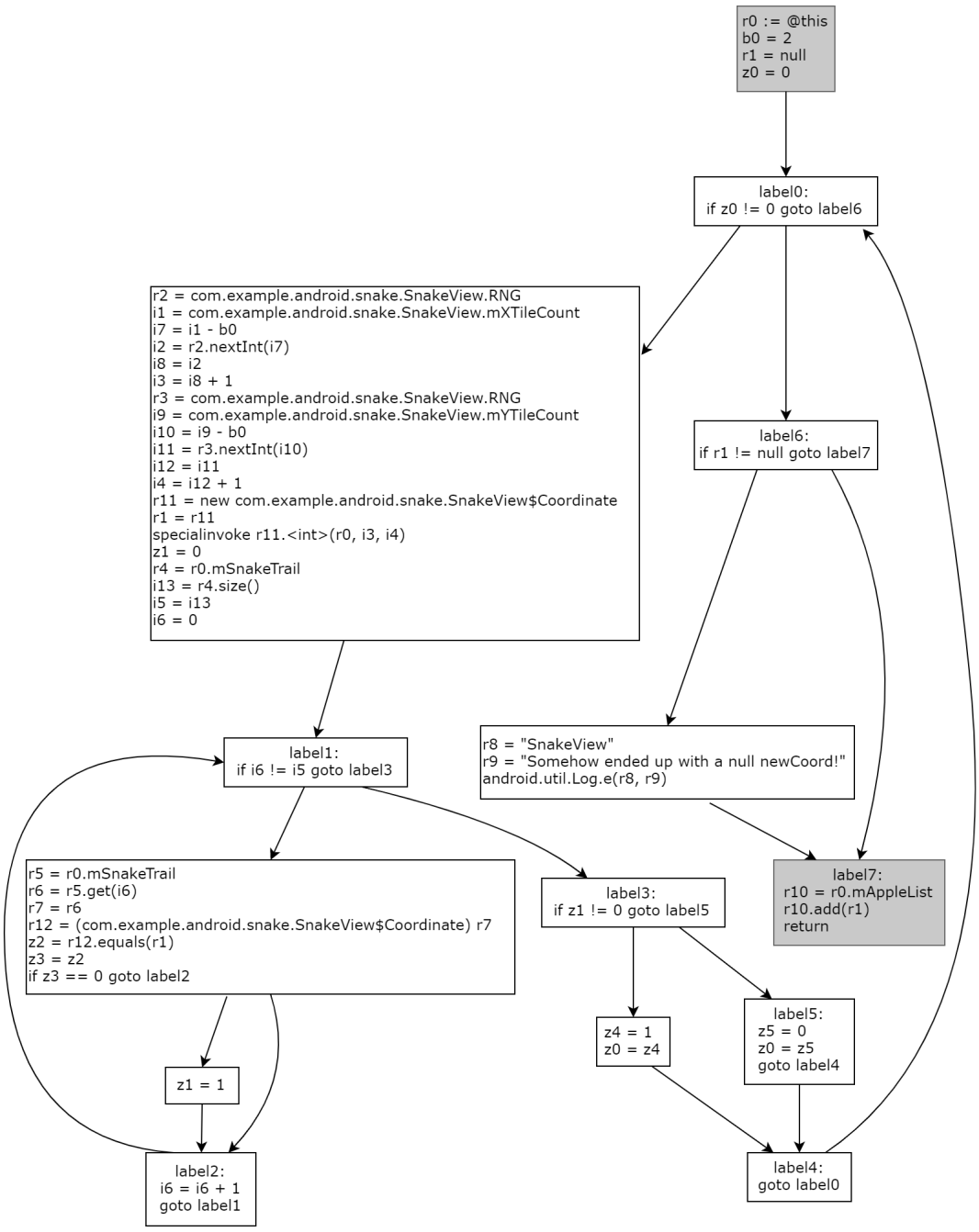


Figure 2.10: Control Flow Graph for addRandomApple Method [ABM12]

the first instruction. Thereby, if the first Dalvik instruction is a jump or if the jump's target correspond to a non-yet generated Jimple statement, they redirect it to the nop instruction. The tool corrects the Jimple jump instructions once the whole Dalvik bytecode of the method has been processed. During the Jimple optimization step, the previously added Jimple nop instruction is removed.

