國立臺灣大學管理學院資訊管理系

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

一個進階的PHP網頁應用程式安全驗證之靜態分析工具

An Improved Static Analyzer for Verifying PHP Web Application Security

葉睿元

Rui-Yuan Yeh

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 99 年 7 月

July, 2010

國立台灣大學資訊管理研究所碩士論文

指導教授：蔡益坤 博士

一個進階的PHP網頁應用程式安全驗證之靜態分析工具

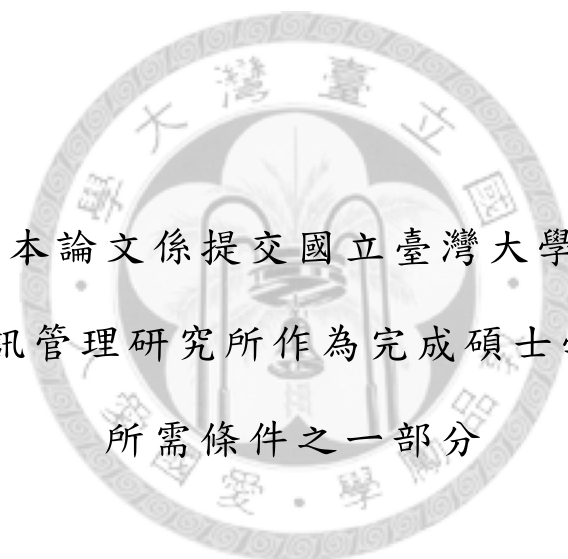
An Improved Static Analyzer for Verifying PHP Web Application Security

研究生： 葉睿元 撰

中華民國九十九年七月

一個進階的PHP網頁應用程式安全驗證之靜態分析工具

An Improved Static Analyzer for Verifying PHP Web Application Security



本論文係提交國立臺灣大學

資訊管理研究所作為完成碩士學位

所需條件之一部分

研究生：葉睿元 撰

中華民國九十九年七月

謝辭

碩士班這兩年，對自己技術能力和做事態度的成長非常多，也多了許多回憶，很感謝這過程中相處過的人、事、物。在即將離開實驗室的此時，反而有種安定和歸屬感。

首要感謝的是指導老師蔡益坤老師，兩年前進到實驗室時，很多技術的基本功都還不紮實。這兩年從老師身上學到非常多，除了技術和研究能力，還有老師做人處事的態度。謝謝老師常很有耐心和我討論分析方法，以及養成許多寫文章和做簡報的重要習慣。會記得老師常提的「累積」，持續累積自己的技術、經驗，和老師常叮嚀的英文能力。

接著是博班學長明憲，讓我見識到何謂「強者」的紮實功力，無論是看論文的理解和行雲流水的程式能力。謝謝你常在我論文看不懂時讓我看的更懂和偶爾更不懂，你是我努力成長的好模範，也謝謝你每週更新的布袋戲。

還有同屆的好夥伴，昇峰，這兩年有你陪伴真的很好，智斌祝你事業突飛猛進，我們之中你最有機會開創事業了，怡文和晉碩喜事時記得通知大家。還有常幫忙和一起討論的學弟妹任峰、奕翔、辰旻，CANTU 之後就靠你們照顧了，也祝你們研究和論文順利。

最後要感謝我的家人和女友，雖然求學過程一直離家越來越遠，但因為有你們經濟和精神上的支持，才能順利完成碩士班學業。

葉睿元　謹識

于台灣大學資訊管理研究所

民國九十九年七月

論文摘要

學生：葉睿元　　　　　　　　　　　　　　　　　　2010 年 7 月

指導教授：蔡益坤

一個進階的 PHP 網頁應用程式安全驗證之靜態分析工具

近年來，網站應用程式的重要性持續成長。隨著許多網路服務透過網站應用程式來完成，這些網站應用程式成為駭客主要的攻擊目標之一。除此之外，程式設計師在撰寫網頁應用程式時，常處於上線壓力和安全程式寫作知識不足的情形下。這些情況帶來了網站安全的威脅和重要資料外洩的可能。由於網站應用程式弱點錯綜複雜，尤其當弱點相關程式碼跨越不同函式和檔案時，沒有自動化工具的輔助，要找出所有可能的網站程式弱點是非常困難的。 PHP 是最熱門的網站開發語言之一，針對 PHP 網站應用程式安全，已有許多的程式分析技術被發展出來，特別是靜態分析方法。

在本論文中，我們根據一個現存的分析雛型工具，延伸設計和實作出一個進階的 PHP 靜態分析工具。此工具將 PHP 程式轉譯為 CIL 中介表示 (CIL intermediate representation) ，並在 CIL 中介表示上進行污染資料流分析。在 PHP5 中，新的物件導向特色帶來新的弱點位置，我們支援 PHP5 新的程式語言特色，並在轉譯過程中盡可能地保留程式原始碼語意。在靜態分析上，我們設計和實作了跨函式分析 (interprocedural analysis) 和別名分析 (alias analysis) 演算法。跨函式分析讓污染資料流分析範圍穿越函式，並提供更完整和精準的弱點掃瞄結果。別名分析能找出名稱不同但對應到相同記憶體位置的變數別名關係。最後以偵測出跨越物件和別名關係的跨網站指令碼 (XSS) 弱點來呈現此方法的有效性，我們也透過動態執行 PHP 原始碼和 CIL 中介表示確認這些弱點的存在。

關鍵字：靜態分析、資料流分析、網站應用程式安全、別名分析、安全性弱點、驗證

THESIS ABSTRACT

GRADUATE INSTITUTE OF INFORMATION MANAGEMENT
NATIONAL TAIWAN UNIVERSITY

Student: Yeh, Rui-Yuan
Advisor: Tsay, Yih-Kuen

Month/Year: July, 2010

An Improved Static Analyzer for Verifying PHP Web Application Security

The importance of Web applications has increased continually in recent years. As more and more services are delivered through Web applications, they have become a major target of security attacks. In addition, Web applications are often implemented by programmers with time-to-market pressure and limited security skills. These situations result in an increasing security threat that may lead to the compromise of sensitive information. Due to the fact that security vulnerabilities are often rather intricate, especially when the relevant code spans many different functions and source files, finding all potential vulnerabilities without the assistance of an automated tool is impractical. PHP is one of the most popular languages for Web application development. To detect security vulnerabilities in PHP Web applications, many program analysis techniques, in particular by static analysis approaches, have been developed.

In this thesis, we design and implement a static code analysis tool for PHP that improves over an existing analyzer. Our tool translates a PHP program into a CIL program and applies taint analysis on the CIL representation. We support most PHP5 features and preserve the semantics of the source program in our translation. The new object-oriented features in PHP5 bring new vulnerable points in programs. We also design and implement interprocedural analysis and alias analysis algorithms which provide support for object-oriented features of PHP. Our interprocedural analysis allows taint analysis to cross function boundaries and provide more precise and complete analysis results. Alias analysis can discover the relationship between variables that are mapped to the same memory location in program. Finally, we demonstrate the effectiveness of our approach by detecting XSS vulnerabilities that cross object and alias relationships. We also confirm these vulnerabilities by executing our CIL representation as well as the original PHP source programs.

Keywords: Static Analysis, Dataflow Analysis, Web Application Security, Alias Analysis, Security Vulnerability, Verification.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation and Objectives . . . 2
- 1.3 Thesis Outline . . . 3

2 Related Work **4**
- 2.1 Pixy: An Open Source Static Analysis Tool . . . 4
  - 2.1.1 Aliases in PHP . . . 5
  - 2.1.2 Intraprocedural Alias Analysis . . . 5
  - 2.1.3 Interprocedural Alias Analysis . . . 6
  - 2.1.4 Discussion . . . 9
- 2.2 A Static Analysis Algorithm by Xie and Aiken . . . 10
  - 2.2.1 Intrablock Analysis . . . 11
  - 2.2.2 Intraprocedural Analysis . . . 14
  - 2.2.3 Interprocedural Analysis . . . 15
  - 2.2.4 Discussion . . . 15
- 2.3 Saner: Composing Static and Dynamic Analysis . . . 17
  - 2.3.1 Sanitization-Aware Static Analysis . . . 17
  - 2.3.2 Testing Sanitation Routines . . . 17
  - 2.3.3 Discussion . . . 18
- 2.4 Stranger: An Automata-Based PHP String Analysis Tool . . . 19
  - 2.4.1 Stranger Architecture . . . 19
  - 2.4.2 Discussion . . . 20
- 2.5 A Static Analyzer by Chung . . . 21
  - 2.5.1 Conversion of PHP Variables and Arrays . . . 21
  - 2.5.2 Conversion of Accessing and Assigning Variables . . . 22
  - 2.5.3 Conversion of PHP Foreach Statement . . . 22
  - 2.5.4 Conversion of PHP User-Defined Functions and Built-In Functions 23
  - 2.5.5 PHP Dynamic File Inclusion . . . 23
  - 2.5.6 Taint Dataflow Analysis . . . 23
- 2.6 Summary of related tools . . . 25

3 Preliminaries **26**
- 3.1 Critical Web Application Security Vulnerabilities . . . 26
  - 3.1.1 Injection . . . 27
  - 3.1.2 Cross-Site Scripting (XSS) . . . 28

3.1.3 Effective Defenses for SQL Injection and XSS 29
3.1.4 Security Misconfiguration 30
3.1.5 Unvalidated Redirects and Forwards 31
3.2 Static Single Assignment Form 31
3.3 The C Intermediate Language 32
3.3.1 Control Flow Graph 33
3.3.2 Data Flow Analysis Framework 33
3.3.3 Points-to Analysis 34
3.4 Abstract Syntax Tree 35

4 Translation and Static Analysis **39**
4.1 Translation of PHP5 Language Features to CIL 40
4.1.1 Adjustments for Basic Structures and Auxiliary Functions 41
4.1.2 Translation of Class and Object 42
4.1.3 Translation of Inheritance 44
4.1.4 Translation of Magic Functions 46
4.1.5 Translation of Try-Catch Exception 48
4.1.6 Translation of Reference Assignment and Object Assignment . . . 50
4.1.7 Translation of Static Member and Class Constant 52
4.1.8 Translation of Namespace 54
4.2 Interprocedural Analysis and Alias Analysis 55
4.2.1 Interprocedural Taint Analysis 55
4.2.2 Alias Taint Analysis 64
4.2.3 Basic Dynamic Vulnerability Confirmation 66

5 Implementation and Experiments **67**
5.1 Implementation 68
5.2 Experimental Results 69

6 Conclusion **71**
6.1 Contributions 71
6.2 Further Work 72

Appendix **73**

Bibliography **79**

List of Figures

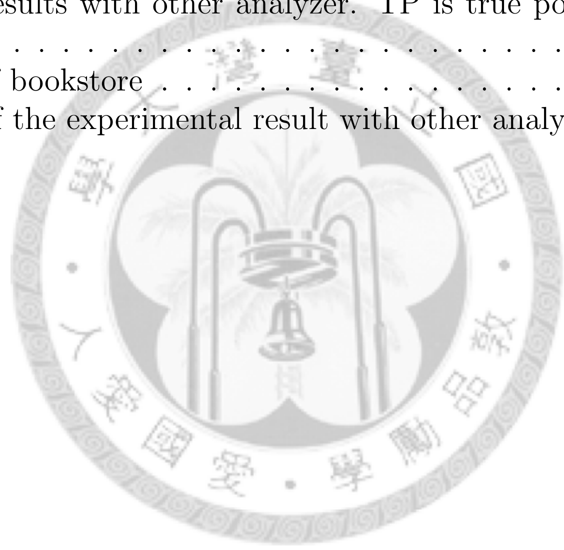
2.1 Simple aliasing in PHP. 5
2.2 Alias analysis information. 6
2.3 Aliases between the callee's formal parameters. 7
2.4 Aliases between global variables and caller's local variables. 9
2.5 Analysis process 10
2.6 Language definition. 12
2.7 Value representation. 12
2.8 Expressions. 13
2.9 Simulation state 13
2.10 Evaluation rules 14
2.11 The analysis process in Saner 17
2.12 Stranger architecture 19
2.13 The conversion of a PHP array 22
2.14 The auxiliary functions of variable assignments. 23
2.15 The algorithm of taint dataflow analysis 24

3.1 Mapping from 2007 to 2010 Top 10 27
3.2 Typical scenario for SQL Injection 28
3.3 Typical scenario for stored XSS 29
3.4 Straigh-line code conversion 31
3.5 SSA example 32
3.6 The mapping from PHP AST operator to C AST operator 36
3.7 The mapping from PHP AST expression to C AST expression 37
3.8 The mapping from PHP AST statement to C AST statement 38

4.1 Analysis architecture 39
4.2 Variable assignment representation 41
4.3 Translation of class and object 42
4.4 PHP class example 43
4.5 Translation of inheritance 44
4.6 PHP inheritance example 45
4.7 Translation of magic functions 46
4.8 PHP magicgfunction example 47
4.9 Translation of Try-Catch Exception 48
4.10 PHP try-catch example 49
4.11 Translation of alias features 50
4.12 PHP alias example 1 51

4.13 PHP object assignment example 52
4.14 Translation of static member 53
4.15 PHP namespaces example 54
4.16 Interprocedural analysis 56
4.17 The intraprocedural analysis algorithm 57
4.18 The function summary computation algorithm 58
4.19 The interprocedural analysis algorithm 60
4.20 Interprocedural analysis example 61
4.21 Function summary example 62
4.22 Whole variable taint information of Figure 4.2.1 63
4.23 An example of analysis report 63
4.24 A alias translation example 64
4.25 Alias group example 65
4.26 A alias cross object XSS example 65

5.1 The integrated analysis architecture of CANTU 67
5.2 The work flow diagram in this thesis 68
5.3 Comparison results with other analyzer. TP is true positive, FN is false negative. 69
5.4 The archive of bookstore 70
5.5 Comparison of the experimental result with other analyzer 70



Chapter 1

Introduction

1.1 Background

Web applications have become one of the most important communication channels in real world. E-commerce refers to the use of Internet and Web applications to transact business. Commercial transactions involve the exchange of value (e.g., money) across organizational or individual boundaries for products and services. Currently, the importance of Web applications is still increasing and lots of operations in industries rely on Web applications, including travel reservations, music, news, software, and finance.

As more and more services are delivered through Web applications such that these applications have become a major target of security attacks. Server-side programming is one of the most important component that support today's WWW environment. Server-side scripting languages like PHP, ASP.Net, and JSP can generate Web pages and database query dynamically. The dynamically scripting language features provide the flexibility to generate Web pages based on user's inputs and requirements.

However, these advantages also make it harder to capture dynamically generated pages that contain malicious inputs. Vulnerabilities in website allow attackers with opportunities to compromise the Web application. According to statistics provided by OWASP [20], the amounts of vulnerabilities increase continually in recent years and also lead to the compromise of sensitive information.

Furthermore, these security vulnerabilities are often rather intricate, especially when the relevant code spans many different functions, source files and includes value flows between parameters and arrays. Finding all potential vulnerabilities without the assistance

of an automated tool is impractical.

PHP enriches many popular Web applications such as Facebook [28], Wikipedia, and Yahoo. According to statistics from TIOBE [24] in June 2010, PHP is ranked 4 of the popularity of programming languages (the top three are JAVA, C, and C++). NIST NVD (National Vulnerability Database) stores all vulnerabilities found in computer software [18]. PHP-related vulnerabilities on NIST NVD amounted to 30% in 2009.

In this thesis we focus on Web application vulnerabilities that arise from insecure source code and develop automated static analysis tools for verifying PHP Web application security.

1.2 Motivation and Objectives

Vulnerabilities in a Web application may cause serious security issues, such as exposure of sensitive information or database corruption. The most common sources of vulnerabilities are the lack of proper validation for user input. Programmers are not always aware of the available protection mechanisms and security concepts.

In current commercial tools, many false positives come from imprecise source list, sink lists, and specific language features. In Academia, lots of scholars have invested in Web security research and proposed many worthwhile research topics. Chung has designed and partly implemented a static source code analyzer for verifying PHP Web application security, which translates PHP program into CIL and apply taint analysis algorithm on the CIL intermediate representation [7].

Formerly, the static analyzer by Chung [7] provides no support for PHP5 features, interprocedural analysis, and alias analysis. The new features in PHP5 such as object-oriented features also bring new vulnerable points in program (called sensitive sinks). The latest version of PHP is 5.3.2 (March 2010). It has many new language features like namespaces, object passed by reference...etc. Without interprocedural analysis, taint analysis couldn't cross function boundaries that result in restricted vulnerability reports. Alias features may result in some false negatives such as reference assignment for user input before tainted data entering sink function.

