國立臺灣大學管理學院資訊管理系(所)

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

網頁應用程式滲透測試案例之自動化產生

Automatic Generation of Penetration Test Cases for Web Applications

游昇峰

Sheng-Feng Yu

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 99 年 8 月

August, 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

網頁應用程式滲透測試案例之自動化產生

Automatic Generation of Penetration Test Cases for Web Applications

研究生： 游昇峰 撰

中華民國九十九年八月

網頁應用程式滲透測試案例之自動化產生

Automatic Generation of Penetration Test Cases for Web Applications

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：游昇峰 撰

中華民國九十九年八月

謝辭

光陰荏苒，轉瞬間兩年的碩士班生涯就這樣好不容易跌跌撞撞的過了。猶記得碩士班入學口試的緊張心情到放榜時的豁然開朗，好像才剛發生不久似的。這兩年裡，雖然表現沒有比別人好，但我卻學到、得到很多寶貴的人事物。

感謝這兩年教導我、培育我的指導教授蔡益坤老師，從老師身上學到許多做人處事與做學問的道理及態度，老師一絲不苟、孜孜矻矻的研究精神，提醒著我隨時隨地都要能秉持著追根究底的態度探索知識最根本的原理。

感謝實驗室一起奮鬥打拼的同學智斌、睿元、怡文。超罩的台大一哥，智斌，多謝兩年來的鼎力相助，祝福你的網站能夠闖出一番名堂。睿元，謝謝你平時的砥礪讓我能夠順利畢業。怡文，感謝你平時的幫忙，祝福你未來工作順利。還有學弟妹們，任峰、奕翔、辰旻也要謝謝你們貼心的幫忙，實驗室未來大好榮景就靠你們了。老大哥明憲，謝謝你常常在我碰到困難時，總能適時的幫助我順利解決問題，如果能有你半個腦袋，我大概可以打遍天下無敵手，還有畢業的學長姐們，正一、晉碩、敔傑、依珊在碩一時給予的幫助，學弟銘感五內。

此外也要感謝我的家人對我無私無悔的奉獻，能在我忙碌於課業壓力之下給予我支持的力量，以及朋友的鼓勵和關懷更是我調劑課業壓力最大的良藥，最重要的是這一路來有喬芳的陪伴。這兩年的學生生活裡，感謝台大資管所每位認真教學的老師們，雖然過程艱辛卻充滿收穫，頂著台大的光環畢業，使我不敢怠慢，期許自己未來對社會能有所貢獻，還有好多需要學習。加油。

感謝有你們成就此時此刻的我，願大家平安喜樂，謝謝!!

游昇峰　謹識
于台灣大學資訊管理研究所
民國九十九年八月

論文摘要

學生：游昇峰　　　　　　　　　　　　　　　　2010 年 8 月

指導教授：蔡益坤

網頁應用程式滲透測試案例之自動化產生

隨著網頁應用程式蓬勃發展，網頁應用程式的安全性日趨重要。目前有許多檢測工具可以幫助程式設計師找出程式中的安全性漏洞，然而依據檢測工具使用分析方法的不同各有其優缺點。分析方法大約分為兩類，一類是使用靜態分析方法直接對程式碼進行分析找出程式中可能的弱點，另一類則是透過動態分析方法在執行程式的環境下進行分析。其中，靜態分析方法為了評估程式所有可能的狀態，必須藉由抽象化的技術來表述，然而這樣的方式卻不可避免地造成分析結果有誤報的情形。至於動態分析方法的問題則在於程式執行的狀態下進行分析很難能夠完全涵蓋程式所有可能執行的路徑，因此分析結果易有漏報的情形。一般而言，嚴謹的程式碼檢測流程需要專家檢視工具分析的結果去排除誤報的情況，然而這樣的動作是非常耗時的，此外專家的知識也會影響判斷的正確性。

本篇論文的貢獻是整合靜態分析與動態測試方法產生滲透測試案例，並且透過自動化執行產生的測試案例來確認弱點，藉此減少專家需要檢視的弱點數量。主要方法是利用廣度優先演算法針對每個弱點使用反向資料流分析找出所有可走到弱點發生位置的源頭，並且針對每條從源頭到弱點發生位置的路徑蒐集限制條件式，再根據弱點種類附加相對應的攻擊字串，最後透過限制式解算器求出各個路徑是否存在一組可行解能夠滿足蒐集的限制條件式。如果存在一組可行解表示攻擊者可攻穿此條路徑的弱點，我們依據限制式解算器算出的結果產生滲透測試案例。接著透過自動化測試的方式執行測試案例，確認真實存在一個攻擊情境可攻穿工具找出的弱點。整體而言，藉由整合靜態分析與動態測試產生測試案例，並且在網站的架構下能夠自動化展示攻擊情境來確認弱點的存在，提供檢測者一個較具準確性的檢測結果。

關鍵字：測試案例、自動化測試、靜態分析、安全性漏洞、網頁應用程式。

THESIS ABSTRACT

Graduate Institute of Information Management
National Taiwan University

Student: Yu, Sheng-Feng Month/Year: June, 2010
Advisor: Tsay, Yih-Kuen

Automatic Generation of Penetration Test Cases for Web Applications

As our daily life increasingly relies on the Web, security of Web applications has become more and more important. There exist quite a few analysis tools that can help programmers find vulnerabilities in Web applications, but there is still much room for improvement. These tools can be roughly divided into two groups by their analysis approaches. One uses static analysis, while the other uses dynamic analysis. The biggest difference between the two groups is that static analysis does not execute the Web applications when performing an analysis, but dynamic analysis does. Besides, static analysis needs to exercise over-approximation techniques to evaluate possible states of the program, which might introduce false positives to the analysis results. On the other hand, dynamic analysis encounters difficulties when it has to generate dynamically as many test cases as possible to cover all paths in the program. The results of dynamic analysis usually contain false negatives because of lower path coverage rates. In general, a rigorous code review process requires human experts to manually inspect the analysis result from analysis tools. It is an essential but time-consuming and error-prone task.

In this thesis, we propose an approach for combining static analysis and dynamic testing to confirm the true vulnerabilities and hence reduce the number of vulnerabilities that human experts have to examine. We apply backward data flow analysis to explore all executable paths of the corresponding vulnerabilities in the target program. In the process of exploring all possible paths by the breadth-first search algorithm, our approach collects simultaneously constraint information along a path. Afterward, we append an attack pattern to the sink variable and try to generate test cases by manipulating constraint solvers to solve collected constraints. Furthermore, given a generated test case, we provide a Web-based testing which can automatically execute the test case and confirm the existence of vulnerabilities. On the whole, our approach integrates static analysis and dynamic testing to provide test cases generation and Web-based test cases execution, producing high-confidence results.

Keywords: Test Cases, Automatic Testing, Static Analysis, Security Vulnerability, Web Applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Web applications are playing an increasingly important role in our life. People like to go online shopping and accomplish various services that might need their confidential information such as a credit card number. Because these information are valuable, service providers have obligation to protect these resources by all means. But, to err is human, it is hard to establish a correct software. Programmers only can do their best to make robust Web applications as much as possible.

With the growth of the amount of Web applications, Web application security has become a significant issue. According to the survey [9], 82% of software security vulnerabilities come from Web applications. Among them, injection and Cross-Site Scripting vulnerabilities account for 16% and 19% of Web application vulnerabilities respectively. Similarly, the latest 2010 report [4] indicates the riskiest vulnerability in recent year is injection flaws, which ranked two in the 2007 report [3]. Injection flaws mean that an attacker can use a malicious input string to fabricate a vicious query, and then the query engine executes it faithfully. At the end, the Web application was defeated. There are more detail descriptions about Web application vulnerabilities in the section 2.1.

Under the circumstances, service providers can detect Web application vulnerabilities by manual approaches or automated approaches. Because Web application vulnerabilities usually are covered in a large number of codes and quite complicated. Through manual approaches to detect vulnerabilities is a time-consuming and error-prone task that may lead to some potential vulnerabilities are overlooked (called false negative). On the other hand, using automated approaches is really an easier and quicker way to find vulnerabilities. Nowadays, there are many kinds of analysis tools to help service providers reduce numbers of vulnerabilities to the best of their abilities before deploying the Web application. However, these tools have several insufficiencies which could be improved like precision and capability.

1.2 Motivation and Objectives

1.2.1 Motivation

Analysis tools can be roughly divided into two groups by the analysis methodology. One is static analysis which needs source code to analyze in the process, and the other is dynamic analysis by running the Web application and generating as much test cases as possible to penetrate it for strengthening confidence in the security of the Web application. In fact, most analysis tools regard static analysis and dynamic analysis as two separation parts; in other words, these tools did not take advantage of two kinds of results to reinforce their analysis confidence. We know that dynamic analysis might have false negatives because it has to generate dynamically as many test cases as possible to cover all paths in the program. As to static analysis, it can help us find most all of vulnerabilities in the target application, but it usually introduces a lot of false positives because of over-approximation philosophy.

Table 1.1: Dynamic Analysis versus Static Analysis

	Pros	Cons
Dynamic analysis	More confident results	Lower path coverage rates Higher false negative rates
Static analysis	More complete results	Higher false positive rates

We take Figure 1.1 as an example of the false positive result from performing static analysis. Figure 1.1 shows a program fragment adapted from a real Web application, a membership authorization management system written in PHP. First, the program assigns the value of the variable $_POST['usergroup'] to the variable $usergroup at Line 02, and assigns the value of the variable $_POST['id'] to the variable $id at Line 03.

```
01<?php
02  $usergroup = $_POST[“usergroup”];
03  $id = $_POST[“id”];
04  if ($usergroup == 1) {
05    //guest
06    echo “Hello, guest”;
07    displayRegistration();
08  }
09  else if ($usergroup == 2) {
10    //member
11    echo “Hello! $id”;
12    displayBasicFun();
13  }
14  else {
15    // admin or supervisor
16    if ($id == “admin”) {
17      echo “Hello! $id”;
18      displayMemberAuthorityManagementFun();
19    }
20  }
21?>
```

Figure 1.1: Motivating Example

Next, the program checks users' group to execute corresponding work. Most all of static analyzers claim that there are two vulnerabilities at line 11 and 17. However, we could easily understand that the vulnerability at line 17 is a false positive alarm. Because if the variable $id can pass through the if block at Line 16, it means that the variable $id is satisfied with the boolean condition. So, the variable $id has a constant value, “admin”, inside the if block.