Dexpler was evaluated with test cases and an Android application called Snake. They executed all the test cases with generated Dalvik bytecode and the tool produced the expected results. The sample application includes 11 classes, 39 methods and it was written in 550 lines of code. They generate Jimple code from the Dalvik bytecode of the Snake application. Then from the Jimple code, Soot generates Java bytecode. Finally from the generated Java bytecode, they reassembled the application to Dalvik, launched it on the android emulator and verified that the game was correctly working. Furthermore, they generated a call graph of the Snake application and a control flow graph of the method *AddRandomApple* as shown in figure 4. After analyzing the produced call graph and CFG, they concluded that it correctly corresponds to the meaning of the original code.

Chapter 3

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:

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

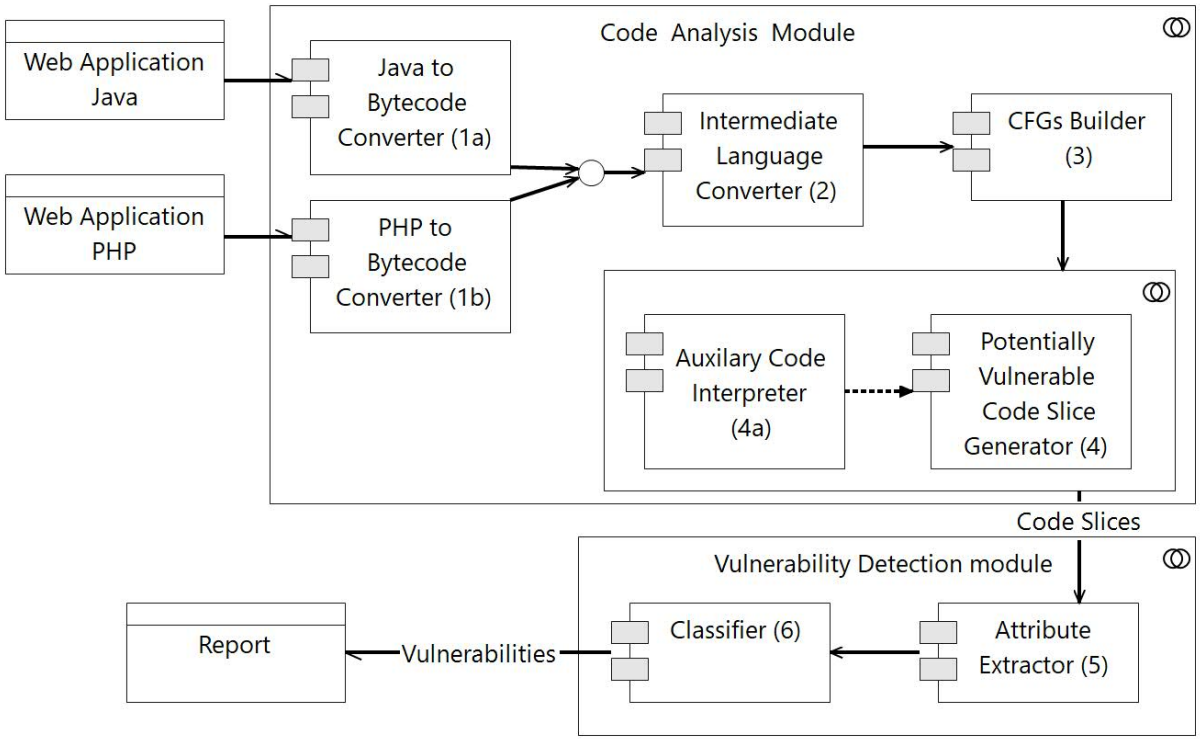


Figure 3.1: Architecture of the MERLIN tool (for PHP and Java code)

