Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Cyber Security

by

Steven M. Kelbley
B.S. Biological Sciences, Wright State University, 2012

2016
Wright State University
I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Steven M. Kelbley ENTITLED Detecting PHP-based Cross-Site Scripting Vulnerabilities Using Static Program Analysis BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

Junjie Zhang, Ph.D.
Thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Adam Bryant, Ph.D.

Robert E. W. Fyffe, Ph.D.
Vice President for Research and Dean of the Graduate School
ABSTRACT


With the widespread adoption of dynamic web applications in recent years, a number of threats to the security of these applications have emerged as significant challenges for application developers. The security of developed applications has become a higher priority for both developers and their employers as cyber attacks become increasingly more prevalent and damaging.

Some of the most used web application frameworks are written in PHP and have become major targets due to the large number of servers running these applications worldwide. A number of tools exist to evaluate PHP code for issues, however most of these applications are not targeted at vulnerability detection. At the same time, Cross-Site Scripting (XSS) vulnerabilities continue to be identified in existing software threatening the security of client data. Providing tools to software developers which can identify these XSS vulnerabilities in code during the development process could reduce the number of vulnerabilities that make it into production code and thus threaten users.

This thesis proposes a solution for the problem of identifying non-persistent XSS vulnerabilities in PHP code by demonstrating a system which is capable of finding these vulnerable code paths. This is achieved through the use of static taint analysis, whereby a number of known sources of untrusted data are defined, along with several sensitive sinks which may present a vulnerability if untrusted data is used at these locations. Any data acquired from these taint sources and subsequent propagation of the data is tracked.

Code analysis is performed on an Abstract Syntax Tree (AST), an intermediate representation which permits conversion to and from source code. This allows individual line numbers to be tracked for the purpose of clearly displaying taint flow to the user allowing them to visualize how the information flow could result in an unsafe condition and take
appropriate action to remedy the vulnerability.

This program is capable of analyzing non-object-oriented PHP code and supports most of the common language constructs. Initial testing has shown the program to be highly successful at identifying non-persistent XSS attacks in the supported subset of the PHP language, with future development efforts targeting expanded support for more elements of the language including object-oriented programming.
Contents

1 Introduction ......................................................... 1
  1.1 Motivation .................................................... 1
  1.2 Background .................................................. 2
    1.2.1 Static Analysis ........................................ 2
    1.2.2 Dynamic Analysis ...................................... 3
    1.2.3 Taint Analysis .......................................... 3
    1.2.4 Cross-Site Scripting (XSS) ............................. 4
  1.3 Goals .......................................................... 5
  1.4 Organization .................................................. 6

2 Related Work ....................................................... 7

3 Design ............................................................. 11
  3.1 Problem Formulation ......................................... 11
  3.2 Intermediate Representation ................................ 11
  3.3 Syntax Rules .................................................. 12
  3.4 Taint Flow Analysis ......................................... 13
    3.4.1 Sources ................................................ 13
    3.4.2 Sinks .................................................. 14
    3.4.3 Sanitizers .............................................. 14
    3.4.4 Approximation ........................................ 15
  3.5 Evaluating Nodes ............................................. 20

4 Implementation .................................................... 24
  4.1 Variable Taint ............................................... 24
  4.2 Function Definitions ....................................... 27
  4.3 Taint Propagation ........................................... 27
    4.3.1 Taint Propagation Functions .......................... 28
    4.3.2 Hard-coded Expressions ................................. 29
    4.3.3 Variable Assignments .................................. 29
    4.3.4 Binary Operations ..................................... 30
    4.3.5 Loops .................................................. 31
List of Figures

5.1 Effect of AST Node Count on Running Time ............... 46
List of Tables

3.1 PHP Syntax ............................................................... 12
3.2 Expression Syntax Rules ............................................. 21
3.3 Statement Syntax Rules .............................................. 21
4.1 Variable Taint Data Structure ................................. 26
5.1 Taint Flow ................................................................. 46
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example XSS Vulnerability</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Taint Overapproximation</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Branch Execution with Default</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Example Code</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Example AST</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Variable Taint Data Structure</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Sample Taint Generation</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Assignment Operation Example</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Assignment Operation Evaluation</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Binary Operation Example</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Binary Operation Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>While Loop Example</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>While Loop Evaluation</td>
<td>32</td>
</tr>
<tr>
<td>4.9</td>
<td>For Loop Example</td>
<td>32</td>
</tr>
<tr>
<td>4.10</td>
<td>For Loop Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>4.11</td>
<td>Foreach Loop Example</td>
<td>34</td>
</tr>
<tr>
<td>4.12</td>
<td>Foreach Loop Evaluation</td>
<td>34</td>
</tr>
<tr>
<td>4.13</td>
<td>If-Else Conditional Example</td>
<td>35</td>
</tr>
<tr>
<td>4.14</td>
<td>If-Else Conditional Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>4.15</td>
<td>Switch Example</td>
<td>37</td>
</tr>
<tr>
<td>4.16</td>
<td>Switch Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>4.17</td>
<td>Function Example</td>
<td>39</td>
</tr>
<tr>
<td>4.18</td>
<td>Function Evaluation</td>
<td>39</td>
</tr>
<tr>
<td>4.19</td>
<td>Sinks Example</td>
<td>40</td>
</tr>
<tr>
<td>4.20</td>
<td>Sinks Evaluation</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Example 1 Program Output</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Example 2</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>PHP Array Gaps</td>
<td>48</td>
</tr>
<tr>
<td>A.1</td>
<td>Program Code</td>
<td>54</td>
</tr>
<tr>
<td>B.1</td>
<td>Example 1</td>
<td>76</td>
</tr>
</tbody>
</table>
Acknowledgment

I would like to take this opportunity to extend my thanks to my advisor, Dr. Junjie Zhang, for his guidance and mentoring while I completed this thesis. His experience and expertise has been invaluable in assisting my work. He has greatly contributed to my education at Wright State and I appreciate his support. I would also like to thank Dr. Krishnaprasad Thirunarayan and Dr. Adam Bryant for volunteering their time to serve on my thesis committee and sincerely appreciate their input and expertise in evaluating this work. I am thankful for the time everyone has invested to assist me in this final chapter of my graduate education. I would like to thank my family for their support during my academic career and helping provide me with the opportunity to study at Wright State University. Finally, I would like to thank my wife for supporting me while I pursued further education and for tirelessly believing in me.
Abbreviations

AST — Abstract Syntax Tree
CFG — Control Flow Graph
CMS — Content Management System
DBMS — Database Management System
HTML — Hypertext Markup Language
IR — Intermediate Representation
IT — Information Technology
OOP — Object Oriented Programming
PoC — Proof-of-Concept
PHP — PHP Hypertext Preprocessor
SQL — Structured Query Language
SQLI — SQL Injection
XSS — Cross-Site Scripting
Introduction

In this thesis, a solution is proposed that can analyze non-object-oriented PHP code for reflected Cross-Site Scripting (XSS) vulnerabilities. Specifically, we will discuss the implications of XSS vulnerabilities in PHP code and ways in which malicious actors can inject unsafe scripts into pages as a result of poor coding practices. We will demonstrate the use of an abstract syntax tree (AST) in our analysis of code to allow feedback to be displayed in a human-readable manner, allowing programmers to rapidly identify the code which permits the XSS attack so that it can be corrected.

1.1 Motivation

An ever-increasing number of critical applications are moving to web-based systems as developers attempt to create applications that can be utilized on a wide range of devices via a web browser. As a consequence, this broad adoption has brought increased attention to the vulnerabilities in these programs that can be exploited by malicious users[1]. The OWASP Foundation [2] analyzed over 500,000 vulnerabilities in 2013 to discern what types were most commonly exploited, and determined that SQL injection (SQLI), issues with Authentication and Session Management and Cross-Site Scripting (XSS) were the top three threats to web applications. Scott and Sharp[3] contend that web application vulnerabilities are inevitable regardless of the language used and the implementation of the various services on which the application relies.

Among the most popular languages for web applications is PHP, which was used on
over 240 million sites as of January 2013 [4]. Many high-traffic web applications and web development frameworks (and their plugin modules) are written in PHP, including Wordpress\(^1\), Drupal\(^2\), Laravel\(^3\) and CodeIgniter\(^4\). Wordpress alone accounts for over 73 million sites on the internet with over 300 million unique visitors each month[5]. However, the widespread use of frameworks like Wordpress has attracted malicious individuals due to the uniformity of these installations and the inevitable delay between security patches being committed and system administrators deploying the patches.

With such a high number of websites running on programs and frameworks written in PHP, static analysis of PHP code can prove to be a valuable tool to the developers of these software programs. The rapid identification of vulnerabilities in code is paramount to the security of these sites and presents a major challenge for developers and system administrators. In the current cybersecurity climate, it is critical to ensure web applications can securely process and store client data.

### 1.2 Background

Several analytical techniques are utilized in this work. Below we provide some background information on these approaches.

#### 1.2.1 Static Analysis

Static analysis is the term that describes the examination of program code without executing the program[6]. This approach can evaluate all of the code in a program for possible vulnerabilities and provides a thorough approach for identifying issues that human code inspection may fail to detect. As the analysis is not dependent on a single execution.
path, it is possible to evaluate all branches individually. Static analysis serves as a jumping-off point for using further analytical methods to solve the problem at hand.

1.2.2 Dynamic Analysis

Dynamic analysis is an analytical technique in which a program is run with test inputs and the execution is monitored for interesting (or undesirable) output or behavior[7]. Since the outputs are dependent on the inputs, identification of vulnerabilities is dependent on generating test inputs that reach the code path on which the vulnerability lies; therefore, the careful creation of test inputs is required to ensure sufficient code coverage. Additionally, any instrumentation code added for the purpose of performing dynamic analysis could potentially cause problems in programs where time-sensitive operations occur. For these reasons, static analysis was chosen as we seek to evaluate as much of the code as possible for vulnerabilities.

1.2.3 Taint Analysis

In this thesis, we use static analysis to evaluate program flow paths in source code. The goal of this analysis is to identify points where execution could potentially result in tainted input (data retrieved from an untrusted source) being displayed within the client browser[8]. There are multiple representations in which the program could be analyzed: source code, abstract syntax trees and various intermediate representations all provide valuable information about how the program functions.

Parsing source code allows easy identification of culprit lines of code, however this requires a significant amount of work to handle the specific syntax of the PHP language. Intermediate representations (IR) permit simple flow analysis and are commonly used by compilers for this purpose, but converting the IR back to source code may result in different syntax than was used in the original program and could make it difficult for users to find errors in the original source code. Abstract syntax trees represent the source code as a tree
data structure which abstracts away specific syntax elements of the code, such as grouping of parentheses, and allows for easy traversal of the tree (and thus the source code). Additionally, the AST is readily converted back to the exact original source code, making it easy to point out specific line numbers for vulnerabilities. As such, we elected to use an AST-based approach.

Taint flow analysis allows us to track the flow of this tainted data through the AST (and thus the program), identifying where it is generated, where it is transferred to other data structures, and where it may be used in sensitive operations such as outputting to the screen. Due to the nature of information flow analysis and the use of a single trace of the execution, the results are an approximation of the data flow. However, this does not diminish the value of the data output by the program, as it would still be capable of clearly identifying sensitive sinks for data and which taint sources generated the untrusted data in question, allowing users to see the problem and determine how best to fix the vulnerability.

1.2.4 Cross-Site Scripting (XSS)

Cross-site scripting vulnerabilities have been a major threat to web applications for a number of years, ranking at second and third on the list of most critical vulnerabilities in 2010 and 2013, respectively[9]. These vulnerabilities pose a risk to clients, as they can trick users into executing untrusted code by luring them into clicking a link that contains parameters which cause code to be printed back to the page displayed on the client. This code can then cause adverse effects on the client ranging from minor annoyance to major security vulnerability, depending on the desired effect of the exploit’s author(s)[10].