1.2.2 Objectives

According to our code review experience, we know that code analysis process always needs man with security knowledge involved to go through the analysis results. This is an essential but a time-consuming and error-prone task. Most off-the-shelf analysis tools regard static analysis and dynamic testing as two separation parts, and few tools support an overall code review process from performing analysis to doing testing. The objective in this thesis is emphasized on that enhances the credibility of analysis results and enriches the capability of analysis tools. Through our approach provides a way to minimize time cost of manual code review after using analysis tools. In order to achieve our goal, we use data flow analysis to generate test cases automatically, and these test cases can exploit all possibly executable paths with the corresponding vulnerabilities of Web applications. Because there are fewer static analysis tools can generate test cases which could really exploit weaknesses. These test cases not merely are produced when the existence of vulnerabilities. We want to use dynamic testing to confirm the existence of the vulnerability with executing the penetration test as simulating attackers' behavior. The most important idea is that this thesis combines static analysis and testing approaches to produce a more confident analysis report.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

- In Chapter 2, introducing several related literatures, and discussing what progress researches could do until now. Compare these literatures what they are better or worse than others.
- In Chapter 3, talking about background information regarding this thesis, including Web application vulnerabilities and basic analysis and testing approaches.
- In Chapter 4, presenting overview approaches of test cases generation and test cases execution.
- In Chapter 5, describing the implementation work and experimental results.
- In Chapter 6, summarizing our approaches. Finally, mentioning our contributions and pointing out which parts we could improve in future work.

Chapter 2

Related Work

2.1 Literature Review

There are numerous methods for detecting vulnerabilities in Web applications, but in general, they could be classified into three groups. One is using a static analysis approach, another is testing approach, which executing the program to do analysis. The other is a hybrid approach using static analysis and testing.

- Static analysis approach
  - Taint-based

    Taint-based method is based on some of the data flow analysis that tracks data flow from user inputs (called source) to a sensitive operation (called sink). This type has been applied in numerous papers [15, 16, 26]. Furthermore, most of these literatures assume that if there is a sanitization function performed between source and sink, the result is totally safe or not connecting to the sanitization policy. These policies may result in false alarms.
  - String-based

    A string-based static analysis focuses on evaluating the possibility value of variables at any program points. Using this information to make sure that whether the Web application exists attack strings in sensitive operations or not. Many papers [19, 23, 28] use this method. For example, in [28] its forward analysis does this way.

- Testing approach [25]

  The concept of testing focuses on simulating the runtime environment, and on this environment testers implement their testing plan. Generally speaking, it can be separated as different types by testing methods or levels. For instance, we divides testing methods into two types, black box testing and white box testing.

  - black box testing

    Black box testing means testers don't know what the testing target implements in the internal. Therefore, it is efficient when testing on large applications, and testers don't be equipped with knowledge of implementation details. However, on the other hand, it is hard to cover all possible executing paths which may contain flaws. This paper [8] describes the current state of automated black box testing.

  - white box testing

    Comparing to black box testing, testers knows the internal structure of testing targets. So, testers could use an easier way to find target's weaknesses and cover a extensive range of executing paths.

  In recent years, there are several papers [6, 12, 13, 18, 24] using testing approach to detect Web application security and concentrating on automatically generating test cases, which could be used to exploit vulnerabilities.

- Hybrid approach

  Hybrid approach is to combine the first two, and it could take advantage of both merits and narrow down both defects. However, there is little research [7, 14] has been done.

In the following sections, we will introduce several much related literatures.

2.2 Generating Vulnerability Signatures

Yu *et al.* [28] proposed an automata-based string analysis to automatically generate a vulnerability characterization which contains all possible nocuous input values given an attack pattern. The overall workflow is as Figure 2.1.

In this paper, they need PHP source code and an attack pattern in a regular expression format before performing string analysis. An attack pattern is used to identify if the PHP source code contains vulnerabilities or not. For the XSS attack, the attack pattern could be indicated as Σ* <script Σ*.

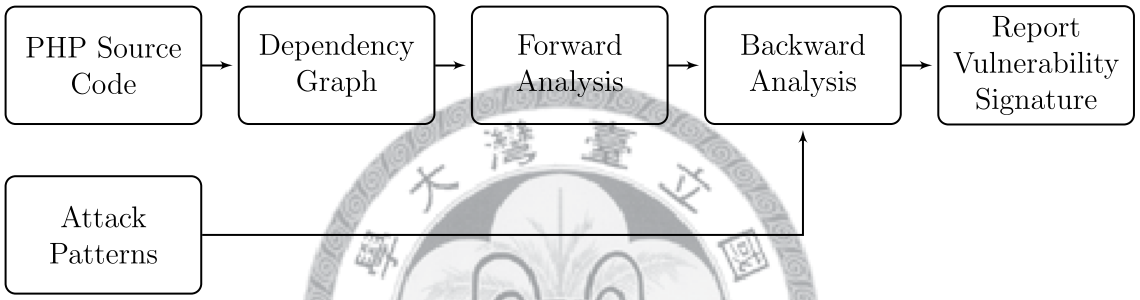


Figure 2.1: An Analysis Flow in [28]

2.2.1 Dependency Graph

In the process of the string analysis, it needs to know the data flow of variables. So, it has to construct a dependency graph (G) to indicates the data flow of the program.

Formal definition:

- G = $\langle N, E \rangle$, where N is a set of nodes which is finite and has two types.
- E $\subseteq$ N $\times$ N is a set of directed edges, and is finite.

Each node n $\in$ N could be:

1. a *normal* node contained input, constant, variable.

2. an *operation* node contained concat and replace. A concat node has two predecessors, one is prefix node (n.p) and the other is suffix node (n.s). It stores the concatenation value of two predecessors.

Besides, they define Succ(n) = {n' | (n, n') ∈ E}, and Pred(n) = {n' | (n', n) ∈ E}, where n ∈ N. Moreover, Root(G) = {n | Pred(n) = ∅}, and Leaf(G) = {n | Succ(n) = ∅}. So, if n is concat node, Pred(n) = {n.p, n.s}. So, if there is program written as Figure 2.2 and its corresponding dependency graph is Figure 2.3.

```
<?php
$article = $_GET[‘article’];
$san_article = preg_replace(
“/[ˆA-Za-z0-9 .-@://]/”, “”, $article);
$content = “content:”.$san_article;
echo $content;
?>
```

Figure 2.2: An PHP Program

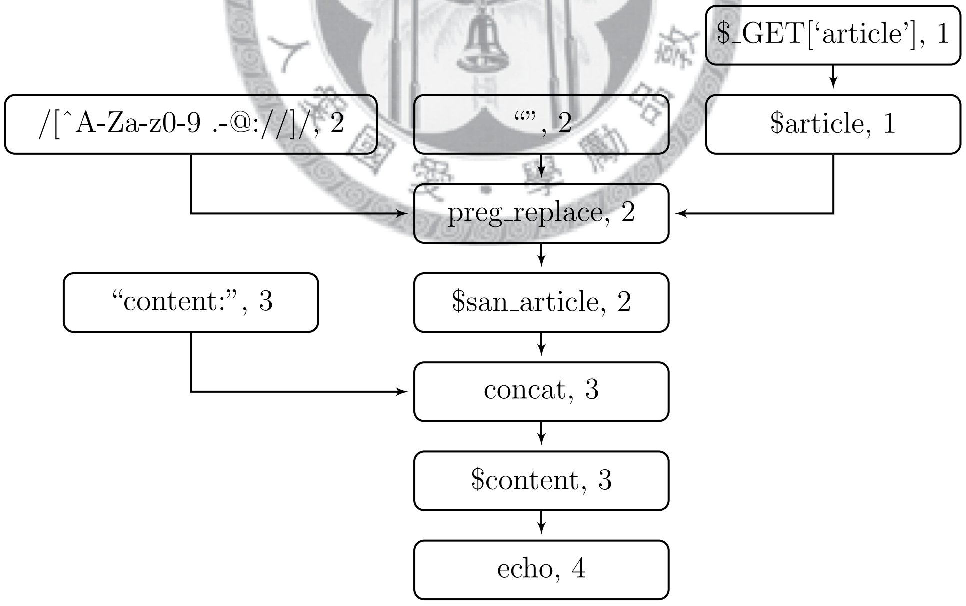


Figure 2.3: A Dependency Graph

2.2.2 Vulnerability Analysis

```
VULANALYSIS(G, Sink, Attk)
{
  Init(POST, PRE);
  set Vul := {};
  FWDANALYSIS(G, POST);
  for each n $\in$ Sink do
    tmp: = POST[n] $\cap$ Attk;
    if L(tmp) $\neq \emptyset$ then
      Vul := Vul $\cup$ {n};
      PRE[n] := tmp;
    end if
  end for
  if Vul $\neq \emptyset$ then
    BWDANALYSIS(G, POST, PRE, Vul);
    for each input n do
      Report the vulnerability signature PRE[n];
    end for
    return "Vulnerable";
  else
    return "Secure";
  end if
}
```

Figure 2.4: Vulnerability Analysis

After generating a dependency graph, then starting to perform vulnerability analysis. Figure 2.4 is the overview of the analysis structure. First, creating two automata vectors POST and PRE. The POST[n] vector stores values which node n can accept in DFA data structure. As for PRE[n], its values can be taken to exploit the vulnerability associated with the given attack pattern. After executing the forward analysis, if the language from the intersection of the POST vector of sink nodes and the attack pattern is not empty, then it means that there exist vulnerabilities in this program at the sink nodes. For finding any possible input values to exploit the vulnerabilities, the paper extends analysis with a backward analysis, which using finite state transducers to model PHP built-in functions and using widening techniques to accelerate the computation of a fixpoint.

2.2.3 Conclusion

In this paper, the authors bring up using automata-based string analysis to character a vulnerability signature. There exists a difficult problem which is to reverse PHP built-in functions, and the authors develop automata-based functions to stimulate PHP built-in functions' pre-image computation. Broadly, it tackles the pre-image computation's problem, and calculates all possible values of variables at any program point. But it does not handle that if a program includes branch or loop statements, which exist commonly in a program. In other words, if taint input data extracted from the vulnerability signature do not match boolean expressions in the branch points from input nodes to sink nodes, and it means that the vulnerabilities will never occur.

2.3 Automatic Creation of Test Cases

Kieżun *et al.* [18] proposed a testing method to create attack vectors which could exploit vulneralbilities including SQL Injection and Cross-Site Scripting. They observe if an attack vector could flow from user input into a sensitive operation or not. The whole analysis process is as Figure 2.5.

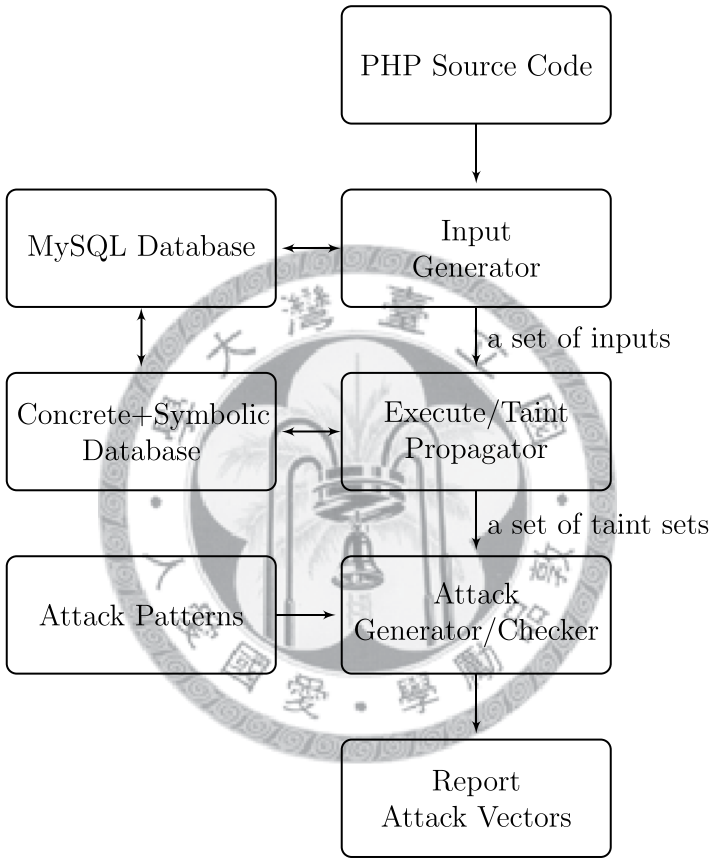


Figure 2.5: An Analysis Architecture in [18]

2.3.1 Technique Components

The following is the explanations of components which are more important.

