Next Generation Black-Box Web Application Vulnerability Analysis Framework

by

Tejas Sunil Khairnar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2017 by the
Graduate Supervisory Committee:

Adam Doupé, Chair
Gail-Joon Ahn
Ziming Zhao

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Web applications are an incredibly important aspect of our modern lives. Organizations and developers use automated vulnerability analysis tools, also known as scanners, to automatically find vulnerabilities in their web applications during development. Scanners have traditionally fallen into two types of approaches: black-box and white-box. In the black-box approaches the scanner does not have access to the source code of the web application whereas a white-box approach has access to the source code. Today's state-of-the-art black-box vulnerability scanners employ various methods to fuzz and detect vulnerabilities in a web application. However, these scanners attempt to fuzz the web application with a number of known payloads and to try to trigger a vulnerability. This technique is simple but does not understand the web application that it is testing. This thesis, presents a new approach to vulnerability analysis. The vulnerability analysis module presented uses a novel approach of Inductive Reverse Engineering (IRE) to understand and model the web application. IRE first attempts to understand the behavior of the web application by giving certain number of input/output pairs to the web application. Then, the IRE module hypothesizes a set of programs (in a limited language specific to web applications, called AWL) that satisfy the input/output pairs. These hypotheses takes the form of a directed acyclic graph (DAG). AWL vulnerability analysis module can then attempt to detect vulnerabilities in this DAG. Further, it generates the payload based on the DAG, and therefore this payload will be a precise payload to trigger the potential vulnerability (based on our understanding of the program). It then tests this potential vulnerability using the generated payload on the actual web application, and creates a verification procedure to see if the potential vulnerability is actually vulnerable, based on the web application's response.

ACKNOWLEDGEMENTS

When I started researching for this project I thought this technique is not going to work. But slowly when I started to understand the research direction I became very clear about how this technique is going to work and how am I going to make it work. Now I believe that "Nothing is Impossible".

Not a part in this thesis would have been possible without the help and guidance of my thesis advisor, the best professor at ASU and committee chair - Dr. Adam Doupé. This project was entirely his idea and recently he received the NSF CAREER award for this project. Congratulations Adam. I would like to thank Adam for teaching me everything during my entire coursework of Masters.

I would like to thank Dr. Gail-Joon Ahn, for being part of the committee, for all his help and valuable inputs.

I would also like to thanks Dr. Ziming Zhao for being part of my committee and helping me complete my thesis with his suggestions and feedback.

Apart from this I would like to thank my dear friend Kevin Liao, who worked on this project with me. He was always available for any kind of help I required to understand the AWL language. He also help me in completing this document with his valuable inputs.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Web applications form the most important part of the security ecosystem. Today, almost every business has web application as their first face to the world. Thus, security of these web application is of paramount importance. To secure their web application, most companies hire security professionals (white-hat hackers) to penetration test their web applications to identify vulnerabilities to patch them accordingly. While this process is effective at finding vulnerabilities in web applications, it is incredibly costly, and, more fundamentally, it does not scale: For instance, these manual tests cannot be performed every time the code changes, thus leaving a window for attackers to find and exploit vulnerabilities.

Black-box vulnerability scanners are tools which attempt to automate the penetration testing process. By treating the web application as a black-box (no knowledge of the source code of the application), these tools automatically discover previously unknown vulnerabilities. In essence, these tools aim to empower the average developer with little to no security expertise to discover vulnerabilities in her web application before an attacker, thus improving the entire security ecosystem.

Our research aims to create a novel black-box vulnerability analysis framework, called Next Generation Black-Box Vulnerability Analysis Framework, that is able to find previously unknown vulnerabilities in a web application. We use a recently proposed novel technique called Inductive Reverse Engineering (IRE) [34], which leverages recent advances in program synthesis and inductive programming communities to reverse engineer a model of the web application's functionality in a black-box manner. This model of the web application can be thought of as *abstracted source code*.

This thesis work presents a framework to automatically identify potential XSS vulnerabilities in the *abstracted source code* using static program analysis techniques. Once a potential vulnerability is identified in the *abstracted source code*, we verify the existence of the vulnerability on the real web application. In this way we are able to automatically find XSS vulnerability in web applications using IRE.