To solve these problems and enhance our analyzer, in this thesis we concentrate on

these objectives:

- Translate PHP5 features to CIL and preserve the program semantics as precisely as possible.
- Design and implement interprocedural analysis algorithm to provide more complete vulnerability report.
- Design and implement alias analysis algorithm to capture the alias relationships in program.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

- In Chapter 2, we discuss several related tools , approaches for detecting PHP Web application vulnerabilities.
- In Chapter 3, we introduce some preliminaries regarding this thesis , common Web vulnerabilities, C Intermediate Language , and static single assignment form.
- In Chapter 4, we give an overview of our translation approach for PHP5 features, interprocedural analysis, and alias analysis approach for PHP Web applications.
- In Chapter 5, we describe the implementation detail of our analyzer, present the experiment results of our evaluations.
- In Chapter 6, we summarize our contributions and indicate possible research direction in the future.

Chapter 2

Related Work

Currently, there are existing several approaches and tools for detecting vulnerabilities in Web applications in recent years. Jovanovic *et al.* introduced an alias analysis algorithm for PHP and implemented a tool named Pixy [14]. Pixy is targeted at detecting cross-cite scripting (XSS) and SQL injection vulnerabilities in PHP applications. Xie and Aiken addressed the problem of detecting SQL injection vulnerabilities statically in PHP scripts [31]. They designed an analysis algorithm for finding security vulnerabilities in PHP by using three-tier analysis architecture with block and function-summaries. Balzarotti *et al.* proposed a tool, called *Saner* with a novel approach to combine static and dynamic analysis techniques for identifying vulnerabilities in PHP programs [4]. Fang Yu *et al.* developed *Stranger*, which a string analysis tool for PHP Web applications [32]. In the following sections of this chapter, we introduce these approaches which related to our analysis approach in this thesis. Besides, we summarize the translation for PHP basic language features into CIL and intraprocedural taint analysis algorithm.

2.1 Pixy: An Open Source Static Analysis Tool

Jovanovic *et al.* proposed an alias analysis algorithm for PHP [14]. They have built a tool named Pixy [13] for detecting SQL injection XSS vulnerabilities in PHP applications. More details about Pixy could be seen in technical report [12]. Pixy is based on a data flow analysis and applies an alias analysis algorithm.

Two or more variables are *aliases* if their values are stored at the same memory location. Two variables are *must-aliases* if they are aliases in every path that is taken by the program. Variables are *may-aliases* if these variables are aliases only for some

program paths. In this section we will give a brief summary for aliases in PHP and alias analysis algorithms.

2.1.1 Aliases in PHP

An alias of a variable is a variable that refers to the same memory location. In PHP, aliases between variables can be introduced by using the *reference operator* "&". Figure 2.1 shows an example of an alias relationship between variables $a and $b. $a is a string with initial value 'test' and then $b is $a's alias variable. On line 3 $a becomes a tainted variable and $b also holds a tainted value through reference operator. Without alias information, taint analysis cannot figure out that the assignment on Line 3 does affect $b. On line 4 *echo* $b may lead to the XSS vulnerability. Apparently, lacking of aliasing information can lead to false negatives.

```
1:  $a = 'test';       // $a:  untainted
2:  $b =& $a;          // $a and $b:  untainted
3:  $a = $_GET['msg']; // $a and $b:  tainted
4:  echo $b;           // XSS vulnerability
```

Figure 2.1: Simple aliasing in PHP.

2.1.2 Intraprocedural Alias Analysis

Figure 4.17shows the main concept of their algorithm. The program annotated with alias information after the execution of the corresponding code line. The alias information is annotated at the end of each line, ("u") means *must-alias* and ("a") means *may-alias*. On Line 1, the set of alias information u and a are empty. After the reference assignment on Line 2, variables $a and $b are aliases. The algorithm records this information by adding the two variables to must-alias information set. On Line 4, a second group is created after reference assignment $c to $d. This new group is extended by variable $e as result of the statement on Line 5. Finally, the algorithm merges the alias information that coming from two different paths after the if-construct. Because the if-construct path may not always be executed, all must-aliases set created inside the if-construct must be moved into may-aliases after the if-construct block.

```
1:  $b=10;          // u{} a{}
2:  $a =& $b;       // u{(a, b)} a{}
3:  if ($a==$b) {
4:    $c =& $d;     // u{(a, b) (c, d)} a{}
5:    $d =& $e;     // u{(a, b) (c, d, e)} a{}
6:  }
7:  echo $a;        // u{(a, b)} a{(c, d) (c, e) (d, e)}
```

Figure 2.2: Alias analysis information.

Instead of using sets of variables, the algorithm records may-aliases by means of unordered variable pairs. Hence, the must-alias group (c,d,e) is split into the three may-alias pairs (c,d), (c,e), and (d,e). The reason for this asymmetric recording of must-alias and may-alias information is that it simplifies the algorithms necessary for interprocedural analysis.

2.1.3 Interprocedural Alias Analysis

When encountering a function call, it provides some information to the caller and callee of the function call. From callee's view, the analysis has to provide:

- Aliases between global variables.
- Aliases between callee's formal parameters.
- Aliases between global variables and callee's formal parameters.

And from caller's view, the analysis has to provide:

- Aliases between global variables
- Aliases between global variables and caller's local variables.

In the following sections, we will discuss these issues mentioned above in detail.

- Aliases between global variables

  The alias relationships between global variables are important for both the caller and the callee. The callee must know about a how global variables are aliased at

the time the function call is performed. The caller must be informed about how the global aliasing information has been modified by the callee.

The approach provides all the alias information of global variables when invoking a function, and perform the intraprocedure alias analysis in it. When the function calls another function, it passes all the alias information about the global variables to the callee.

- Alias between the callee's formal parameters
  Aliases between formal parameters appear when there exists an alias relationship between the corresponding actual call-by-reference parameters. For example, the function "callee" has two call-by-reference parameters ,$fp1 and $fp2, as shown in Figure 2.3. The actual parameter in function "caller" are $cp1 and $cp2 which are must-alias when calling callee and results in that the formal parameters $fp1 and $fp2 are must-alias as well.

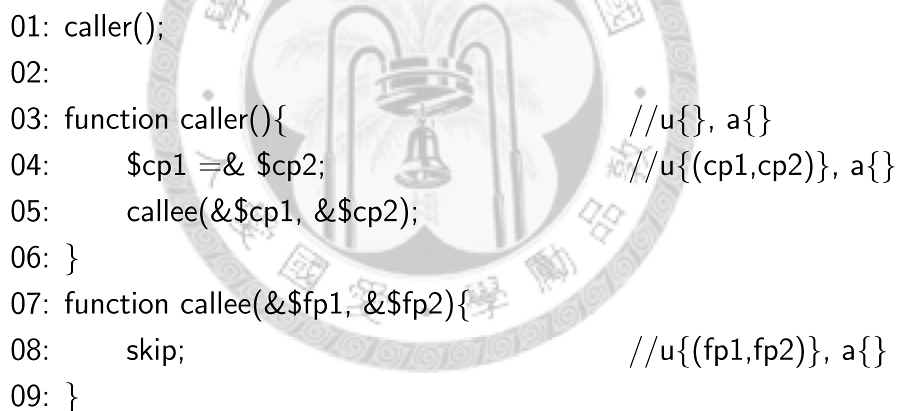


Figure 2.3: Aliases between the callee's formal parameters.

- Aliases between Global Variables and the Callee's Formal Parameters
  This issue is the same with the second issue so the method of finding alias information is similar with it. Although the corresponding actual parameter of the callee's formal parameter has three cases: It is a global variable, It is a must-alias of a global variable or It is a may-alias of a global variable, we need not to consider this condition. In the analysis process, we just simply find the actual parameter's alias

relation, if it has must- or may- alias relation with any global variable, the corresponding formal parameter also has the same alias relation with the same global variable.

- Aliases between Global Variables and the Caller's Local Variables

  In this case, the situation is more complex because of the alias between the caller' s local variables and global variables can be modified by the callee in several ways. For all the global variables and formal parameters which has alias relations with caller's local variable or the specific global variable, every changes of the alias relations may change the alias relations between them. Note that the basic rule of interprocedure analysis is only stores information about global variables and its own local variables inside functions. Thus, it is necessary to use another variable to store the change of alias relations.

  In their approach, for each global variables, Pixy creates a local variable, *shadow variable* in every function. The auxiliary shadow variable and the corresponding global variable are must-alias in the functions. Figure 2.4 shows the alias information of $er.s_g1, $er.s_g2, $ee.s_g1 and $ee.s_g2. On Line 5, the function "callee" changes the global variable $g1's alias relation. Without shadow variables, caller can only know that $g1 and $g2 are alias. After invoking the callee, it can not differ that whether $g1 is aliased to $g2 or not. So caller can not say that $er1 and $g1 are still alias or not. With shadow variables, caller can see that after invoking callee, the shadow variable $er.s_g1 and $g1 are not alias. Thus, the alias information knows that $er1 and $g1 are not alias.

Another important issue is the handling with the recursive function calls. It is hard to statically decide the number of the level of the recursive call chain will reach at runtime. In this situation, the approach have to store every local variables in every function instance. The static analysis would face infinite number of variables and the analysis would not terminate. Pixy provides a solution to solve this problem, that is, the analysis only stores information about global variables and its own local variables inside functions. This rule makes sure that the number of variables in the analysis process is finite and thus can perform static analysis in this condition. More details about approaches and

```
01: caller();
02: skip;                                      //u{(g1,g2)}, a{}
03:
04: function caller(){                         //u{(g1,er.s_g1), (g2,er.s_g2)}, a{}
05:       $er1 =& $GLOBALS['g1'];              //u{(g1,er.s_g1,er1), (g2,er.s_g2)}, a{}
06:       callee();
07:       skip;                                //u{(er1,er.s_g1), (g1,g2,er.s_g2)}, a{}
08: }
09: function callee(){                         //u{(g1,ee.s_g1), (g2,ee.s_g2)}, a{}
10:       $GLOBALS['g1']; = & $GLOBALS['g2'];  //u{(g2,ee.s_g2,g1)}, a{}
11: }
```

Figure 2.4: Aliases between global variables and caller's local variables.

algorithms could be found in *Web application security* by jovanovic [11]

2.1.4 Discussion

Pros

Jovanovic *et al.* proposed an alias analysis algorithm that fits with PHP's variable reference property. It improves the accuracy of alias analysis in PHP web application.

Cons

The analysis did not support for object oriented features of PHP. This means that object or member variables never appear as elements of alias relationships. Besides, reference statements that contain arrays or array elements are not considered. A tainted value propagated through alias relationships between array elements and objects cannot be detected.

2.2 A Static Analysis Algorithm by Xie and Aiken

Xie and Aiken address the problem of statically detecting SQL injection vulnerabilities by applying a three-tier architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level [31]. Given a PHP source file, their tool carries out static analysis in the following steps:

- Parse a PHP source into abstract syntax trees (ASTs)
- For each function in the program, convert the AST of the function body to a CFG
- Summarize the collective effects of statements in a basic block into a block summary
- Use a reachability analysis to combine block summaries into a function summary
- Use a interprocedural analysis, when a statement calls to other user-defined functions

Figure 2.5 illustrates how the analysis process works.

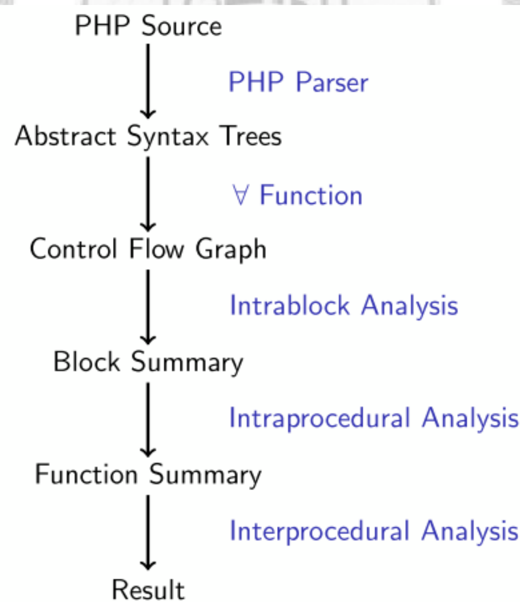


Figure 2.5: Analysis process

2.2.1 Intrablock Analysis

The simulation starts in an initial state, which maps each variable x to a symbolic initial value x_0. It processes each statement in the block in order, updating the simulator state to reflect the effect of that statement and using information contained in the final state of the simulator to summarize the effect of the block into a **block summary**.

Block summary Characterizes behavior of CFG block into block summary represented as six-tuple $\langle \mathcal{E}_\mathcal{B}, \mathcal{D}_\mathcal{B}, \mathcal{F}_\mathcal{B}, \mathcal{T}_\mathcal{B}, \mathcal{R}_\mathcal{B}, \mathcal{U}_\mathcal{B} \rangle$:

- **Error set**
  The set of input variables which must be sanitized before entering the current block
- **Definitions** ($\mathcal{D}_\mathcal{B}$):
  The set of memory locations defined in the current block
- **Value flow** ($\mathcal{F}_\mathcal{B}$):
  The set of location pairs (l_1, l_2) where string value of l_1 on entry becomes a substring of l_2 on exit
- **Termination predicate** ($\mathcal{T}_\mathcal{B}$):
  True if the block contains exit statement or calls another termination function
- **Return value** ($\mathcal{R}_\mathcal{B}$):
  Representation for return value
- **Untaint set** ($\mathcal{U}_\mathcal{B}$):
  Sanitized locations for each successor

Figure 2.6 gives the definition of a small imperative language captures a subset of PHP constructs that the authors believe is relevant to SQL injection vulnerabilities.

They model three basic type **basic types** of PHP values:

- **String**: Most fundamental type
- **Boolean**: Common result of validation checks
- **Integer**: Less emphasized in the simulation In addition, the authors introduce a special $\top$ **type** to describe objects whose static types are undetermined.

$$
\begin{array}{ll}
\texttt{Type } (\tau) & ::= \texttt{str} \mid \texttt{bool} \mid \texttt{int} \mid \top \\
\texttt{Const } (c) & ::= \texttt{string} \mid \texttt{int} \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \\
\texttt{L-val } (lv) & ::= x \mid \texttt{Arg\#i} \mid lv[e] \\
\texttt{Expr } (e) & ::= c \mid lv \mid e \texttt{ binop } e \mid \texttt{unop } e \mid (\tau)e \\
\texttt{Stmt } (S) & ::= lv \leftarrow e \mid lv \leftarrow f(e_1, ..., e_n) \mid \texttt{return } e \mid \texttt{exit} \mid \texttt{include } e \\
\texttt{binop} & \in \{+,\ -,\ \texttt{concat},\ ==,\ !=,\ <,\ >,\ \ldots\} \\
\texttt{unop} & \in \{-,\ \neg\}
\end{array}
$$

Figure 2.6: Language definition.

Expressions can be constants, *l-values*, *unary* and *binary operations*, and *type casts*. A statement can be an *assignment*, *function call*, *return*, *exit*, or *include*.

The following figures illustrate the Introblock simulation algorithm.

Figure 2.7 for value representations.

$$
\begin{array}{ll}
\texttt{Loc } (l) & ::= x \mid l\texttt{[string]} \mid l[\top] \\
\texttt{Init-Values } (o) & ::= l_o \\
\texttt{Segment } (\beta) & ::= \texttt{string} \mid \texttt{contains}(\sigma) \mid o \mid \bot \\
\texttt{String } (s) & ::= \langle \beta_1, ..., \beta_n \rangle \\
\texttt{Boolean } (b) & ::= \texttt{true} \mid \texttt{false} \mid \texttt{untaint}(\sigma_0,\ \sigma_1) \\
\texttt{Loc-set}(\sigma) & ::= \{l_1,\ \ldots,\ l_n\} \\
\texttt{Integer } (i) & ::= k \\
\texttt{Value } (v) & ::= s \mid b \mid i \mid o \mid \top
\end{array}
$$

Figure 2.7: Value representation.

Figure 2.8 for Expressions.

$$\texttt{cast}(k, \texttt{bool}) = \begin{cases} \texttt{true} & \texttt{if } k \neq 0 \\ \texttt{false} & \texttt{otherwise} \end{cases}$$

$$\texttt{cast(true, str)} = \langle \text{``1''} \rangle$$

$$\texttt{cast(false, str)} = \langle \rangle$$

$$\texttt{cast}(v = \langle \beta_1, ..., \beta_n \rangle, \texttt{bool}) =$$

$$\begin{cases} \texttt{true} & \texttt{if } (v \neq \langle \text{``0''} \rangle) \wedge \bigvee_{i=1}^{n} \neg \texttt{is_empty}(\beta_i) \\ \texttt{false} & \texttt{if } (v = \langle \text{``0''} \rangle) \vee \bigwedge_{i=1}^{n} \texttt{is_empty}(\beta_i) \\ \top & \texttt{otherwise} \end{cases}$$

Figure 2.8: Expressions.