- Input Generator

  It executes the program with concrete values and collects symbolic constraints in the process. Its goal is to generate numerous of input sets, which cover all possible control flow path from an user input through a program termination point by solving constraints.

- Execute/Taint Propagator

  To run the program with each input set, which created by the input generator. During the process of executing, the taint propagator dynamically monitors the input data flow by modifying the PHP interpreter engine. When it reaches a sensitive operation, then outputs a taint set by recording those parameters, whose values flow into the sink, for each input set.

- Attack Generator/Checker

  The purpose of the attack generator is to substitute the values of each taint set for attack patterns. Then, executing the program again with new input sets, which contains malicious string, and judging whether these new input sets can result in a real attack by finding differences when run two related input sets. One is harmless and the other is malicious.

- Concrete+Symbolic Database

  It is used for tracking taint data flow through a database. Therefore, it can be applied to detect stored XSS.

2.3.2 An Example of the Analysis Flow Process

This section describes an example to make the analysis process more clearly. Figure 2.6 is a snippet program of a message board system modified from this paper. The input generator runs the program and generates some input sets. At the beginning, the first input set has nothing information. Then, recursively running the program, it encounters a branch point on Line 1. The first input is not satisfied with the boolean expression, and the input generator jumps to Line 5 then the program terminates; simultaneously, the input generator collects a constraint, “ $_GET[‘mode’] == “show” ”, in the way. Next round, the input generator solves the collected constraints and creates a new input set which has different path from the previous ones. After exploring all probable paths, the input generator outputs various input sets. One input set is as Figure 2.7. The following step is that the executor executes the program again with those input sets. The taint propagator dynamically tracks whether there is any value of user input may flow into a sink. If yes, then it outputs an outcome, called a taint set. In Figure 2.7, the taint propagator detects that the value of the msg variable flow into the sensitive function (echo) on Line 13 in Figure 2.6. Thus, the taint set of this sensitive function for this input set contains the msg variable. The attack generator generates a corresponding malicious input set which replaces the value of the variable from the taint set with a XSS attack pattern. Now, there are two input sets; one is Figure 2.7, the other is Figure 2.8. The attack checker takes the input set from Figure 2.8 as input date, and then executes the program. If the attack checker judges this input set is a real attack, then signals it as an attack vector.

```
01 if ($_GET[‘mode’] == “show”){
02     showMessage();
03 }
04 else{
05     exit;
06 }
07
08 function showMessage(){
09     if (!isset($_GET[‘msg’])){
10         exit;
11     }
12     $show_msg = $_GET[‘msg’];
13     echo “ Your message is ‘$show_msg’ ”;
14 }
```

Figure 2.6: A Generating Attack Vectors Example

```
mode → show
msg  → 1
(1 means don't care)
```

Figure 2.7: Input Set: I

```
mode → show
msg  → <script>alert("XSS")</script>
```

Figure 2.8: Input Set: I'

2.3.3 Conclusion

In this paper, we can learn how a testing method is used for detecting Web application vulnerabilities. The analysis results are very useful because testers can take these input sets to convince programmers that their program contains vulnerabilities. But there are some difficulties in generating attack vectors. In the goal of testing approach, they hope the analysis can produce many input sets to cover as much paths as possible. However, it is a hard problem, especially for scripting languages which usually contain dynamic language features. Furthermore, when the taint propagator encounters a PHP built-in function, this paper assumes it is a sanitization function. So, the taint propagator will return an empty taint set when programmers use PHP built-in functions, which are satisfied with the sanitization policy of the taint propagator. So, it may lead to a false negative. On the other hand, this paper does not focus on the correctness of a sanitization function. So, it cannot generate an attack vector for a program with wrong sanitization functions.

2.4 Composing Static and Dynamic Analysis

Balzarotti *et al.* [7] proposed a combining static and dynamic analysis method. This paper focuses its goals on analyzing the correctness of the applied sanitization through a program. First of all, it uses static analysis to find out whole probable vulnerabilities, which may contain false positive. Subsequently, it reconstructs a sanitization graph and employs dynamic techniques to penetrate the program with attack patterns. Eventually, to identify whether the program is comprised of incorrect sanitization functions. Figure 2.9 is a brief illustration of the analysis procedure. Then, we discuss in more detail about static and dynamic analysis components.

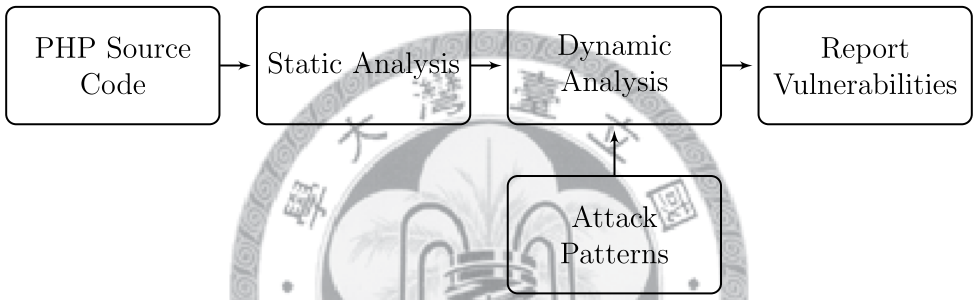


Figure 2.9: An Analysis Process in [7]

2.4.1 Static Analysis

In this paper, it makes use of the data flow technique to report a vulnerabilities result in a conservative approach. It is based on [16] and improved the data flow analysis by using more precise string analysis. It not only consider variables are taint or not, but also record possible values of variables by using finite state automata. In their automata representation, each edge represents either a tainted character (using a dashed line) or an innocuous character (using a solid line). Figure 2.10 is an example of automata, which describing a "hi" string and all possible string. In Figure 2.10, the right hand side automata can stand for an user input like $_GET['id'].

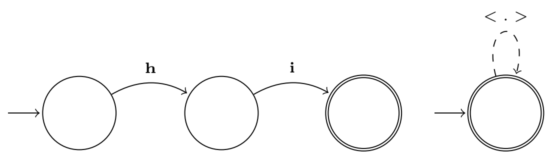


Figure 2.10: Automata Samples

Because it needs to compute values of variables, it modifies the dependency graph in [16]. Its dependency graph represents the dependencies of a variable at a specific program point like [28] did. But there are some differences in nodes and edges. In this paper, each edges in the dependency graph has the opposite direction from 2.3. Further, it contains SCC nodes which use for modeling cyclic string operations. Besides, the operation nodes are separated into two groups. One group can be precisely modeled by finite state transducers. The other group is the inverse of the former and is handled in a over-approximation way in this analysis. Figure 2.11 is the algorithm of dependency graph decoration. It is worth noting that SCC nodes are handled similarly to the second kind of operation nodes.

```
decorate(Node n) {
    decorate all successors of n;
    if n is a string node:
        decorate n with an automaton for this string
    else if n is an <input> node:
        decorate n depending on type of input
    else if n is an operation node:
        simulate the operation's semantics
    else if n is a variable node:
        decorate n with the union of n's successor automata
    else if n is a SCC node:
        decorate n with a star automaton
        (the taint value of its transition depends on
        the successor nodes)
}
```

Figure 2.11: The Decoration Algorithm of a Dependency Graph

2.4.2 Dynamic Analysis

The static analysis result contains lots of false positive, which needs manual inspections to verify. Consequently, this paper introduces a dynamic analysis to compensate inadequacies in the static analysis. For the purpose of verifying custom sanitization functions, the paper constructs a sanitization graph which is a slice of the interprocedural dataflow graph. A sanitization graph is the data structure, which keeps the sequences of sanitization functions. Figure 2.12 is the sanitization graph of Figure 2.2. Like the backward analysis in [28], this paper only pays attention to the variables flow into a sink.

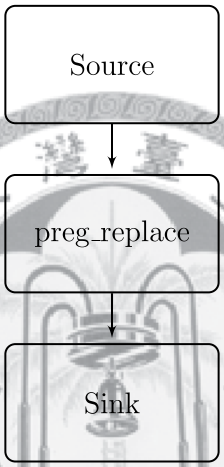


Figure 2.12: A Sanitization Graph

Moreover, depending on the sanitization graph, the paper extracts all possible paths, which are from a source to a sink. For each path, it generates a block of code by concatenating the PHP statements. Then, using the PHP interpreter to evaluate the code with attack patterns. Finally, verifying the correctness of the program with test oracles.

2.4.3 Conclusion

The intention of this paper is to verify the correctness of using custom sanitization functions. This paper uses attack patterns to test if there exists an attack pattern which could bypass the custom sanitization functions and become a real attack scenario or not. However, custom sanitization functions are maybe tricky, it has to work out a special case which can only make it big. So, the repository of attack patterns is a critical point if it is not sufficient to support designing an attack scenario. Therefore, it could make a conclusion that some vulnerabilities are true positive if it can find an attack string, which can frustrate custom sanitization functions. Moreover, in the process of generating code of the dynamic analysis phase, the paper does not take branch and loop statements into account. This way maybe leads to a false positive.

Chapter 3

Preliminaries

In this chapter, we introduce fundamental knowledge of this research, including Web application vulnerabilities, analysis approaches and finite state transducers.

3.1 Common Web Application Vulnerabilities

Table 3.1: OWASP Top 10 Application Security Risks in 2010

OWASP Top 10 - 2010	
A1	Injection
A2	Cross-Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure Direct Object Reference
A5	Cross-Site Request Forgery (CSRF)
A6	Security Misconfiguration (NEW)
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transport Layer Protection
A10	Unvalidated Redirects and Forwards (NEW)

With the development of Web applications, its security have become a essential key factor in a successful Web application. Many third-party organizations track the trend of the Web application security, and the most well-known institution is OWASP (The Open Web Application Security Project). OWASP is a nonprofit and open community, and there are lots of branches in the worldwide. Moreover, much on-going projects have been conducted in full swing, the most famous projects are the Top Ten project and the WebGoat project. The former raises awareness about what is the critical flaws in Web applications in recent years. As for the latter, it depends on the Top Ten project for designing a platform where people can learn what is the root cause of vulnerabilities by exploiting vulnerabilities during the attack process.