3.1 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 [jph] when processing PHP code. JPHP's original purpose was not to compile Java bytecode, but to execute bytecode. Therefore, we had to make some changes in the tool to be able to dump the produced Java bytecode. Furthermore, since the execution of the resulting bytecode was not relevant for the scope of our work, we also changed JPHP to not execute the bytecode. This procedure required us to understand how JPHP generated Java bytecode in order to be able to make the necessary changes. 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. Since JPHP is also able to execute the resulting bytecode, the tool modifies the format of some instructions to be able to correctly interpret and consequently execute the resulting Java bytecode. 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

```
$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

Figure 3.2: Translation of the 3rd line of the code in Figures 2.1 and 2.2

tool.

Then, the Soot framework [PLH11] analyzes and converts Java bytecode into Jimple, a typed 3-address intermediate code representation. Each source code instruction is broken down into several separate Jimple instructions; variable names are changed and temporary symbols are generated. It is a called a 3-address intermediate code because an instruction has at most three operands and one operator. A translation example of the third line of code from Figures 2.1 and 2.2 respectively into Jimple code is shown in the Figure 3.2. In this figure, $r45 generated from PHP code and $r12 generated from Java code are examples of temporary symbols. In the case of PHP, the JPHP tool transforms the *mysqli_query* function into the *jphp.runtime.invoke.InvokeHelper.call* function. The essential arguments of the *jphp.runtime.invoke.InvokeHelper.call* function to be able to correctly analyze the data flow of the original function are: the third and the fourth which indicate the original function name; and the fifth which represents an array composed of arguments of the *mysqli_query* function. The remaining arguments are important for JPHP to be able to execute the compiled Java bytecode, but do not add any relevant information for our analysis. In the case of Java, the source code instruction was separated into two simpler Jimple code instructions.

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 de-

```
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

Figure 3.3: Examples of instructions in the simplified representation (resulting from the code in Figures 2.1 and 2.2)

clared within a class defined in the generated Java bytecode.

3.2 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 successively parsed by MERLIN. During parse, the tool interprets the tokens, making it possible to properly process each Jimple instruction found. This way, MERLIN is able to interpret all Jimple instructions and convert them into a simplified representation, that, in turn, are translated into an attribute. When

Table 3.1: Entry points, sanitization functions and sensitive sinks necessary to detect SQL injection vulnerability in Figures 2.1 and 2.2

Vuln	Language	Entry point	Sanitization function	Sensitive sink
SQLi	PHP	$_GET	mysqli_escape_string mysqli_real_escape_string	mysqli_query
	Java	HttpServletRequest.getParameter	StringEscapeUtils.escapeSql	Statement.executeQuery

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 3.3.

A simplified representation of a instruction is composed of two parts: instruction tag and name. The instruction tag can have the following values: *cast*, *object*, *function*, *if*, *constant*, *array*, *parameter*, *source*, *throw* and *return*. In the case of the examples of the Figure 3.3, the first instructions will receive the instruction tag *source* and last instruction will be *function*. The name is used to indicate function, object, array, constant, parameter, or global variable names (which can correspond to entry points). In the event that an instruction is a cast, if, through or a return, it is not specified a name. In the PHP example, the name given in the first line belongs to the super global variable $_POST. Whereby, in the Java example, it is specified the function name that corresponds to an entry point.

In order to detect potential vulnerabilities, MERLIN needs a configuration file. The configuration file includes variables and functions that correspond to entry points, sanitization functions and sensitive sinks for the source language. For instance, in order to detect the vulnerabilities in Figures 2.1 and 2.2, MERLIN requires a configuration file for each programming language with the information specified in the Table 3.1. The Table 3.1 contains the entry point ($_GET in PHP and *HttpServletRequest.getParameter* in Java), sanitization functions (*mysqli_escape_string* and *mysqli_real_escape_string* in PHP and *StringEscapeUtils.escapeSql* in Java) and the sensitive sink (*mysqli_query* in PHP and in *Statement.executeQuery*Java). 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 and classes from Java and PHP. For instance, the tool is capable of correctly processing functions and classes that create and manipulate arrays (e.g, *array* function in PHP and *ArrayList* class in Java). Supporting the processing of these functions and classes was quite challenging. One of the challenges found was to correctly process the different methods of adding, removing and replacing

elements. Another challenge found is related with static analysis. When the approach uses static analysis, it is not possible to determine the value of some variables, since they vary with the execution of the program. When undefined variables are used to access the indexes of these structures, it is not possible to determine with certainty the result of these methods, which can affect accuracy. In such cases, the type of processing that minimizes the lost of information is always privileged, for example no element is deleted when the specified index to be removed is an undefined variable. 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. As a result the tool performs a more complete and accurate analysis.

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.

3.3 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 with Turing's Halting problem [Lan92]. 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.

3.3.1 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 [MNC14b]. 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 fundamental since a vulnerability only exists if the code slice has an entry point.
- 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 white spaces. In some cases, these functions can untaint strings depending on how they are used and on the vulnerability considered.