Figure 2.9 for Simulation state maps memory locations to their value representations.

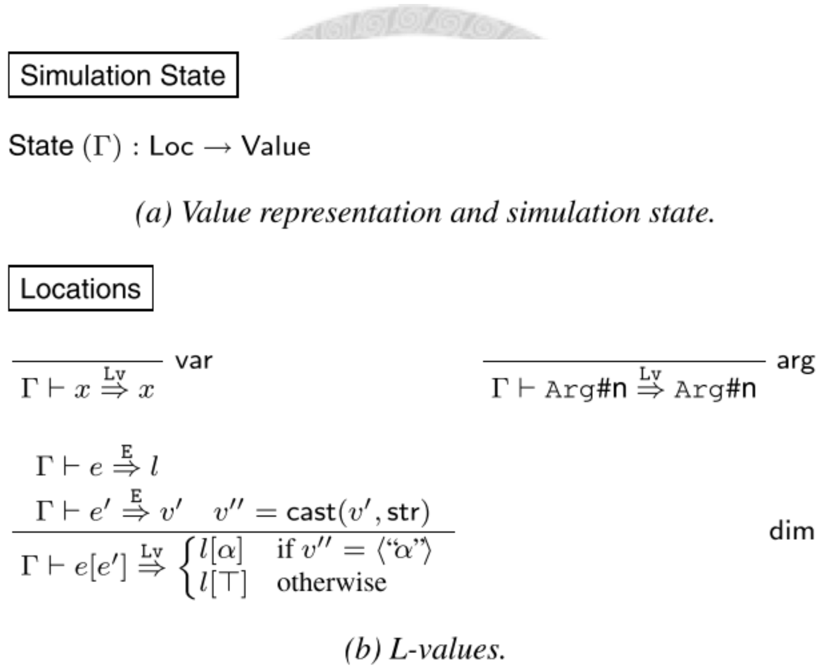


Figure 2.9: Simulation state

Figure 2.10 for evaluation rules.

Evaluation Rules:

$$\frac{\Gamma \vdash lv \overset{\text{Lv}}{\Rightarrow} l}{\Gamma \vdash lv \overset{\text{E}}{\Rightarrow} \Gamma(l)} \quad \textsf{L-val}$$

$$\frac{\Gamma \vdash e_1 \overset{\text{E}}{\Rightarrow} v_1 \quad \textsf{cast}(v_1, \textsf{str}) = \langle \beta_1, \ldots, \beta_n \rangle \qquad \Gamma \vdash e_2 \overset{\text{E}}{\Rightarrow} v_2 \quad \textsf{cast}(v_2, \textsf{str}) = \langle \beta_{n+1}, \ldots, \beta_m \rangle}{\Gamma \vdash e_1 \ \textsf{concat}\ e_2 \overset{\text{E}}{\Rightarrow} \langle \beta_1, \ldots, \beta_m \rangle} \quad \textsf{concat}$$

$$\frac{\Gamma \vdash e \overset{\text{E}}{\Rightarrow} v \quad \textsf{cast}(v, \textsf{bool}) = v'}{\Gamma \vdash \neg e \overset{\text{E}}{\Rightarrow} \begin{cases} \textsf{true} & \text{if } v' = \textsf{false} \\ \textsf{false} & \text{if } v' = \textsf{true} \\ \textsf{untaint}(\sigma_1, \sigma_0) & \text{if } v' = \textsf{untaint}(\sigma_0, \sigma_1) \\ \top & \text{otherwise} \end{cases}} \quad \textsf{not}$$

(c) Expressions.

Figure 2.10: Evaluation rules

2.2.2 Intraprocedural Analysis

Intrablock Analysis Bases on block summaries computed in the previous step, the intraprocedural analysis computes the following function summary $\langle \mathcal{E}_{\mathcal{F}}, \mathcal{R}_{\mathcal{F}}, \mathcal{S}_{\mathcal{F}}, \mathcal{X}_{\mathcal{F}} \rangle$ for each function. Intraprocedural analysis yields a summary for each function:

- **Error set** ($\mathcal{E}_{\mathcal{F}}$):

  The set of memory locations which must be sanitized before invoking this function

- **Return set** ($\mathcal{R}_{\mathcal{F}}$):

  The set of parameters or global variables which may be a substring of the return value *$\mathcal{R}_{\mathcal{F}}$ is only computed for functions that may return string values.

- **Sanitized values** ($\mathcal{S}_{\mathcal{F}}$):

  The set of parameters or global variables which are sanitized on function exit

- **Program exit** ($\mathcal{X}_{\mathcal{F}}$):

  The current function terminates program execution on all paths

2.2.3 Interprocedural Analysis

Due to the dynamic nature of PHP (e.g., include statements), the authors analyze functions on demand. Then, the function summary is memorized to avoid redundant analysis. If the interprocedural analysis encounters a cycle, the current implementation uses a dummy "no-op" summary as a model for the second invocation (i.e., the authors do not compute fixpoints for recursive functions.). In theory, this is a potential source of **false negatives**, which can be removed by adding a simple iterative algorithm that handles recursion.

For each function call $f(e_1, ..., e_n)$ encountered during intrablock simulation, the following is determined:

- **Preconditions**:
  The set of parameters and global variables which must be sanitized before calling f (substitute actual parameters for formal parameters in $\mathcal{E}_\mathcal{F}$)
- **Postconditions**:
  The set of sanitized input parameters and global variables (substitute actual parameters for formal parameters in $\mathcal{S}_\mathcal{F}$)
- **Exit condition**:
  If callee marked as exit function ($\mathcal{X}_\mathcal{F}$ is true), modify and mark block
- **Return value**:
  Use untaint representation ($\mathcal{S}_\mathcal{F}$) if returns a boolean value, or function summary ($\mathcal{R}_\mathcal{F}$) representation if returns a string value

2.2.4 Discussion

Pros

Xie and Aiken proposed a three-tier architecture to handle dynamic features such as dynamic typing and code inclusion in PHP. The analysis uses an interprocedural analysis and thus models information flow that can cross function boundaries. They also demonstrate a effective approach for program analysis.

Cons

Recursive function calls are simply ignored with a "no-op" summary that may result in potential false negatives. In this approach does not perform alias analysis for alias relationships between variables in program.

2.3 Saner: Composing Static and Dynamic Analysis

Balzarotti *et al.* developed a tool, called *Saner* with a novel approach to combine static and dynamic analysis techniques to identify vulnerabilities in PHP programs. Their approach works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process. The last stage is identifying whether the sanitization routines in program is correct or not.

Figure 2.11 illustrates the main analysis process.

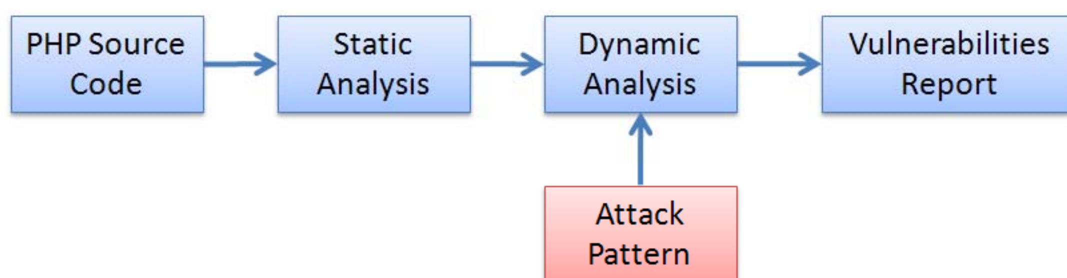


Figure 2.11: The analysis process in Saner

大 學 數 位 學 安

2.3.1 Sanitization-Aware Static Analysis

It is based on *Pixy* and improved the data flow analysis by using more precisely string analysis. It not only considers variables are taint or not, but also record possible values of variables by finite automata. They support language-based replacement but only bounded computation for loops and approximate variables in a loop as arbitrary strings. On the other hand, Fang Yu *et al.* incorporate the widening operator to handle this problem [34].

2.3.2 Testing Sanitation Routines

In static analysis phase, the result is conservative and may generate false positives. During the dynamic stage, they test the effectiveness of the sanitization routines along the paths between a source and the corresponding sink. By executing the corresponding sanitation

routines and using attack strings as input, the testing result could be used to confirm whether the sanitization is effective or not.

2.3.3 Discussion

Pros

Saner addresses the shortcomings of current static analysis tools and combines dynamic analysis to reduce the number of false positives. This approach also can identify incorrect sanitization routines.

Cons

Limitation in *Saner* is bounded computation for loops and approximate variables in a loop as arbitrary strings. This condition may lead to false positives.

2.4 Stranger: An Automata-Based PHP String Analysis Tool

Stranger is an automata-based string analysis tool developed by Fang Yu *et al.* [32] for PHP web applications which can detect XSS, SQLI and MFE vulnerabilities (OWASP Top 10). It takes a PHP program as input and automatically analyzes it and outputs the possible XSS, SQLI and MFE vulnerabilities in the program. In addition to that, for each input that leads to vulnerability, it outputs an automaton that characterizes all possible string values for this input which may exploit the vulnerability.

There are two types of static code analyses to detect vulnerabilities.

- Taint analysis
- String analysis

2.4.1 Stranger Architecture

The following figure 2.12 is redrawn from Stranger [33] and shows the architecture of stranger and its different components.

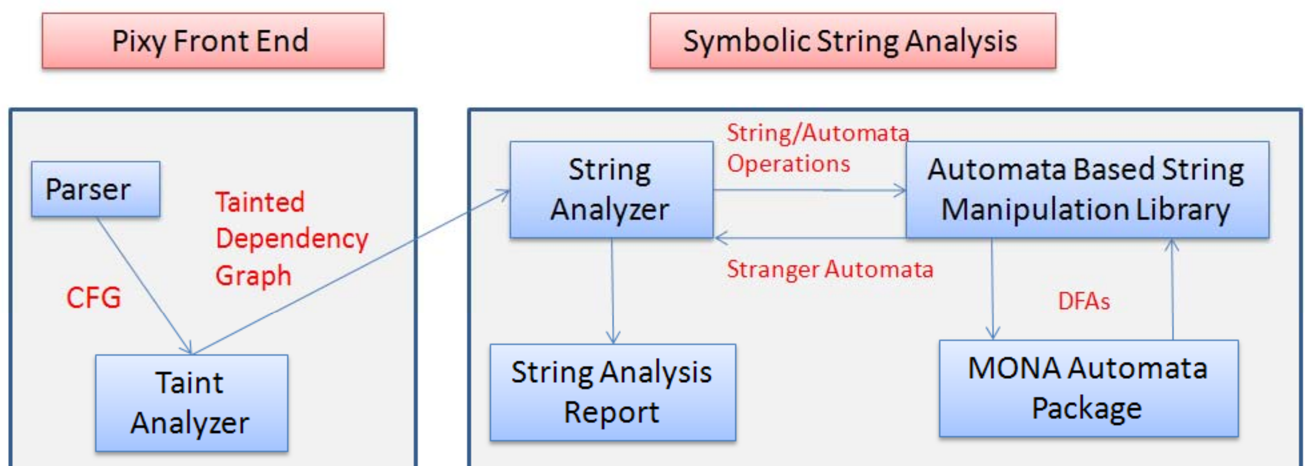


Figure 2.12: Stranger architecture

The first part parser executes literal analysis, type analysis, . . . etc and then build a full **Control Flow Graph** for further analyses. In taint analysis part is inherited from

Pixy with some modifications. If a sink is found to be tainted by taint analysis then its Dependency Graph is passed to String Analyzer. A Dependency Graph for a certain sink shows all program points for inputs that this sink value depends on and how data flow through the program CFG from these program points towards this sink.

In next stage, string analysis will calculate all possible string values that may reach the tainted sink node in its dependency graph. It will start from the input nodes in the sink's dependency graph, calculates all possible string values for each node (depending on child nodes) and propagate these values until it reaches the sink node.

The string values for input nodes will be approximated as Σ^*. There are some existing string analysis approaches [6] [15]. The possible string values at each node in the dependency graph are represented by an automaton using a symbolic representation with BDDs. The automaton that represents all possible values that may reach the sink node will be printed in a dot format to the standard output

After stranger discovers a vulnerability, it will run (an optional) backward analysis to characterize all possible input values that may result in this vulnerability. The result is an automaton for each input node in the dependency graph that represents (an over approximation of) all possible values that can be used at this input node to compromise the vulnerability in the vulnerable sink. This automaton will be output to the standard output in dot format.

2.4.2 Discussion

Pros

Stranger not only detects vulnerabilities in the program but also calculates possible string values for program points. This approach demonstrates an effective algorithm in checking the correctness of sanitization operations in PHP applications.

Cons

For each built-in functions in target program, string analysis stage have to establish a mapping automata-based operation. The number of built-in functions in real world PHP program paths will result in significant costs.

2.5 A Static Analyzer by Chung

Chung have designed and partly implemented a static source code analyzer for verifying PHP Web application security, which translates PHP program into CIL and apply taint analysis algorithm on the CIL intermediate representation [7]. The analyzer is implemented with the Objective Caml programming language [17] and CIL library. More details could be found in Chung [7]. The static analyzer operates in two main phases.

- Translate PHP program into CIL
- Apply intraprocedural taint analysis

In the following of this section, we summarize these translations for PHP4 features.

2.5.1 Conversion of PHP Variables and Arrays

In PHP, the programmer does not need to make a variable declaration before the using of the variable. A variable's type is determined by the type of the value on the right hand side variable through variable assignments. C *union* and *struct* constructor are used to represent it.

An array in PHP is actually an ordered map that associates keys to values. Besides, the indices of a PHP array can be a string or an integer. Chung use *hashtable* to implement the structure of PHP arrays instead of using the origin C array. Chung use two hashtables which are named "*index*" and "*val*" to record a variable's indices and its corresponding value. The pseudo code in Figure 2.13 shows how we convert a PHP array variable. The first argument is a struct variable V stores current dimension information, the second argument is the array's index values IDX_i from left to right. Chung abbreviate hashtable "index" to *HASH_IDX* and hashtable "val" to *HASH_VAL* below.

```
Function resolve_arr(V, IDX_i)
1 Check IDX_i in V.HASH_IDX
2   If IDX_i exists in V.HASH_IDX then
3     return the corresponding index value OdrIDX
4   else
5     OdrIDX := count(V.HASH_IDX)+1
6     Insert (IDX_i, OdrIDX) into V.HASH_IDX
7 Check OdrIDX in V.HASH_val
8   If OdrIDX exists in V.HASH_VAL then
9     return the corresponding struct variable VAR
10   else
11     VAR : = malloc a new struct variable
12     insert (OdrIDX, VAR) into V.HASH_VAL
13 If IDX_i is the last array index then
14   return VAR
15 else
16   call resolve_arr(VAR, IDX_i+1)
```

Figure 2.13: The conversion of a PHP array

2.5.2 Conversion of Accessing and Assigning Variables

About variable assignments, Chung represents it with auxiliary functions written in C. The auxiliary functions access the corresponding fields in struct variable and record its current type in val_type field. In the conversion, a variable's type is evaluated and use a corresponding auxiliary function to assign the variable. If the type of a variable cannot be evaluated in parsing phase, the variable will be assigned with "=" operator. Figure 2.14 is the list of auxiliary assignment functions.

2.5.3 Conversion of PHP Foreach Statement

In PHP, *foreach* statement simply gives an easy way to iterate over arrays orderly. There are two syntaxes of *foreach* statement.

```
foreach (array_expression as $value)
    statement
```

VariableType/Variable kind	Single variables	Array variables	Variable variables
boolean	var_assign_bool	arr_assign_bool	vv_assign_bool
int	var_assign_int	arr_assign_int	vv_assign_int
float	var_assign_float	arr_assign_float	vv_assign_float
string	var_assign_str	arr_assign_str	vv_assign_str
resource	var_assign_res	arr_assign_res	vv_assign_res

Figure 2.14: The auxiliary functions of variable assignments.

```
foreach (array_expression as $key => $value)
    statement
```

Chung convert *foreach* with *for* constructor and some auxiliary functions.

2.5.4 Conversion of PHP User-Defined Functions and Built-In Functions

In PHP, user-define functions do not have to specify the return type and its argument type. Even the function takes an integer as its parameter, it can be invoked with a string type variable without problems. Thus the return type of all user-define functions is assumed *mixed* or void type. All function's parameter is also mixed variable type. If the actual parameter is a constant value, transform it to mixed type with some auxiliary functions.