According to the latest report [4], we can know the ten most crucial Web application security risks in Table 3.1. There are several differences between the report in 2007 [3] and in 2010. The main difference is that the report in 2010 adopts a concept of risk management methodology to rank Web application security risks. It considers attack techniques, the level of weaknesses and business impact associated with each security risk. The following describes the injection and Cross-Site Scripting risk in detail.

3.1.1 Injection

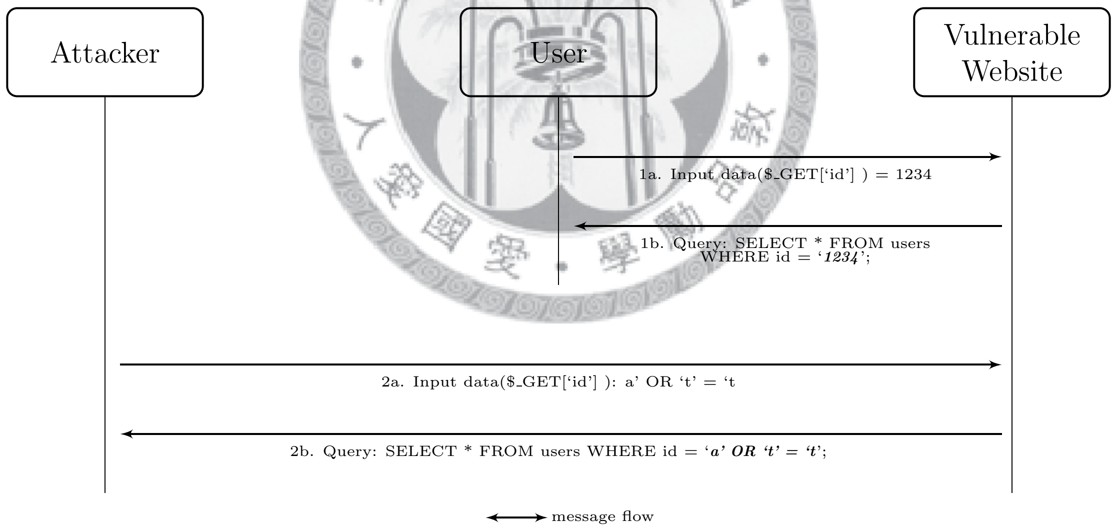

Attacker
User
Vulnerable Website
1a. Input data($_GET['id']) = 1234
1b. Query: SELECT * FROM users WHERE id = '*1234*';
2a. Input data($_GET['id']): a' OR 't' = 't
2b. Query: SELECT * FROM users WHERE id = '***a' OR 't' = 't***';
message flow

Figure 3.1: Typical SQL Injection Scenario

The vulnerability occurs when an untrusty data was sent to an interpreter as a segment of query or command. As a result, the interpreter will execute the query or command faithfully, then this action will lead to critical information leakage and damage. There are some types of injections: SQL, LDAP, XPath, SXLT, HTML, XML and OS command injection. Figure 3.1 is a typical scenario of SQL Injection. In general, programmers escape special characters such as a single-quote(') in user input data before executing this query or command to prevent SQL Injection attack.

3.1.2 Cross-Site Scripting (XSS)

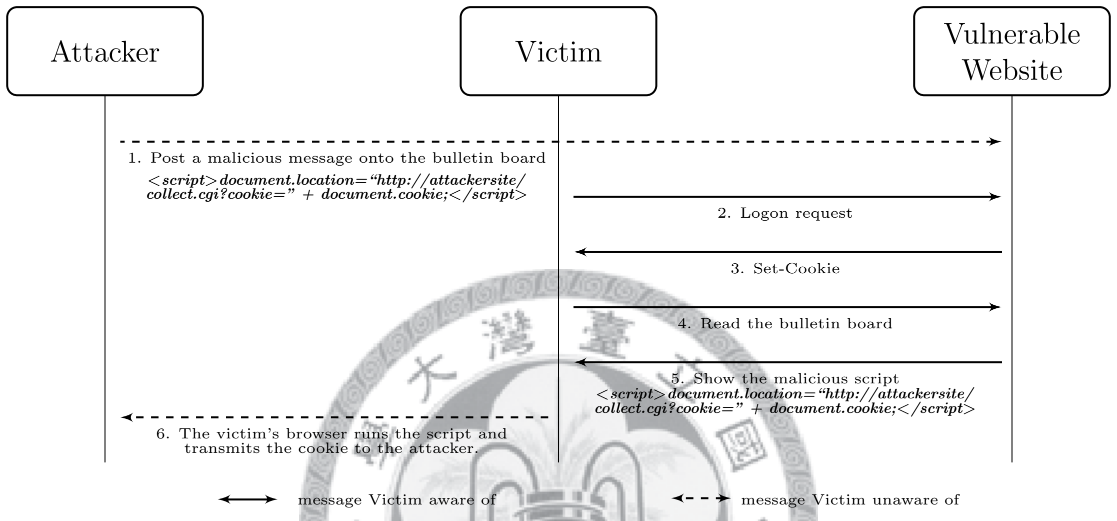

Attacker
Victim
Vulnerable Website
1. Post a malicious message onto the bulletin board
<script>document.location="http://attackersite/collect.cgi?cookie=" + document.cookie;</script>
2. Logon request
3. Set-Cookie
4. Read the bulletin board
5. Show the malicious script
<script>document.location="http://attackersite/collect.cgi?cookie=" + document.cookie;</script>
6. The victim's browser runs the script and transmits the cookie to the attacker.
message Victim aware of
message Victim unaware of

Figure 3.2: Typical Cross-Site Scripting Scenario

This kind of flaws takes place whenever invalidated data were responded to the victim's browser where those tainted data would trigger the script engine in the browser to carry out a harmful act like stealing victim's cookie or conducting phishing attacks. There are three basic types of Cross-Site Scripting: reflected, stored and DOM-based XSS.

A reflected XSS attack is the easiest way to perform. If an attacker found a website equipped with a reflected XSS vulnerability, the attacker can spam emails where included a malicious script to anyone. Then, when an innocent user clicks the link in the email, the vulnerable website will execute the malicious code. A stored XSS vulnerability means that a victim's browser displays malicious scripts which have stored without validated in a database of the vulnerable website such as the scenario in 3.2. The third type of XSS is a weakness that an attacker tampers codes or variables in scripts to intrude a victim's browser. Generally, programmers can encode or filter special characters like a left angle bracket(<) before either responds output strings to a client or stores user input data in a database.

3.2 Analysis Approaches

3.2.1 Static Analysis

Generally speaking, static analysis is a process of analyzing a program without executing it. Static analysis can be used for solving many problems such as type checking, bug finding and program verifying. Using static analysis has some advantages, one is that for some defects like race condition, it is hard to detect by using testing methods. But static analysis can deal with the condition by an easier way. Traditionally, static analysis has a trade-off issue between precision and scalability, and most of static analysis approaches adopt a conservative strategy which might lead analysis results to false alarms. Especially, using static analysis to detect Web applications has such problems because scripting languages possess dynamic features such as generating a Web page dynamically. In the process of performing static analysis for Web application, it has to deal with this situation, so abstraction and over-approximation techniques are inevitable. Figure 3.3 is the workflow of an analysis tool in using static approach from [10].

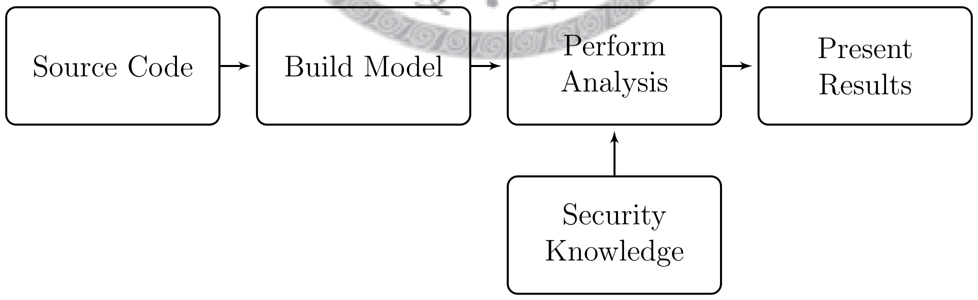


Figure 3.3: The Workflow of Analysis Tools Using a Static Analysis

First of all, testers have to get the Web application source code, which the elementary component in the static analysis. Then, they need to translate them to a model where testers could perform analysis. Furthermore, testers applies any type of analysis like taint analysis or alias analysis on the model depending on what they aim for. To go into details, in the procedure of constructing a program model, it needs to go through three steps. First, it has to do a lexical analysis, which takes an action of converting important language features of source code into a series of tokens. These tokens are already pre-defined. For example, if there is a code such as "if (items) sum = unitprice * items;". After lexical analyzing, it will output a token stream as "IF LPAREN ID(items) RPAREN ID(sum) EQUAL ID(unitprice) TIMES ID(items) SEMI". Futher, using a context-free grammar (CFG) to define a language, and match the token stream with the context-free grammar. A context-free grammar is a grammar which consists of a set of production rules that describe nonterminal or terminal symbols in the language. "Context-free" means that nonterminal symbols could be rewritten regardless of the context where they occur. For example, Figure 3.4 is production rules of a context-free grammar. After done a lexical analysis, it could get a parsing tree corresponding to the code, see Figure 3.5.

```
stmt:= if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval | expr TIMES expr
assign_stmt := lval EQUAL expr SEMI
lval := ID
```

Figure 3.4: Production Rules of a Context-free Grammar

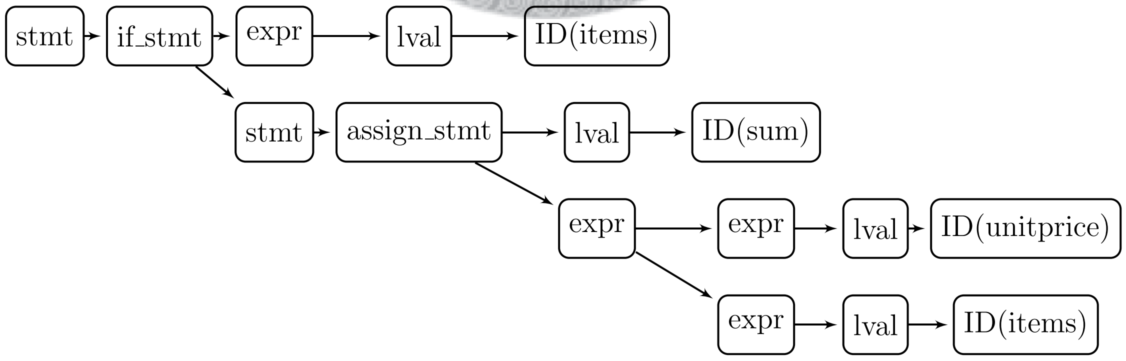


Figure 3.5: A Parsing Tree

However, a parsing tree is not convenient to perform complex analysis, because the parse tree has nonterminal symbols that are meaningless. So, for obtaining meaningful components of a language, it transform a parsing tree to an abstract syntax tree (AST) which provides a standard view of the program for succeeding analysis. Given an abstract syntax tree, it can stand for the original program, and applying kinds of analysis to solve problems that testers focus on.