In summary, this thesis make the following contributions:

- A novel approach of using static program analysis and constraint solving techniques to identify and verify vulnerabilities in Inductive Reverse Engineered programs (*abstracted source code*).
- We implement a prototype of next generation black box vulnerability analysis tool to detect XSS vulnerability in web applications.
- We demo the feasibility of our approach evaluating our prototype on broken web applications available publicly.

Chapter 2

BACKGROUND

This chapter talks about the background of the problem. It describes the novel concept of Inductive Reverse Engineering, a brief history about Cross-Site Scripting vulnerability, static program analysis techniques to find this vulnerability in *abstracted source code* and constraint solvers.

2.1 Inductive Reverse Engineering

The basic idea of Inductive Programming is simple and was explained in Ameral's original paper on the topic [2]: Given a set of required input/output pairs, create a program that generates the correct output given the input. This problem is not as easy as it sounds. Even if you have an `if` statement that maps each given input to the output, it does not fulfill the true goal of having a minimal program do it. One solution to this is enumerate through all possible programs, in an order from least complicated to most complicated, and the first program that solves the input/output constraints is the correct program. This idea of Inductive Programming was further developed in a project started by Gulwani [19, 20] which became a feature first introduced in Excel 2013 called Flash Fill, which detects when a user is manually performing a data transformation on the previous columns' data. Using only a single input/output example, Flash Fill can automatically discover a program that satisfies the input/output example, and then show this transformation applied to the rest of the data. The user can then choose to either accept the transformation or enter more input/output examples, so that the transformation can be refined. This work has been extended to support table

lookups [22] and semantic string transformations [53]. In our work, this approach has been used to Inductively reverse engineer the source code of a web application using the input/output pairs. This limits the universe of possible programs to learn. The way these techniques of Inductive Programming work is by defining a domain-specific language of programs which we describe in Chapter 4 of this thesis. Next we use version space algebra [42, 33] to use a data structure to represent the set of all possible programs that can generate a given output from a given input. Further this technique gives us a graph of each input/output pair. Then we partition the graphs so that they apply to only the inputs in a given partition and no other partitions. These boolean formulas are also part of the language, and their expressiveness describes the possible set of boolean conditions. The final result is a series of boolean conditions and a version space for each boolean condition that corresponds to the given input/output set. Using these techniques on web application, from the input/ouput pairs and the boolean conditions dervied from that we are able to generate an *abstracted source code* of the web application which satisfies the input/output pairs and is in our domain-specific language.

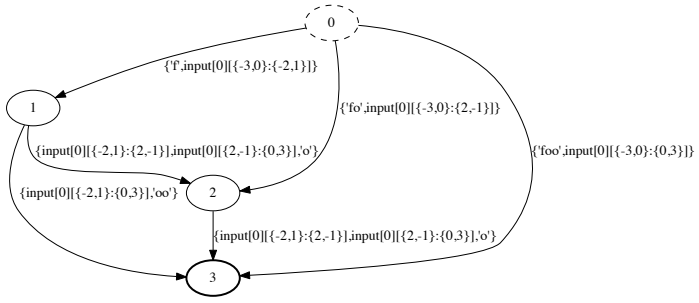

Figure 2.1: Version Space Represented as a Graph for All Possible Programs that Output `"foo"` Given the Input of `"foo"`.

2.2 Cross Site Scripting

Cross Site Scripting (XSS) is a vulnerability in which an attacker is able control the JavaScript that her victim executes, in the context of a web application. Specifically, a XSS vulnerability allows an attacker to circumvent the browser's Same-Origin Policy, and an XSS exploit allows an attacker to steal the user's cookies, phish the user's credentials, or perform actions on the web application on behalf of the victim. The Open Web Application Security Project (OWASP) ranks XSS as the third most important vulnerability on the web today [47]. *Server-side* means that the XSS vulnerability occurs because of a vulnerability in the server-side code. There are two types of XSS attackes defined:

1. Reflected XSS — in this type of XSS attack the injected script, by the attacker, is reflected from the web server, such as in an error message, search results, or any other response that includes some or all of the user input sent to server as part of request.

2. Stored XSS — in this type of XSS attack the injected script, by the attacker, is stored on the server permanently, such as in the database. This injected script is executed the next time any user navigates to the page.