XSS attacks provide an avenue by which attackers can bypass the same-origin policy implemented by web browsers which prevents scripts loaded from one origin (a combination of hostname, port and Uniform Resource Identifier) from accessing data used by scripts from another origin[11]. An XSS attack would appear to be coming from the same origin as the site with the vulnerability, and would therefore be permitted to access cookies
and other data stored by the target site.

A simple example of vulnerable code is shown below in Listing 1.1. Two values are retrieved from the PHP built-in array $_POST at the indices ‘name’ and ‘zip_code’. If the user has supplied a ZIP code, then they are thanked by name; otherwise, they are informed that the ZIP code field is required.

Listing 1.1: Example XSS Vulnerability

```php
$name = $_POST['name'];
$zip = $_POST['zip_code'];
$response = '';
if($zip){
    $response = "Thank you for your submission, " . $name . "\n";
} else {
    $response = "Your ZIP code is required\n";
}
echo $response;
```

Preventing XSS attacks entails sanitizing data before it is returned to the client browser for output. In the case of the example above, if the ‘$response’ variable had been provided to a sanitization function like ‘htmlspecialchars’, any code that was passed in via the ‘name’ parameter would be displayed on the page for the user in HTML-escaped form; the <script> tag would be converted to &lt;script&gt; and the script would no longer be valid Javascript capable of being executed.

1.3 Goals

In this thesis, we designed and implemented a system which is capable of finding reflected XSS vulnerabilities by performing static analysis on PHP code using taint analysis to identify potentially unsafe operations on input data. As much of the PHP language as possible should be handled to create an accurate representation of the data flow of the target
program, excluding operations that would not play a direct role in reflected XSS attacks (e.g. file operations or database queries). Excluding OOP code (for reasons mentioned previously) and these ‘unrelated’ operations, a high percentage of commonly-used elements of the PHP language are supported in this work.

The system should be capable of reliably detecting operations which could result in untrusted input being displayed on a web page, a condition which allows XSS to occur. We specifically target reflected (non-persistent) XSS, in which data is passed into the page directly from the client. The intermediate representation chosen should allow the ability to be converted back into source code for display to the user of the software; this capability allows us to precisely show from where the untrusted input originated, all operations in which it was propagated, and finally where the untrusted input was used in an operation which resulted in the content being displayed in the web page.

The supported set of syntax elements of PHP, and thus the scope of this work, was chosen based on the most commonly-used elements of the language. Standard conditional and loop constructs are supported, and most use-cases of language elements were handled. Support for object-oriented code was omitted as it presented a separate non-trivial problem and was left for later work.

1.4 Organization

Chapter 1 describes the motivation for this work, background on used techniques, the goals of the project and the procedure for validating results. Chapter 2 provides an overview of related work in this research area and details approaches taken by other authors towards solving this problem. Chapter 3 details the approach used in this work and the techniques employed for vulnerability detection. Chapter 4 describes the implementation of the work as well the data structures generated by the PHP-Parser library. Chapter 5 discusses results, limitations and desired future work. Chapter 6 concludes the paper.
Related Work

A number of authors have utilized static taint analysis to identify vulnerabilities and shown that it can be an effective tool for source code vulnerability detection. Huang et al.\cite{12} created the Web application Security by Static Analysis and Runtime Inspection (WebSSARI) tool which utilized static analysis to identify application vulnerabilities. Rather than focusing on control flow as many previous authors had, the authors instead utilized information flow as a mechanism for finding vulnerable code paths. An AST was used as the intermediate representation, with support for included files during the AST generation. A control-flow graph and symbol table were utilized to allow the analysis engine to determine types for variable and function pre- and post-conditions. Following this, the AST was evaluated for insecure data flow paths. WebSSARI was designed to be modular such that different parsers and lexers could be written and utilized permitting different languages to be analyzed, though the authors focused their efforts on PHP.

Jovanovic et al.\cite{13} used static analysis in their Pixy PHP analysis tool, which was capable of automatically detecting XSS and SQLI vulnerabilities using taint flow analysis. The program utilized an AST as the intermediate representation which was generated by the open-source PHPParser library. The authors used transfer functions to track the propagation of taint through data structures, and handled some major problems inherent with static analysis of PHP code, including variable references, literal value analysis, and included files as well as emulation of built-in PHP functions. The authors did not add support for object-oriented PHP code, and use of member variables and object functions were always
assumed to be untainted.

Gauthier and Merlo[14] developed the Access Control Models Analyzer (ACMA), a tool which tackled a different problem from the OWASP Top 10[2]: Access Control Vulnerabilities. While the vulnerability is different from other works reviewed in this paper, some of the approaches and techniques are similar. The authors utilized both a CFG generated by a parser, which was itself generated by JavaCC, as well as an AST. They compared taint analysis to access control model checking in situations where one boolean privilege exists and drew from this methodology when developing their control flow taint analysis and suggested that sanitization functions are analogous to access control routines in their function (preventing unauthorized actions). The program successfully identified a number of access control vulnerabilities, performing almost 900 times faster than existing solutions.

Zhao and Gong[1] developed a system which utilizes the AST intermediate representation (IR) generated by the HipHop Virtual Machine (HHVM)\(^1\). The authors converted the code into the AST IR using the HHVM, created a control flow graph (CFG) and performed static analysis on the AST. The data structure used in tracking was detailed and provided an interesting way of storing the relation between variables in a tree-like data structure consisting of storage and index nodes. This static analysis was followed up with dynamic analysis which utilized a URL crawler to identify code paths. These URLs were then deconstructed to create lists of values for known parameters which allowed the code analysis to target known valid code paths. Fuzz testing was then performed by reconstructing various parameter lists, fixing some parameters with known values, and fuzzing the non-fixed parameters. The program was tested against 55 custom vulnerable code samples (each with a corresponding "fixed" sample) which were designed to reflect specific constructs in the PHP language. The results showed a high (80%) rate of detection for specific language elements but only a moderate (44%) rate of detection for specific injection attacks.

Dahse and Holz[15] developed a precise static analysis tool named RIPS which can

---
\(^1\)http://hhvm.com/
detect new vulnerabilities in PHP code. RIPS is an open-source PHP static analysis project which was designed to automatically detect 15 types of common vulnerabilities. Unlike many other tools it was developed to model PHP built-in functions and simulates execution of these functions for a more complete analysis. It provides a GUI in the form of a web interface as well as graphical representations of control flow similar to those found in commercial programs such as IDA Pro\(^2\). Along with these novel features, the authors defined over 230 sensitive sinks in the application and added support for file inclusions. Additionally, RIPS provides an exploit creator which can assist researchers in designing proof-of-concept (PoC) exploits. However, RIPS only supports PHP 4 and lacks support for OOP. As these limitations were apparently caused by fundamental limitations in the design, the program was abandoned and subsequently completely rewritten and is now a commercial offering\(^{[16]}\).

Nunes et al.\(^{[17]}\) designed the phpSAFE system to detect XSS and SQLI vulnerabilities in PHP-based plugins for web frameworks, specifically targeting the content management system (CMS) WordPress. The authors highlight the lack of OOP support in various other implementations of PHP static analysis tools like Pixy and RIPS, as this prevents these programs from analyzing code from the most-used PHP web frameworks as these frameworks are written with objects. With this in mind, Nunes et al. designed phpSAFE specifically targeting support for OOP and handling of CMS plugins. The authors used data flow analysis to track the propagation of tainted data through the program. phpSAFE differs from previous implementation of PHP static analysis tools as it handles specific cases that occur with plugin code, such as files with no main function.

Medeiros et al.\(^{[18]}\) evaluated PHP code for input validation vulnerabilities using static analysis, also using taint flow analysis to track the spread of untrusted input through an application. Their approach used analysis of an AST in conjunction with data mining, code correction and user feedback to the developers to show vulnerable code along with the sug-

\(^2\)https://www.hex-rays.com/products/ida/index.shtml
gested correction. As the authors pointed to several circumstances in which it is difficult to determine whether or not a vulnerability exists; for example, in a substring operation on a tainted string, it uncertain whether the new string should be tainted. Machine learning was incorporated in conjunction with data mining to reduce the rate of false positives by providing a machine learning algorithm with samples of code which humans classified as vulnerable or not vulnerable, allowing the system to reduce the number of false positives based on certain ”symptoms” in the code. A wide range of vulnerabilities were supported for detection, from XSS and SQLi vulnerabilities to remote file inclusion, local file inclusion, directory traversal, source code disclosure, PHP code injection and operating system command injection.

Bandhakavi et al.[19] targeted the JavaScript language with their VEX framework, which was designed to analyze Mozilla Firefox extensions for patterns that could result in information flow vulnerabilities. The authors identified five specific types of information flow from sources to sinks that constitute patterns which may result in a vulnerability. In reviewing JavaScript extensions, it was noted that they tend to be very complex. The extensions generally have a significant number of functions and rely heavily on OOP as well as utilize dynamic programming extensively. The authors proposed an abstract heap design for their program, which permits accurate tracking of the relationships between objects and functions by utilizing a graph. An AST was used as the code intermediate representation, and the VEX program walked the AST to create the abstract heap. The authors note that VEX found three previously unknown vulnerabilities in JavaScript extensions, though posit that adding pointer analysis in future revisions could enhance the precision of the program.
Design

A number of challenges presented in the design of this program. Below we detail these challenges and the solutions which were developed for solving them.

3.1 Problem Formulation

In this work, we propose a system that can identify XSS vulnerabilities in PHP code. The basis for finding these issues lies in the analysis of data flow through code. There exist a number of “sources” and “sinks” through which tainted data can flow in a PHP application. Our method of static analysis involves converting the source code to an intermediate representation which may then be more readily interpreted.

3.2 Intermediate Representation

It was determined that program analysis would be simplified through the use of an intermediate representation. We elected to utilize an abstract syntax tree for this purpose due to the ease with which it can be converted to and from source code. Additionally, the tree structure lends itself well to a recursive analysis algorithm, which simplifies some of the challenges with handling scopes for multiple branches. The AST is constructed of nodes which may be either statements or expressions. Nodes may or may not have children; if any children exist, those must be recursively analyzed in a depth-first manner before the result of the parent node can be determined.
3.3 Syntax Rules

Interpretation of the program requires a set of rules which describe the language. The behavior of each operation is modeled based on a set of rules which specifies the results from and impacts of supported operations. The list of these operations in the PHP language is given in Table 3.1 below. This list was selected so as to cover the most commonly-used elements of the PHP programming language.

Table 3.1: PHP Syntax

Expressions ::= 
| \( x \) | variable 
| \( x[e] \) | array index access 
| \( e \ op \ e \) | binary operation 
| \( function(p_1, \ldots, p_n) \) | function definition 
| \( f(a_1, \ldots, a_n) \) | function call 

Statements ::= 
| \( break \) | break 
| \( return \ e \) | return 
| \( x := e \) | assignment 
| \( if \ e \ then \ S_1 \ else \ S_2 \) | conditional 
| \( while \ e \ then \ S \) | while 
| \( do \ S \ while \ e \) | do-while 
| \( for(S_1; \ e; \ S_2) \ do \ S_3 \) | for 
| \( foreach \ e \ then \ S \) | foreach 
| \( switch \ e_0 \ case \ e_1 \ then \ S_1 \) | switch
3.4 Taint Flow Analysis

Tracking the flow of tainted data through the program is paramount to the task of identifying XSS vulnerabilities. In this work, we use taint flow analysis to record where and how tainted data flows in a program so that we can reliably detect when sensitive operations are performed on it. This process begins with identifying ways in which taint is generated, propagated and removed. There are three major classes of operations that can result in the input, output or removal of taint from the system: sources, sinks and sanitizers (respectively).