3.2.2 Testing

Testing is another way to help developers suppress vulnerabilities before the application deployment in the software development life cycle. The following Figure 3.6 is an overview of a testing process [21].

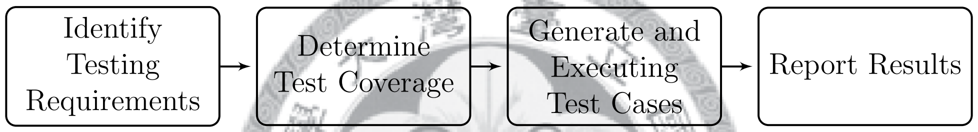

Identify Testing Requirements
Determine Test Coverage
Generate and Executing Test Cases
Report Results

Figure 3.6: A Fundamental Testing Process

In the first step, testers have to identify what is the problems they want to test. For example, they can establish basic requirements of a security policy the application has to conform. During the second step, testers determine what is the scope they need to test depending on the requirement. For instance, to test whether all branches presented in the application contains SQL Injection or Cross-Site Scripting vulnerabilities. In the third step, generating test cases by artificial or automatic methods. Next, testers need to determine oracles for each test case and run test cases. Finally, testers assess the results with pre-defined oracles to recognize flaws in the application.

Chapter 4

Approach

4.1 Overview

In this thesis, we propose an approach which combines static analysis and dynamic testing. This approach generates test cases to simulate attack scenarios and reports highly confident results. The problem we deal with can be divided into two parts.

- How to generate test cases automatically
- How to execute test cases automatically within a Web-based architecture

Figure 4.1 is the architecture of our approach. We generate test cases for target programs, and then take a test case as an input to perform dynamic testing. The following sections describe details of our approach.

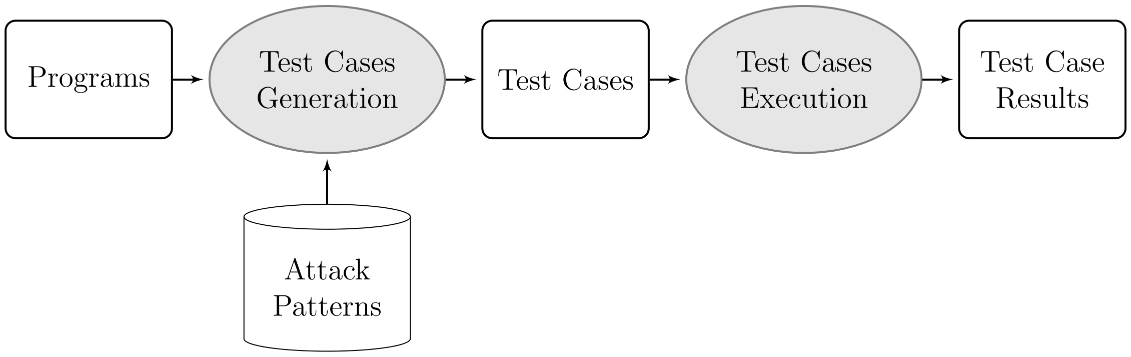
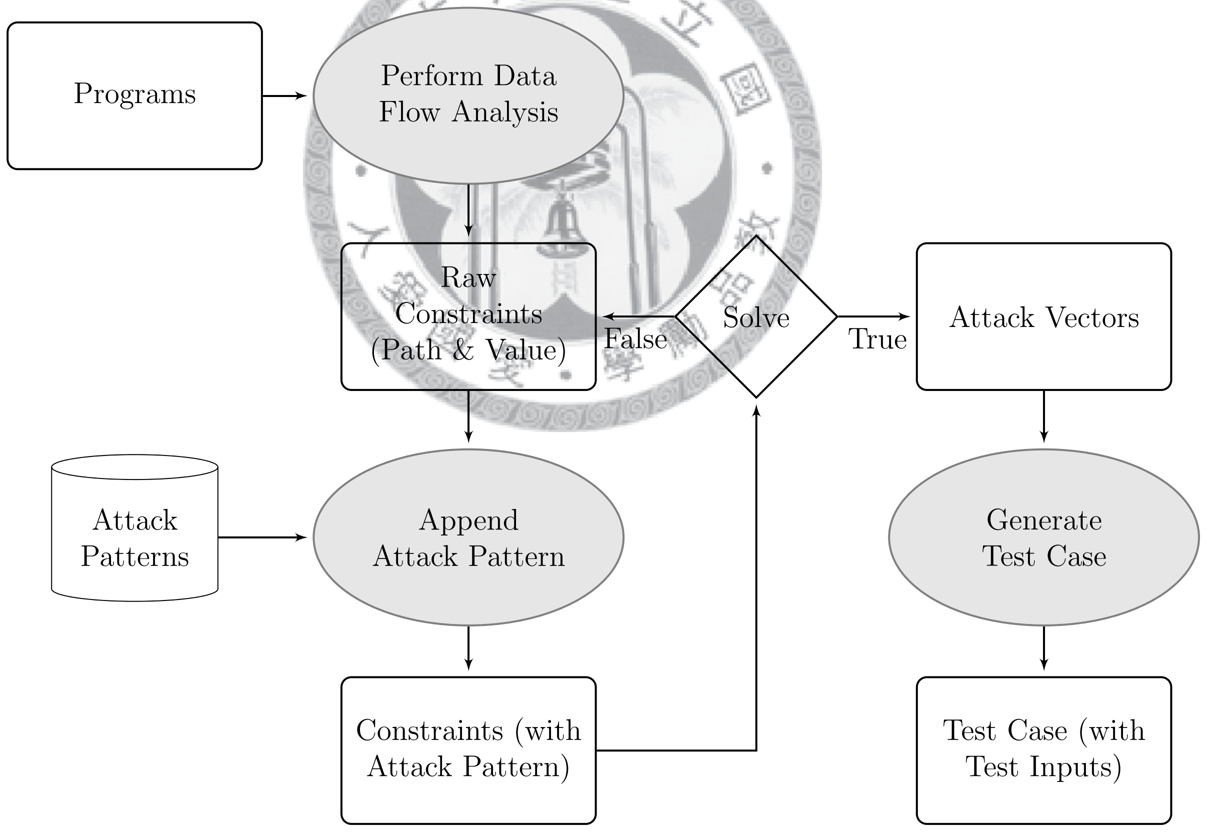


Figure 4.1: An Approach Architecture

4.2 Test Cases Generation

Because we want to generate test cases which are corresponding to different paths they took, our approach have to explore all executable paths in programs. So, the main method to generate test cases is using backward data flow analysis in the breadth-first search method. In the process of expanding all possible paths, the approach simultaneously collects constraint information along a path. When a path has no predecessor, then it invokes a constraint solver to solve collected constraints. If the solver can solve the constraints, the approach generates a test case according to the path, or it applies another attack pattern and invokes the solver again until all attack patterns are used or constraints are solved. Figure 4.2 is the flow diagram of test cases generation.

4.2.1 Flow Diagram



Programs
Perform Data Flow Analysis
Raw Constraints (Path & Value)
False
Solve
True
Attack Vectors
Attack Patterns
Append Attack Pattern
Generate Test Case
Constraints (with Attack Pattern)
Test Case (with Test Inputs)

Figure 4.2: The Flow Diagram of Test Cases Generation

4.2.2 Pseudo-code for Algorithm

Figure 4.3 is the algorithm of our analysis. The inputs of the algorithm contain a program, sink statements and attack patterns. The results of the algorithm are several test cases.

- First, the algorithm constructs the control flow graph of the program.
- Second, it does preprocessing works to collect constraints for each sink statements and puts them as a path in the input queue.
  - A path contains two parts; one is the identifier of the currently processing statement and the other is collected constraints.
- Next step is to check if the input queue is empty.
  - If the input queue is empty, it means that there are no vulnerabilities in the program.
  - Otherwise, the algorithm takes one node from the input queue and checks if the node has predecessors.
    - If the node has no any predecessors, it means that the path comes to the end and the algorithm appends an attack pattern to the collected constraints. Then the algorithm transforms the collected constraints to conform the static single assignment form (SSA form). Afterward, the algorithm invokes a constraint solver to solve transformed constraints.
    - If the node has predecessors, it means that there are different paths which can go to this processing node. So, the algorithm collects constraints of the preceding statement and concatenates these two constraints of the processing node and the preceding statement. Subsequently, the algorithm puts the new constraints and the identifier of the preceding statement as a new path node to the input queue.

```
parameter: Program $P$, SinkStmts $S$, AttackPatterns $AP$
result: Test cases in $P$
constructCFG($P$); // construct control flow graph
inputQueue := emptyQueue();
// path contains : (1) stmtid: current processing statement id
//                 (2) constraints: collected constraints along the path
// preprocessing
foreach sink in $S$ do
  path.stmtid := sink;
  path.constraints := collectConstraints(sink);
  enqueue(inputQueue, path); // put a sink in the queue
end
// perform backward data flow analysis in Breadth-First Search
while notEmpty(inputQueue) do
  predecessorQueue := emptyQueue();
  enqueue(predecessorQueue, path.stmtid.predecessor);
  // the statement has no predecessor
  if empty(predecessorQueue) then
    apQueue := emptyQueue();
    enqueue(apQueue, $AP$);
    while notEmpty(apQueue) do
      ap := dequeue(apQueue);
      apConstraints := appendAP(sinkvar, ap); // append an attack pattern to the sink variable
      constraints := concatConstraints(path.constraints, apConstraints);
      solverInput := translateToSSA(constraints); // translate input to conform SSA form
      if satisfiable(solverInput) then // invoke constraint solver
        result := solve(solverInput);
        generateTestCase(result); // generate a test case
        break;
      end
    end
  end
  // the statement has predecessor
  while notEmpty(predecessorQueue) do
    stmtid := dequeue(predecessorQueue);
    newPath.stmtid := stmtid;
    newConstraints := collectConstraints(stmtid);
    newPath.constraints := concatConstraints(path.constraints, newConstraints);
    enqueue(inputQueue, newPath);
  end
end
```

Figure 4.3: Data Flow Analysis

4.2.3 Constraints Specification Language

We applied the Kaluza constraint solver in our implementation, which is developed by a team in Berkeley in 2010 [20]. We introduce the simplified syntax of the Kaluza core language as Figure 4.4. Also, we give an example to interpret mapping relationship between a program in Figure 4.5 and constraints in Figure 4.6.

```
Constraints   ::=  Constraints Constraint
Constraint    ::=  LHS OP RHS ";"
                |  LHS SOP """ ConstantStr """ ";"
                |  LHS \in "/" RegExp "/" ";"
                |  LHS \notin "/" RegExp "/" ";"
LHS           ::=  Var
RHS           ::=  Var
                |  Int
ConstantStr   ::=  [a-zA-Z0-9][a-zA-Z0-9]*
Int           ::=  [0-9][0-9]*
OP            ::=  SOP
                |  >
                |  >=
                |  <
                |  <=
SOP           ::=  ==
                |  !=
```