Ww have targeted to detect reflected XSS vulnerability in web applications using our AWL Vulnerability Analysis module. To find stored XSS vulnerability we have to consider the state of the web application to understand the application's behavior. This is part of our future work.

2.3 Static Program Analysis

As the name goes, static program analysis analyzes the program without executing it. These techniques are generally used to find bugs or to analyze a program for syntax

issues, one example is a compiler. However, there are various other areas where these techniques can be used efficiently such as finding security vulnerabilities. Leveraging to the large body of work done on static program analysis techniques [5, 39, 26, 36, 24, 12, 64, 58, 63, 61, 45, 50, 35, 57] we proceed to build our vulnerability analysis framework. These techniques build a control flow graph of the program which explains how the program flows through various functions for different inputs. Further, much research [28, 9, 37] has already been done in using these control flow graphs, to find vulnerabilities in web applications.

2.4 Constraint Solvers

Constraint solving plays an important role in program analysis for the purpose of test generation for coverage [51], bug finding [4, 65], and vulnerability detection [9, 30]. The reason for this is solver-based analysis tools enable more precise analysis with the ability to generate bug-revealing inputs. Z3str2 [66] is a constraint solver for the quantifier-free theory of string equations, the regular-expression membership predicates, and linear arithmetic over the length functions. Z3str2 is implemented as a string theory plug-in of the powerful Z3 SMT solver [13]. Thus, ad-hoc sanitization routines (which will manifest themselves as either string transformations or boolean constraints) can be analyzed.

In our work, we are using Z3-str2 [66], to check if XSS exploits can reach a sensitive output given specific boolean constraints. The constraint solver also helps us to generate more precise payloads depending upon the substring conditions in the output for any given input.

Chapter 3

SYSTEM DESIGN

In this chapter we explain the System Design of our tool. Our tool mainly consists of three modules.

1. Crawler
2. Inductive Reverse Engineering Module
3. Abstracted Web Language Static Analysis Module

The design of these modules is represented in the Figure 3.1. The Crawler module crawls the web application with various input-output pairs. The IRE module utilizes the technique of Inductive Reverse Engineering to generate the *abstracted source code* of the web application. The AWL module further uses this abstracted web language program generated from the IRE module [34] to perform vulnerability analysis.

3.1 System Architecture

This section discusses the architecture of the Abstracted Web Language(AWL) vulnerability analysis module of the Black-Box tool. The AWL vulnerability analysis module is the second major part of the Next Generation Automated Black-Box Vulnerability Analysis Framework after the IRE module. This module is responsible to find vulnerabilities in the abstracted source code of web application generated by the IRE module. The vulnerability analysis architecture Figure 3.2 shows various modules responsible for vulnerability identification in the reverse engineered source code.

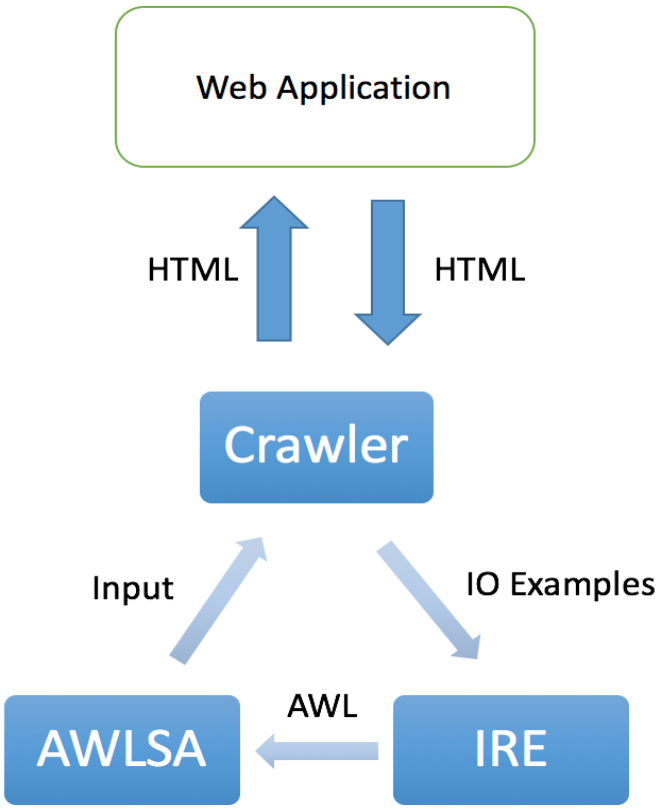

Figure 3.1: System Design

3.2 System Components

This section explains each system component of the prototype's system architecture.