3.4.1 Sources

Sources are code elements that are assumed to generate untrusted input to a program. Untrusted inputs could result from reading data from a file (which may be changed outside of the program), accepting user input (which may have malicious content) or referencing data that was passed into the program via certain superglobal\(^1\) variables (some of which can be set by clients). In the case of this program, we consider only the last case, as those receive input directly from the user’s browser. Of the superglobal variables, several are implicitly considered tainted due to their being set based on values returned from the client browser: \$_GET, \$_POST, \$_FILES, \$_COOKIE, \$_REQUEST and \$_SERVER.

The \$_GET array receives its values from URL parameters passed in with a query, which could be malicious if a user clicks an untrusted link with malicious parameters attached. \$_POST contains data passed in through POST requests (typically form data), which could potentially include script data. \$_FILES is an associate array of items uploaded with a POST request which contains string fields that may be manipulated. \$_COOKIE can be set based on prior user input and may also be tainted. \$_REQUEST by default contains the contents of \$_GET, \$_POST and \$_COOKIE, all of which are considered unsafe.

$_SERVER contains information such as the request URL, which may be script and could pose a threat if displayed on the page.

For our purposes, we consider any reference to a key within one of these arrays to be tainted. When an expression is evaluated for taint and one of these sources is present, it results in the entire expression being considered tainted and the taint is propagated via the rules listed in the previous section.

### 3.4.2 Sinks

There are sensitive operations in PHP code where the use of untrusted data could result in a XSS attack. In this particular case we are specifically concerned with operations that can display information on a web page, as this could result in malicious script being executed in a client browser. There are a number of functions that output to the page, however for the purpose of simplicity we only cover the non-formatted string output functions, `echo()` and `print()`. Both functions take expressions as arguments, however the print function only allows one argument while the echo function allows any number of arguments.

Upon encountering either of these sinks being utilized in a program being analyzed, their expression arguments are evaluated recursively to determine if any taint sources are being used to create the output. If any are found, the taint flow history from that source is retrieved and displayed to the user of this program, presenting them with a textual representation of how the tainted information flowed from a source to this sink.

### 3.4.3 Sanitizers

There are a number of ways in which taint can be removed from a variable or expression. Regular expressions can be used to filter out special characters, custom functions can be written that encode script tags and PHP built-in functions for escaping special characters and encoding strings can all be used to sanitize output to prevent XSS attacks.

Unfortunately, it would be an entire work in itself to support every possible mechanism
of taint sanitization as programmers might write their own custom sanitizer functions or regex matching strings and handling these could require writing a regex interpreter, which would be outside the scope of this project. A reasonable compromise was chosen in which a number of built-in functions that are capable of preventing script from being output to the page are recognized by default, with the option to manually add additional functions by name to this list.

The two PHP built-in functions recognized are `htmlspecialchars` and `htmlentities`, which HTML encode special characters. When a sanitizer function is invoked in code being analyzed, a function is called which performs actions based on how that function operates. In the case of the two functions just listed, the function immediately returns False to indicate that the return from these functions is considered untainted. Other, more specialized functions could be added to this list and handled appropriately allowing for a high level of flexibility in handling their emulation and impact on data flow.

### 3.4.4 Approximation

Due to the nature of static analysis reducing the flow of a program to a single execution path, any results generated from statements that result in a multiple code branches must be condensed into a single result at the end of that basic block. In this work, we chose to analyze each branch individually then merge the results at the end of the split. As each branch is analyzed, the statements are evaluated within the context of a current environment.

Each branch begins with an environment that contains all variable and taint history from the actions leading up to the conditional, however changes that occur do not effect any other branch. At the end of the branching, a logical OR operation is performed on the taint value of a given variable across all possible branches; if any branch contains that variable in a tainted state, then that variable is tainted as a result. This ensures that any taint of variables is preserved at the end of the branching, regardless of from which branch it originated. Listing 3.1 below shows an example of this approach.
In this simple example, at the time of execution the current environment is $\delta$, which contains simply the variable $\var$. The first branch of the conditional works on a copy of the environment, $\delta'$, and then stores into the variable $\var$ tainted data from PHP’s $_POST superglobal array, which contains parameters passed along with the POST request. This environment is retained until the end of the conditional so that it may be compared with the others. In the elseif branch, a separate copy of the environment is made, $\delta''$, and $\var$ is assigned a string literal which is considered untainted. However, at the end of the conditional, there will only be one execution path.

To accommodate this scenario, the environment from each conditional is compared with the others. Any variables declared in the branches would go out of scope at the end of execution of the branch, and are subsequently discarded. For each variable from the parent environment $\delta$, the tainted status of that variable from all environments $\delta$, $\delta'$ and $\delta''$ are used in a logical OR operation such that if any environment contains a tainted version of the variable, then the result at the end of the conditional indicates that the variable is now tainted going forward. If more than one branch tainted the variable, the line numbers of all taint sources are recorded so that the user knows that multiple locations generated the tainted data. This is implemented via the `merge` function, which accepts as arguments three parameters: the parent environment $\delta_p$, the branch environment $\delta_b$, and a boolean representing whether at least one branch of the conditional must execute.

The boolean parameter in the `merge` function handles a specific case that can arise with multiple branches:
• A variable is tainted before entering a conditional construct

• The conditional has a default condition

• Every branch results in the variable being in an untainted state at the end of execution

In this case, due to the presence of a default statement which guarantees at least one branch will be executed, we can safely say that the variable will be untainted at the end of the branching. An example of this is given in Listing 3.2 below. The variable $var is tainted on the first line, however since all branches in the conditional result in $var being untainted and the presence of a default condition (an else statement) ensures execution of at least one branch, it is guaranteed that the variable will no longer be tainted at the end of the conditional block.

Listing 3.2: Branch Execution with Default

```php
1 $var = $_GET['data'];
2 if(condition1){
3   $var = 0;
4 } elseif(condition2) {
5   $var = 1;
6 } else {
7   $var = 2;
8 }
9 echo $var;
```

The analysis of PHP code in this project was not performed on the source code directly, but rather on an AST generated from the source code. This format succinctly represents the structure and logic of the code while abstracting away the specific syntax of the language, making the program analysis simpler as the resulting tree is significantly easier to work with than source code. The open source library PHP-Parser\(^2\) was used in this project to provide this functionality. PHP-Parser generates an AST of nodes that can be evaluated more easily, in this case using a recursive approach.

\(^2\)https://github.com/nikic/PHP-Parser
The PHP-Parser library defines three types of nodes: statements, expressions, and scalars. Statement nodes are those which do not return a value. Function definitions, loops, and conditionals are all considered to be statements. Statements can occur as a result of other statements; for example, an if statement can contain a series of statements that execute as a result of the condition evaluating to true.

Expressions are nodes that return values, such as variables, boolean operators, concatenation operations, array indices, and assignment operations. Expression nodes can occur inside other expressions; for example, if we let \( a = b || (c || d) \), the resultant node is an expression node which contains an inner expression node which must first be evaluated before the result of the outer expression can be evaluated.

 Scalars are a special case of an expression, representing things like hard-coded integers, strings and predefined constants called ”magic constants”. These are generally assumed to be safe from tampering, as they should not have an unsafe value by default. The value of these could be manipulated after the initial assignment to a variable, however such manipulation would be assumed to be safe barring the use of any tainted data in that process.

As an example, in Figure 3.3 below a small section of PHP code is shown that retrieves a value from the superglobal \$_POST array and stores it to a variable $var before finally echoing it to the page as part of a concatenated string. Figure 3.4 below shows the AST that is generated from this code. In this example, the assignment expression Expr_Assign stores the result of the expression Expr_ArrayDimFetch to the variable given by the expression Expr_Variable. The result from the Expr_ArrayDimFetch is dependent on the result of two more nodes, an Expr_Variable and an index into that variable given by the Scalar_String (which is also an expression).

The second operation is somewhat more complex. The Stmt_Echo accepts an array of expressions, in this case an array containing one item. The only entry is a concatenation operation, the Expr_BinaryOp_Concat. This expression concatenates the result of another
concatenation operation with the Scalar_String whose value is simply an exclamation point.

**Listing 3.3: Example Code**

```php
1 $var = $_POST['input'];
2 echo "You entered " . $var . "!\n";
```

**Listing 3.4: Example AST**

```plaintext
array(
  0: Expr_Assign(
    var: Expr_Variable(
      name: var
    )
    expr: Expr_ArrayDimFetch(
      var: Expr_Variable(
        name: _POST
      )
      dim: Scalar_String(
        value: input
      )
    )
  )
  1: Stmt_Echo(
    exprs: array(
      0: Expr_BinaryOp_Concat(
        left: Expr_BinaryOp_Concat(
          left: Scalar_String(
            value: You entered
          )
          right: Expr_Variable(
            name: var
          )
        )
        right: Scalar_String(
          value: !
        )
      )
    )
  )
)
```

19
This is a simple example that demonstrates the structure of the AST and how it relates to the original PHP source code. More complex samples have been utilized to test this program, as vulnerable PHP code in the wild is much more elaborate.

3.5 Evaluating Nodes

Based off the syntax rules established previously, we have defined semantic-based inferences that describe how nodes are evaluated. Tables 3.2 and 3.3 below detail these inferences.
Table 3.2: Expression Syntax Rules

<table>
<thead>
<tr>
<th>Syntax Rule</th>
<th>Semantic-Based Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>When a variable is used in an expression, the taint status of the variable is returned so that the expression as a whole may be evaluated as a combination of each of the parts</td>
</tr>
<tr>
<td>$x[e]$</td>
<td>Array index accesses are checked against the variable declaration data structure; the key can be either a scalar value or a variable</td>
</tr>
<tr>
<td>$e_1 = e_2 \text{ binaryop } e_3$</td>
<td>Binary operations are performed by evaluating each of the operands and performing a logical OR operation of their results; if either of the operands is tainted, then the result is tainted</td>
</tr>
<tr>
<td>$f[name] = function(p_1, ..., p_n)S$</td>
<td>Function definitions store the entire contents of the function definition in an array with an index of the function name; this permits easy access to evaluate the code later when that function is called</td>
</tr>
<tr>
<td>$function(p_1, ..., p_n)$</td>
<td>Function calls receive the taint status of each argument and analyze the statements of the function in an environment that reflects the taint status of those arguments</td>
</tr>
</tbody>
</table>

Table 3.3: Statement Syntax Rules
<table>
<thead>
<tr>
<th>Syntax Rule</th>
<th>Semantic-Based Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>break</code></td>
<td>Break statements prevent control fallthrough in switch statements, resulting in analysis of the next case statement in line (break statements in loops are ignored to ensure all statements inside the loop are processed)</td>
</tr>
<tr>
<td><code>return e</code></td>
<td>Return statements evaluate the expression provided after the statement for taint; the result of this evaluation is set as the returned taint status from the function call</td>
</tr>
<tr>
<td><code>x = e</code></td>
<td>The expression being assigned to the variable is evaluated; if <code>e</code> is untainted then <code>x</code> is marked untainted and the taint history of <code>x</code> is cleared, otherwise <code>x</code> will be marked tainted and a unique join of the taint history of each component in <code>e</code> is stored as the taint history of <code>x</code></td>
</tr>
<tr>
<td><code>if e_1 then S_1 elseif e_2 then S_2 else S_3</code></td>
<td>The statements <code>S</code> in each conditional block are evaluated in their own scope, and modify only their own scope; the results are merged at the end and can taint or untaint variables in the parent environment</td>
</tr>
<tr>
<td><code>while e then S</code></td>
<td>The statements <code>S</code> are evaluated in a new scope; at the end of the loop, any local variables are discarded and any newly-tainted variables are marked as such in the parent scope</td>
</tr>
<tr>
<td><code>do S while e</code></td>
<td>The statements in a do-while loop are evaluated similarly to the while loop above</td>
</tr>
<tr>
<td><code>for(S_1; e; S_2) then S_3</code></td>
<td>The initialization (<code>S_1</code>) and afterthought (<code>S_2</code>) statements of a for loop are evaluated in a new scope, after which the body of the loop is evaluated; at the end of the loop, local variables are discarded and newly-tainted variables are marked tainted in the parent scope</td>
</tr>
<tr>
<td><code>foreach(e_1 as e_2) do S</code></td>
<td>If any items in <code>e_1</code> are tainted, then <code>e_2</code> is marked tainted in the local scope; the statements in the loop are then evaluated, local variables are discarded and newly-tainted variables are marked in the parent scope</td>
</tr>
<tr>
<td><code>switch(e) case e_n : S_n</code></td>
<td>Each case is evaluated in turn in its own scope, continuing until a break is found or the switch ends. Locals are then discarded and any new taint is merged back into the parent scope.</td>
</tr>
</tbody>
</table>

The `eval` function contains the logic for handling each type of node. It accepts as an
argument a node from the AST which may be either a statement or an expression. When evaluating expressions the function will return a boolean which indicates the taint status of the expression, whereas evaluation of statements will not return a value but can modify the various data structures.
Implementation

The goal of this project was to implement support for a core set of functionality in the PHP language (Figure 3.1 in Chapter 2 shows the supported code elements). In this work, we use static analysis of PHP code to find vulnerabilities which could allow an XSS attack against a web site. Additionally, we present any identified issues to the user so that the vulnerabilities may be remedied.