There are a large number of built-in functions in PHP. While converting them to CIL, check the function prototype declaration based on PHP manual [22] first, then rename them in this form: PHP_*function name*. This mechanism can help us to figure out the function's right semantic in CIL. It also avoids facing same function name from different source language (e.g. C#).

2.5.5 PHP Dynamic File Inclusion

When the analyzer meet file inclusion, it will ask the user what files should be included because these files cannot be determined during parsing phase.

2.5.6 Taint Dataflow Analysis

In the analyzer, taint dataflow analysis [5] has two steps:

- forward dataflow analysis
- backward dataflow analysis

Step1 is a *forward dataflow analysis* which record **variable dependence information** and **taint information**; Step2 is a *backward dataflow analysis* which find tainted variables in sink functions and trace back its taint source. This analysis algorithm can find taint-style vulnerabilities like XSS and SQL injection. Figure 2.15 shows the pseudocode of the algorithm.

```
Traverse the CIL AST nodes
Foreach node n in CIL AST
If n = instruction node
  Vname := lvalue's name in this instruction.
  Rv := right hand side value in the instruction.
  T_flag  := Check rv is tainted or untatinted.
  Obtain new variable information vi (vname, rv, t_flag)
  Update local variable information map with vi
  Update whole variable information map with vi and line number
else if n = joint node of "if statement"
  Merge variable information maps from previous paths
else if n = joint node of "loop statement"
  if old variable information = new variable information then
    Algorithm has reached a fixed point, go to next node
  else
    Merge old variable information with new variable information map
    Revisit this loop block
```

Figure 2.15: The algorithm of taint dataflow analysis

2.6 Summary of related tools

Pixy is able to detect taint-style vulnerabilities and support precise alias analysis for PHP applications. Xie and Aiken show us typical and effective three-tier analysis architecture and an approach to cross function boundaries. *Stranger* combines *pixy* and their string analysis technique that could check the effective of sanitization function. *Saner* demonstrate an integrated static and dynamic analysis approach to reduce false positives. The pros and cons in these related approaches and tools show us typical benchmarks for developing our static analysis approach.

Chapter 3

Preliminaries

3.1 Critical Web Application Security Vulnerabilities

In this section we will describe the latest common Web application vulnerabilities and present the two critical vulnerabilities, cross-site scripting (XSS) and Injection [9].

The OWASP (an acronym derived from "**O**pen **W**eb **A**pplication **S**ecurity **P**roject") Top 10 represents a broad consensus about what are the most critical web application security flaws [21]. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.

The following is the latest released OWASP Top 10 for critical Web application vulnerabilities. According to the OWASP Top 10-2010 document [1] (April 2010), the most common Web security vulnerabilities are

1. Injection
2. Cross-Site Scripting (XSS)
3. Broken Authentication and Session Management
4. Insecure Direct Object Reference
5. Cross-Site Request Forgery (CSRF)
6. Security Misconfiguration (NEW)
7. Insecure Cryptographic Storage

8. Failure to Restrict URL Access

9. Insufficient Transport Layer Protection

10. Unvalidated Redirects and Forwards

Figure 3.1 is redrawn from OWASP [21] and shows the mapping from 2007 Top 10 to 2010 Top 10.

OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	A1 – Injection
A1 – Cross Site Scripting (XSS)	A2 – Cross Site Scripting (XSS)
A7 – Broken Authentication and Session Management	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	A6 – Security Misconfiguration (NEW)
A10 – Failure to Restrict URL Access	A7 – Insecure Cryptographic Storage
<not in T10 2007>	A8 – Failure to Restrict URL Access
A8 – Insecure Cryptographic Storage	A9 – Insufficient Transport Layer Protection
A9 – Insecure Communications	A10 – Unvalidated Redirects and Forwards (NEW)
A3 – Malicious File Execution	<dropped from T10 2010>
A6 – Information Leakage and Improper Error Handling	<dropped from T10 2010>

Figure 3.1: Mapping from 2007 to 2010 Top 10

According to statistics provided by OWASP [20], the most two common types of attacks are Cross-site scripting (XSS) [27] and SQL injection [29].

Injection and XSS are still the most prevalent in Web application attacks. We describe the two most critical vulnerabilities in detail and introduce the two new members in 2010 Top 10, Security Misconfiguration and Unvalidated Redirects and Forwards.

3.1.1 Injection

Injection means user's inputs are used directly as command arguments to interpreter. Interpreter then takes strings as command to operations such as SQL, OS Shell, LDAP, . . . etc. Injection flaws usually lead to the compromise of database such that attacker may read, modified, even drop database tables.

Figure 3.2 shows a typical scenario for SQL injection. The vulnerable website takes user input as parameter of SQL query and replies the result of database query. When user’s input is **5018**, The vulnerable website replies the result of **select * from user where id=’5018’**. But let’s suppose attacker’s input is “**0’ or ’1’=’1**”, the query becomes **select * from user where id=’0’ or ’1’=’1’** which has the effect of all user information.

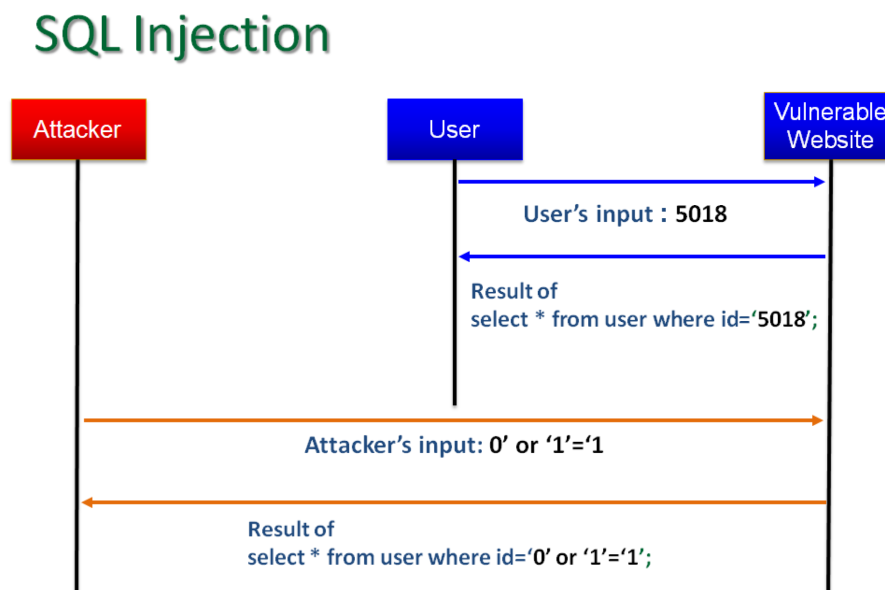


Figure 3.2: Typical scenario for SQL Injection

3.1.2 Cross-Site Scripting (XSS)

XSS occurs when any time raw data from attacker is sent to an innocent user’ s browser directly. There are two typical types of XSS:

- Reflected XSS
- Stored XSS

One of the main purposes of XSS attack is to steal the credentials (e.g., the cookie) of an authenticated user. XSS vulnerabilities not only steal sensitive information but also could redirect user to phishing or malware site.

Figure 3.3 show a typical scenario for Stored XSS.

The attacker can post a malicious message onto the bulletin board. When victim visits the vulnerable website, at first the vulnerable website set cookie for victim. Once victim reads the bulletin board and then browser executes the malicious JavaScript which may steal the cookie of victim and transmit the cookie to the attacker.

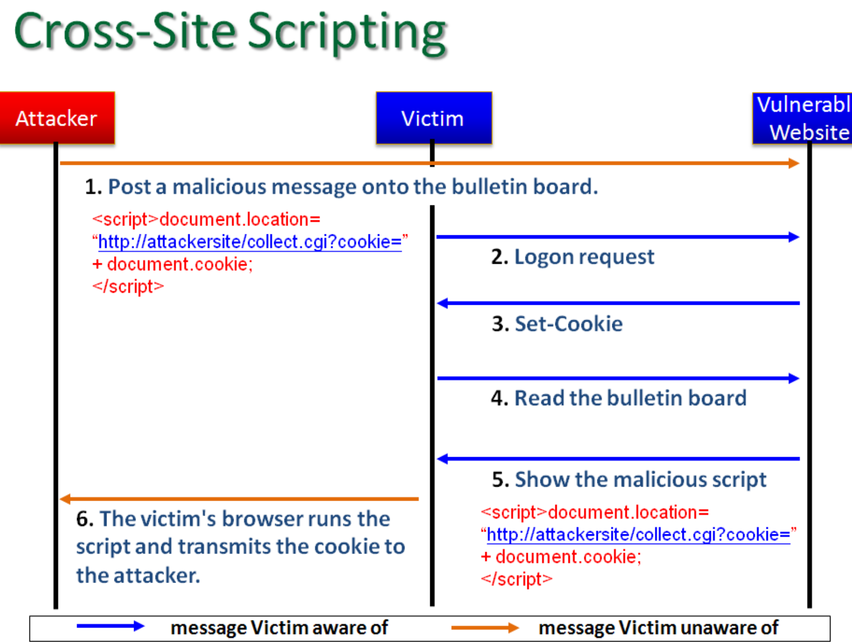


Figure 3.3: Typical scenario for stored XSS

3.1.3 Effective Defenses for SQL Injection and XSS

The following are some basic possible protection mechanisms of Cross-site scripting (XSS) and SQL injection vulnerabilities in PHP [23].

For SQL injection,

- In PHP.ini configuration, let magic_quote_gpc= On
- Use *addslashes*() related functions to escape malicious characters
- Server-side prepared statement

When magic_quote_gpc=On, all ′ (single-quote), ” (double quote), \ (backslash) and NULL characters are escaped with a backslash automatically. These characters usually are materials for composing malicious injection inputs. *addslash*() related functions

returns a string with backslashes before characters that need to be quoted in database queries etc. (Ex ′,” , \, NULL). Both methods can prevent basic SQL injections but at the same time lose the valid usage of these characters (e.g. single quote in Merry X’mas). Prepared statements are the ability to set up a statement once, and then execute it many times with different parameters. The following is a typical example:

```
SELECT * FROM Country WHERE code = ?
```

The “?” is what is a called a placeholder. When you execute the above query, you would need to supply the value for it, which would replace the “?” in the query above. This method exists in MYSQL extension version and needs some configurations.

For cross-site scripting (XSS),

- Encoding strings before output to browser.
- Customized Whitelist and Blacklist characters.

The first method is using *htmlentities*(), *htmlspecialchars*() related functions to encoding strings. Script tags are converted into tokens that are no longer interpreted by a browser as the start of JavaScript code.

The second method is using string functions to sanitize malicious characters such as <, >, %, / etc. But how to organize a proper and effective sanitization functions is still difficult to programmers.

When programmers do not have sufficient knowledge of secure programming, It is also difficult for programmers to discover potential vulnerabilities by themselves. One mechanism for discovering vulnerabilities is to review code manually, but this mechanism is impractical and cost lots of time.

3.1.4 Security Misconfiguration

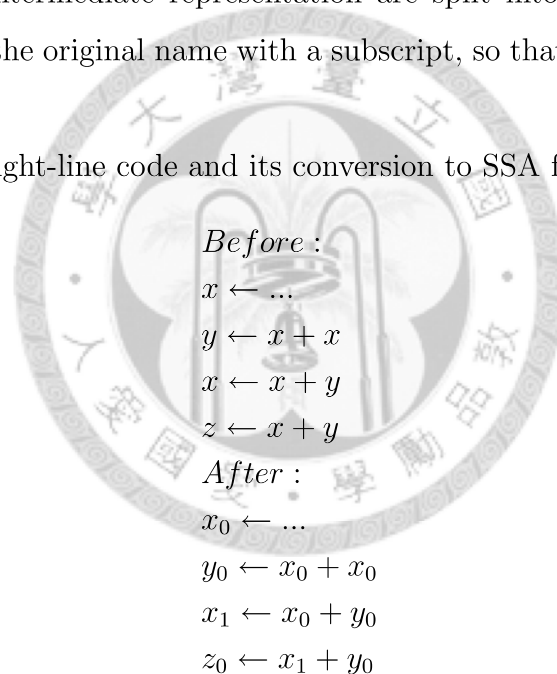
Web applications rely on a secure foundation. The configuration of server before online should be protect and refuse access of unauthorized directories or Web pages. This vulnerability usually lead to the opportunities for attacker to install backdoor through missing network or server patch or Unauthorized access to default accounts.

3.1.5 Unvalidated Redirects and Forwards

In Web application, redirects are very common and frequently include user supplied parameters in the destination URL. When the inputs aren' t validated, attacker can send victim to specific sites. This vulnerabilities usually result in victim be redirected to phishing site or malware site.

3.2 Static Single Assignment Form

static single assignment form (often abbreviated as **SSA form** or **SSA**) is an intermediate representation in which every variable is assigned exactly once [30]. In compilers SSA form usually is used to facilitate program analysis and optimization. Existing variables in the original intermediate representation are split into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version.
Figure3.4 shows a straight-line code and its conversion to SSA form.



$$
\begin{aligned}
&Before:\\
&x \leftarrow ...\\
&y \leftarrow x + x\\
&x \leftarrow x + y\\
&z \leftarrow x + y\\
&After:\\
&x_0 \leftarrow ...\\
&y_0 \leftarrow x_0 + x_0\\
&x_1 \leftarrow x_0 + y_0\\
&z_0 \leftarrow x_1 + y_0
\end{aligned}
$$

Figure 3.4: Straigh-line code conversion

When a variable may have two or more definitions in a program, for example, if construct in program, the ϕ-function is added. The value of a ϕ-function depends on the path which program executed. Converting a program to SSA form is useful for many dataflow problems and dataflow analysis. Figure3.5 shows the mapping from source program to SSA form.

Source program:
$a = 5$;
$b = 7$;
if(b <10)
  a=15;
else
  $b = b + 1$;

SSA form:
a_1=5;
b_1=7;
if(b_1 <10)
  $a_2 = 15$;
else
  $b_2 = b_1$+1;
$a_3 = \phi(a_1, a_2)$
$b_3 = \phi(b_1, b_2)$

Figure 3.5: SSA example

3.3 The C Intermediate Language

In this section, we summarize and introduce the C intermediate language, our CIL intermediate representation. CIL (C Intermediate Language) is a high-level representation that permit program analysis of C programs. The main author is George Necula from University of California, Berkeley. Necula *et al.* published CIL in 2002 [16].

Up to now, there are numbers of changes and updates for CIL. The latest version is 1.3.7 (April 24,2009). CIL will parse and typecheck a program, and compile it into a simplified, highly-structured subset of C.

In our translation,below is the main types that are used to represent C programs in CIL:
The CIL file is Top-level representation of a C source file.
file
global
typ

compinfo

varinfo

fundec

exp

instr

stmt

...

file includes many global definition such as *funcdec*(function definition). *funcdec* composed by *svar*(function name), *sformals*(parameters),*sbody*(function statements)... and so on. This highly structured representation help us apply many program analysis approaches and CIL also provide many useful modules for analysis. In the following, we will introduce some library of CIL modules we used or studied in this thesis.

3.3.1 Control Flow Graph

A control flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution. Control flow graph module provides practical functions for compute statement's predecessor and successor. CFG is fundamental information for data flow analysis

In CIL, function *computeFileCFG* could compute the CFG for an entire file. This will fill in the preds and succs fields of Cil.stmt. We could use this information in Cil.stmt for data flow analysis.

3.3.2 Data Flow Analysis Framework

The Dataflow module contains a parameterized framework for forward and backward data flow analysis. In our analysis we provide transfer function and analysis with the module. For example, for CIL instructions, we describe how to update SSA information as function for assignment or function invocation.

In our taint analysis, we use FowardsDataFlow to compute program variable information. For modules in CIL, we could use them by provide corresponding operations for each analysis material the module needed. BackwardsDataFlow could be used to trace back and collect information from sink back to source. CIL provide the analysis

framework and we could integrate it with our analysis approaches.

3.3.3 Points-to Analysis

The module ptranal.ml contain basic interprocedural points-to analyses for CIL. function *Ptranal.analyze_file* could builds the points-to graph for the CIL file and stores it internally.

For example, *Ptranal.may_alias*: If **true** the two expression may have the same value. We could invoke corresponding functions for CIL type in module to do program analysis.

3.4 Abstract Syntax Tree

In compiler [2], an abstract syntax tree (AST) [26], or just syntax tree, is a tree we use to represent the abstract (simplified) syntactic structure of source code written in a certain programming language. The syntax is abstract that it represent the structure of program rather than every detail syntax in source code. For example, grouping parentheses is implicit in the tree structure, and a syntactic construct such as an *IF*-cond-*Then* expression may be denoted by a single node with two branches.