3.2.1 IRE Module

The IRE module is the one which has the logic for reverse engineering the source code of web application. The logic of the IRE module is based on the concept of Inductive Programming, where you try to generate all possible programs for a given input/output pair. This module works by fuzzing the web application with input/out-

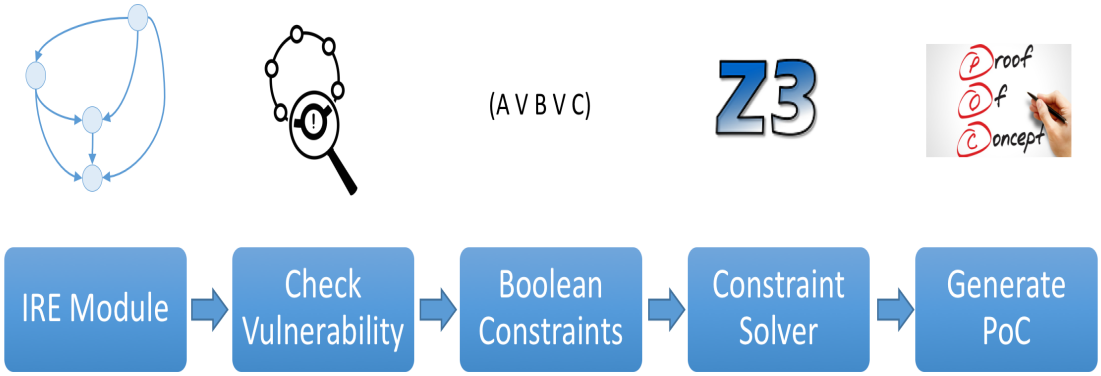

Figure 3.2: System architecture

put pairs and try to generate all possible programs that can represent that particular application. These input/output pairs are stored in the form of directed acyclic graph's by the IRE module. As part of the fuzzer the input strings are randomly generated of small letters*[a-z]*, capital letters*[A-Z]*, Digits*[0-9]* and special characters. The fuzzer in the IRE module plays an important role because, the more number of string variants we use, the more we can learn about the application and reverse engineer its functionality.

3.2.2 Check Vulnerability

This module takes the directed acyclic graph's generated by the IRE module as input and finds vulnerability over the same. Our algorithm 1 traverses over all the DAG's generated by the IRE module. This module flags any theory as vulnerable if it is of the type Sub String of the input given by the user. It does not flag theories which contains constant string in it. This is because we believe that the output generated by the inputs does not vary in the HTML or CSS content of the response.

Algorithm 1 Check vulnerable

```
function CHECK_VULNERABLE(b, η̃, η^s, η^t )
    while η^s ≠ η^t do
        if theory == ConstStr() : then return False
        if theory == SubStr() : then
            vuln_theory = theory
            end_indicies = theory.indicies
            payload = GENERATE_PAYLOAD(b, end_indices)
            if payload == None then return False
            if payload ≠ None then
                b̂ = POC_GENERATOR(payload)
                POC_VALIDATOR(b,b̂)
```

The input to our vulnerability module is the Abstracted Web Language synthesized code generated by the IRE module. This input is the DAG we refer to in Figure 7.2. Because we have to find vulnerabilities over these dags, we take these DAGs one at a time as input to our algorithm. Now, we traverse over this graph completely to check if there are any theories along the edges which might be vulnerable. If the user is entering any input and that input is somehow reflected in any HTML input tag in the response then we call it as substring theory. Now, to make our scanner to find as many vulnerabilities as it can we assume each SubStr theory as vulnerable and try to generate XSS payload for any such theory. The next step in our algorithm is to find if there is an PHP SubStr manipulation done on the server side of user input. This means, we are trying to see if the server is chopping of the user input and then displaying it or adding some random characters in the start. These cases are correctly captured by our IRE module and presented on the edges of the DAG. All