4.1 Variable Taint

The taint status of variables is tracked in a map data structure where the key is the name of the variable, and the value is an array of attributes of that variable, specifically the taint status (recorded as a boolean) and the taint flow history of that variable (represented as an array of line numbers through which the taint was propagated). This structure is illustrated in Listing 4.1 below.
Listing 4.1: Variable Taint Data Structure

```php
vars = array(
    'key_1' => array(
        'tainted' = <bool>,
        'taint_flow' = array(1
            <integer_1>,
            <integer_2>,
            ...
            <integer_N>
        )
    ),
    'key_2' => array(
        'tainted' = <bool>,
        'taint_flow' = array(1
            <integer_1>,
            <integer_2>,
            ...
            <integer_N>
        )
    )
)
```

This structure allows us to not only track whether or not a variable has been contaminated with untrusted input, but also to store a history of lines through which the tainted data has flowed to reach a given variable. Storing this data allows us to later show the user visually where the vulnerability occurred to give them a better idea of the conditions that led to the unsafe usage of tainted data.

A simple example of this data structure follows, in order to demonstrate how the data is structured. In the case of the code shown in Listing 4.2 below, the following Table 4.1 results:

Listing 4.2: Sample Taint Generation

```php
$my_arr['val'] = 0;
$b = $_GET['val'];
$var = $b;
```
Table 4.1: Variable Taint Data Structure

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Tainted Taint Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_arr['val']</td>
<td>False  NULL</td>
</tr>
<tr>
<td>b</td>
<td>True  2</td>
</tr>
<tr>
<td>var</td>
<td>True  2,3</td>
</tr>
</tbody>
</table>

In this example, the array index $my_arr['val']$ is assigned a scalar value, which is assumed to be untainted. The variable $b$ then receives data from a taint source, and both the new taint status and the line on which the taint originated are stored with the variable. Then $var$ receives the content of $b$, which results in taint propagating to this new variable. In addition, when considering the new taint history of $var$, an array is built containing the taint history of all expressions used in this new assigned value (in this case just line 2). We then add the current line 3 to this array and trim the array to ensure only unique values, in case two expressions have taint that flows through the same line.

This approach allows us to record not only the immediate source of variable taint but also the history of any operations through which this taint has been transmitted until it arrived at the new variable assignment. Once the tainted variable $var$ is used at a sensitive sink, the program can then display to the user exactly how the tainted data flowed through the program to get to $var$ and provides a clearer picture of what steps need to be taken to remedy this situation.

As can be seen in Table 4.1, standard scalar variables are represented simply by their name, whereas an index into an array is stored as the array name along with the key. An interesting issue arises due to the nature of array index assignments in PHP, as PHP arrays are actually ordered maps. Note that this poses issues with array access by integer indices, something which will be discussed further later in the paper along with potential solutions.
4.2 Function Definitions

In order to facilitate the handling of function calls in the code being analyzed, a map was utilized which stored the code defining the function. The name of the function was used as the key which maps to the function code, allowing easy access to the function code whenever a call to that function was made for the purpose of evaluating the function code with the given parameters. An initial pass through the file fills this array as PHP does not require functions to be defined before use; this ensures that functions defined at the end of a file are available to be analyzed from calls made prior to the function definition.

4.3 Taint Propagation

After the definitions of taint flow were defined and the data structures created, we needed to translate the syntax rules for each code construct into the analysis code in the program. Identification of different node types was performed using a switch statement on the class of a node (retrieved from the PHP built-in function `get_class`), and the fully-qualified class names were used for the case statements; for example, the Assignment operation has a node class of `PhpParser\Node\Expr\Assign`.

At the beginning of execution, two global variables are initialized to empty arrays: `$vars` and `$functions`. The former is the data structure where variables will be marked as tainted and the history of their taint flow recorded, while the latter will contain the nodes representing function definitions for easy access later. Additionally, a number of flag variables were set that allow tracking of specific conditions, such as `$has_return` which is True if a function returns data, `$returns_taint` which is True if a function’s return statement evaluates to True (or rather is tainted), and `$break_hit`, which identifies when a switch statement pass has hit a break statement.
4.3.1 Taint Propagation Functions

There are a number of important functions that assist in managing the propagation of taint from nodes. The merge_vars_arrays function takes two copies of the $vars array; the first copy is an updated copy (or a new scope) we’ll call a child array, the second is the parent from which the child was derived. Remember that the keys in these arrays are a variable name, and the values are an array which contains the tainted status of the variable as well as line numbers through which the taint flowed. The function then iterates through each key (a variable name) in the parent array and checks if the key exists in the child. If it does not, it adds the key and value from the parent array to a new result array. If the value does exist in both, the key is stored to the result array and the value is calculated by calling merge_vars_props on the values from both the parent and child array. This continues until all keys have been processed.

The merge_vars_props function performs the merge on the properties of these tainted variables. The function determines the new ”tainted” status of the variable as the union of the taint status from the parent and child array values. The taint flow history from both the parent and child $vars arrays are merged, keeping only unique values; this lets us track any new propagation in this child array.

Additionally, both the merge_vars_arrays and merge_vars_props functions accept an optional third argument, $mandatory, which states whether or not the child array is from a code segment which is guaranteed to be executed. For example, in a do-while loop, it is guaranteed that the code will be executed at least once, so when calling merge_vars_arrays the flag is set to True (which will be passed subsequently to merge_vars_props). In this way, if a variable is tainted before a code section which is guaranteed to execute at least once and then untainted at the end of the block, we can safely assume that the variable will be untainted after the block has completed.
4.3.2 Hard-coded Expressions

The simplest nodes to evaluate are the String scalar, the ConstFetch expression (for using a defined constant), and the InlineHTML nodes. These are all defined by the programmer and hard-coded in the application, therefore they are presumed to be untainted as we assume the developer does not have malicious intent. As a result, when evaluated these nodes will always return a value of False.

4.3.3 Variable Assignments

Evaluating the assignment of values to variables is critical in tracking taint propagation. The assignment operation in PHP is unique in that it is a statement but also behaves like an expression as it returns a value. For example, $a = ($b = 4) + 5 is a valid statement in PHP and results in $b being set to 4 and $a being set to 9. We support this usage of the assignment operator for completeness.

In the example Listing 4.3 below, we illustrate the flow of taint from the $_POST superglobal variable to the variable $a. From there the taint transfers to $b before being used in an echo statement. The implementation for supporting this operation is shown in Listing 4.4 below. First the name of the variable is retrieved; if the assignment is not directly to a variable then the $var_name function composes the variable name (an example being an index into an array, such as $arr[1]), otherwise the name is set to the variable name. If the target variable is not yet in the $vars array, the data structure for the variable is initialized. The expression being assigned is then evaluated, and the taint status and taint flow history are transferred to the target variable name.

Listing 4.3: Assignment Operation Example

```php
1 $a = $_POST['name'];
2 $b = $a;
3 echo "Hello, " . $b;
```
Listing 4.4: Assignment Operation Evaluation

```php
case "PhpParser\Node\Expr\Assign":
    // Get the name of the variable to which we're assigning
    $var_name = '';
    if(!$node->var instanceof PhpParser\Node\Expr\Variable){
        // Get the name of the variable to which we're assigning
        $var_name = get_var_name($node->var);
    } else {
        // Get the name of the variable to which we're assigning
        $var_name = $node->var->name;
    }

    if(!array_key_exists($var_name, $vars)){
        init_var($var_name);
    }

    if(eval_node($node->expr)){
        taint($var_name);
        if(is_a($node->expr, "PhpParser\Node\Expr\FuncCall")){
            $history = $returns_taint_history;
        } else {
            // Get the name of the variable to which we're assigning
            $history = get_taint_flow($node->expr);
        }
        $flow_to_add = array_merge($history, array($currentLine));
        add_taint_flow($var_name, $flow_to_add);
    } else {
        // Remove taint and clear taint_flow array
        untaint($var_name);
        clear_taint_flow($var_name);
    }

    return;
```

4.3.4 Binary Operations

Binary operations are also trivial to evaluate. Listing 4.5 shows an example of a binary operation resulting in tainted output; the concatenation of a hard-coded string along with a value received from a tainted source results in a new string, which is now tainted. This string is subsequently used in a sensitive sink, resulting in an XSS vulnerability. If either operand in a binary operation is tainted, then the result is considered tainted. As all binary operations follow this pattern, a regular expression was used to trigger evaluation of this case. Note that the body of the switch statement has been included in Listing 4.6 below to show how the ternary operator in the case statement triggers execution.
4.3.5 Loops

Loops are also relatively easy to evaluate. An example of a loop resulting in an XSS vulnerability is shown in Listing 4.7 below, which repeats a user-entered string five times.

Processing a while loop (shown in Listing 4.8) for taint flow begins by saving a copy of the current global $vars array, so that we are able to identify later which variables were defined in the scope of this loop (or local variables). The condition is then evaluated, as assignment operations can occur as part of the conditional expression. Then each statement in the body of the while loop is recursively evaluated to determine taint propagation. At the end of the loop, the merge_vars_arrays function (explained above) is called to propagate taint from the loop scope back to the parent scope and to delete local variables.

Listing 4.7: While Loop Example

```php
<?php
// Repeats a string five times
$output = '';
$ctr = 0
while($ctr < 5)
{
    $output = $output . $_POST['str'];
    $ctr++;
}
```
The do-while loop is processed in much the same way as the while loop, however the `merge_vars_arrays` function is called with the `$mandatory` flag set to True, since the body of the loop will be executed at least once.

For loops consist of three main components with which we are concerned: the initialization, the loop action(s), and the loop statements. An example for loop is given below in Listing 4.9, which shows a simple for loop that initializes the variables `$i` and `$j` (the latter being tainted), prints them out on each iteration, then increments `$i`. The procedure for evaluating this loop is given in Listing 4.10. After saving a copy of the parent `$vars` array, the initial conditions are evaluated (in our example, this results in `$j` becoming tainted). The loop action (`$i++`) is then processed, followed by the statements in the body of the loop. Once all statements have been processed, the results are merged back into the parent array.