This makes abstract syntax trees different from concrete syntax trees, traditionally called parse trees, which are often built by the parser part of the source code translation and compiling process. Once built, additional information is added to the AST by subsequent processing, e.g., semantic analysis.

The concrete syntax consists of a linear sequence of characters and/or tokens, along with a set of rules for parsing them. The abstract syntax generally doesn't have to worry about issues such as parser ambiguity, operator precedence, etc. Note that the abstract syntax tree just reveals the lexical and syntactical structure of the program text, what blocks and statements are lexically contained within in what. This may, or may not, be related to the semantic structure of the program. For example, in most object oriented languages with inheritance, the inheritance hierarchy is not revealed by examining the arrangement of the abstract syntax tree.

In this thesis, an important phase is the conversion from PHP AST to C AST. Main translation methods in our analyzer are implemented in this conversion. The following we display a complete mapping from PHP AST to C AST. Fiqure 3.6 for operator , Fiqure 3.7 for expression, and Fiqure 3.8 for statement.

PHP AST	C AST	Description
Plus	ADD	Addition
Minus	SUB	Subtraction
Times	MUL	Multiplication
Div	DIV	Division
Mod	MOD	module
Concat	Function call: strcat()	String concatenation
Eq	EQ	Equal
Neq	NE	Not equal
Le	LE	Less then or equal
Lt	LT	Less then
Ge	GE	Greater then or equal
Gt	GT	Greater then
Andand	AND	Logical operator: And
Oror	OR	Logical operator: Or
Xor	XOR	Logical operator: Xor
Not	NOT	Logical operator: Not
Pif	Convert to expression: QUESTION (e1,e2,e3)	Condition operator ?:
Bitnot	NOT	Bitwise not
Bitand	BAND	Bitwise and
Bitor	BOR	Bitwise or
Bitxor	XOR	Bitwise xor
Bitshiftl	SHL	Bitwise shift left
Bitshiftr	SHR	Bitwise shift right
Cast Phptype	CAST (type,init)	Variable type casting

Figure 3.6: The mapping from PHP AST operator to C AST operator

PHP AST	C AST	Description
Lvalue lv	Described in Chapter 4	Variables
Null	CONST_INT 0	NULL value
Const const_str	VARIABLE const_str	Constant variables
Int inum	CONST_INT inum	Integer
Float fnum	CONST_FLOAT fnum	Float
String str	CONST_STRING str	String
Bool bval	CONST_INT bval	Boolean
Prim (op, [e1;e2])	BINARY(op, e1, e2)	Operator with two arguments
Prim (op, [e1])	UNARY(op, e1)	Operator with one argument
App (fname, parameters)	CALL(fname, parameters)	Function call
NewArray array_init	Convert to a sequence of array variable assignments	Initial a new array
ClassMethod (c_name, m_name, parameters)	CALL(c_name_m_name, parameters)	Call class method
Method (exp, m_name, parameters)	CALL(c_name_m_name, parameters)	Call object method
NewObj (c_name, init_e)	Described in Chapter 4	Declare an new object
RefAssign (lv, exp)	Described in Chapter 4	Assign with pointer
LAssign (lv, exp)	Described in Chapter 4	Variable assignment
LOpAssign (lv, op, exp)	Described in Chapter 4	Variable assignment with operator
ListAssign (lv, exp)	Convert to a sequence of array variable assignments	List assignment
UClassMethod (c_name, exp, parameters)	N/A	Call class method with dynamic name
UMethod (exp1, exp2, parameter)	N/A	Call object method with dynamic name
UNewObj (c_name_exp, init_e)	N/A	Declare object with dynamic name
IncludeExp filename	File inclusion preprocessor will handle this	File inclusion
FileBlock f_block	File inclusion preprocessor will handle this	File inclusion block
Define (v_name, exp)	Declare variable and initialize	Define a constant variable

Figure 3.7: The mapping from PHP AST expression to C AST expression

PHP AST	C AST	Description
ExpSt exp	See figure 3.7	Expression
Echo exp	CALL(”printf”, exp)	Output funciton
Unset lv	CALL(”unset”,lv)	Unset a variable
BlockSt stmt_list	BLOCK(stmt_list, loc)	A block statement
While (exp, stmt)	WHILE(exp, stmt, loc)	While loop statement
DoWhile (stmt, exp)	DOWHILE(exp, stmt, loc)	DoWhile loop statement
For of (e1, e2, e3, stmt)	FOR(e1, e2, e3, stmt, loc)	For loop statement
If (exp, stmt1, stmt2)	IF(exp, stmt1, stmt2, loc)	If statement
Assert (exp, str)	N/A	Assert function
Function (func_name, parameter, body, ref)	Described in Chapter 4	Function declaration
Return exp	RETURN (exp, loc)	Return statement
Break loc	BREAK loc	Break statement
Continue loc	CONTINUE loc	Continue statement
Global lv_list	Declare variable	Global variable declaration
Static lv_list	Declare static variable	Static variable declaration
Switch (exp, stmt)	SWITCH (exp, stmt, loc)	Switch statement
Class declare_class	Described in Chapter 4	Class declaration

Figure 3.8: The mapping from PHP AST statement to C AST statement

Chapter 4

Translation and Static Analysis

Our static analyzer for verifying PHP Web application operates in three main phases. As the following diagram, the first phase is parsing PHP program into C AST. The second phase translates C AST to the CIL intermediate representation. The third phase is applying static analyses on the CIL intermediate representation. This analysis architecture can support different languages by providing language parser to C AST individually. Figure 4.1 demonstrate our analysis architecture of analyzer.

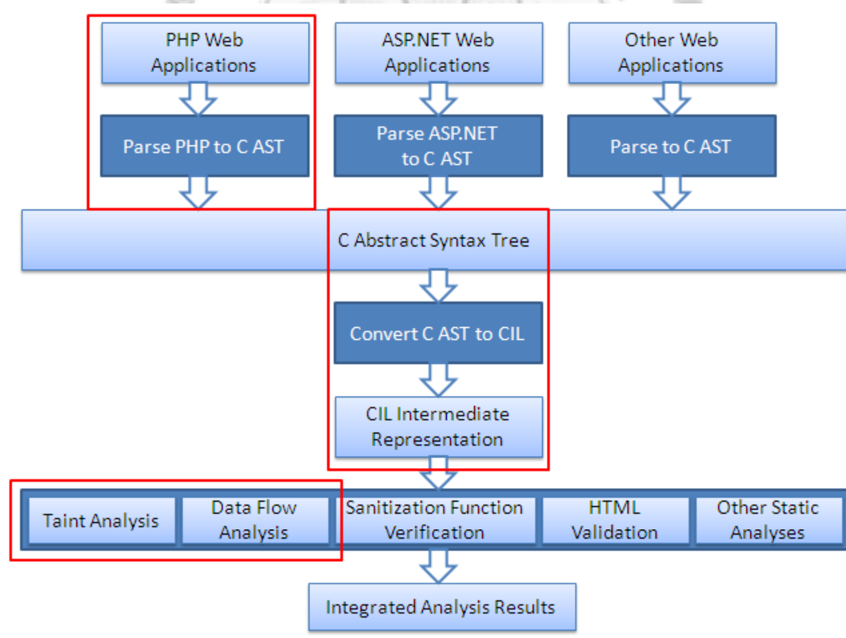


Figure 4.1: Analysis architecture

The foundation of analysis architecture is translation from PHP to CIL representation. We apply static analysis algorithms to support a more precise and accurate analysis result. We go deep into further program analysis and integrate with automatic penetration test-case generation. In our translation, we try to preserve PHP5 features as precisely as possible in the translation. With precise translation and auxiliary functions, our CIL intermediate representation could also be compiled and executed. To ensure our translation is precise, we compile and execute our translation result and observe the runtime behavior as well as original source PHP program. The CIL representation also help us to preform program analysis with a well-structured CIL file and CIL modules.

This section is structured as follows:

1. Translation for PHP5 language features to CIL representation
2. Inteprocedural taint analysis
3. Alias taint analysis
4. Basic dynamic vulnerability confirmation

4.1 Translation of PHP5 Language Features to CIL

Currently, our analyzer supports basic features of PHP4. The main object oriented features such as inheritance and new features in PHP5 is not supported by now. The latest version of PHP is 5.3.2 (March 2010) and it has many new language features. We extend our PHP analyzer to support major features of PHP5. In the translation we also preserve program semantics in conversion from PHP to CIL as precisely as possible. The following sections describe how we handle these new features.

4.1.1 Adjustments for Basic Structures and Auxiliary Functions

To present more language features such as alias behavior, we modeled PHP variables as *pointers*. We also give memory allocations for each variable used in PHP program. Function return type and parameter type become pointers to the struct "variable" For function declaration and operations, we also adjust the return type and parameter type as struct variable pointer. Finally we correct some auxiliary function and make corresponding adjustments. The following figure 4.2 shows our new variable assignment representation.

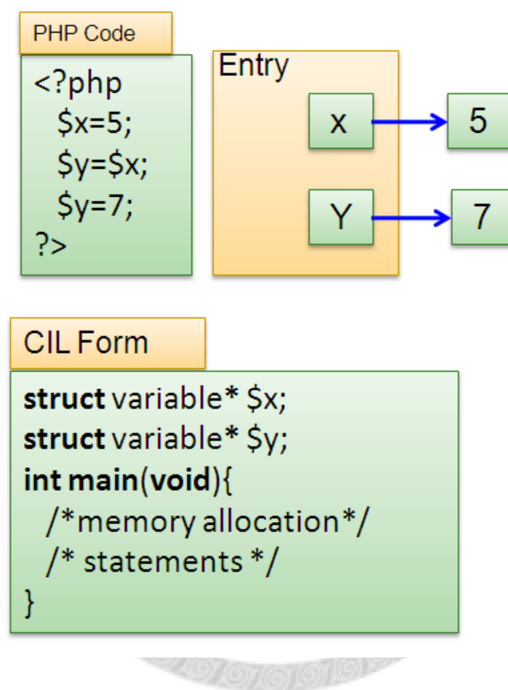


Figure 4.2: Variable assignment representation

4.1.2 Translation of Class and Object

There are no class constructors in CIL. Thus, we use C struct constructor and function constructor to represent it. The main approach is converting class members to struct constructor and declaring each member in struct. We extract methods in class and rename these functions in the format *class name_method name*. Because struct constructor only admits member declarations in CIL, we make memory allocations and each member initialization in a auxiliary function constructor named *class name_constructor*.

When meeting an instance of a class, we invoke the object constructor function to initialize the object and memory allocation. Figure4.4 illustrates the conversion.
We also compile and execute the intermediate representation of class and object to confirm our translation. The following figure 4.3 shows a translation example as graphic presentation.

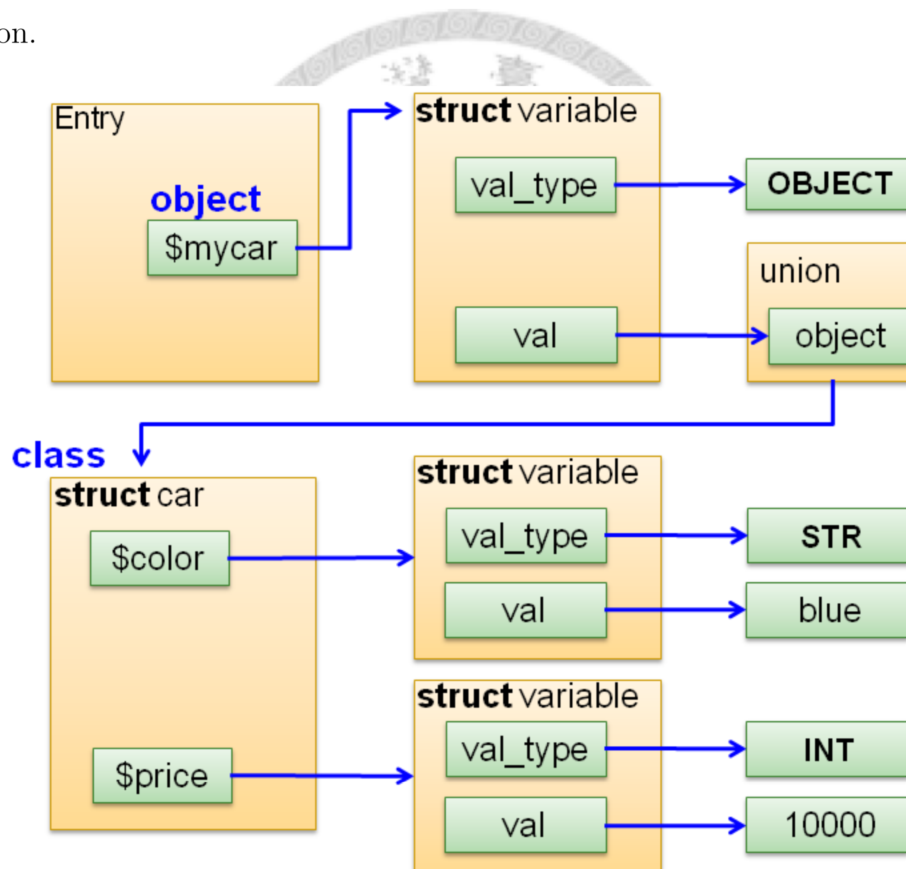


Figure 4.3: Translation of class and object

```
PHP code:
class car{
  var $color;
  var $price=10000;
  function set_color($c){
    $this->color=$c;
  }
}
$mycar=new car;
$mycar->set_color("blue");
echo $mycar->color;
echo $mycar->price;

CIL form:
struct car{
  struct variable *color;
  struct variable *price;
}
void car__constructor(struct variable $this){
  ((struct car*)$this->val.object)->price=
  (struct variable*)malloc(sizeof(struct variable));
  var_assign(((struct car*)$this->val.object)->price,struct_int(10000));
  ((struct car*)$this->val.object)->color=
  (struct variable*)malloc(sizeof(struct variable));
  return;
}
void car_set_color(struct variable *$this, struct variable *$c){
  (struct car*)($this->val.object)->color=$c;
  return;
}
int main(){
  #initial statements and memory allocation
  struct variable* $mycar;
  $mycar->val.object=malloc(sizeof(struct car*));
  car__constructor ($mycar);
  var_assign(((struct car*)$mycar->val.object)->price, struct_int(20000));
  car_set_color($mycar,struct_str("blue"));
  printf("%s", var_get_str(((struct car*)$mycar->val.object)->color);
  printf("%s", var_get_str(((struct car*)$mycar->val.object)->price);
}
```


Figure 4.4: PHP class example

4.1.3 Translation of Inheritance

Inheritance is a well-established programming principle. PHP also makes use of this principle in its object model. When we extend a class, the subclass inherits all of the public and protected methods from the parent class. Unless a class overrides those methods, they will retain their original functionality.

We use struct, function, and specify unique nested struct name and functions to represent the inheritance features [10]. The main approach is keeping parent class statements in translation, When meet inheritance statement, we duplicate and rename parent's members then become subset of child class members. For superclass's methods, we invoke the corresponding superclass's method in the subclass's method. The rename format is *child name parent name_method name*. We also execute the intermediate representation to confirm our inheritance translation.

The following figure 4.5 shows a translation example as graphic presentation.