Algorithm 2 Payload Generation

```
function GENERATE_PAYLOAD(B, end_indices)
    if end_indices == None then
        Z3_SOLVER(B, None)
    if end_indices < 0 then
        string = GEN_Z3CONSTRAINT(end_indices)
        Z3_SOLVER(B, string)
    if end_indices > 0 then
        string = GEN_Z3CONSTRAINT(end_indices)
        Z3_SOLVER(B, string)

function GEN_Z3CONSTRAINT(end_indices)
    generate z3_constraint according to value of end_indices
    return newConstraint

function Z3_SOLVER(z3_string, newConstraint)
    solve(z3_string, newConstraint)
    if SAT then
        return Precise Payload String
    if UNSAT then
        return Payload cannot be generated
```

Algorithm 3 Proof-of-Concept generator

function PoC_GENERATOR(PAYLOAD)

Fuzz the application with the payload to generate DAG **return** $\hat{b}$

function PoC_VALIDATOR(b,$\hat{b}$)

dag3 = b.intersect($\hat{b}$)

if $dag3 == None$ **then return** Cannot generate PoC.

if $dag3 \neq None$ **then return** Vulnerable. PoC generated.

these DAG's we know are partitioned using boolean classifiers which we mentioned earlier. We use these boolean classifiers to generate Z3 constraints. In other words, we convert these boolean classifier into various strings which can be given as input to Z3 Constraint Solver. Z3 is a very powerful SMT solver which can solve constraints and also generate strings which Satisfy those constraints. Our XSS payload which satisfies all the boolean classifiers is generated by Z3. This payload generated by Z3 will be precise according to the boolean constraints. Hence we are sure at this point of algorithm that this payload will satisfy all the required input conditions, that the IRE module inferred from, of the web application. Now we again fuzz the web application to generate input/output pairs for input payload. These steps are done in order to generate proof-of-concept that the precise payload given by Z3 will actually trigger a XSS vulnerability in the web application. The poc_generator function fuzzes the web application with input as payload and generated the required DAG. The poc_checker function will now take the complete DAG learnt by the IRE module and the DAG generate by poc_generator function and will try to intersect them. If an intersection is found then we check if the same theory is present on the intersected DAG as we observed in the vulnerable DAG. If that is the case then our scanner outputs that the XSS payload will definitely trigger vulnerability in the web application.

3.2.3 Boolean Constraints

The IRE module in our prototype generates the *abstracted source code* on the basis of the domain specific language designed by the IRE module. IRE module uses various boolean classifiers in order to reverse engineer the source code. These boolean classifiers are dependent on our language and currently consists of various regular expressions defined in our language. The AWL vulnerability analysis module further uses these boolean classifier to generate Z3-constraints.

3.2.4 Constraint Solver

The idea behind using Z3-str2 constraint solver is to generate a precise payload that can satisfy as input to the web application. Using the IRE module we already know the behavior of the web application. This behavior reflects exactly if the web application is reflecting the same input string to the user as is, performing any sub-string operation like chopping of certain number of characters at the start of user input or chopping of certain number of characters at the end of user input. Due to this behavior it is very important to generate precise payload which will get executed as reflected XSS payload on the web browser. Even a single character mismatch, for instance 't>' of the XSS payload '<script>alert('XSS')</script>', gets chopped off by the server side logic then our payload may not get executed and would flag as not vulnerable. Therefore, in this case we use Z3 to solve this constraint too for us. It generates precise string of the required length which will satisfy all these constraints. This example, explains very clearly why we make use of Z3 constraint solver in our tool. Algorithm 2 describes how we convert the boolean constraints into Z3-str2 constraints.