**Listing 4.8: While Loop Evaluation**

```php
<?php

case "PhpParser\Node\Stmt\While_":

    $saved_vars = $vars;

    eval_node($node->cond);

    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }

    // Merge any updates back in to the main array, remove any locals
    $vars = merge_vars_arrays($vars, $saved_vars);

    return;

?>
```

**Listing 4.9: For Loop Example**

```php
<?php

for($i = 0, $j = $_POST['j']; $i < 10; $i++){

    echo $i;

}```
The foreach function operates on a somewhat different premise than the other loops. The foreach loop iterates over each entry in an array and uses each value as the loop variable. Listing 4.11 shows an example of a foreach loop where one of the array values is tainted, while the code for handling a foreach loop is given in Listing 4.12. The $vars array is backed up, after which the array to be iterated over is searched to find any tainted values. Since the array indices are stored as array_name[key_name] in the $vars array, we use a regular expression to identify values stored in the array. If any tainted values are found, the whole array is considered to be tainted for the purposes of this analysis (as each value will be used). We then create a variable named as the $value and set it to the taint status of the array found previously, adding any taint history from the tainted variable to this new loop variable. We are then able to evaluate the statements in the foreach loop, merging the variable taint status at the end and removing local variables.
4.3.6 Conditionals

Both the if-else and switch conditional statements are supported in this project. As with the do-while loop, a scenario occurs with these conditionals that can guarantee at least one branch will be executed. An else statement at the end of an if conditional as well as a...
default case in a switch statement both ensure that a minimum of one branch will be taken. Below we detail the use and implementation of these conditionals in our project.

Listings 4.13 and 4.14 below show a sample if-else construct and the code which analyzes these statements. The example below shows how the variable $var is tainted in only the first branch of the if-else conditional but not the others. The evaluation for the *if* conditional saves the $vars array as standard, but also creates a new $merged_vars array which will save the results of merging each branch with the previous results. We then process each statement inside the *if* block, merging the results of the temporary $vars array into $merged_var. For each *elseif* statement, a new $vars array is created from the original parent $vars array and the statements in the *elseif* block are evaluated in that environment before merging the results into the $merged_vars array. Finally, the *else* block (if one exists) is processed and the results merged. Once all blocks are processed, the merged variables array is itself merged back with the parent array. As was noted earlier in this chapter, variables can be untainted if the following two conditions are met:

- An *else* branch exists
- The variable is untainted in each branch of the conditional

If both conditions are met, then it can be safely assumed that every possible outcome results in the variable having taint removed and that there is no way for the conditional block to execute without untainting the variable.

Listing 4.13: If-Else Conditional Example

```php
$var = ''; $cond = 1;
if($cond == 0{
    $var = $_GET['data'];
} elseif($cond == 1){
    $var = 'stuff';
} else {
    $var = 'things';
}
echo $var;
```
The other conditional construct we handle is the switch statement. Listings 4.15 and 4.16 below show an example of a switch statement in PHP and our implementation of taint support. In the example we assign a value to $result based on which case is executed, though the absence of a break statement in the "first" case causes control fallthrough resulting in the "second" case being executed as well. If no cases match, the "default" case is executed.

Support for this construct requires a slightly different approach, as the possibility of fallthrough when break statements are not present means that the statements from multi-
ple cases could be executed. In addition to the standard saving of $vars and creating the
$merged_vars array, we also save the portion of the AST containing the cases, permitting us
to jump back to evaluate previous cases if fallthrough occurs. From this point, we evaluate
each case individually, continuing as long as a break statement is not found. Once a break
is encountered, the $break_hit flag is set and execution will not continue to the next case.
As with the if – else example, the results of each branch are merged at the end, and the
presence of a default case permits variables to be marked untainted so long as all branches
result in the variable being untainted.

Listing 4.15: Switch Example

```
$var = "second";
$result = ''; switch($var){
  case "first":
    $result = "First case executed";
  case "second":
    $result = "Second case executed";
    break;
  default:
    $result = $_REQUEST['data'];
}
echo $result;
```

Listing 4.16: Switch Evaluation

```
case "PhpParser\Node\Stmt\Switch_":
  $saved_vars = $vars;
  $merged_vars = array();
  $cases = $node->cases;
  $is_default = False;
  // Evaluate each case statement sequentially.
  for($i = 0; $i < count($cases); $i++){
    // A null condition means this is the "default" case
    if(is_null($cases[$i]->cond)){
      $is_default = True;
    }
    for($j = $i; $j < count($cases) && !$break_hit; $j++){
      // Evaluate the statements for this case statement
      foreach($cases[$j]->stmts as $stmt){
```
eval_node($stmt);
}

$merged_vars = merge_vars_arrays($merged_vars, $vars);

// Reset the vars array
$vars = $saved_vars;

// Reset the break flag
$break_hit = False;

if($is_default){
    $vars = merge_vars_arrays($saved_vars, $merged_vars, True);
} else {
    $vars = merge_vars_arrays($saved_vars, $merged_vars);
}

return;

4.3.7 Functions

Our program also supports taint analysis of functions defined in the target code. The initial function definition is handled by the function data structure outlined previously in this document. Prior to starting analysis of the code, the AST is traversed searching for any function definitions (as PHP does not require functions to be defined before use). Any that are found are added to the function datastructure, where the key is the function name and the value is the AST node corresponding to the function code. At the time a function is called, the name of the function is checked to see if it is a built-in sanitization function (other built-in functions are currently ignored); if it is, a False value is returned. If the function is in the list of user-defined functions, the arguments are each analyzed to determine if they are tainted. A new scope is set up in which the function’s parameters are tainted accordingly to the argument taint status, all other variables are removed (since this is a local function scope), and the statements in the function are analyzed. If the function has a return statement, a taint status of the expression from the return node is determined and returned as the result of the function call. At this point, local variables from the scope of the function are removed.
Listing 4.17 shows an example of a defined function, its use and return of tainted data while Listing 4.18 shows the code whose operation is outlined above. Comments have been omitted for brevity, however the full text of the code can be found in Appendix A.

Listing 4.17: Function Example

```php
function gen_taint($val1, $val2)
{
    $val3 = $val1 . $_POST['taint'];
    return $val3;
}
$var = gen_taint('0', '1');
echo $var;
```

Listing 4.18: Function Evaluation

```php
case "PhpParser\Node\Expr\FuncCall":
    $has_return = False;
    $returns_taint = False;
    $func_name = $node->name->parts[0];
    $args = $node->args;
    if(in_array($func_name, $special_functions) || ...
        function_exists($func_name)){
        $has_return = True;
        $returns_taint = handle_function($func_name, $args);
    } else {
        $saved_vars = $vars;
        $tmp_vars = array();
        if(!array_key_exists($func_name, $functions)){
            return;
        }
        $params = $functions[$func_name]->params;
        for($i=0; $i<count($args); $i++){
            // Get name of param
            $var_name = $params[$i]->name;
            $tmp_vars[$var_name] = array(
                'tainted' => eval_node($args[$i]->value),
                'taint_flow' => get_taint_flow($args[$i]->value),
            );
            if($tmp_vars[$var_name]['tainted']){
                array_push($tmp_vars[$var_name]['taint_flow'], ...
                    $currentLine);
            }
        }
        $vars = $tmp_vars;
        $stmts = $functions[$func_name]->stmts;
        foreach($stmts as $stmt){
```
4.3.8 Sinks

There are two sinks we consider for our discussion of reflected XSS attacks; the `echo` and `print` statements. Both of these statements write data out to a client’s browser and are locations where malicious code could be injected onto a page. We will discuss only the `echo` statement, as both `print` and `echo` are functionally equivalent with the exception that the `print` statement accepts only one argument while `echo` accepts a list. Listing 4.19 illustrates a simple use of the `echo` statement which retrieves a name from the user and prints it out onto the page. Listing 4.20 shows how `echo` statements are evaluated. The statement node contains a list of expressions, each of which are evaluated to check for taint. If any of the expressions are tainted, the taint history from the tainted node is gathered for display and a message is printed to the user detailing from where the taint originated, where it flowed and the current line number with this sink.

Listing 4.19: Sinks Example

``` php
$name = $_POST['name'];
echo "Hello, " . $name;
```

Listing 4.20: Sinks Evaluation

``` php
case "PhpParser\Node\Stmt\Echo_":
    foreach($node->exprs as $expr){
        if(eval_node($expr)){
            echo "\neval_node: Found taint in echo\n";
        }
```
$history = get_taint_flow($expr);
$history[] = $currentLine;
$history = array_unique($history);

printf("%5s | %s
", "Line", "Content\n");

foreach($history as $line){
    printf("%5d | %s\n", $line, trim($lines[$line-1]));
}

echo "\n\n";
return;
}
Evaluation

This thesis work was evaluated using samples of PHP code that are known to contain XSS vulnerabilities. The output of the program was compared with expected results from these known samples to assess the accuracy of the experimental results.

5.1 Samples

Several real-world code examples with known vulnerabilities were pulled from open source repositories for testing. Finding samples of vulnerable PHP code to use for the analysis portion of this project proved challenging. In deciding to not support OOP in this version, much of the publicly available code became unsuitable for use in testing as most modern PHP programs are written utilizing classes. Therefore, we had to manually craft code samples that would be representative of real-world code resulting in XSS vulnerabilities. Over a dozen samples of varying complexity were created to test the efficacy of this system. To ensure coverage of features, samples were used that included loops (for, while, do-while), conditionals (if, if-else, if-elseif-else), function declarations, function calls (both built-in and declared) and nested conditionals and loops. These examples may be relatively short compared to actual application code that would be found on a web server, however they have been written so as to validate the core functionality of the project.

Listing 5.1 in Appendix B contains vulnerable code pulled from a now-defunct Wordpress plugin. Our program identified eight locations in the code where XSS vulnerabilities could occur and which might require closer inspection. The output of the program is shown
below in Listing 5.1.

Listing 5.1: Example 1 Program Output

```
1 eval_node: Taint propagated on line number 9
2
3 eval_node: Found taint in echo
4  Line| Content
5     | 9| $fasc_plugin_ver = $_GET['ver'];
6     | 31| <script language="javascript" type="text/javascript"
7     |     src="../wp-includes/js/tinymce/tiny_mce_popup.js
8     |     ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
9
10 eval_node: Found taint in echo
11  Line| Content
12     | 9| $fasc_plugin_ver = $_GET['ver'];
13     | 32| <link rel="stylesheet" href="../css/button-styles.css
14     |     ?ver=<?php echo $fasc_plugin_ver; ?>" />
15
16 eval_node: Found taint in echo
17  Line| Content
18     | 9| $fasc_plugin_ver = $_GET['ver'];
19     | 33| <link rel="stylesheet" href="../css/font-awesome.css
20     |     ?ver=<?php echo $fasc_plugin_ver; ?>" />
21
22 eval_node: Found taint in echo
23  Line| Content
24     | 9| $fasc_plugin_ver = $_GET['ver'];
25     | 34| <script src="jquery.minicolors.min.js
26     |     ?ver=<?php echo $fasc_plugin_ver; ?>"></script>
27
28 eval_node: Found taint in echo
29  Line| Content
30     | 9| $fasc_plugin_ver = $_GET['ver'];
31     | 35| <link rel="stylesheet" href="jquery.minicolors.css
32     |     ?ver=<?php echo $fasc_plugin_ver; ?>">
33
34 eval_node: Found taint in echo
35  Line| Content
36     | 9| $fasc_plugin_ver = $_GET['ver'];
37     | 36| <link rel="stylesheet" href="popup.css
38
39 eval_node: Found taint in echo
40  Line| Content
41     | 9| $fasc_plugin_ver = $_GET['ver'];
42     | 37|<link rel="stylesheet" href="jquery.minicolors.min.js
43  ```
Additionally, a much shorter sample from the O’Reilly PHP learning resources\(^1\) was observed to have an XSS vulnerability. On line 19 of the code sample in Listing 5.2 below it can be seen that the value of \$_POST[‘my_name’] is printed directly onto the page.