Figure 4.4: The Formal Specification of Constraints

```
<?php
  if($_GET['mode'] == "add"){
    if(!isset($_GET['msg']) || !isset($_GET['poster'])){
      exit;
    }
    $my_msg = $_GET['msg'];
    $my_poster = $_GET['poster'];
    if (strlen($my_msg) > 16){
      echo "Thank you for posting the message $my_msg";
    }
  }
?>
```

Figure 4.5: An Example of PHP Program

```
T_1 == var_0xINPUT_mode;
T_1 == "add";
T_2 == var_0xINPUT_msg;
T_2 != "";
T_3 == var_0xINPUT_poster;
T_3 != "";
T_mymsg == T_2;
T_myposter == T_3;
Len(T_mymsg) > 16;
T_mymsg \in /.*<script>.*/;
```

Figure 4.6: The Collected Constraints of Figure 4.5

In the analysis process, we collected two kinds of constraints in programs. One is path constraints, like Boolean expression in IF statements or SWITCH CASE statements. The other is value constraints, like assignment statements. In Figure 4.5, path constraints contains "$_GET['mode'] == "add"", "!isset($_GET['msg'])", "!isset($_GET['poster'])" and "strlen($my_msg) > 16". Value constraints include "$my_msg = $_GET['msg'];", "$my_poster = $_GET['poster'];" and "echo "Thank you for posting the message $my_msg";". We use Table 4.1 to represent the relation between Figure 4.5 and Figure 4.6. It is worth talking about the final row in Table 4.1. As mentioned above, we append different attack patterns to generate test cases in our approach. In this example, the appended attack pattern is ".*<script>.*".

Table 4.1: The Mapping between A Program and Constraints

Program	**Constraints**
$_GET['mode']	T_1 == var_0xINPUT_mode;
$_GET['mode'] == "add"	T_1 == "add";
$_GET['msg']	T_2 == var_0xINPUT_msg;
!isset($_GET['msg'])	T_2 != "";
$_GET['poster']	T_3 == var_0xINPUT_poster;
!isset($_GET['poster'])	T_3 != "";
$my_msg = $_GET['msg'];	T_mymsg == T_2;
$my_poster = $_GET['poster'];	T_myposter == T_3;;
strlen($my_msg) > 16	Len(T_mymsg) > 16;
echo "Thank you for posting the message $my_msg";	T_mymsg \in /.*<script>.*/;

4.2.4 Static Single Assignment Form

Before solving collected constraints, we have to translate those constraints in static single assignment form. For example in Figure 4.7, there is one vulnerability at Line 08 when the $id variable is equal to constant strings, "bob" or "admin". In our approach, we generate two executable paths in this example, and we could see each collected constraints in Table 4.2. We found that if we do not translate constraints into static single assignment form, the constraints in path 1 are unsolvable. In fact, attackers can exploit the vulnerability at Line 08 along two different paths. So, we have to translate collected constraints into static single assignment form. The simplified method is as following.

- Step 1: Create two hash tables, one stores distinct variables in LHS and the other stores matched variables in LHS and its substitutions.
- Step 2: Take a constraint from collected constraints of one path.
- Step 3: Check it if the syntax of the constraint is conformed to the format, "LHS op RHS". If conformed, then go to step 4, or go back to step 2 to take another constraint.
- Step 4: Take a look at the variable in RHS. If the variable in RHS is matched in the later hash table, then replace it to its substitution. Or if not matched, do nothing.
- Step 5: Take a look at the variable in LHS. If the variable in LHS is matched in the former hash table, then check if this variable has occurred in the later hash table. If the variable in LHS is matched in the later hash table, then create a new mapping substitution, replace this variable to its new substitution, and update the substitution of this variable in the later hash table. Or if not matched in the later hash table, add this variable to the later hash table and also create a substitution and replace this matched variable to its substitution. Or if not matched in the former hash table, then add this variable to the former hash table.
- Step 6: Continue step 2 to 5 until all constraints are processed, and we can transform collected constraints of paths in static single assignment form.

```
<?php
   $id = $_GET['id'];
   $name = $_GET['name'];
   if($id == "bob"){
      $id = "admin";
   }
   if($id == "admin"){
      echo "Hi $name";
   }
?>
```

Figure 4.7: An Tricky Example of PHP Program

Table 4.2: Two Executable Paths

Program	**Constraints in Path 1**	**Constraints in Path 2**
$_GET['id']	T_1 == var_0xINPUT_id;	T_1 == var_0xINPUT_id;
$id = $_GET['id']	T_id == T_1;	T_id == T_1;
$_GET['name']	T_2 == var_0xINPUT_name;	T_2 == var_0xINPUT_name;
$name = $_GET['name']	T_name == T_2;	T_name == T_2;
$id == "bob"	T_id == "bob";	
$id = "admin"	T_id == "admin";	
$id == "admin"	T_id == "admin";	T_id == "admin";

4.3 Test Cases Execution

After generating test cases, we want to automatically run test cases under a Web-based architecture and make sure those can really exploit the vulnerability. There are many test tools which can automatically demonstrate test scenarios for Web applications. Traditionally, these tools always need to install components in where you want to execute these tests. It is rare that executing test cases takes place under a Web-based architecture. Moreover, we want those people who execute test cases do not need to install tools at their computers. So, we conceive an approach which can fulfill these requirements which are automatically demonstrating test scenarios under a Web-based architecture and installing nothing in the client-side. There are four components in our approach to execute a test case.

- Target Application : the target of test cases
- JavaScript Code : the main component in our approach which is used to manipulate DOM objects in a Web page. The fragment code is shown in Appendix A1.
- Test Case : a test case from the step of test cases generation
- GetStep Code : retrieving step information from a test case. The fragment code is shown in Appendix A2.

The whole process of test cases execution is described in the following. First, we need to build applications under test (AUT). This step is to duplicate the origin environment where the application is to our server and make sure it can be run on our server properly. After the process of test cases generation, we could have some test cases if the application is vulnerable. Each test case is an attack scenario which we want to demonstrate and confirm it as a true positive. So for a test case, we have to understand which of Web pages need to be instrumented and this information are described in the test case. Depends on these information, we instrument those Web pages with the JavaScript Code. After instrumentation, we redirect the Web page to the beginning page which described in the test case and start to demonstrate the attack scenario. We show the process of instrumentation in Figure 4.8. After redirecting the beginning page, the instrumented JavaScript Code uses the Ajax method to automatically send an asynchronous Httprequest from a client to our server. The Httprequest requests a step to manipulate by communicating with the server-side code (the GetStep Code). According to the value of the SESSION variable, we know which step has to be dealt with and return it to the JavaScript Code. After acquiring the untreated step information, the JavaScript Code manipulates DOM objects of the Web page depends on these information. After manipulating this step, the instrumented JavaScript Code automatically send another asynchronous Httprequest to our server until all steps in the test case are processed. We show programs of the GetStep Code and the JavaScript Code in Figure 4.9 and Figure 4.10 respectively. Moreover, Figure 4.11 describes the flow diagram of test cases execution mechanism. This approach can apply to any server-side language because the JavaScript Code is independent with a server-side language.

```
parameter: TestCase TC
// check the existence of the test case
if checkFileExist(TC) then
   // collect pages which need to be instrumented
   instrumentPages := getInstrumentPagesInfo(TC);
   foreach page in instrumentPages do
      // instrument the JavaScript code
      instrumentJS(page);
   end
   // get the beginning page in this test case
   beginningPage := getBeginningPageInfo(TC);
   // keep the processed step information in the SESSION variable
   // which is a global variable, the default value is 0
   SESSION.step := 0;
   // redirect to the beginning page and start to demonstrate an attack scenario
   redirectToPage(beginningPage);
end
```

Figure 4.8: The Process of Instrumentation

```
parameter: TestCase TC, SESSION S
result: Step information in TC
// check the existence of the test case
if checkFileExist(TC) then
   // get the processed step number in the SESSION variable
   stepNum := getStepNum(S);
   // get the next step information in the test case
   // a step contains: (1) action: which action the JavaScript Code takes
   //                  (2) targetId: a DOM object's id
   step := getStepInfo(TC, stepNum+1);
   S.step := stepNum+1;
   return step;
end
```

Figure 4.9: The Program of The GetStep Code

```
// automatically send an asynchronous Httprequest to the GetStep Code,
// and then get a step infomation
step := sendAJAX(GetStep);
while notEmpty(step) do
  // according to the action and the target id
  // to manipulate the DOM object of the Web page
  doAction(step.action,step.targetId);
  // get a next step
  step := sendAJAX(GetStep);
end
```