3.2.5 PoC & Payload Generator

The payload generator module generates various XSS payload strings which are to be tested with the constraint solver. These XSS payloads are taken from the OWASP XSS cheat sheet [48]. The algorithm to generate the payload is described in Algorithm 2. Section 10 of this thesis document explains how this payload generation module can be extended to fuzz of all different types of XSS payload out there in the wild. The PoC generator module generates a proof-of-concept which is satisfying the constraints and is actually triggering the XSS vulnerability in the application and also test the payload against the web application to see if it is actually vulnerable. This module again uses the IRE module to generate the graph of input output pairs for the web application while the difference is this time it uses XSS payload string as input. Further, this module confirms the vulnerability by intersecting the generated graph with the previously generated graph from random input/output pairs. According to our theories if this intersection happens, then the XSS payload should trigger the vulnerability in the application. If we can see the same string in both the graph being reflected we can be sure that the web application is vulnerable and the payload used can trigger the vulnerability. The PoC checker checks if the same string is present as the longest string on the intersected graph as well as the graph generated during initial crawling. The Algorithm 3 describes the way we generate proof-of-concept for the precise payload generated.

Chapter 4

ABSTRACTED WEB LANGUAGE

This chapter introduces you to the Abstracted web language. The goal of AWL is to be a domain-specific language for web applications. This is the most important part of our research. Our input-output analysis depends on this language. The assertion we claim that, using the input-output pairs we can generate all possible programs which can generate you the same input-output pairs, depends on this language. The more expressive we make our language the more behavior we would learn of the web application. The concept, syntax and semantics of AWL was inherited from IRE module developed by Liao [34].

4.1 Syntax of Language for describing AWL Programs

In this section we explain the syntax of abstracted web language. This language is defined by us in order to understand the behavior of the web application. According to the syntax of AWL language we can represent all the programs which can generate the input-output pair. These programs can be represented as a big switch statement of programs to generate the given input-output pair. We represent all the boolean conditions by b_1 and $\tilde{x_1}$, ... $\tilde{x_n}$ represents all the directed acyclic graphs generate for given input-output pair. These together represent a big set of synthesized programs in AWL. Now, the DAG is a concatenation of all the html output. Thus output can further described as made up of HTML start tag concatenated with HTML data concatenated with HTML end tag and HTML self-closing tags. We also represent HTML attributes by in our syntax of AWL language. Each DAG is made up of nodes, start node, end node, edges and W, the web application itself which takes in the path,

method, and query parameters. These atomic expression are the main language we have defined. It consists of

- SubStr of input and position of strings
- ConstStr which represent string which is constant for all the given inputs
- TwoInputMath which perform Math operations on given input
- ConstMath which performs Math operations on given input and a constant

4.2 Semantics of Language for describing sets of AWL Programs

In this section we describe the semantics of our AWL language for describing sets of AWL programs. We explain them as below:
T represents all the HTML token in the output we get for the given input. Data is the HTML data which is inside the generic HTML tags. For example a <form> tag with input type and value. EndTag are the ending tags in every HTML, for instance />. In the recent developments in HTML5, we observe self-closing tags. For instance in the closing tag />, the / is completely optional in HTML5. Hence in our language semantics we consider that too. $(\tilde{A_1}, \tilde{e_1})$ represent the HTML attributes. For example, this set would contain all the attributes in a form tag.

Chapter 5

IMPLEMENTATION

In this chapter we explain how we implemented the prototype in Python programming language. To implement the scanner we had a system setup as follows:

5.1 System Configuration

We used one machine to implement the tool. The details about this machine are as follows:

- MacBook Pro

```
CPU: Intel Core i7 @ 3.1GHz
Cache Size: 4M
No. of Cores: 2
Total Memory: 8GB
Disk Space: 256GB
```

5.2 Platforms and Software

Our scanner is built to run on any machine which has Python and certain packages installed on it. To perform the experiments we ran our scanner on test-applications as well as published vulnerable applications. These applications were hosted on our own machine to complete the experiments. The platforms and softwares used to build this tool are enumerated in Table 5.1:

Operating system	macOS Sierra
Web Server	Apache - 2.4.23
Constraint-Solver	Z3-str2

Table 5.1: Platforms and software used for our tool.

5.3 Languages

We used Python 2 to build the system. The reasons for choosing this language are: it has numerous libraries for fuzzing an application and it is easy to use the data structures in this language. The major libraries which were used for our tool are listed in Table 5.2:

Library	**Functionality**
Requests	HTTP Request Generation
Beautiful Soup	HTML Parsing
Subprocess	Spawn new process

Table 5.2: Libraries that we used and their functions.