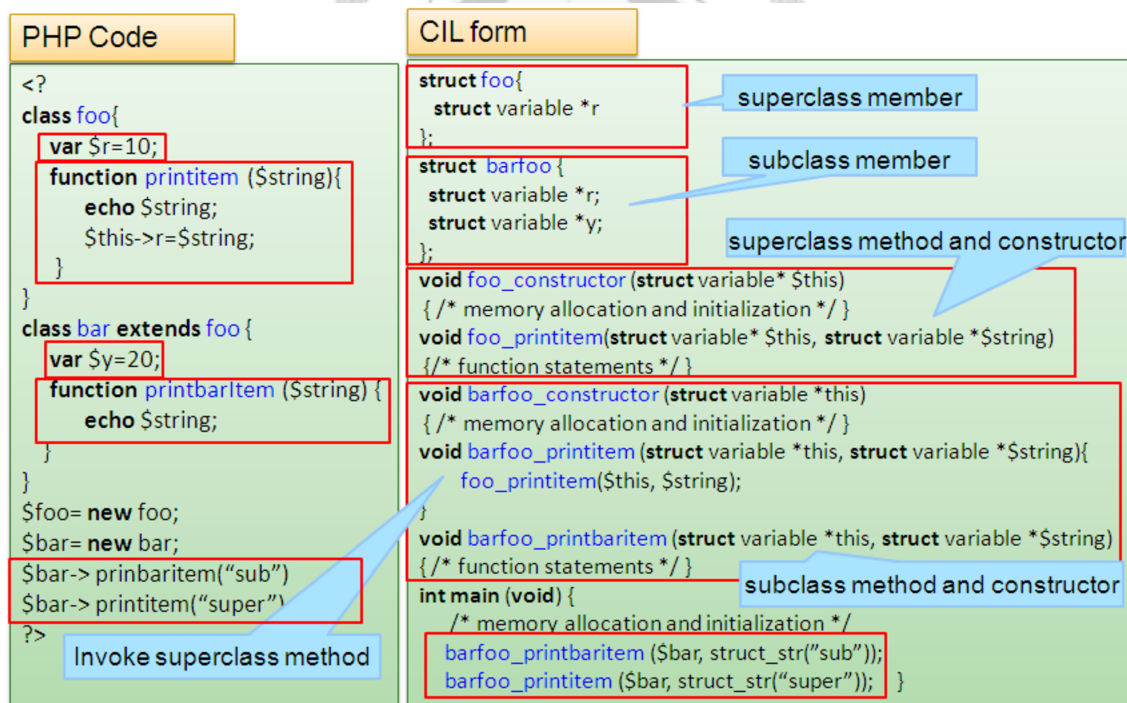


Figure 4.5: Translation of inheritance

Figure4.6 shows a conversion example.

```
PHP code:
<?
class foo{
     var $r=10;
     function printitem ($string){
           echo $string;
      }
}
class bar extends foo {
     var $y=20;
     function printbarItem ($string) {
           echo $string;
    }
}
$foo= new foo;
$bar= new bar;
?>
CIL form:
struct foo{
   struct variable *r
};
struct  barfoo {
  struct variable *r;
  struct variable *y;
};
void foo_constructor (struct variable* $this)
 { /* memory allocation and initialization */ }
void foo_printitem(struct variable* $this, struct variable *$string)
 {/* function statements */ }
void barfoo_constructor (struct variable *this)
 { /* memory allocation and initialization */ }
void barfoo_printitem(struct variable *this, struct variable *$string){
        foo_printitem($this, $string);
}
void barfoo_printbaritem (struct variable *this, struct variable *$string)
{ /* function statements */ }
int main (void) {
  /* memory allocation and initialization */
  /* statements */
}
```

Figure 4.6: PHP inheritance example

4.1.4 Translation of Magic Functions

The function names __construct, __destruct, __call, __callStatic, __get, __set, __isset, __unset, __sleep, __wakeup . . . etc which is magical in PHP classes. Programmer cannot have functions with these names in any of classes unless we want the magic functionality associated with them. PHP reserves all function names starting with __ (double underscore) as magical. These function is executed prior to any serialization and load automatically. We translate the most typical two magic functions __construct and __destruct. We can use constructor and destructor to initialize member value and write code that will be executed when the object is destroyed.

In our translation, our approach is collecting magic functions and makes corresponding operations. For __construct, we translate function statements and rename function name as *class name_construct*. When object initialization, we invoke corresponding magic construct functions. For __destruct, we translate it and rename function name as *class name_destruct* and invoke this function at the end of the program automatically.

The following figure 4.7 shows a translation example as graphic presentation.

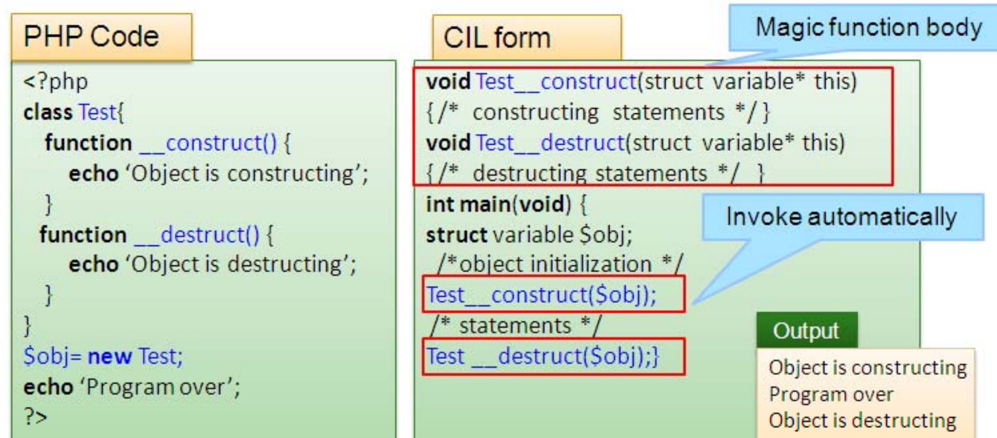


Figure 4.7: Translation of magic functions

Figure4.8 shows a conversion example.

```
PHP code:
class grade{
  var $score=1000;   function __construct(){
    echo "object initialization programs";
  }
  function __destruct(){
    echo "object destruction programs";
  }
  $obj=new Test;
  echo "program over";
}
CIL form:
struct grade{
  struct variable *score;
}
void grade__constructor(struct variable *$this){
  #constuction statements }
void grade__construct(struct variable *$this){
  #destruction statements }
int main(){
  $obj->val.object=malloc(sizeof(struct grade));
  grade__construct($obj);
  grade__constructor($obj);
  printf("program over");
  grade__destruct($obj);
  return 0;
}
```

Figure 4.8: PHP magicgfunction example

4.1.5 Translation of Try-Catch Exception

PHP5 has an exception model similar to that of other programming languages. By using an exception, you can get more control the simple trigger_error notices we were stuck with before. Code may be surrounded in a try block, to facilitate the catching of potential exceptions. Each try must have at least one corresponding catch block.

In C language, there is not directly exception handling feature to represent try-catch exception. At first, we translate the throw exception, try block, catch operation statements to CIL. We label each block by throw, try, and catch keywords, and then we coordinate with a *cexception* interface with *setjmp.h* to represent the try-catch behavior [8].

The following figure 4.9 shows a translation example as graphic presentation.

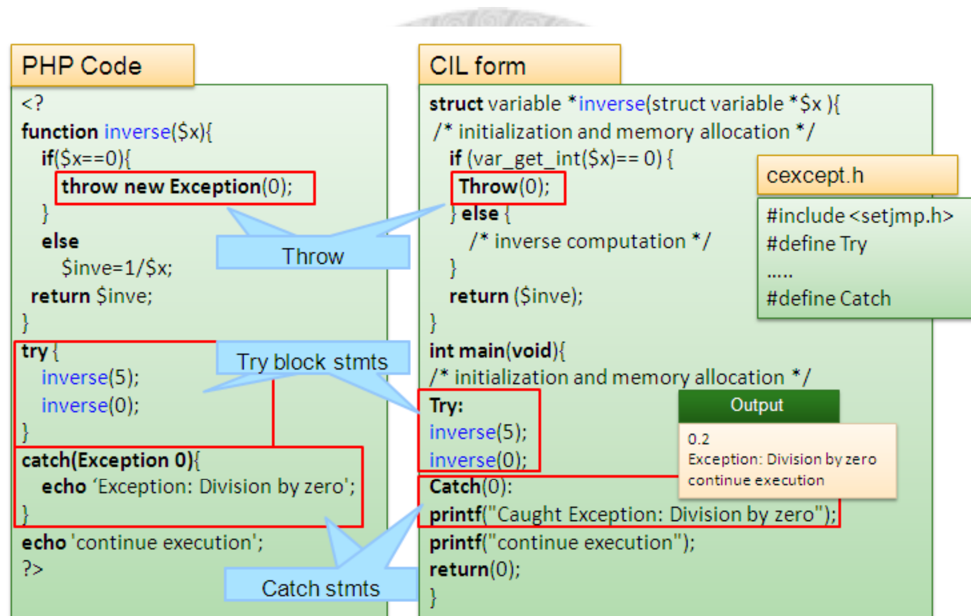


Figure 4.9: Translation of Try-Catch Exception

Figure4.10 shows a conversion example.

```
PHP code:
<?
  function inverse($x){
  if($x==0){
  throw new Exception(0);
  }
  else
  $inve=1/$x;
  return $inve;
 }
  try {
  inverse(5);
  inverse(0);
}
  catch(Exception 0){
  echo 'Caught Exception: Division by zero';
  }
  echo 'continue execution';
?>
CIL form:
struct variable *inverse(struct variable *$x ){
  /* initialization and memory allocation */
  if (var_get_int($x) == 0) {
    Throw(0);
  } else {
         /* inverse computation */
  }
  return ($inve);
}
int main(){
/* initialization and memory allocation */
Try:
inverse(5);
inverse(0);
Catch(0):
printf("Caught Exception: Division by zero");
printf("continue execution");
return(0);
}
```

Figure 4.10: PHP try-catch example

4.1.6 Translation of Reference Assignment and Object Assignment

PHP references allow you to make two variables to refer to the same content. A reference between two variables establishes a run-time alias by using the reference operator "&". There are two basic conditions in conversion of PHP reference. The first condition is the "&" used in variable assignment. The second condition is the "&" invoked with function call parameters. We use pointer assignment to represent the alias behavior of first condition. For second condition, we differentiate call by value or call by reference in parsing stage.

Figure 4.11 shows a translation example as graphic presentation.

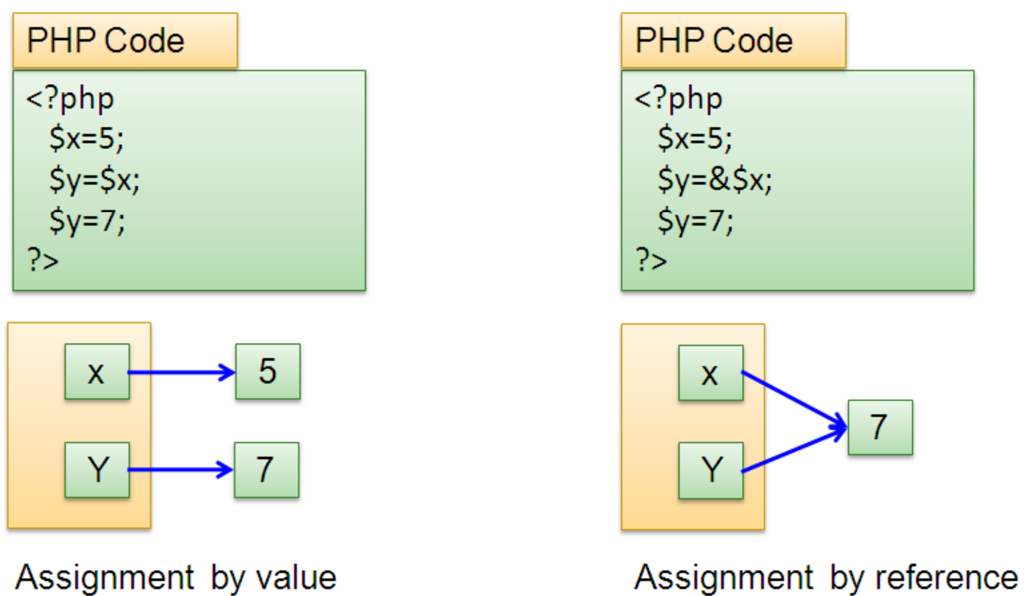


Figure 4.11: Translation of alias features

Figure4.12 illustrates the conversion of first condition.

```
PHP code:
<?php
   $a=10;
   $b=&$a;  // alias {$a,$b}
   $a= 20;
   echo $b;   //  20
?>
CIL form:
int main(void){
  /* #initialization and memory allocation */
  var_assign($a, struct_int(10);
  $b=$a;
  var_assign($a, struct_int(20);
  printf("%s", var_get_str($b);
  return(0);
}
```

Figure 4.12: PHP alias example 1

In PHP4, all assignment will pass the value, including objects. In PHP5, all objects are now passed by reference. In our translation, we model each variable as struct variable pointer. When we met object assignment, we use pointer assignment to represent the semantics. We also compile and execute the representation to confirm our translation. Figure4.13 illustrates the conversion.

```
PHP code:
<?
class person {
   var $sex;
   var $height;
}
$joe=new person();
$joe->sex='male';
$betty =$joe;
$betty->sex='female';
echo$joe->sex;//Will be'female'
?>
CIL form:
struct person {
   struct variable *height ;
   struct variable *sex ;
};
void person__constructor(struct variable *$this ) {
  /* #object initialization */
}
int main(){
/* #initialization and memory allocation */
person__constructor($joe);
tmp___17 = struct_str((char *)"male");
var_assign(((struct person *)$joe->val.object)->sex, tmp___17);
$betty = $joe;
tmp___20 = struct_str((char *)"female");
var_assign(((struct person *)$betty->val.object)->sex, tmp___20);
tmp___21 = var_get_str(((struct person *)$joe->val.object)->sex);
printf("%s", tmp___21);
return(0);
}
```

Figure 4.13: PHP object assignment example

4.1.7 Translation of Static Member and Class Constant

For situations where we would like to associate some data with an entire class of objects, but the data is not constant, we can create member variables that are global to the entire class. These are called static member variables. In CIL representation, we represent this shared data in class method representation and memory allocation in object construction. In object operations, objects could share the global data in class. The following example is using static member to count number of objects by magic function automatically. In

PHP5, we can define constant values by keyword *const*. In CIL representation, we use *const* in C to represent the constant value. Figure4.14 illustrates the conversion.

```
PHP code:
class circle {
  static $count = 0;
  var $r=10;
  function __construct(){
   $count=$count+1;
  }
}
$object1=new circle();
$object2=new circle();
?>
CIL form:
void circle__constructor(struct variable *$this ) {
  /* #object initialization */
}
void circle___construct(struct variable *$this ) {
  struct variable *$count ;
  var_assign($count, struct_int($count->val.inum+ 1));
  return;
}
int main(void){
  /* initialization and memory allocation */
  circle__constructor($object1);
  circle___construct($object1);
  circle__constructor($object2);
  circle___construct($object2);
  return(0);
}
```

Figure 4.14: Translation of static member

4.1.8 Translation of Namespace

PHP namespaces provide a way in which to group related classes, functions and constants. We add the namespace as prefix of variable name, struct name, and function name. Figure4.15 illustrates the conversion.

```
PHP code:
<?
namespace blog:
  const HELLO = 'Hello blog';
  $a=15;
namespace blogrock:
  const HELLO = 'Hello rock';
  $a='constant';
  }
}
?>
CIL form:
char *blog_HELLO= "Hello blog";
char *blogrock_HELLO= "Hello rock";
struct variable *blog_$a;
struct variable *blogrock_$a;
}
int main(void){
  /* initialization and memory allocation */
  var_assign(blog_$a, struct_int(15));
  var_assign(blogrock_$a, struct_str("constant"));
  return(0);
}
```

Figure 4.15: PHP namespaces example

4.2 Interprocedural Analysis and Alias Analysis

After we finished the translation from PHP to CIL, we can apply static analysis algorithm on our CIL intermediate representation. We address the most prominent types of web application vulnerabilities such as XSS and SQL Injection. At first we compute control flow graph of CIL file and obtain the information of each statement's predecessors and successors. Then we can apply dataflow analysis to compute variables taint information in program. We extend the analysis approach by Chung [7] as intraprocedural analysis algorithm. We design an interprocedural analysis algorithm to allow taint analysis to cross function and object boundaries and provide more complete analysis results. In our dataflow analysis, we also build up alias group graph as basic alias information for alias analysis. Then we merge the interprocdural taint analysis algorithm and alias analysis to support more precise and complete taint analysis vulnerability results. Finally, we build up runtime environment by auxiliary functions and execute our CIL representation. This helps us to do vulnerability confirmation and could be combined with automatic test case generation. The rest of this section is structured as follows:

- Interprocedural taint analysis.
- Alias taint analysis.
- Dynamic vulnerability confirmation.