Figure 4.10: The Program of The JavaScript Code

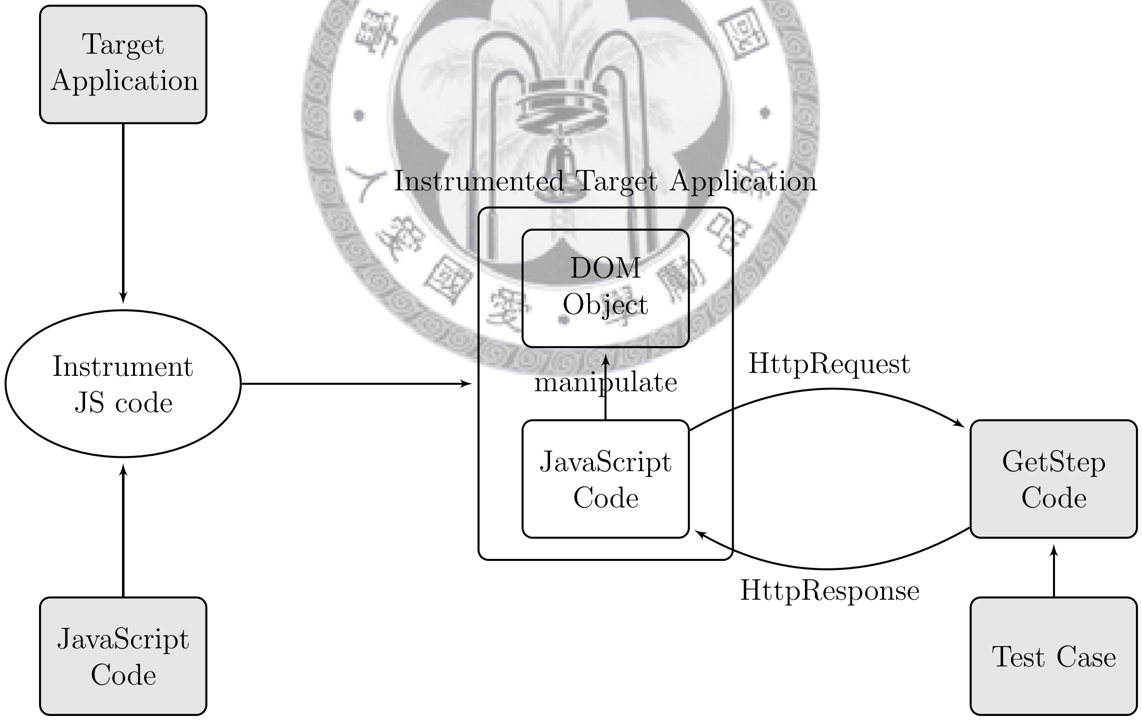


Figure 4.11: The Flow Diagram of Test Cases Execution

Chapter 5

Implementation and Evaluation

5.1 Implementation

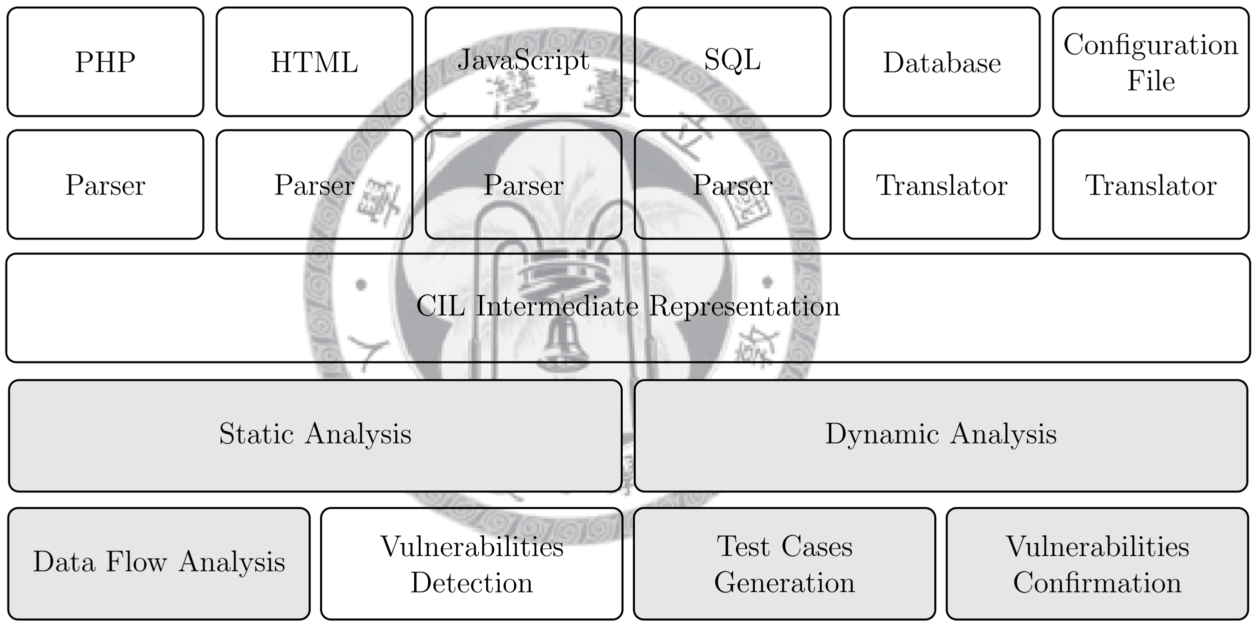


Figure 5.1: Architecture of the Environment

Our approach is implemented as a part of CANTU environment which depicted in Figure 5.1. The transformation approach from target programs to programs in CIL format is purposed by [11, 22, 27]. Our effort acts on the below right part of the Figure 5.1, which is filled in light gray. We focus on using data flow analysis to generate test cases and applying dynamic testing to confirm the existence of vulnerabilities under a Web-based architecture. Also, we generate test cases to describe attack scenarios.

5.1.1 Test Cases Generation

Under the CANTU architecture, the target programs will be transformed in C Intermediate Language. Relying on the efforts of [22, 27], the input program of our analysis algorithm is a program in C Intermediate Language. The program in CIL contains complete information of the target application, and we apply our approach on it. Furthermore, sink statements, one parameter of analysis input, are obtained from the taint analysis result in [27]. The analysis algorithm is implemented in OCaml. Besides, we need a constraint solver to solve collected constraints which could be in types of arithmetic or string. We find that most constraint solvers can only solve one type of constraints at a time. For example, choco [1] and lp_solve [2] are arithmetic solvers, and hampi [17] is a string solver. There are few solvers which could solve arithmetic and string constraints at the same time. Finally, we find out the Kaluza constraint solver, which is developed by a team in Berkeley in 2010 [20]. It supports a rich set of constraints over string, integer and Boolean variables, including most possible constraints that Web applications may have. So, we adopt the Kaluza constraint solver in our implementation.

5.1.2 Test Cases Specification

IEEE 829-2008 [5] is an IEEE standard that specifies the format of a set of documents for use in eight defined stages of software testing, each stage potentially producing its own separate type of document. In the standard, a test case means a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a particular requirement.

The format of our test cases is referenced from the IEEE 829-2008 standard, and we modify some elements to meet our needs in automatically executing these test cases. We formally define a test case in Figure 5.2, and a test case is represented in the XML format. The contents of a test case include its identifier, vulnerability type, scenario and expected result. The following introduces the motive of each element, and we show an actual test case in Appendix B2.

- TestCase : contains identifier, vulnerability type, scenario and expected result
- TCId : the identifier of a test case
- Vul : vulnerability type that the test case describes
- Scenario : a sequence of steps
  - TestStep : contains step identity, target page, action, target component and type string when the action is to type
  - StepAct : actions we handled, included browse, type, click and confirm action
  - StepTarget : the target of the action
  - StepTypeStr : for typing in the target
- ExpValue : for confirming the test case result
  - ExpType : types of confirming methods
  - ExpInfo : the expected string

We also formally define a test case result in Figure 5.3. The contents of a test case result include a test case identifier, project name, project version and actual result. We show a test case result in Appendix B3.

TestCase	::=	“<TestCase>” TCId Vul Scenario ExpValue “</TestCase>”
TCId	::=	“<tcId>” ConstantStr “</tcId>”
Vul	::=	“<vulnerability>” VulSpec “</vulnerability>”
VulSpec	::=	“XSS”
	|	“SQLI”
Scenario	::=	“<scenario>” TestStepList “</scenario>”
TestStepList	::=	TestStep
	|	TestStepList TestStep
TestStep	::=	“<step>” StepID StepPage StepAct StepTarget StepTypeStr “</step>”
StepID	::=	“<id>” Int “</id>”
StepPage	::=	“<page>” FileName “</page>”
FileName	::=	ConstantStr “.” FileTypeSpec
FileTypeSpec	::=	“php”
	|	“html”
StepAct	::=	“<action>” StepActSpec “</action>”
StepActSpec	::=	“browse”
	|	“type”
	|	“click”
	|	“confirm”
StepTarget	::=	“<target>” TargetID TargetName “</target>”
TargetID	::=	“<id>” ConstantStr “</id>”
TargetName	::=	“<name>” ConstantStr “</name>”
StepTypeStr	::=	“<typingString>” ConstantStr “</typingString>”
ExpValue	::=	“<expectedValue>” ExpType ExpInfo “</expectedValue>”
ExpType	::=	“<type>” ExpTypeSpec “</type>”
ExpTypeSpec	::=	“title”
	|	“document”
ExpInfo	::=	“<info>” ConstantStr “</info>”
ConstantStr	::=	[a-zA-Z0-9][a-zA-Z0-9]*
Int	::=	[0-9][0-9]*

Figure 5.2: The Formal Specification of A Test Case

TestCaseResult	::=	“<TestCaseResult>” TCId Project Result “</TestCaseResult>”
TCId	::=	“<tcId>” ConstantStr “</tcId>”
Project	::=	“<project>” ProjectName ProjectVersion “</project>”
ProjectName	::=	“<projectName>” ConstantStr “</projectName>”
ProjectVersion	::=	“<projectVersion>” ConstantStr “</projectVersion>”
Result	::=	“<result>” ResultSpec “</result>”
ResultSpec	::=	“” | “Passed” | “Failed”
ConstantStr	::=	[a-zA-Z0-9][a-zA-Z0-9]*
Int	::=	[0-9][0-9]*

Figure 5.3: The Formal Specification of A Test Case Result

5.1.3 Confirm XSS and SQL Injection

We use prepared attack patterns and its corresponding expected result to confirm the penetration test passed or failed. So, for each attack pattern, they have their own corresponding expected result we need to know. First, we have to consider how a vulnerability will be exploited, and then under the context to build an attack pattern which can help us to confirm the result.
The following describes the root cause of the existence of vulnerabilities

- XSS
  - If there is XSS vulnerability, it means that the return string to the client-side browser contains malicious input which included angle brackets and “script” string.
  - For example, <script> ... </script>, and “...” could be any words which can be composed by JavaScript grammar
- SQL Injection
  - If there is SQL Injection vulnerability, it means that the parameterized SQL query contains user input and the input has unwilling characters (single quote) and SQL reserved words which make the SQL query has a different structure and meaning which not conformed to programmers’ original intentions.
  - For example, a SQL query like
    SELECT name FROM user WHERE id = ‘ ... ’. “...” is the values from user input, and the most well-known injection string is ’ OR ‘1’ = ‘1 which contains single quotes and SQL reserved words and the string makes the **WHERE** clause is a tautology.

The following is our collected attack patterns which one for penetrate the XSS vulnerability and another for SQL Injection vulnerability. In Table 5.1, we could realize that

- XSS
  - **<script> ... </script>** : the root element of a XSS vulnerability
  - **document.title="xss";** : For the purpose of confirming, the Web page title has the "xss" string.
- SQLI
  - For example, a SQL query like
    SELECT name FROM user WHERE id = ' ... '
  - **' AND '1' <> '1'** : makes the first **SELECT** statement has no any result
  - **UNION** : concatenates two SQL statement results
  - **CONCAT(parameter1, parameter2)** : returns the concatenation of parameter1 and parameter2
  - **#** : a comment sign
  - **' AND '1' <> '1' UNION SELECT CONCAT('sql-','injection') #** : For the purpose of confirming, the Web page document has the "sql-injection" string.

Table 5.1: Attack Patterns

Vulnerability Type	Attack Pattern	Expected Value	
		Type	Info
XSS	<script>document.title="xss";</script>	title	xss
SQLI	' AND '1' <> '1' UNION SELECT CONCAT('sql-', 'injection') #	document	sql-injection

5.2 Evaluation

We designed five programs which are shown in Appendix C to evaluate the correctness of our analysis. Table 5.2 shows the experimental results of our analysis. The following describes the contents of each column's header.

- Program : the test program
- Num of Line : the total number of program lines
- C/F Tool : commercial static analyzers
  - Num of Vul : the total number of vulnerabilities which the tool detected in the test program
  - Line of Vul : the number of lines of the vulnerability
- Our Analyzer : our tool in [11, 27]
  - Num of Vul & Line of Vul : the same in the above
  - Test Case Generation : the results of our approach
    - Num of Path : the total number of paths which can reach the vulnerability
    - TP (solve) : the total number of true positives (the solver can solve)
    - FP : the total number of false positives
    - FN : the total number of false negatives

The following describes the motive of each program.