5.4 OWASP Broken Application

In order to test our scanner we had to find some publicly available web applications which are designed to be vulnerable. The best resource to find such applications is the OWASP website. There are certain limitations to our vulnerability scanner which are discussed in the section8.3. Considering those we reversed engineered the web applications to serve our needs viz., stateless web application. Table 5.3 enumerates the web applications we used in order to test our tool. We choose these web applications because they are released as broken web application on OWASP web site [47]. This

Sr No	Web Application	Source URL
1	test-vul	https://github.com/tejas619/tejas-test-vul
2	WackoPicko	https://github.com/adamdoupe/WackoPicko
3	xvwa	https://github.com/s4n7h0/xvwa
4	btslab	https://github.com/CSPF-Founder/btslab

Table 5.3: Web applications used for our experiments

helps us in proving our results because the payload used to exploit these websites are also released by the authors of these broken web application.

5.5 Problems Faced

This section describes some issues we faced while implementing the design and idea of our tool.

- Support of Regular expressions in Z3

  After going through the documentation of Z3-SMT solver, we found that it would support only limited use of regular expressions in constraint solving. We were not able to generate regular expression defined in our language using Z3. Hence, we decided to look for any supporting library built on top of Z3 which would help us defining and solving regular expressions in our language. After a good amount of research we found that there is certain amount work done over supporting regular expressions in constraint solving. Amongst the related tools were

  - Hampi [30] - solver for string constraints
  - STP [17] - constraint solver for theory of quantifier-free-bit-vectors

- Z3-str2 [66] - constraint solver for the quantifier-free theory of string equations, the regular-expression membership predicates, and linear arithmetic over the length functions

Looking at our requirements and the work done in above tools, we decided to use Z3-str2.

- Detect False Positives

  Automated vulnerability analysis tool are a creation of a human who has knowledge of penetration testing a web application. Hence these tools can be intelligent only to a certain extent due to which often times they generate false positives. In our case we had to find a way in which we can detect false positives given by our tool. Hence we decided to implement the PoC checker module which actually checks if the XSS payload give by payload generator triggers the vulnerability in the web application.

Chapter 6

EVALUATION

This chapter describes our evaluation of the system, with the experiments conducted, results and screen captures of the output of our scanner.

6.1 Results

This section discusses our results after running our scanner on various test applications mentioned in the Section 5.4.

6.2 Experiments on Vulnerable application

We conducted our experiments on the latest versions of 3 open source PHP vulnerable applications: xvwa, btslab and wackopicko. Table 6.1 summarizes our findings for these 3 applications. The table shows that we were successfully able to generate precise payload, generate proof of concept and flag the application vulnerable to reflected XSS attack. We further explain our results for WackoPicko application, for which our application was not able to generate results, in the Section 7.

Sr No	**Web Application**	**Precise Payload**	**Proof of Concept**	**Vulnerable**
1	test-vul	✓	✓	✓
2	WackoPicko	×	×	×
3	xvwa	✓	✓	✓
4	btslab	✓	✓	✓

Table 6.1: Experiment results

6.3 Implementation Output Results

These are some of the screenshots showing how our scanner asks the user for web application URL as input to scan, perform fuzzing operation and output results to the user. Figure 6.1 shows how our scanner accepts the URL of the web application to be scanned as input parameter. Once given in the right format the Crawler component of the tool fetches the parameters on the page and gives you a list of query parameters which it is going to fuzz for outputs.

Figure 6.1: Screen Capture of our Tool in action - 1

Figure 6.2 shows the boolean classifiers generated by the IRE module for the given web application. This screenshot output is for the **test-vul** application we describe about in the running example chapter. In the Figure we can also see Z3-str3 constraints generated for the boolean classifier. These are further fed to Z3-str2 which generates the precise payload.