Table 3.2: Attribute Examples (part of Java name functions are omitted due to space constraints)

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	

- 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 valid when the tool is dealing with SQL injection vulnerabilities. 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 input is converted to another format, it may no longer pose a threat to a web application.

- Others: this category includes instructions that are capable of untainting data by the way they condition data flow (using ifs), by using operators that perform automatic type conversions or by performing type casting. The user of the tool does not need to specify which instructions should be considered in this category, as the tool is already programmed to deal with these instructions regardless of the programming language in which the web

application is written.

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 3.2.

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.

3.3.2 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.

3.3.3 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 is the true positives and FP is 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 is the false negatives. *F-score* is a harmonic mean of the precision and recall metrics, and it is computed according to the following formula:

$$F - score = 2 * P * R/(P + R)$$

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 is 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.

3.3.4 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 [Nat]. 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

Table 3.3: 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

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 from 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 3.3. 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 3.4. 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

Table 3.4: 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

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.

3.4 Implementation

We developed a script that performs all stages of the approach automatically. The script was developed in Python in a Windows environment. This script can be executed from the command line. Our script can receive as input a single file or a folder with several files. It is also possible to specify the programming language in which the web application is written. The script also accepts as input a file or a web application already compiled in Java bytecode. However, if the source code of the web application is provided, the script transforms each file of the web application into Java bytecode. It is used the appropriate bytecode converter for the programming language in which is written the web application. Our script uses the compiler javac when the application is written in Java and JPHP when is written in PHP. Many Java web applications are written using Maven, therefore our script also supports the execution of maven in order to be able to produce Java bytecode from them. Furthermore, Java web applications built upon Maven often have *jsp* files which can execute Java code. These files can also contain vulnerabilities and put the web application at risk. Thus, the script analyzes the file pom.xml of the Java web application to verify whether the Jetty Jspc Maven Plugin is used. In the case it is not used, our script suggests adding the plugin dependence to the pom. This plugin generates

Java bytecode from jsp files. So by using this plugin, the tool can generate Java bytecode from jsp files and thereby, MERLIN can perform a more complete analysis of the web application.

Next, all files generated with Java bytecode are analyzed by Soot. Soot produces Jimple code and CFGs of the web application. Soot produces multiple CFGs, as it takes into account the various exceptions that can occur during the execution of the program. Because of performance reasons and since the remaining CFGs do not add any valuable data for the detection of the vulnerabilities, we only consider the CFG corresponding to the normal flow, without exceptions. Soot produces CFGs in *dot* format.

Then, the Potentially Vulnerable Code Slice Generator together with the Auxiliary Code Interpreter, both developed in Java, will analyze the Jimple code together with the CFGs to search for potentially vulnerable code slices. This stage requires to provide two configuration files: a file with the sources, sanitization functions and sensitive sinks; and a file with the attributes to be considered. We created the configuration files for Java and PHP, which required extended research. The list of entry points, sanitization functions and sensitive sinks regarding each class of vulnerability used in the implementation of MERLIN is found in the Appendix A. It should be noted that as a preventive measure, we chose to consider as sources, the input from files and the command line, since we do not know whether it is tainted or not.

The module Attribute Extractor, that was also developed in Java, generates attribute vectors from potentially vulnerable code slices. The attribute vectors are produced in *csv* format. Finally, the machine learning classifier implemented in Python uses the attribute vectors to classify the code as vulnerable or not vulnerable. In case a vulnerability is detected, the file and the method where the vulnerability is found, as well as the class of the vulnerability detected, is indicated in the command line.

Chapter 4

Evaluation

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

The evaluation involved the comparison with two other tools and was based on the following code bases, adding to more than 700 thousand lines of code:

- a set of 13 code samples that we created on purpose for testing the tool (8 written in PHP and 5 written in Java);
- two hundred code samples from the SRD database (100 written in PHP and 100 written in Java);
- 12 real world web applications (8 written in PHP and 4 written in Java).

4.1 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. The web applications were designed by us especially for this purpose. An example of two code samples vulnerable to SQLi are shown in Figure 2.1 and 2.2. The tool was able to correctly identify the vulnerabilities in both code samples, so the answer to the first question is positive.