- Ex1 : We can handle a basic case which only contains assignments.
- Ex2 (branch) : We can handle a case which contains one branch.
- Ex3 (falsepositive) : We show our value by writing a case which makes most static analyzers report a result which contains a false positive.
- Ex4 (multibranches) : We make sure our analysis could really explore many paths.
- Ex5 (different paths) : We express that only specific paths can exploit the vulnerability.

We compare our analysis result with commercial tools, and we can find that these tool results contain false positives in Ex3 and Ex4. So, if applying our approach in the process of static analysis, the analysis result can be more confident. We also execute test cases which generated by the process of test cases generation, and the Web-based testing can exactly automatically execute and confirm these test cases. We show the target programs with the instrumented JavaScript code, an actual test case and a test case result in Appendix B.

Table 5.2: Experimental Results

Program Name	Num of Line	C/F Tool		Our Analyzer					
		Num of Vul	Line of Vul	Num of Vul	Line of Vul	Test Case Generation			
						Num of Path	TP (solve)	FP	FN
Ex1	4	1	3	1	3	1	1	0	0
Ex2 (branch)	12	1	6	1	6	1	1	0	0
Ex3 (falsepositive)	15	2	6	2	6	1	1	0	0
			12		12	1	0	True	0
Ex4 (multibranches)	20	4	6	4	6	1	1	0	0
			10		10	1	0	True	0
			14		14	2	0	True	0
			18		18	2	2	0	0
Ex5 (different paths)	17	2	13	2	13	2	1	0	0
			15		15	4	2	0	0

Chapter 6

Conclusion

Many tools have been developed that use static or dynamic analysis to detect security vulnerabilities in Web applications. However, few tools can support a comprehensive code review process from analysis to testing. In this thesis, we proposed an approach that combines static analysis and dynamic testing for detecting Web application vulnerabilities to produce high-confidence analysis results and enhance the capability of analysis tools. Generally speaking, a rigorous code review process requires human experts to manually inspect the analysis result which are produced by analysis tools. This is an essential but time-consuming and error-prone task. Our approach could reduce the number of vulnerabilities that a human expert has to examine and hence avoid errors made by men and minimize the cost of manual code review.

Our approach can be divided into two parts. One is test cases generation, and the other is test cases execution. In the part of test cases generation, we used backward data flow analysis to expand all executable paths in the target program, and generated test cases by solving constraints which are collected when performing the analysis. Further, we took these test cases as an evidence of the vulnerability and applied the automatically Web-based testing to confirm the vulnerability is a true positive.

6.1 Contributions

We summarize our contributions as follows.

- Combine static analysis and dynamic testing

As we mentioned above, quite a few analysis tools can detect vulnerabilities in Web applications, but none of them integrate analysis and testing process. Moreover, because of dynamic features of web applications, in the process of performing static analysis for Web application, it has to apply abstraction and over-approximation techniques which might introduce false positives to the analysis results. As to applying dynamic testing to detect vulnerabilities, it may encounter path coverage problems. So, our approach combined two ways to make use of both advantages.

- Propose an automatic testing method on a Web-based architecture
  Traditionally, automatical testing needs to install tools to execute test cases. In this thesis, we proposed a Web-based approach which needs no installing tools in client-side, and it also can achieve the objective of automatic testing.

- Provide an easier way to confirm vulnerabilities automatically
  We designed different types of attack patterns to help us confirm the existence of vulnerabilities and convince the programmers that their programs have weaknesses.

6.2 Further Work

Our work may be extended in two ways

- Improve the efficiency of test cases generation by
  - sharing information of common sub paths and applying program slicing technique to remove all the code that cannot affect the outcome of the constraints. Besides, we only need to know if the target program contains an exploitable path from a source to a sink. So, if there are numerous exploitable paths from a source to sink, we can merge the same source and sink paths.

- Enhance capabilities of test cases execution by
  - dealing with more features of DOM objects and offering more ways to confirming the existence of vulnerabilities. For example, we just provide test information of steps and let users manipulate Web pages by themselves.

Bibliography

[1] Choco solver. http://www.emn.fr/z-info/choco-solver/index.html.

[2] lp_solve solver. http://lpsolve.sourceforge.net/5.5/.

[3] OWASP Top 10 - 2007 the ten most critical Web application security vulnerabilities. http://www.owasp.org/index.php/Top_10_2007.

[4] OWASP Top 10 - 2010 the ten most critical Web application security risks. http://www.owasp.org/index.php/Top_10_2010.

[5] IEEE Standard for Software and System Test Documentation. *IEEE Std 829-2008*, pages 1–118, 2008.

[6] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic Web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 261–272. ACM, 2008.

[7] 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.

[8] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box Web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, to appear.

[9] Cenzic. Web application security trends report. http://www.cenzic.com, Q3-Q4, 2009.

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

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

[12] Thanh-Binh Dao and Etsuya Shibayama. Idea: Automatic security testing for Web applications. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, pages 180–184. Springer-Verlag, 2009.

[13] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 151–162. ACM, 2007.

[14] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *Proceedings of the 2007 31st Annual IEEE International Computer Software and Applications Conference - Volume 01*, pages 87–96. IEEE Computer Society, 2007.

[15] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52. ACM, 2004.

[16] 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.

[17] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the 2009 International Symposium on Software Testing and Analysis*, pages 105–116. ACM, 2009.

[18] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and Cross-site scripting attacks. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 199–209. IEEE Computer Society, 2009.

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

[20] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, 2010.

[21] Hossain Shahriar and Mohammad Zulkernine. Automatic testing of program security vulnerabilities. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*, pages 550–555. IEEE Computer Society, 2009.

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

[23] Gary Wassermann and Zhendong Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41. ACM, 2007.

[24] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for Web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 249–260. ACM, 2008.

[25] Wikipedia. Software testing. http://en.wikipedia.org/wiki/Software_testing.

[26] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, pages 179–192. USENIX Association, 2006.

[27] Rui-Yuan Yeh. An improved static analyzer for verifying PHP Web application security. Master's thesis, National Taiwan University, 2010.

[28] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 199–209. ACM, 2009.

Appendix

A. Test Cases Execution Components

A1. A Fragment Code of JavaScript Code

```
function sendAJAXrequest(){}
function getStep(){
    sendAJAXrequest();
    xmlHttp.onreadystatechange = getStep_r;
}
function getStep_r(){
    doAction();    // manipulate DOM object
    setTimeout("getStep()", 2000);    // get next step
}
function doAction(){
    if (action == "type"){
        // insert a string into a target object
    }
    else if (action == "click"){
        // click a target object
    }
    else if (action == "confirm"){
        // confirm this test case
    }
}
getStep();    // initially invoke
```

A2. A Fragment Code of GetStep Code

```
<?php
  //get a step from a test case file
  if (file_exists(“testcase.xml”)) {
    $stepNo = $_SESSION[“testStep”];
    $stepNo++;
    $xml = simplexml_load_file(“testcase.xml”);
    $action = $xml->scenario->step[$stepNo]->action;
    $targetId = $xml->scenario->step[$stepNo]->target->id;
    $targetName = $xml->scenario->step[$stepNo]->target->name;
    $typeStr = $xml->scenario->step[$stepNo]->typingString;
    echo $action.“__”.$targetId.“__”.$targetName.“__”.$typeStr;
    $_SESSION[“testStep”] = $stepNo;
  }
  else {
    exit(‘Failed to open ’.$path.‘.’);
  }
?>
```

B. An Example of A Test Case

B1. Target Programs with the Instrumented JavaScript Code

```
//a.php
<html>
<head>
<title>Reflected XSS</title>
</head>
<body id=“aaa”>
  Enter your name:
  <form id=“form1” action=“b.php” method=“GET”>
    <input id=“txtname” type=“text” name=“txtname” size=30><br>
    <input id=“btn1” type=“submit” value=“enter”><br>
    <input id=“btn2” type=“reset” value=“reset”>
  </form>
</body>
</html>
<br/><!–instrument code –><br/>
<script src=“../../simulate.js”></script>
```

```
//b.php
<html>
<head>
<title>Reflected XSS2</title>
</head>
<?
  $name = $_GET[“txtname”];
  echo “Hi, ”;
  echo $name;
?>
</body>
</html>
<br/><!–instrument code –><br/>
<script src=“../../simulate.js”></script>
```

B2. An Actual Output of A Test Case

```
<TestCase>
  <tcId>tc1</tcId>
  <vulnerability>XSS</vulnerability>
  <scenario>
    <step>
      <id>1</id>
      <page>a.php</page>
      <action>browse</action>
      <target><id></id><name></name></target>
      <typingString></typingString>
    </step>
    <step>
      <id>2</id>
      <page>a.php</page>
      <action>type</action>
      <target><id>txtname</id><name>txtname</name></target>
      <typingString>&lt;script&gt;document.title = “xss”&lt;/script&gt;</typingString>
    </step>
    <step>
      <id>3</id>
      <page>a.php</page>
      <action>click</action>
      <target><id>btn1</id><name>btn1</name></target>
      <typingString></typingString>
    </step>
    <step>
      <id>4</id>
      <page>b.php</page>
      <action>confirm</action>
      <target><id></id><name></name></target>
      <typingString></typingString>
    </step>
  <expectedValue>
    <type>title</type>
    <info>xss</info>
  </expectedValue>
</TestCase>
```

B3. An Actual Output of A Test Case Result

```
<TestCaseResult>
  <tcId>tc1</tcId>
  <project>
    <projectName>myProject1</projectName>
    <projectVersion>v1</projectVersion>
  </project>
  <result>Passed</result>
</TestCaseResult>
```

C. Test Data in the Evaluation Section

C1. Ex1

```
<?php
  $name = $_GET['name'];
  echo "Welcome $name";
?>
```

C2. Ex2(branch)

```
<?php
  $group = $_GET['group'];
  $name = $_GET['name'];
  if ($group == 1)
  {
    echo "Hello! $name";
  }
  else
  {
    echo "str";
  }
?>
```

C3. Ex3(falsepositive)

```
<?php
  $group = $_GET['group'];
  $name = $_GET['name'];
  if ($name != "admin")
  {
    echo "Hello! $name";
  }
  else
  {
    if ($group == 99)
    {
      echo $name;
    }
  }
?>
```

C4. Ex4(multibranches)

```
<?php
  $id = $_GET['id'];
  $name = $_GET['name'];
  if ($id != "123")
  {
    echo $id;
  }
  else
  {
    echo $id;
  }
  if ($name == "abc")
  {
    echo $name;
  }
  else
  {
    echo $name;
  }
?>
```

C5. Ex5(different paths)

```
<?php
  $id = $_GET['id'];
  $group = $_GET['group'];
  $name = $_GET['name'];
  if ($group == 1)
  {
    $id = "guest";
  }
  if ($name == $id)
  {
    if ($group == 99)
    {
      echo $id;
    }
    echo "Hello! $name";
  }
?>
```