### Listing 5.2: Example 2

```php
<?php
// Code from https://github.com/oreillymedia/Learning_PHP/
blob/master/code/forms/forms-122.php

// Logic to do the right thing based on
// the request method
if (\$_SERVER[‘REQUEST_METHOD’] == ‘POST’) {
    // If validate_form( ) returns errors, pass them to ...
    show_form( )
    if (\$form_errors = validate_form( )) {
        show_form(\$form_errors);
    } else {
        process_form( );
    }
} else {
    show_form( );
}
// Do something when the form is submitted
```

\(^1\)https://github.com/oreillymedia/Learning_PHP/blob/master/code/forms/forms-122.php
function process_form() {
    print "Hello, ".$_POST['my_name'];
}

// Display the form
function show_form($errors = array()) {
    // If some errors were passed in, print them out
    if ($errors) {
        print 'Please correct these errors: <ul><li>QPushButton';
        print implode('</li><li>', $errors);
        print '</li></ul>';  
    }
    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;  
}

// Check the form data
function validate_form() {
    // Start with an empty array of error messages
    $errors = array();
    // Add an error message if the name is too short
    if (strlen($_POST['my_name']) < 3) {
        $errors[ ] = 'Your name must be at least 3 letters long.';  
    }
    // Return the (possibly empty) array of error messages
    return $errors;  
}

5.2 Effectiveness

Of the samples studied, all known XSS vulnerabilities were detected. A number of samples were pulled from public code repositories including Wordpress plugins and PHP learning resources, while others were crafted manually to test core functionality of the system. Table 5.1 below demonstrates the some of the sources and sinks identified and the
Table 5.1: Taint Flow

<table>
<thead>
<tr>
<th>File</th>
<th>Taint Source</th>
<th>Sensitive Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>forms-122.php</td>
<td>$_POST['my_name']</td>
<td>print $_POST['my_name']</td>
</tr>
<tr>
<td>popup.php</td>
<td>$fasc_plugin_ver = $_GET['ver']</td>
<td>echo $fast_plugin_ver;</td>
</tr>
<tr>
<td>edit-styles.php</td>
<td>$_GET['gid']</td>
<td>echo $_GET['gid']</td>
</tr>
</tbody>
</table>

files in which the taint flow was found.

5.3 Running Time

Each real-world sample was evaluated fifty times and the runtime was determined for each test using the GNU time command. The time was calculated by taking the sum of the user-mode time (User) and the time spent in system calls (Sys). The average of these fifty trials for each code sample was then calculated. Below is a graph of the average runtime for a sample against the number of nodes in the AST representation of that program.

![Figure 5.1: Effect of AST Node Count on Running Time](image)
Discussion

We believe the program has been shown to be effective on the provided samples of PHP code and on the test cases designed to thoroughly test how the program handles analysis of specific elements of the PHP language. As was mentioned in the previous section a relatively small quantity of testable code was available for analysis due to the heavy reliance on OOP by modern PHP projects, handling for which was omitted from this project as it was left for future work. Implementing support for this feature of the language would require some reworking of the code to support classes, attributes and class functions, however it would be feasible to add to the existing codebase and is required to process more complicated code such as that from frameworks like Wordpress and Drupal (and their plugins).

Persistent XSS vulnerabilities are not supported in the program to as we specifically target reflected XSS vulnerabilities. Addition of this functionality would be simple, requiring identification of any read/write operations to a database management system (DBMS) so that the flow of data into and out of the database could be evaluated for taint. Whether or not data from the database is considered tainted would be dependent on the current state of the database and if data already present could be assumed to be correctly sanitized. This ”explicit trust” could be implemented as a flag in the program to allow users to specify whether or not read operations from the data source should be assumed to be safe, as users with existing databases may not be able to trust that all data stored there is not tainted.

Support for arrays currently assumes that square bracket notation will be utilized,
allowing for the tainting of specific indices within the array. Use of the array() construct would require some reworking of the array handling code. This arises due to the fact that an array assigned with the `array(key_1 \rightarrow value_1, ..., key_n \rightarrow value_n)` notation has two methods of accessing values: by key, and by integer index. If keys are specified, the integer indices are automatically generated and assigned based on the order of key-value pair assignments. However, manual index specification can permit ”gaps” in the array, such as the following example in Listing 6.1 from the PHP documentation[20]. While it does not appear to be common practice to use this method of array creation among PHP developers, support for this would be feasible to implement.

**Listing 6.1: PHP Array Gaps**

```php
<?php
$array = array(
    "a",
    "b",
    6 => "c",
    "d",
);?
var_dump($array);
?>
```

Resulting array:

`array(4) {
[0]=> string(1) "a"
[1]=> string(1) "b"
[6]=> string(1) "c"
[7]=> string(1) "d"
}`

Functions defined within the code to be analyzed are currently evaluated effectively. Aside from specific PHP built-in functions which are known to remove script from a string, in this current implementation any other built-in functions are assumed to have a return and
return taint only if any arguments are tainted. PHP-Parser does not implicitly analyze built-in functions, so the PHP functions would need to be either emulated or the source code provided and analyzed to determine both whether the function passes taint and what other effects it might have.

Support for file includes was also excluded from the scope of this project. Addition of this feature would require changes to scoping as any statements executed in an included file would need to be reflected within the current environment, however this could be added to the creation of the AST and the code included inline and would be simple to implement.
Conclusion

This thesis has described a prototype system for performing static analysis on PHP code with the goal of identifying XSS vulnerabilities. We demonstrate the use of the system on examples that display the core functionality of the program, detailing both the implementation and challenges that exist in analyzing PHP code. A number of tasks exist for future work, including adding support for object-oriented programming, handling formatted output statements (for completeness), differentiating between local and global variables, and supporting the alternate array definition notation listed in the previous section. These changes would all be beneficial and increase the amount of code that the program could successfully analyze for vulnerabilities.
Bibliography


Appendix A

Program Code

Listing A.1: Program Code

```php
#!/usr/bin/env php

<?php

// Bootstrap generated by composer
require 'vendor/autoload.php';

// Increase debug level
ini_set('xdebug.max_nesting_level', 3000);

$debug = 0;
$scope_debug = False;

// Load the PhpParser library
use PhpParser\Error;
use PhpParser\ParserFactory;
use PhpParser\PrettyPrinter;
use PhpParser\NodeTraverser;
use PhpParser\NodeVisitor\NameResolver;
use PhpParser\NodeDumper;

// Use our custom Node Visitor class
require __DIR__ . '/node_visitor.php';

// Array of tainted arrays (PHP "superglobals")
$tainted_arrays = array("_GET",
    "_POST",
    "_FILES",
    "_COOKIE",
    "_REQUEST");
```
// Array of special functions that introduce or remove taint
 specials_functions = array( 'htmlspecialchars',
                        'htmlentities');

/*
 In PHP, printing a bool returns "1" if true and "" if false
 This function fixes this PHP "feature" and returns the ...
 strings "True" or "False"
*/
function print_bool($val){
    if($val){
        return "True";
    } else {
        return "False";
    }
}

/*
 Small function to return an X if the given value is true, ...
 used for printing the taint array.
*/
function x_if_true($val){
    if($val){
        return "x";
    } else {
        return "";
    }
}

/*
 Function to pretty-print the $vars array, since print_r ...
 takes up too much space in the output
*/
function print_vars($vars_arr = NULL){
    global $vars;
    if(is_null($vars_arr)){
        $vars_arr = $vars;
    }

    $mask = "|%-20.20s|%-5s|%-50s
";
    printf($mask, "Variable", "Taint", "Taint Flow (Line ... Numbers)"");
    printf($mask, "-----------------------", "-----", ...
             """""""""""");
    foreach($vars_arr as $key => $value){
        $taint_flow = implode(', ', $value['taint_flow']);
        printf($mask, $key, x_if_true($value['tainted']), ...
                $taint_flow);
    }
At the first sighting of a variable, initialize the data ... structure for it in the $vars array

function init_var($var_name){
    global $vars;
    // echo "init_var: First usage of variable ". $var_name . ",\n";
    $vars[$var_name] = array(
        'tainted' => False,
        'taint_flow' => array(),
    );
}

function is_tainted($var_name){
    global $vars;
    if(array_key_exists($var_name, $vars)){
        return $vars[$var_name]['tainted'];
    } else {
        return False;
    }
}

function taint($var_name){
    global $vars;
    $vars[$var_name]['tainted'] = True;
}

function untaint($var_name){
    global $debug;
    global $vars;
    if($debug){
        echo "untaint: Removing taint from variable $var_name\n";
    }
    $vars[$var_name]['tainted'] = False;
function get_var_name($node)
{
    global $debug;

    switch(get_class($node))
    {
        case "PhpParser\Node\Expr\Variable":
            return $node->name;
        case "PhpParser\Node\Expr\ArrayDimFetch":
            $array_name = get_var_name($node->var);
            $key_name = get_var_name($node->dim);
            // Differentiate between arrays indexed by strings and numbers
            if(is_a($node->dim, "PhpParser\Node\Scalar\LNumber") &&
                $var_name = $array_name . "[$" . $key_name . "]");
            else if(is_a($node->dim, "PhpParser\Node\Scalar\String_") &&
                $var_name = $array_name . "['" . $key_name . "'");
            }
            else
                $var_name = $array_name . "['" . $key_name . "]");
    }

    // $var_name = $array_name . "['" . $key_name . "'");
    // $var_name = addslashes($var_name);
    if($debug > 2)
    {
        echo "get_var_name: " . $var_name . 
    }
    return $var_name;

    case "PhpParser\Node\Scalar\String_":
        return $node->value;
    case "PhpParser\Node\Scalar\LNumber":
        return $node->value;
    }
    }

    /*
    Helper function to get the taint flow information when given a variable name. Can also be used if you already have the variable name. Returns an array of line numbers.
    */
function _get_taint_flow($var_name) {
    global $debug;
    global $vars;
    return $vars[$var_name]['taint_flow'];
}

/*
 * For a given AST, find the history of taint flow and ...
 * return it as an array of line numbers
 */
function get_taint_flow($node) {
    global $debug;
    global $vars;
    global $tainted_arrays;

    if(is_null($node)) {
        return array();
    }

    $var_name = get_var_name($node);

    $node_class = get_class($node);

    switch($node_class) {
        case "PhpParser\Node\Expr\Variable":
            if(is_tainted($var_name)) {
                return _get_taint_flow($var_name);
            } else {
                return array(); // return an empty array, ...
            }
            break;

        case "PhpParser\Node\Expr\ArrayDimFetch":
            // Get the names of the array, the dim, and ...
            $array_name = get_var_name($node->var);
            $key_name = get_var_name($node->dim);
            $var_name = get_var_name($node);

            // Special case if it's a tainted array, just ...
            if(in_array($array_name, $tainted_arrays)) {
                return array($node->getLine());
            }
            break;

        // If it's just an array indexed with a scalar, ...
        if($node->dim instanceof PhpParser\Node\Scalar){
            return array($node->getLine());
        }
    }
}

58
return _get_taint_flow($var_name);
// Otherwise, check the taint history of the ...
} else {
    return array_merge(taint_flow($var_name), ...
    _get_taint_flow($key_name));
}

    case (preg_match("/BinaryOp.*,\$/", $node_class) ? ...
        $node_class : !$node_class):
        return array_merge(get_taint_flow($node->left), ...
            get_taint_flow($node->right));

    case "PhpParser\Node\Scalar\String_":
        return array();

    case "PhpParser\Node\Scalar\LNumber":
        return array();

    case "PhpParser\Node\Expr\ArrayItem":
        return array_merge(get_taint_flow($node->key), ...
            get_taint_flow($node->value));

    default:
        return array();
    }
}

    /*
      Append taint flow history to a variable, given an array ...
      of integers corresponding to line numbers
      */
    function add_taint_flow($var_name, $history) {
        global $vars;
        // Remove duplicate line number values, and assign to ...
        // this var
        $vars[$var_name]['taint_flow'] = ...
            array_unique($vars[$var_name]['taint_flow'] + $history);
    }

    /*
      Variable $var_name is untainted, so clear the taint flow ...
      */
    function clear_taint_flow($var_name) {
        global $vars;
$vars[$var_name]['taint_flow'] = array();