Table 4.1: 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

4.2 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 - DVWAP, Mutillidae, bWAPP, WackoPicko, Java Vulnerable Lab, HackMe; and, a data set that included 6 widely used web applications - MantisBT, phpMyAdmin, DokuWiki, MISP, Pinpoint and Spring OAuth2.

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 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 4.1. 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

Table 4.2: Taint nalysis of Real World Web Applications with seeded vulnerabilities

webapp	language	#loc	#files	#TP	#FP	#FN	P (%)	R (%)	F-Score (%)
DVWAP	PHP	14,895	353	25	10	2	71.43	92.59	80.65
Mutillidae	PHP	142,515	919	49	17	39	74.24	55.68	63.63
bWAPP	PHP	24,070	198	336	0	264	100	56	71.79
WackoPicko	PHP	1,916	48	17	2	2	89.47	89.47	89.47
Java Vulnerable Lab	Java	1,795	60	73	30	5	70.87	93.59	80.66
HackMe	Java	824	17	33	0	12	100	73.33	84.61
Total/Avg	Java+PHP	186,015	1,595	533	59	328	84.34	76.77	78.47

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 these set of web applications with taint analysis, instead of using a machine learning classifier to identify vulnerabilities. In this case, the tool always a reported a vulnerability whenever there was a sensitive sink, an entry point and there was no sanitization function. The results obtained during this analysis are shown in the Table 4.2. These results are slightly better when compared to the results obtained with the machine learning classifier. These results show that the classifier sometimes classifies as vulnerable, code that contains an entry point and a sanitization function. However, it also shows that it correctly classifies as non vulnerable, code that uses a set of operation capable of untaint input.

We also tested the tool with a set of widely used open source web applications. These web applications belong to a database called *Secbench* [RA17]. 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 4.3. 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.

Table 4.3: 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

4.3 Comparison with other tools

We selected two tools – WAP [MNC14a] and Achilles [SDD+19] – 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 [SDD+19]. 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 otherwise 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 4.4. 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%. However, it should be noted that the selected

Table 4.4: 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

```
$tainted = $_GET['UserData'];
if (filter_var($tainted, FILTER_VALIDATE_INT))
    $tainted = $tainted;
else
    $tainted = "";
$var = require("pages/'" . $tainted . "'.php");
```

(a) Safe code sample from SRD database

```
$tainted = $_GET['UserData'];
if (filter_var($tainted, FILTER_VALIDATE_EMAIL))
    $tainted = $tainted;
else
    $tainted = "";
$var = require("'" . $tainted . "'.php");
```

(b) Unsafe code sample from SRD database

Figure 4.1: Examples of code samples tested by MERLIN and WAP

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. Another factor that has impact on the obtained results is related with functions that in some cases can untaint instructions and, in others, they cannot. Figure 4.1 shows code samples from the SRD database that were processed by MERLIN and WAP. The code sample in Figure 4.1.(a) is not vulnerable to file disclosure and the code sample in Figure 4.1.(b) is vulnerable. However, the attribute vectors generated by MERLIN are the same in both cases, since there is only a change on the parameter used in the function *filter_var*. Therefore, MERLIN classifies both code samples as vulnerable. Instead, WAP classifies these code samples as not vulnerable, since the tool aims to reduce the number of false positives.

Chapter 5

Conclusions

This document presents an approach to improve security in web applications, by detecting vulnerabilities in web applications written in different languages using machine learning. This approach includes the following steps: converting source code to intermediate code, analyzing intermediate code, extracting features from potentially vulnerable code slices and classifying them as vulnerable or not vulnerable. The detection of potentially vulnerable code is performed at an intermediate level of the code and therefore the tool is not specific to a high-level language. Moreover, the identification of vulnerabilities is performed automatically by the machine learning classifiers. The implemented approach was evaluated with code samples from SRD database and real world web applications written in Java and PHP. The performed evaluation shows that the tool is capable of detecting different types of vulnerabilities in both languages. Furthermore, it also shows it can detect vulnerabilities in real world web applications.

5.1 Future Work

The assessment carried out on the tool showed promising results. However, there are still a few aspects in the developed tool that can be improved and that were not implemented due to time constraints.

It would be interesting to test the tool with web applications written in more different programming languages, such as Python, Scala, JavaScript, among others. This would make the tool more versatile and useful in the area of web development, where several programming languages are used. In addition, it further completed the demonstration that MERLIN is a static analysis tool capable of detecting vulnerabilities in multiple languages, which means, it can process web application regardless of the programming language in which they are written.