4.2.1 Interprocedural Taint Analysis

The following figure 4.19 shows the main process of interprocedural analysis. In our translation, we integrate the file inclusion program and then generate a main function in CIL file. The *main* function body includes all semantics of original PHP program instructions and statements. As the left-hand picture, we start the *forward dataflow analysis* from the *main* function. The main intention in our dataflow analysis is computing program **variable dependence information** and **taint information**. We record this information as total SSA taint information. We process each statement in program in order, updating the SSA taint information. When we encounter function call in dataflow analysis, as the first edge in figure4.19, we apply interprocedural algorithm to update the SSA informa-

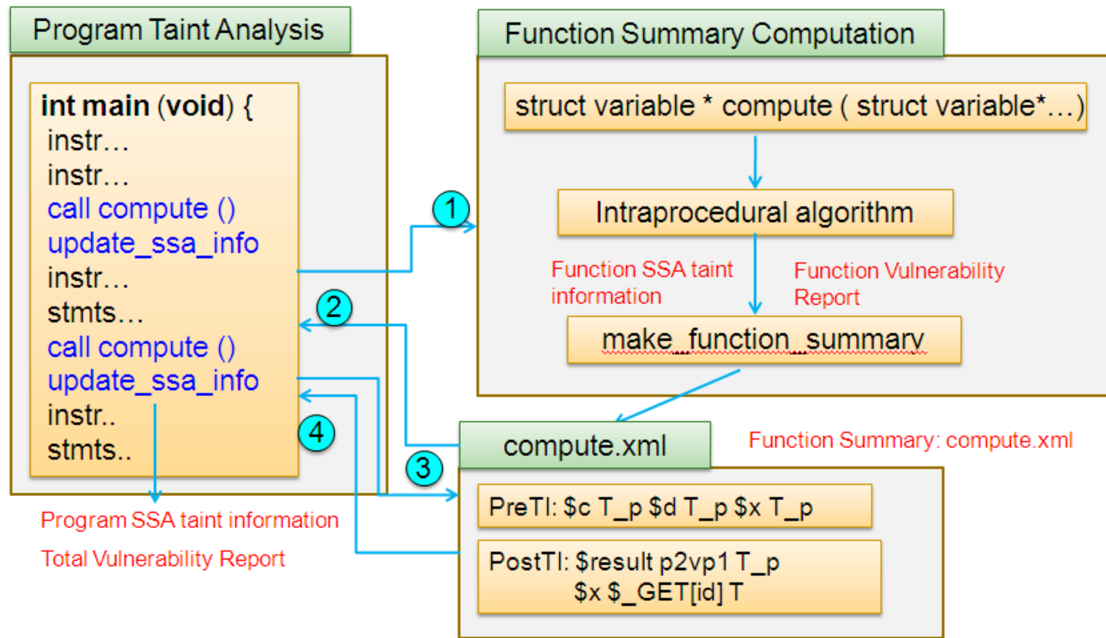


Figure 4.16: Interprocedural analysis

tion by function summary. If the function is a user-define function and invoked the first time, we invoke the intraprocedural analysis and function summary computation algorithm to compute function summary and taint analysis for the function. As the second edge in figure4.19, we obtain the function summary with precondition, postcondition, and function SSA taint information. Function summary will be used in interprocedural analysis. We generate function vulnerability report by function SSA taint information. This information in function is also materials for generating vulnerability trace flow. After the *main* function obtains the function summary, then we can cross the function and update the SSA taint information by function summary and interprocedural algorithm. As the third and fourth edge, when the same function invoked again, the analysis take function summary and update SSA taint information to cross the function. The summarized process of interprocedural taint analysis composed by the following three algorithms, intraprocedural analysis algorithm (extended from Chung [7]) , make_function_summary algorithm, and interprocedural algorithm.

```
Algorithm Intraprocedural (fdec)
Traverse the CIL AST nodes
foreach node n in CIL AST
if n = instruction node
  if instruction node= function call then
    interprocedural (ssa_info, fun_summry)
else   Vname := lvalue's name in this instruction.
  Rv := right hand side value in the instruction.
  T_flag := Check rv is tainted or untatinted.
  Obtain new variable information vi (vname, rv, t_flag)
  Update local variable information map with vi
  Update whole variable information map with vi and line number
else if n = joint node of "if statement"
  Merge variable information maps from previous paths
else if n = joint node of "loop statement"
  if old variable information = new variable information then
    Algorithm has reached a fixed point, go to next node
  else
    Merge old variable information with new variable information map
    Revisit this loop block
make_function_summary (fdec, ssa_info)
```

Figure 4.17: The intraprocedural analysis algorithm

Figure 4.17 lists the steps for intraprocedural analysis. At first, we record *variable information* (includes variable name, possible value set, and taint flag) for each statement. T represents tainted and U represents untainted. When we meet a branch statement like '**if**', we compute variable information for every branch respectively. Then we merge (union) each result at the branch join point. This approximation will result in some false positives because there may not every branches will be reached in program execution. In test-case generation stage, we will analyze the '**if**' expression in branch statement and solve the integer and string value. This could help us reduce the false positives. More details could be found in Yu [35]. When we meet a **loop** statement, we compute the variable information repeatedly until we reach the fixed point. We record whole variable information with line number. This mechanism is similar to *SSA* [3] and it helps us with tracing back tainted variables. When we have each variable's taint information with line

number, we trace back tainted variables in sink functions. According to possible value sets in out SSA info, we also provide the complete trace-back path in vulnerability report.

```
Algorithm make_function_summary (fdec, f_ssainfo)
Precondition computation:
foreach sformals in fundec
  PreTI:= add (sformal.varname, T_p)
foreach varname in f_ssainfo
  if (check_global varname)
  then PreTI := add (varname, T_p)
  else go to next varname
Postcondition computation:
  foreach global_var in f_ssainfo
  PostTI := add (varname, possible_varset, t_flag)
if return_var in fdec.return_stmt
  PostTI := add (varname, possible_varset, t_flag)
foreach varname in f_ssainfo
  if (check_object varname) //object constructor
    PostTI := add ($this, possible_valset, t_flag)
return PreTI,PostTI
```

Figure 4.18: The function summary computation algorithm

Figure 4.18 shows the steps for function summary computation. We focus on the taint information passed into function and returned after the function call. We compute precondition and postcondition to describe function_summary.
For precondition, we focus on

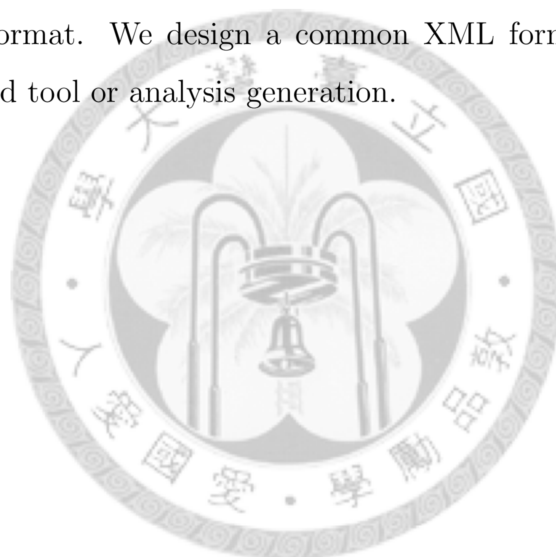
- Parameter variable taint information.
- Global variable taint information.

We compute and prepare that information should be passed through into the function. Function parameters are added to precondition and set *t_flag* as *T_p* for taint information will be provided when invocation. For each global variable used in function, we also put this variable into precondition.

For postcondition, we focus on

- Return variable taint information.
- Global variable taint information.

We compute postcondition that includes global variables, return variables, and object constructor $this. The postcondition consists of return variable name, possible value set, and t_flag. If the variables' t_flags in postcondition could be decided (such as possible value with $_GET array or constant value), then return the corresponding t_flag. For t_flag that could not be decided in function summary we return T_p for interprocedural analysis to infer the t_flag cross the function invocation. In the end, we output function summary with XML format. We design a common XML format for writing function summary with front-end tool or analysis generation.



```
Algorithm Interprocedural (ssa_info, fun_sumry)
Input:  (F.preTI, F.postTI), (actual_preTI) // actual_preTI from ssa_info
Output:  (actual_postTI)
if check_fsm_exist (fun_name) then
  begin
  foreach varname in F.preTI
    Obtain variable information vi (vname, possible_varset, t_flag)
    Update local variable information map with vi
    Update whole variable information map with vi and line number
  foreach varname in F.PostTI
    if (t_flag==Taint)
    then ssa_info:=add (varname, possible_varset, T)
    else if (t_flag==Untaint)
    then ssa_info:= add (varname, possible_varset, U)
    else if(t_flag==T_p)
    then t_flag:=taint_infer(possible_varset)
      ssa_info:= add (varname, possible_varset, t_flag)
  if check_exist (lvar)
  then update (lvar, PostTI)
else
  intraprocedural(fundec)
  interprocedural(ssa_info, fun_sumry)
return
```

Figure 4.19: The interprocedural analysis algorithm

For each function call encountered in analysis, Figure 4.19 shows the steps for interprocedural approach to cross function and update SSA taint information correspondingly. At first, we look for function summary by function name, if we obtain the function summary, then we update SSA taint information by function summary. If not found we invoke intraprocedural analysis to compute function summary. For each variable in function precondition, we prepare the variable taint information from current SSA information. This information will be used in taint inference. For each variable in postcondition, we update these variables' taint information to SSA information. When the t_flag is t_p, we infer the t_flag by taint information from possible value set and precondition. Finally, if the there is lvar to take function return value, we update the lvar taint information. After

the steps, program SSA information updates taint information by function summary correspondingly and cross the function.
In the end, we obtain the program main function SSA taint information and then we could provide complete program vulnerability report.

The following is a example to demonstrate how it works. Figure4.2.1 is php program include function operation and number of XSS vulnerabilities. This program has basic

```
   <?
01 $a=$_GET["msg"];
02 $b="constant";
03 $x=16;
04 $sum= compute($a, $b);
05 function compute ($c, $d){
06 $result= $c.$d;
07 echo $c;
08 global $x;
09 $x=$_GET["id"];
10 return $result;
11 }
12 echo $a;
13 echo $sum
14 echo $x;
   ?>
```

Figure 4.20: Interprocedural analysis example

assignments and a function call to make a computation. In line 7, there are a XSS vulnerability in function and three XSS vulnerability with function return value and global variables in line 12, 13, and 14.

Figure4.2.1 shows the function summary of function *compute*. In function *"compute"*,

```
<PreTI>
     <var> $c </var>
        <t_flag> T_p </t_flag>
     <var> $d </var>
        <t_flag> T_p </t_flag>
     <var> $x </var>
       <t_flag> T_p </t_flag>
</PreTI>
<PostTI>
     <return_var> return$result </return_var>
        <possible_varset> p1 </possible_varset>
        <possible_varset> p2 </possible_varset>
        <t_flag> T_p </t_flag>
     <return_var> $x </return_var>
        <possible_varset> $_GET['id'] </possible_varset>
        <t_flag> Taint </t_flag>
</PostTI>
```

Figure 4.21: Function summary example

the precondition includes parameter variables \$c,\$d, global variable \$x, T_flag of these variables will be decided until function invocation. The postcondition includes return variable \$result and global variable \$x taint information. We keep the possible_varset of \$result for t_flag inference. The t_flag of \$x could be decided in function because value comes from \$_GET source.

Figure4.22 shows the program SSA taint information. In figure4.22, function invo-

Variable and line number	Possible value set	Taint information
($a, 2)	($_GET["msg"], 2)	T
($b, 3)	"constant"	U
($x, 4)	16	U
($a, 5)	($a, 2)	T
($b, 5)	($b, 3)	U
($x, 5)	($x, 5)	T
($sum, 5)	(return$result, 5)	T
($a, 13)	($a, 5)	T
($sum, 14)	($sum, 5)	T
($x, 15)	($x, 5)	T

Figure 4.22: Whole variable taint information of Figure 4.2.1

cation in line 5, we prepare and update taint information by interprocedural analysis algorithm. ($a, 5) and ($b, 5) are parameter taint information. After function call, we update global variable $x, function returns *lvar* $sum by function summary and taint inference. Finally, we could identify program vulnerabilities by program SSA taint information.

Our vulnerability report follows the NIST static analysis tool output XML format and could be merged with other static analysis tools easily in the future [19]. Figure4.2.1 shows the report format example of cross-site scripting (XSS).

```
<report tool_name='CANTU' tool_version='1.0'>
    <weakness id='1' tool_specific_id='79'>
       <category cweid=''>Cross-Site Scripting </category>
           <sink path='xss2.php' line='2'/>
           <source path='xss2.php' line='2'/>
           <grade severity='Critical ‘ probability=''/>
                 <output>
                   <textoutput/>
                 </output>
    </weakness>
</report>
```

Figure 4.23: An example of analysis report

4.2.2 Alias Taint Analysis

In PHP5, alias relationship between variables can be introduced by

- Reference Assignment (e.g.$a=&$b) .
- Object Assignment (e.g.$obj1=$obj2).

In our translation, we represent it by pointer assignment. Figure4.2.2 shows an example for our alias representation.

```
PHP Code:
<?php
   $b=&$a;  // alias { $a , $b }
   $a=20;
   echo $b;  // $b is 20.
   $b=40;
   echo $a; // $a is 40.
?>
CIL form:
/* initialization and memory allocation */
   $b=$a;
   var_assign($a,  struct_int(20));
   printf ("%s" , var_get_str($b)); // $b=20
   var_assign($b, struct_int(40) );
   printf ("%s" , var_get_str($a)); // $a=40
```

Figure 4.24: A alias translation example

In our forward dataflow analysis, we build up alias group when we meet pointer assignment instructions. The following figure 4.25 shows the main process of alias analysis. For example, A alias group is composed of $a, $b, and $c and object alias group $object1 and $object2

When our taint analysis arrived *sink* point, if sink function argument is *Untainted* then we check variables in argument's alias group latest taint information. If there is member of alias group tainted, we could identify alias vulnerability. Figure4.2.2 shows a XSS example with object alias.

In this program, $mycar is an instance of class car in line 08. In line 10, $mycar and $a_mycar becomes alias by object assignment. $a_mycar's member $color take user input color in line 12 and output sink is in line 13. In sink argument, $mycar− >color in taint

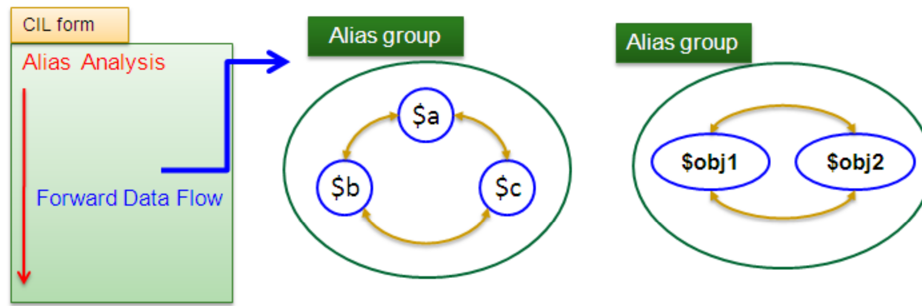


Figure 4.25: Alias group example

```
01 <?php
02 class car{
03 var $color;
04  function setcolor($c){
05  $this->color = $c;
06     }
07    }
08 $mycar = new car;
09 $mycar->setcolor("blue");
10 $a_mycar = $mycar;
11 $usercolor= $_GET['color'];
12 $a_mycar->setcolor($usercolor);
13 echo $mycar->color;
14 ?>
```

Figure 4.26: A alias cross object XSS example

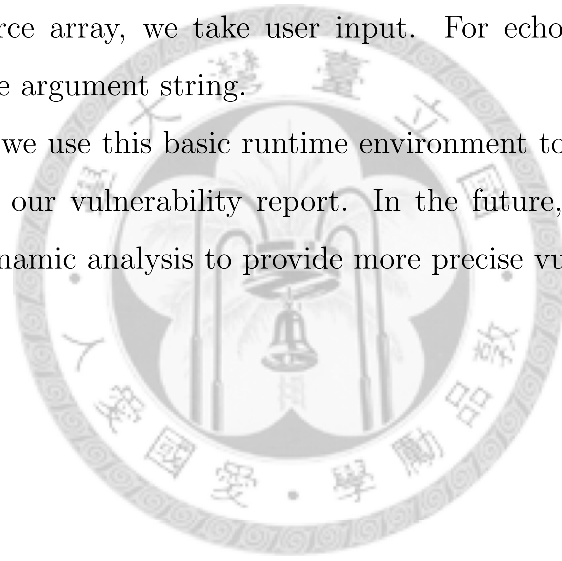
information is *Untainted* with constant string ("blue"). $color's actual value is provided by user. We check alias group taint information and obtain that $a_mycar is **Tainted** and identify this vulnerability. We also compiled and executed our representation to confirm this vulnerability.

4.2.3 Basic Dynamic Vulnerability Confirmation

We apply static analysis approaches on our CIL representation and report vulnerabilities. Static analysis is an effective approach to detect vulnerabilities. Dynamic analysis could help us to confirm vulnerability.

Our CIL intermediate representation is a subset of C and could be compile and executable. We compiled and executed our translation result by GCC just replacing \$ as _ slightly (because C language doesn't admit variable name start from \$). We build up runtime environment by auxiliary functions and execute our CIL representation. Runtime environment could help us check translation and could be used to take user input and output strings. We compare representation's runtime behavior and original program's behavior to confirm our translation. In our vulnerability confirmation, for \$_GET, \$_POST and other tainted source array, we take user input. For echo and other output sink function, we output the argument string.