/*
 * Merge two $vars arrays
 * PHP's array diff only works in one dimension, so we need ...
 * to do this manually
 */

function merge_vars_arrays($new_vars, $old_vars, $mandatory = False)
{
    $result_arr = array();
    // Check each var in the old array and see if it's in the ...
    // new one. If the variable is in the new array and the ...
    // old one, it needs to be updated. Anything not in the ...
    // old array will be thrown out (end of scope).
    foreach($old_vars as $key => $value){
        if(array_key_exists($key, $new_vars)){
            $result_arr[$key] = ...
            merge_vars_props($new_vars[$key], ...
            $old_vars[$key], $mandatory);
        } else {
            $result_arr[$key] = $value;
        }
    }
    return $result_arr;
}

/*
 * When merging vars arrays, merge their properties in a ...
 * smart way. I.e. if either variable is ...
 * tainted, the result is tainted. Additionally, merge their ...
 * taint flow histories.
 * 
 * NOTE:
 * Each variable in the $vars array is associated with a ...
 * number of properties. We need to find ...
 * any differences between the old $vars array and the new ...
 * $vars array and ...
 */

function merge_vars_props($old_props, $new_props, $mandatory = False)
{
    // Compare the properties array for this variable name ...
    // between the old and new array. NOTE: Err on the side ... of caution; if it lost taint, it might still have it ... in another branch, so don't remove it
    $ret_props = array();
    // First handle the tainted property; if either was ...
    tainted, the result should still
if(!$mandatory){
    $ret_props['tainted'] = $old_props['tainted'] || ...
    $new_props['tainted'];
} else {
    if($new_props['tainted'] == False){
        // If it was a mandatory conditional, and the ...
        // variable is still false, that means that every ...
        // branch resulted in the variable being ...
        // untainted; therefore, we can untaint it in the ...
        global $vars array and remove the taint history.
        $ret_props['tainted'] = False;
        $old_props['taint_flow'] = array();
        $new_props['taint_flow'] = array();
    } else {
        // Otherwise, it must be tainted so set it to True
        $ret_props['tainted'] = True;
    }
}

// Now merge the taint_flow arrays
$ret_props['taint_flow'] = ...
    array_unique(array_merge($old_props['taint_flow'], ...
    $new_props['taint_flow']));

    return $ret_props;
}

/************************************************************************
 For special PHP functions, emulate their behavior here. ... 
 This covers functions 
 that always generate of remove taint. 
*/
function handle_function($func_name, $args){
    switch($func_name){
        case "htmlspecialchars":
            return False;
        case "htmlentities":
            return False;
        default:
            echo "Handling function $func_name\n";
            // We'll assume it doesn't remove taint
            foreach($args as $arg){
                if(eval_node($arg)){
                    return True;
                }
            }
            return False;
    }
}
Recursively evaluates nodes from the AST to determine their taint status

Requires: A node of type \PhpParser\Node\*
Returns: True or false for expressions (indicating their taint status), otherwise nothing

```php
function eval_node($node){

    // Tell PHP to let us use our global in this scope
    // What a great language
    global $debug;
    global $nodeDumper;
    global $prettyPrinter;
    global $vars;
    global $functions;
    global $special_functions;
    global $lines;
    global $has_return;
    global $returns_taint;
    global $returns_taint_history;
    global $break_hit;
    global $tainted_arrays;

    // print_r($node);

    $currentLine = $node->getLine();
    $node_class = get_class($node);

    echo "eval_node: Evaluating " . $node_class . "\n";
    switch($node_class){
        case "PhpParser\Node\Expr\Assign":
            // Get the name of the variable to which we're assigning
            $var_name = '';
            if(!$node->var instanceof \PhpParser\Node\Expr\Variable){
                $var_name = get_var_name($node->var);
            } else {
                $var_name = $node->var->name;
            }

            // If it's not in the array, initialize the variable being assigned to as untainted and create an empty array that will track line numbers to show taint flow
            if(!array_key_exists($var_name, $vars)){
```
init_var($var_name);
}

// If the expression being assigned has taint, ...
find the flow and add that to the assignee
if(eval_node($node->expr)){
    echo "eval_node: Taint propagated on line ... number ". $currentLine . "\n";

    // Taint the variable
taint($var_name);

    // Get the taint history of the item(s) being ...
    assigned
    if(is_a($node->expr, ...
        "PhpParser\Node\Expr\FuncCall"){
        $history = $returns_taint_history;
    } else {
        $history = get_taint_flow($node->expr);
    }

    // Add the taint flow history plus the ...
current line num to this variable's flow ...
    $flow_to_add = array_merge($history, ...
        array($currentLine));

    add_taint_flow($var_name, $flow_to_add);

} else {
    // Remove taint and clear taint_flow array
    untaint($var_name);
    clear_taint_flow($var_name);
}

return;

// Echo statements are sinks for taint in XSS
case "PhpParser\Node\Stmt\Echo_":
    foreach($node->exprs as $expr){
        if(eval_node($expr)){
            echo "\neval_node: Found taint in echo!\n";

            $history = get_taint_flow($expr);
            $history[] = $currentLine;

            printf("%5s| %s", "Line", "Content\n");

            foreach($history as $line){
                printf("%5d| %s\n", $line, ...
                    trim($lines[$line-1]));
            }
    }
// Print statements are sinks for taint in XSS
$case = "PhpParser\Node\Expr\Print_":
    if(eval_node($node->expr)){
        echo "\neval_node: Found taint in print!\n";
        $history = get_taint_flow($node->expr);
        $history[] = $currentLine;
        printf("%5s| %s", "Line", "Content\n");
        foreach($history as $line){
            printf("%5d| %s\n", $line, ...
                trim($lines[$line-1]));
            echo "\n\n";
        }
    }
    return;

$case = "PhpParser\Node\Stmt\Do_":
    // Evaluate the while first, then run through our ...
    $saved_vars = $vars;
    eval_node($node->cond);
    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }
    // Merge any updates back in to the main array, ...
    $vars = merge_vars_arrays($vars, $saved_vars);
    break;
$case = "PhpParser\Node\Stmt\For_":
    $saved_vars = $vars;
    // Evaluate the initial condition(s)
    foreach($node->init as $stmt){
        eval_node($stmt);
    }
    // Check the loop action(s)
    foreach($node->loop as $stmt){
        eval_node($stmt);
    }
// Evaluate the statements inside the loop
foreach($node->stmts as $stmt){
    eval_node($stmt);
}

$vars = merge_vars_arrays($vars, $saved_vars);
return;

$vars = $saved_vars;

// Find if any items in the target array are tainted
$tainted = False;
$flow = array();
foreach($vars as $var => $values){
    if(preg_match('/\^\.' . ... get_var_name($node->expr) . '/', $var) === 1){
        if(is_tainted($var)){
            $tainted = True;
            // Now get the taint flow so we can ... see where the source was
            $flow = ...
            array_unique(_get_taint_flow($var) ... + $flow);
        }
    }
}

// Add our loop variable and set the taint status ...
if($tainted){
    init_var(get_var_name($node->valueVar));
    taint(get_var_name($node->valueVar));
}

add_taint_flow(get_var_name($node->valueVar), $flow);
foreach($node->stmts as $stmt){
    eval_node($stmt);
}

// Update any changed vars, remove locally scoped ...
$vars = merge_vars_arrays($vars, $saved_vars);
return;

// Save the original vars so we know what's changed
$saved_vars = $vars;
// Here's where we'll merge the results of taint ...
// from each block
$merged_vars = array();

// Evaluate the initial if condition statements
foreach($node->stmts as $stmt){
    eval_node($stmt);
}

// Merge an empty array with our current vars ...
// array at the end of the If branch
$merged_vars = merge_vars_arrays($merged_vars, ...
// $vars);

if($debug){
    echo "eval_node: Results from Stmt_If:
";
    print_vars();
}

// Evaluate each elseif and merge in the taint ...
// status and history
if(!is_null($node->elseifs)){
    foreach($node->elseifs as $elseif){
        $vars = $saved_vars;
        eval_node($elseif);

        // Merge the results from the If branch ...
        // with this ElseIf branch
        echo "eval_node: Merging result from ...
            ElseIf on line $currentLine\n";
        $merged_vars = ...
        merge_vars_arrays($merged_vars, $vars);
    }
}

// Evaluate the else and merge in the taint ...
// status and history
if(!is_null($node->else)){
    $vars = $saved_vars;
    eval_node($node->else);

    // Merge the results from the Else branch ...
    // with all previous results
    $merged_vars = ...
    merge_vars_arrays($merged_vars, $vars);
}

// NOTE: Now we can update our original vars ...
// array. If the statement is of the form ...
// if->else, and all branches remove taint from a ...
// particular variable, then it is guaranteed ...
// that the taint will be removed and we pass a ...
// True which signals to the function that this ...

is a "mandatory" conditional and at least one ... of these must be executed. The "mandatory" ... flag allows the removal of taint; conditionals ... with only an If or If/ElseIf are not ... guaranteed to execute therefore we cannot ... assume that a "False" tainted result for a ... variable will occur in every case.

    if(!is_null($node->else)){
        $vars = merge_vars_arrays($saved_vars, ...
            $merged_vars, True);
        if($debug){
            echo "eval_node: If: Results from ...
                mandatory merge:\n";
            print_vars();
        }
    } else {
        $vars = merge_vars_arrays($saved_vars, ...
            $merged_vars);
        if($debug){
            echo "eval_node: If: Results from ...
                non-mandatory merge:\n";
            print_vars();
        }
    }

    return;

    case "PhpParser\Node\Stmt\ElseIf_":
    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }

        if($debug){
            echo "eval_node: Results from Stmt_ElseIf\n";
            print_vars();
        }

        return;

    case "PhpParser\Node\Stmt\Else_":
    foreach($node->stmts as $stmt){
        eval_node($stmt);
    }

        if($debug){
            echo "eval_node: Results from Stmt_Else\n";
            print_vars();
        }

        return;

    case "PhpParser\Node\Stmt\Function_":
        // Shouldn't have any redeclarations, because ...
    it'd throw
// a fatal PHP error, but let's just be safe
if(!array_key_exists($node->name, $functions)){
    $functions[$node->name] = $node;
    echo "eval_node: Adding function $node->name ... to function list\n";
}

return;

case "PhpParser\Node\Expr\FuncCall":
    // Reset our return flags
    $has_return = False;
    $returns_taint = False;

    // Get the name of the function we're calling
    $func_name = $node->name->parts[0];

    // Gather args
    $args = $node->args;

    // If it's a special, escaping function, go ...
    // handle it
    // Alternatively, if function_exists returns ...
    // True, it's a built-in
    if(in_array($func_name, $special_functions) || ...
        function_exists($func_name)){
        // Assume it has a return, since parser ...
        // doesn't know for built-ins
        $has_return = True;
        $returns_taint = handle_function($func_name, ...
            $args);
    } else {

        // Save our current variables and remove all ... existing vars
        // (since we're in a function scope now)
        $saved_vars = $vars;

        // Create a safe place to store ...
        // function-local vars because
        // we'll need to wipe the vars array to ...
        // emulate function scope
        $tmp_vars = array();

        // Make sure this is a function we've seen ...
        // defined; if not, print a warning and skip
        if(!array_key_exists($func_name, $functions)){
            echo "eval_node: NOTICE: Function not ... found in defined list: $func_name\n";
            print_r($functions);
            return;
        }
    }
// Now, define the function's parameters ...
// based on the args
$params = $functions[$func_name]->params;

// Iterate through each argument
for($i=0; $i<count($args); $i++){
    // Get name of param
    $var_name = $params[$i]->name;
    $tmp_vars[$var_name] = array(
        'tainted' => eval_node($args[$i]->value),
        'taint_flow' => ...
    );