Another work that can be done is to extend the classes of input validation vulnerabilities

supported by MERLIN. This, in turn, will make the tool even more complete. To support more classes of vulnerabilities, it is only necessary to provide sensitive sinks, sanitization functions and entry points. This can show once again that MERLIN is able to detect new classes of vulnerabilities without having to program knowledge about them.

Although the tool demonstrates that it is capable of detecting vulnerabilities, improvements can still be done to achieve better results. It can be verified if there is another machine learning algorithm, such as a deep learning algorithm, capable of learning more accurately what are the instructions associated with the presence of the vulnerability. The algorithm used to analyze the data flow of CFGs can also be improved. If we improve the scalabilty of the algorithm, without loosing accuracy, the tool will obtain significantly better results when analyzing more complex web applications.

Bibliography

[ABM12] J. Klein A. Bertel and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012.

[aIMAN20] A. Fidalgo annd I. Medeiros, P. Antunes, and N. Neves. Towards a deep learning model for vulnerability detection on web application variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 465–476, 2020.

[BBMV07] Sruthi Bandhakavi, Prithvi 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*, pages 12–24, October 2007.

[BK14] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. In *ACM SIGPLAN Notices 49.10*, 2014.

[che] Checkmarx. `https://www.checkmarx.com/`.

[CKS+11] 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*, pages 101–114, 2011.

[CM04] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[DCKV12] 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.

[DCV10] 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*, pages 111–131, 2010.

[DH14] Johannes Dahse and Thorsten 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.

[E. 08] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla and D. Zanardini. Termination analysis of java bytecode. In *International Conference on Formal Methods for Open Object-Based Distributed Systems (pp. 2-18)*, 2008.

[GIW06] 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.

[GS05] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2005.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[HP11] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach.* Elsevier, 5th edition, 2011.

[HVO06] 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.

[JKK06] 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*, pages 258–263, 2006.

[JLH04] O. Lhoták J. Lhoták and L. Hendren. Integrating the soot compiler infrastructure into an IDE. In *13th International Conference on Compiler Construction*, pages 281–297, 2004.

[jph] JPHP. `https://github.com/jphp-group/jphp`.

[JSMHB13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[ker] Keras. `https://keras.io/`.

[KVR05] C. Kruegel, G. Vigna, and W. Robertson. A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5):717–738, 2005.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LF08] F. Logozzo and M. Fahndrich. On the relative completeness of bytecode analysis versus source code analysis. In *International Conference on Compiler Construction*, 2008.

[Lho02] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, 2002.

[Lho06] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[LL05] V. Livshits and M. Lam. Finding security vulnerabilities in java applications with static analysis. In *SENIX Security Symposium. Vol. 14*, 2005.

[LN] J. Lind-Nielsen. Buddy - a binary decision diagram package. `www.itu.dk/research/buddy/`.

[LN01] 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*, pages 191–200, 2001.

[LZX+18] 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.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

[Med16] I. Medeiros. *Detection of Vulnerabilities and Automatic Protection for Web Applications*. PhD thesis, Faculdade de Ciências, 2016.

[Min05] Y. Minamide. Approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441, May 2005.

[MM14] M. Meucci and A. Muller. OWASP testing guide 4.0. Technical report, OWASP Foundation, 2014.

[MNC14a] Ibéria Medeiros, Nuno F. Neves, and Miguel 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*, pages 63–74, April 2014.

[MNC14b] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Website of WAP tool, January 2014. http://awap.sourceforge.net/.

[MNC16] 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.

[MSB05] L. Shan M. Sridharan, D. Gopan and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, systems, languages, and applications*, pages 59–76, 2005.

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

[NB05] 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.

[NFV15] P. Nunes, J. Fonseca, and M. Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[NMF+18] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[NTGG+05] 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*, pages 295–307, 2005.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.

[PLH11] O. Lhoták P. Lam, E. Bodden and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS)*, 2011.

[RA17] S. Reis and R. Abreu. SECBENCH: A database of real security vulnerabilities. In *International Workshop on Secure Software Engineering in DevOps and Agile Development*, pages 69–85, 2017.

[SDD+19] 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*, pages 114–121, 2019.