In our experiment, we use this basic runtime environment to confirm basic XSS vulnerability and confirm our vulnerability report. In the future, we could also combine static analysis with dynamic analysis to provide more precise vulnerability reports.



Chapter 5

Implementation and Experiments

We implement the approach described in chapter 4 and preform a series of experiments. Our improved analyzer for verifying PHP Web application security is implemented in CANTU (an acronym derived from “Code Analyzer from NTU”). Figure 5.1 shows the integrated analysis architecture of CANTU. In this thesis we address the modules marked 1 and 2.

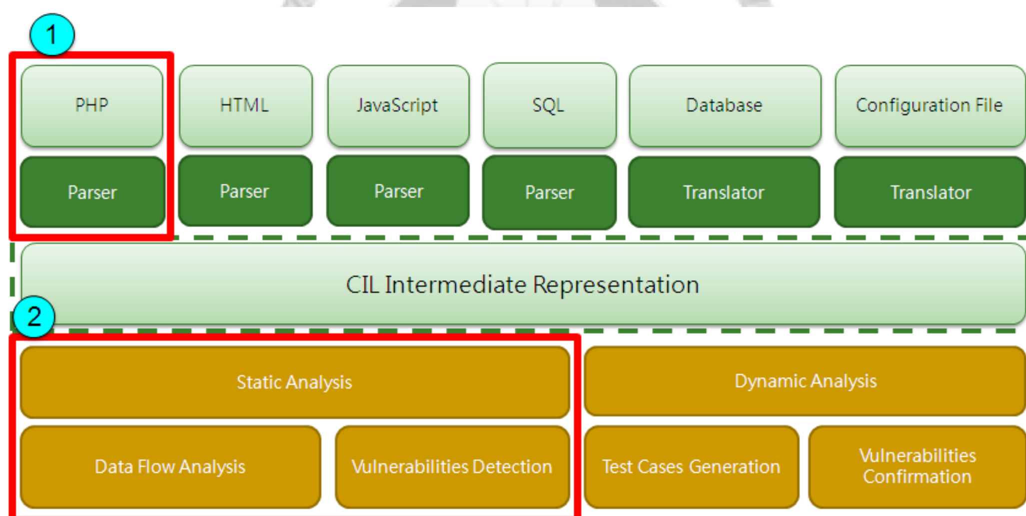


Figure 5.1: The integrated analysis architecture of CANTU

In the CANTU architecture, we focus on PHP translation and static analysis. We improved the prototype analyzer developed by Chung [7]. We implemented the translation approach and analysis algorithms described in Chapter 4.

5.1 Implementation

The front-end Web user interface is written by PHP with LAMP architecture. The back-end static analysis engine is implemented with Objective Caml programming language [17] and CIL library. User could upload source code zip file by front-end Web page. Our static analyzer parses the source code, translates, and applies taint analysis. We generate CIL intermediate representation and vulnerability XML report. User could browse vulnerability report and trace flow in front-end.

Figure 5.2 shows the complete process in this thesis. After our translation, we output CIL intermediate representation and then start our static analysis. Our static analysis includes interprocedural taint analysis and alias analysis to generate vulnerability XML report. In the end, we compile and execute our CIL intermediate representation to confirm our vulnerability report.

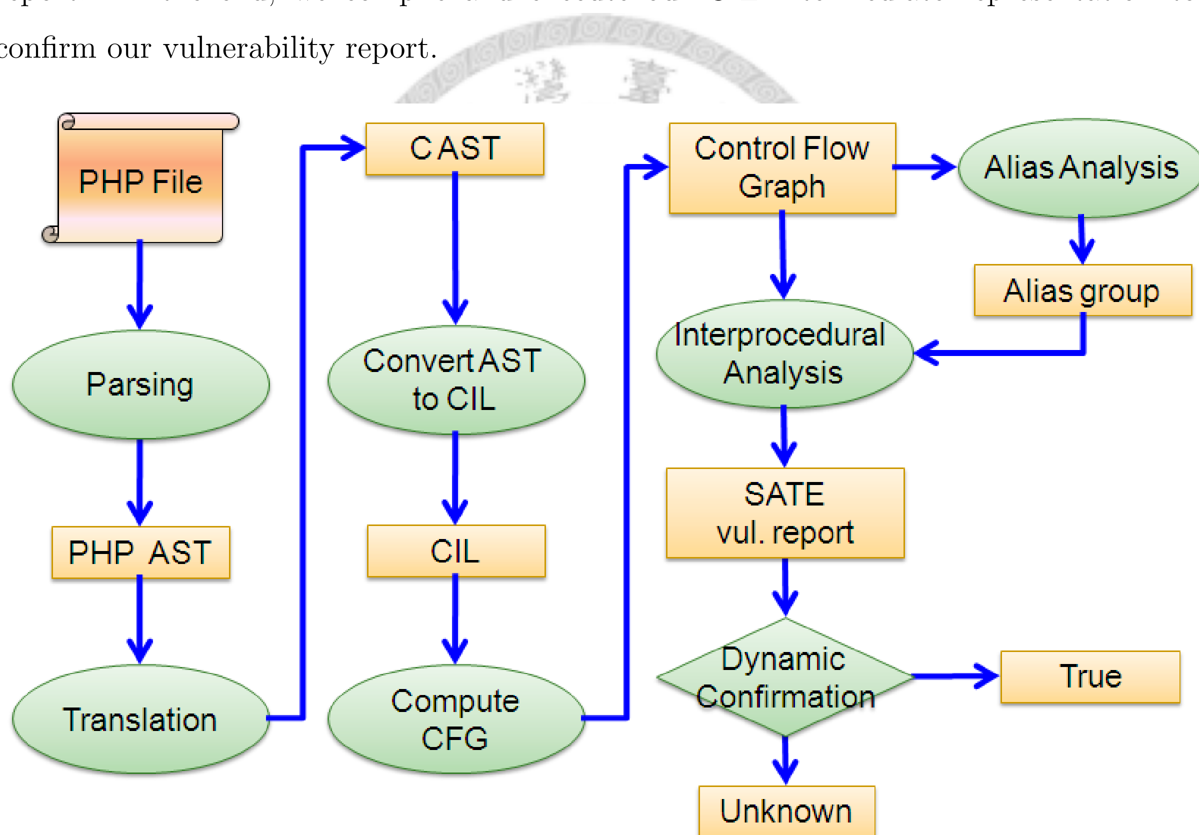


Figure 5.2: The work flow diagram in this thesis

5.2 Experimental Results

We designed six test cases to evaluate the ability of our tool for detecting XSS vulnerability that cross function and alias conditions. These testing programs are available in Appendix.

We compared our analysis result with two commercial tools toolF, toolC, and an open source tool Pixy. Pixy is a static analysis tool for detecting XSS and SQL injection vulnerabilities. Pixy also addresses the alias problems in PHP [14]. Dynamic confirmation is we compile and execute our representation for vulnerability confirmation. Figure5.3 shows the experimental result of our analyzer.

Program name\Analyzer	TooF	ToolC	Pixy	CANTU	
	Result	Result	Result	Result	Dynamic confirmation
xss.php	TP	TP	TP	TP	TP
interfunxss.php	TP	TP	TP	TP	TP
aliasxss.php	FN	TP	TP	TP	TP
aliasinterxss.php	FN	TP	TP	TP	TP
arrayaliasxss.php	FN	FN	FN	TP	TP
aliasobjectxss.php	FN	TP	FN	TP	TP

Figure 5.3: Comparison results with other analyzer. TP is true positive, FN is false negative.

This experiment is performed on 2010.6.3. We observed that ToolF did not handle alias features in PHP security. Programmer could bypass the automated vulnerability detection by reference assignment before tainted data entering sink function. ToolC could handle alias XSS vulnerabilities without alias relation between array elements. As the limitation mentioned in Pixy [14], it provide no support for object-oriented feature and alias contain array elements. We also confirm these vulnerabilities by executing our CIL representation as well as the original PHP source programs.

We also analyzed PHP Web project in real world. The project is a bookstore with basic shopping process and operations. User could register, search and purchase books on the website. Administrator could manage product information and order messages. Figure 5.4 illustrates the architecture of bookstore and the program entry *index.php*.

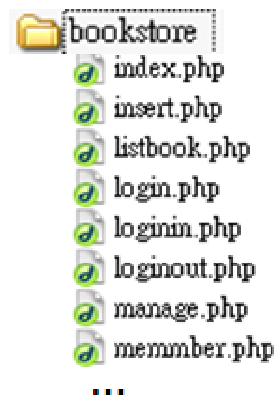


Figure 5.4: The archive of bookstore

Figure 5.5 shows the comparison result with other tools.

Project name\Analyzer	ToolF	ToolC	Pixy	CANTU
bookstore	41	12	33	28

Figure 5.5: Comparison of the experimental result with other analyzer

This experiment is performed on 2010.6.18. In the comparison result, our analyzer presents the ability to detect cross-site scripting (XSS) and SQL injection. ToolC found fewer vulnerabilities because the tool considered data comes from database is untainted. The tainted source assumption may result in some difference in analysis report between tools.

Chapter 6

Conclusion

To detect security vulnerabilities in PHP Web applications more precisely and completely, in this thesis we improved the static analyzer by Chung [7]. We also designed and implemented interprocedural analysis and alias analysis algorithms which provide support for object-oriented features of PHP.

6.1 Contributions

The contributions of this thesis are summarized as follows:

- **A translation of PHP5 features to the CIL Intermediate Representation**
  There are many new language features in PHP5 such as new object-oriented features and exceptions. We studied the new language features of PHP5 and proposed a semantically precise translation to C. Due to the fact that there are no object-oriented features and exceptions in C, we designed several data structures and auxiliary functions to solve this problem and tried to preserve the semantics as precisely as possible.

- **Interprocedural analysis and alias analysis**
  Interprocedural analysis allows taint analysis to cross function boundaries. To support interprocedural analysis, we designed and combined alias analysis with alias relationship in function summary. We improved the analyzer with interprocedural analysis and alias analysis to detect more vulnerability and reduce false positives. Our interprocedural and alias analysis can also cross alias relationship between objects or array elements.

6.2 Further Work

Currently, we have designed and implemented a static analyzer for verifying PHP Web application security. To provide more precise analysis reports and promote the correctness rate, below are some research suggestions for further work.

- **Integrate with a data flow analysis that cuts across JavaScript and PHP.**
  Taint analysis is a common approach to finding security vulnerabilities. It is an approximation analysis technique that may result in false positives. Some false positives come from incomplete dataflow information, e.g., the vulnerable page cannot be requested directly by configuration restriction or parameters passed to the vulnerable page have been sanitized. In Tai's integrated environment [25], the complete control flow crosses PHP, JavaScript, and HTML could reduce false positives in analysis reports. It is worthwhile to integrate these analysis approaches and try to analyze programs with a complicated architecture.

- **Combine with dynamic analysis and automatic penetration test cases generation.**
  With precise translation, our CIL intermediate representation could be compiled and executed. It is worthwhile to develop some dynamic analysis approaches to complement static analysis results. Dynamic analysis can help confirm vulnerabilities and reduce false positives. In Yu's test case generation [35], the expression in an **if** branch condition along an execution path is captured. These captured constraints are then solved to check whether the vulnerable program point is reachable or not. This approach could reduce false positives in static analysis results In the future, it should be worthwhile to extend to more complicated programs.

- **Attempt further CIL program analysis research**
  In our analysis architecture, we chose CIL to be our intermediate representation. The program file structure of CIL is well-constructed and suitable for program analysis. Our CIL representation also could be extended to many CIL program analysis research topics such as liveness analysis.

Appendix

1.xss.php:

```
<?
$a=$_GET[’msg’]; //tainted
echo $a; //xss vulnerability
?>
```

2.interfunxss.php:

```
<?
$a=$_GET[’msg’]; //$a tainted
$b=’constant’;
$x=16;
$sum= add($a,$b);
function add($c,$d){
   $result=$c.$d; //$result tainted
   echo $c; //xss vulnerability 1
   global $x;
   $x=$_GET[’id’]; //$x tainted
   return $result;
}
echo $a; //xss vulnerability 2
echo $sum; //xss vulnerability 3
echo $x; //xss vulnerability 4
?>
```

3.aliasxss.php:

```
<?
$a="name";
$b=&$a;
$a=$_GET[’msg’]; //tainted
echo $b; //xss vulnerability
?>
```

4.aliasinterxss.php:

```
<?
$a="name";
$b=&$a;
add();
function add() {
   global $a;
   $a= $_GET[’msg’]; //tainted
}
echo $b; //xss vulnerability
?>
```

5.aliasarrayxss.php:

```
<?
$book[0]= "Title: Analysis";
$book[1]= "Version: 2";
$book[2]= &$price;  // untainted
$book[3]= "Author: John";
$price="Price:".$_GET["cust_price"]; // tainted
for ($i=0; $i<=5; $i++){
    echo $book[$i]."<br>"; // XSS vulnerability
} ?>
```

6.aliasobjectxss.php:

```
<?
class car {
  var $color;
  function set_color($c){
    $this->color = $c;
  }
}
$mycar = new car;
$mycar->set_color("blue");
$a_mycar = $mycar; // object alias ($a_mycar, $mycar)
$usercolor=$_GET["color];
$a_mycar->set_color ($usercolor); // tainted
echo $mycar->color."<br>"; // XSS vulnerability
?>
```

Bibliography

[1] OWASP Top 10 - 2010 the ten most critical Web application security risks. http://www.owasp.org/index.php, April 19, 2010.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 2006.

[3] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *Proceedings of the 9th International Conference on Compiler Construction CC '00*, pages 110–124. Springer-Verlag, 2000.

[4] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401. IEEE Computer Society, 2008.

[5] Brian Chess and Jacob West. *Secure Programming with Static Analysis.* Addison-Wesley, 2007.

[6] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, 2003.

[7] Chen-I Chung. A static analyzer for PHP Web applications. Master's thesis, National Taiwan University, 2009.

[8] Adam M. Costello and Cosmin Truta. Exception-handling interface for C. http://www.nicemice.net/cexcept/, 2008.

[9] Nico L. de Poel. Automated security review of PHP Web applications with static code analysis. 2010.

[10] Laurent Deniau. *Object Oriented Programming in C*, 2001.

[11] Nenad Jovanovic. *Web application security*. PhD thesis, Technical University of Vienna, 2007.

[12] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. Technical report, Secure Systems Lab Vienna University of Technology, 2006.

[13] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of Web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security PLAS '06*, pages 27–36. ACM, 2006.

[15] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 432–441. ACM, 2005.

[16] George C. Necula, Scott Mcpeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[17] Ocaml. Ocaml programming language. http://caml.inria.fr/, 2008.

[18] National Institute of Standards and Technology. National vulnerability database. http://nvd.nist.gov/.

[19] Vadim Okun, Romain Gaucher, and Paul E. Black. *Static Analysis Tool Exposition (SATE) 2008*. NIST:National Institute of Standards and Technology, June 2009.

[20] OWASP. Common types of software vulnerabilities. http://www.owasp.org/index.php/Category:Vulnerability, 2008.

[21] OWASP. Top 10 2010. http://www.owasp.org/index.php/Top_10_2010, 2010.

[22] PHP. References explained. http://tw.php.net/manual/en/language.references.php.

[23] Chris Shiflett. *Essential PHP Security*. O'Reilly, 2005.

[24] TIOBE Software. Tiobe programming community index for june 2009. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2010.

[25] Chih-Pin Tai. An integrated environment for analyzing Web application security. Master's thesis, National Taiwan University, 2010.

[26] Wikipedia. Abstract syntax tree. http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[27] Wikipedia. Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site_scripting.

[28] Wikipedia. Facebook. http://en.wikipedia.org/wiki/Facebook.

[29] Wikipedia. SQL injection. http://en.wikipedia.org/wiki/SQL_injection.

[30] Wikipedia. SSA form. http://en.wikipedia.org/wiki/Static_single_assignment_form.

[31] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium*. USENIX Association, 2006.

[32] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *ASE*, pages 605–609, 2009.

[33] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, pages 154–157, 2010.

[34] Fang Yu, Tevfik Bultan, Narco Cova, and Oscal H.Ibarra. Symbolic string verfication: An automata-based approach. In *Proceedings of the 15th International SPIN Workshop on Model Checking of Software*, pages 306–324. SPIN, 2008.

[35] Sheng-Feng Yu. Automatic generation of penetration test cases for Web applications. Master's thesis, National Taiwan University, 2010.