    // If it was tainted coming in as an arg, ...
    // add the line of the function call to ... the history
    if($tmp_vars[$var_name]['tainted']){
        array_push($tmp_vars[$var_name]['taint_flow'], ...
            $currentLine);
    }
}

// Leaves only the function local variables
$vars = $tmp_vars;
if($debug){
    echo "Scope inside function $func_name: \n";
    print_vars($vars);
}

// TODO: Handle return values! Add handler ...
for Stmt\Return_ class.
$st sts = $functions[$func_name]->stms;

// Emulate each statement in the function
foreach($st ms as $stmt){
    eval_node($stmt);
}

// Merge any updates back in to the main ... array, remove any locals
$vars = merge_vars_arrays($vars, $saved_vars);

if($has_return){
    return $returns_taint;
} else {
    return;
case "PhpParser\Node\Stmt\Return_":
    $tainted = eval_node($node->expr);

    // Log that this function returns data
    $has_return = True;

    if($tainted){
        // Add the history of the taint sources in this return
        $returns_taint_history = ...
        array_merge($returns_taint_history, ...
                     get_taint_flow($node->expr));

        // Also add this return statement so the whole path can be seen
        $returns_taint_history = ...
        array_merge($returns_taint_history, ...
                     array($currentLine));
    }
    
    // Helps track if any returns generate taint
    $returns_taint = $returns_taint || $tainted;
    return $tainted;

case "PhpParser\Node\Arg":
    return eval_node($node->value);

case "PhpParser\Node\Stmt\Switch_":
    $saved_vars = $vars;
    $merged_vars = array();

    $cases = $node->cases;

    // NOTE: If there's a default, and the end result has a variable untainted, then that variable has actually been guaranteed to be untainted
    $is_default = False;

    // Evaluate each case statement sequentially.
    for($i = 0; $i < count($cases); $i++){
        // echo "Starting on case $i:\n";

        // A null condition means this is the "default" case
        if(is_null($cases[$i]->cond)){
            $is_default = True;
        }
    }
// print_r($cases[$i]);
for($j = $i; $j < count($cases) && !$break_hit; $j++){
    // Evaluate the statements for this case ...
    statement
    foreach($cases[$j]->stmts as $stmt){
        eval_node($stmt);
    }
}

// Merge the results of this branch
$merged_vars = ...
merge_vars_arrays($merged_vars, $vars);

// Reset the vars array
$vars = $saved_vars;

// Reset the break flag
$break_hit = False;

if($is_default){
    $vars = merge_vars_arrays($saved_vars, ...
    $merged_vars, True);
    if($debug){
        echo "eval_node: Switch: Results from ... mandatory merge:
        print_vars();
    }
} else {
    $vars = merge_vars_arrays($saved_vars, ...
    $merged_vars);
    if($debug){
        echo "eval_node: Switch: Results from ...
        non-mandatory merge:
        print_vars();
    }
}

return;

case "PhpParser\Node\Stmt\Break_":
    $break_hit = True;
    return;

case "PhpParser\Node\Stmt\While_":
    $saved_vars = $vars;
eval_node($node->cond);

foreach($node->stmts as $stmt){
eval_node($stmt);
}

// Merge any updates back in to the main array, remove any locals
$vars = merge_vars_arrays($vars, $saved_vars);

return;

// Use regex since all BinaryOps are the same
case (preg_match("/BinaryOp/", $node_class) ? ...
    $node_class : !$node_class):
    if(eval_node($node->left) or ...
        eval_node($node->right)){
        return True;
    }
    return False;

case "PhpParser\Node\Expr\ArrayDimFetch":
    // If our key is the result of a tainted lookup, ...
    this is now tainted
    $dim_tainted = eval_node($node->dim);
    $var_name = get_var_name($node);

    $tainted_arrays)) . "\n";

    if($dim_tainted || in_array($node->var->name, ... $tainted_arrays) || is_tainted($var_name)){
        if($debug){
            echo "eval_node: Taint generated in ...
                ArrayDimFetch on line $currentLine\n";
            return True;
        } else {
            return False;
        }
}

case "PhpParser\Node\Expr\ConstFetch":
    return False;

case "PhpParser\Node\Expr\Variable":
    return is_tainted($node->name);

case "PhpParser\Node\Scalar\Encapsed":
    $tainted = False;
    foreach($node->parts as $part){

if(eval_node($part)){
    $tainted = True;
}
return $tainted;

case "PhpParser\Node\Scalar\String_":
    return False;

case "PhpParser\Node\Stmt\InlineHTML":
    return False;

default:
    return False;
}

// Track variable taint
$vars = array();

// Store functions
$functions = array();

// Track if a function returns any data
$has_return = False;

// Flag for function taint return
$returns_taint = False;

// Track return taint history
$returns_taint_history = array();

// Flag to track when a switch-case has hit a break
$break_hit = False;

// Set up components
$parser = (new ...
    ParserFactory)->create(ParserFactory::PREFER_PHP5);
$traverser = new NodeTraverser;
$prettyPrinter = new PrettyPrinter\Standard;
$nodeDumper = new PhpParser\NodeDumper;

if (isset($argv[1])){
    $targetFile = $argv[1];
} else {

```
die("No file specified!\n");

// Content of file being analyzed
$lines = file($targetFile);

// Add our custom Name Resolver and Node Visitor to the traverser
$traverser->addVisitor(new NameResolver);
$traverser->addVisitor(new NodeVisitor);

// Open the file and parse the statements
try {
    $file_contents = file_get_contents($targetFile);
    // Print the length of the file
    if ( $debug ) {
        echo "Code is ", substr_count($file_contents, ...
                PHP_EOL), " lines long.\n";
    }
    // Parse the file to retrieve an array of statements
    $stmts = $parser->parse($file_contents);
} catch (Error $e) {
    echo 'Parse Error: ', $e->getMessage();
}

// Traverse the nodes and convert to SSA form
echo "\n\nAbstract Syntax Tree:\n";
echo "---------------------\n";
echo $nodeDumper->dump($stmts) . "\n\n\n";

// Iterate over top level nodes; eval_node will recurse into ...
// the lower-level nodes automatically
foreach($stmts as $node) {
    eval_node($node);
}

echo "\n\n---------------------\n";
echo "Vars array at the end of analysis\n";
echo "---------------------\n";
print_vars();

?>
Appendix B

Sample Code

Listing B.1: Example 1

```php
<?php
// Code from ...
// https://wordpress.org/plugins/forget-about-shortcode-buttons/
// this file contains the contents of the popup window
SOURCE = "insert";
$title = "Insert Button";
$insert_text = "Insert";
if(isset($_GET['ver']))
{
    $fasc_plugin_ver = $_GET['ver'];
}
else
{
    $fasc_plugin_ver = "";
}
if(isset($_GET['source']))
{
    if($_GET['source']=='click')
    {
        $source = "click";
        $title = "Edit Button";
        $insert_text = "Update";
    }
}
?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title><?php echo $title; ?></title>
</head>
<title><?php echo $title; ?></title>
```
<div id="tab-1-content" class="fasc-tab-content">
<div class="inputrow main">
<label for="button-text">Text</label>
<div class="inputwrap">
<input type="text" name="button-text" value="" id="button-text" value="" placeholder="Enter your text..." />
</div>
</div>
<div class="clear"></div>
</div>
<div class="inputrow main">
<label for="button-url">URL</label>
<div class="inputwrap">
<input type="text" name="button-url" value="" id="button-url" />
<br/>
</div>
</div>
<div class="inputwrap">
<label for="new-window"><small>Open link in new window?</small></label> <input type="checkbox" id="new-window" name="new-window" value="1" />
</div>
<div class="clear"></div>
</div>

<div class="inputrow">
<div class="inputcol left">
<div class="inputrowc">
<label for="text-color">Text Color</label>
<div class="inputwrap">
<input type="text" id="text-color" name="text-color" value="#ffffff" />
</div>
</div>
</div>
</div>
<div class="inputrowc">
<label for="button-type">Type</label>
<div class="inputwrap">
<select name="button-type">
<option value="fasc-type-flat" selected="selected">Flat</option>
<option value="fasc-type-flat fasc-rounded-medium">Flat Rounded</option>
<option value="fasc-type-glossy">Glossy</option>
</select>
</div>
</div>
<select name="fasc-type" size="1">
  <option value="fasc-type-glossy ... Fascy Round" />
  <option value="fasc-type-popout" ... Round" />
  <option value="fasc-type-popout ... Fascy ... Round" />
</select>
</div>
</div>
</div>
</div class="inputcol right">

<div class="inputrowc">
  <label for="button-color">Button Color</label>
  <div class="inputwrap">
    <input type="text" id="button-color" name="button-color" value="#33809e" />
  </div>
</div>

<div class="inputrowc">
  <label for="button-size">Size</label>
  <div class="inputwrap">
    <select name="button-size" size="1">
      <option value="fasc-size-xsmall"> ... Extra Small" />
      <option value="fasc-size-small"> ... Small" />
      <option value="fasc-size-medium" selected="selected"> ... Medium" />
      <option value="fasc-size-large"> ... Large" />
      <option value="fasc-size-xlarge"> ... Extra Large" />
    </select>
  </div>
</div>
</div>
</div class="clear">
<div class="clear"></div>

<div class="inputrow" style="display:none;">
  <label for="button-align">Alignment</label>
  <div class="inputwrap">
    <select name="button-align">
      <option value="" selected="selected">None</option>
      <option value="left">Left</option>
      <option value="right">Right</option>
    </select>
  </div>
</div>

</div>

<div id="tab-2-content" class="fasc-tab-content">
  <div class="inputrow">
    <div class="inputwrap left">
      <select id="icon-type-select">
        <option value="dashicons-grid">Dashicons</option>
        <option value="fa-web-grid">Web</option>
        <option value="fa-media-grid">Media</option>
        <option value="fa-form-grid">Form</option>
        <option value="fa-currency-grid">Currency</option>
        <option value="fa-editor-grid">Editor</option>
        <option value="fa-directional-grid">Directional</option>
        <option value="fa-brand-grid">Brand</option>
        <option value="fa-medical-grid">Medical</option>
      </select>
    </div>
    <div class="inputwrap right">
      <label><input type="radio" name="fasc-ico-position" value="none" class="fasc-ico-position" ...>
    </div>
  </div>
</div>
<div class="grid-container fontawesome" id="fa-editor-grid">
  <input type="hidden" value="" id="fasc-ico-val" name="fasc-ico-val" />
  <?php require_once("fa-editor-grid.php"); ?>
  <div class="clear"></div>
</div>

<div class="grid-container fontawesome" id="fa-directional-grid">
  <input type="hidden" value="" id="fasc-ico-val" name="fasc-ico-val" />
  <?php require_once("fa-directional-grid.php"); ?>
  <div class="clear"></div>
</div>

<div class="grid-container fontawesome" id="fa-brand-grid">
  <input type="hidden" value="" id="fasc-ico-val" name="fasc-ico-val" />
  <?php require_once("fa-brand-grid.php"); ?>
  <div class="clear"></div>
</div>

<div class="grid-container fontawesome" id="fa-medical-grid">
  <input type="hidden" value="" id="fasc-ico-val" name="fasc-ico-val" />
  <?php require_once("fa-medical-grid.php"); ?>
  <div class="clear"></div>
</div>

<div class="clear"></div>

<div id="tab-3-content" class="fasc-tab-content ... saved-buttons-tab">
  <div class="container-grid">
    <ul>
    </ul>
  </div>
</div>

<div id="tab-4-content" class="fasc-tab-content">
  <strong>Button Attributes</strong>
  <div class="container-grid">
    <ul>
    </ul>
  </div>
</div>
<div id="fasc-footer">
   <a href="javascript:ButtonDialog.insert( ... ButtonDialog.local_ed)" id="insert" ...
      style="display: block; line-height: ... 24px;"><?php echo $insert_text; ?></a>
</div>

</form>
</div>

</body>
</html>