[ST12a] Lwin Khin Shar and Hee Beng Kuan Tan. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1293–1296, 2012.

[ST12b] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting common web application vulnerabilities from input validation and sanitization code patterns. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 310–313, 2012.

[SW06] 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*, pages 372–382, January 2006.

[TPC+13] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg*, pages 210–225, 2013.

[Uma06] N. Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, 2006.

[VVB+09] 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*, 17(3):305–329, 2009.

[WS08] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[WW17] J. Williams and D. Wichers. OWASP Top 10 - 2017 - the ten most critical web application security risks. Technical report, OWASP Foundation, 2017.

Appendix A

Entry points, sensitive sinks and sanitization functions

This appendix shows the entry points, sensitive sinks and sanitization functions considered during the execution of the tool to detect vulnerabilities in web applications.

Table A.1: Entry points, sensitive sinks and sanitization functions used to analyze PHP web applications

Entry points	Vulnerabilities	Sensitive sinks	Sanitization Functions
$_GET, $_POST, $_COOKIE, $_REQUEST, HTTP_GET_VARS, HTTP_POST_VARS, HTTP_COOKIE_VARS, HTTP_REQUEST_VARS, $_FILES, $_SERVERS, $_SERVER, $_SESSION, shell_exec, exec, fgets, fread, stream_get_contents, system	SQL Injection	**MySQL**	
		mysql_query, mysql_unbuffered_query, mysql_db_query, mysqli_query, mysqli_real_query, mysqli_master_query, mysqli_multi_query, mysqli::query, mysqli::multi_query, mysqli::real_query	mysql_escape_string, mysql_real_escape_string, mysqli_escape_string, mysqli_real_escape_string, mysqli::escape_string, mysqli::real_escape_string
		DB2	
		db2_exec	db2_escape_string
		PostgreSQL	
		pg_query, pg_send_query	pg_escape_string, pg_escape_byte
	Remote File Inclusion Local File Inclusion Directory Traversal/ Path Traversal	fopen, file_get_contents, file, copy, unlink, move_uploaded_file, imagecreatefromgd2, imagecreatefromgd2part, imagecreatefromgd, imagecreatefromgif, imagecreatefromjpeg, imagecreatefrompng, imagecreatefromstring, imagecreatefromwbmp, imagecreatefromxbm, imagecreatefromxpm, require, require_once, include, include_once	
	Source Code Disclosure	readfile	
	OS Command Injection	passthru, system, shell_exec, exec, pcntl_exec popen	escapeshellarg
	Cross site scripting	echo, print, printf, die, trigger_error, exit, file_put_contents, user_error	htmlentities, htmlspecialchars, strip_tags, urlencode
	PHP Code Injection	eval	

Table A.2: Entry points, sensitive sinks and sanitization functions used to analyze Java web applications (part of Java name functions are omitted due to space constraints)

Entry points	**Vulns**	**Sensitive sinks**	**Sanitization Functions**
HttpServletRequest.getParameter, HttpServletRequest.getParameterMap, HttpServletRequest.getParameterNames, HttpServletRequest.getParameterValues, HttpServletRequest.getCookies, HttpServletRequest.getHeader, HttpServletRequest.getHeaderNames, HttpServletRequest.getHeaders, HttpServletRequest.getQueryString, ResultSet.getString, BufferedReader.readLine, Properties.getProperty, URLConnection.getInputStream, System.getProperty	SQLI	Statement.execute, Statement.executeQuery, Statement.executeUpdate, Statement.addBatch, Connection.prepareStatement	StringEscapeUtils.escapeSql
	RFI LFI FI DT / PT SCD	ImportTag.setUrl, HttpServletResponse.sendRedirect, io.FileReader, io.File, jspRuntimeLibrary.include, Scanner.next, Files.readAllLines, Files.readAllBytes, io.FileInputStream	
	OSCI	ProcessBuilder.command, Runtime.exec	
	XSS	JspWriter.println, JspWriter.print, PageContextImpl.proprietaryEvaluate, PrintWriter.println, PrintWriter.print, PrintWriter.printf, HttpServletResponse.sendError	StringEscapeUtils.escapeHtml3, StringEscapeUtils.escapeHtml4, Jsoup.clean, Document.text, StringEscapeUtils.escapeHtml, Cleaner.clean
	PHPCI	-	-