First line of Figure 6.3 tells you how the AWLSA module confirms using constraint solver that the payload generated satisfies the boolean constraints. Further, we can see our AWLSA module again fuzzes the web application with payload as input and

```
[[<class '__main__.Or'>, <class '__main__.And'>, <class '__main__.NullMatch'>], [<class '__main__.Or'>, <class '__main__.And'>, <class '__main__.NotMatch'>],
 [<class '__main__.Or'>, <class '__main__.And'>, <class '__main__.NullMatch'>], [<class '__main__.Or'>, <class '__main__.And'>, <class '__main__.NotMatch'>]]
[[input[1] is null, input[1] is null, input[1] is null], [!(input[1] is null), !(input[1] is null), !(input[1] is null)], [input[1] is null, input[1] is null
, input[1] is null], [!(input[1] is null), !(input[1] is null), !(input[1] is null)]]
Pruning 1 unreachable nodes
Theory is SubStr. It is probably vulnerable. Generate an XSS payload for this particular theory and boolean classifier
these are the substring values Input_Index 1 Initial Indicies 0 End Indicies -2
******************
```

Figure 6.2: Screen Capture of our Tool in action - 2

generates required input-output pairs. Then it moves to the Proof of concept logic and outputs whether proof of concept was successfully generated or not.

```
* v-ok
************************
>> SAT
------------------------
x : string -> "<script>alert('XSS');</script>ak"
************************
>> etime(s) = 0.056412


Input payload <script>alert('XSS');</script>ak satisfies
We are in the poc generator function
I[0]: W(/page2.php, POST, {'password': "<script>alert('XSS');</script>ak", 'userid': "<script>alert('XSS');</script>ak"})
O[0]: <html>
<body>
Welcome <script>alert('XSS');</script><br>This is your Welcome page!</body>
</html>
```

Figure 6.3: Screen Capture of our Tool in action - 3

Chapter 7

CASE STUDY

In this section we describe a case study of our scanner over our test application. To explain the functioning in depth we have to first dive into the test application which we have built. We call this test application as **test-vul**.

7.1 Test Application

Our test application is built in PHP and is one of the most simplest web applications. This application is a stateless application, which means it does not maintain any kind of user state in it. It just asks the user for input and prints the input as output in the HTML page.

In the Listing 7.1 we can see that this simple web application only *echo's* the input of the user as a substring stripping of the last two characters from the end. The application checks whether the second text box is not empty. So if the user submits the form with random strings in both the input text boxes, the page-2 of the web application will be the response from the application.

7.2 IRE Module

The IRE module when ran on this web application will fuzz both the user inputs with strings which fall in our abstracted web language. This gives the IRE module an idea about how to generate the synthesized PHP code using the input/output pairs. Considering our language in mind our fuzzer generates the following strings at the input text boxes:

```
<html>
<head>
<title>Login page</title>
</head>
<body>
<h1>Simple Login Page</h1>
<form action="page2.php" method="POST">
Username<input type="text" name="userid">
Password<input type="password" name="password">
<input type="submit" value="Login">
</form>
</body>
</html>
```

Listing 7.1: Example Simple PHP Web Application - Page 1

```
<html>
<body>
<?php
if(!empty($_POST["password"])) {
$res = substr($_POST['userid'], 0, -2);
echo 'Welcome '. $res."<br>";
echo 'This is your Welcome page!';
} else{
echo 'Sorry 404 not found';
}
?>
</body>
</html>
```

Listing 7.2: Example Simple PHP Web Application - Page 2

```
"$\'@!", "/(](-\&`)\"=", "-^(\&", "lunch", "concede", "automobiles", "
    exulted", "Afghanistan"
```

The IRE module then stores all the input/output pairs in the form of a Directed Acyclic Graph, we call it a DAG. The structure of this DAG is similar to a normal graph with nodes and edges. The edges in these DAG's are our theories which we define in our language. These theories are:

- ConstStr - Strings like plain HTML, CSS which are generic to every server response.
- SubStr - Substring of the user input which can be represented in different ways and change according to the applications code.
- TwoInputMath - output is the result of a math operation on two inputs.
- ConstMath - output is the result of math operation with an input and a constant.

Now, our IRE module classifies the input-output pairs based on various boolean constraints defined in our language. These boolean constraints are part of our language which decides whether particular input/output pair should fall into which category. For instance, we have a boolean constraint "/$\hat{}$[a-zA-Z0-9]+$/ = input[0]" which states that the input[0] entered by the fuzzer matches the regular expression which captures all letters and digits. This tells the IRE module to generate separate DAG's for each boolean classifier accordingly. The boolean constraints generated according to our language of the test-vul application are shown in Figure 7.1. These boolean classifiers are a set of Or boolean conditions.

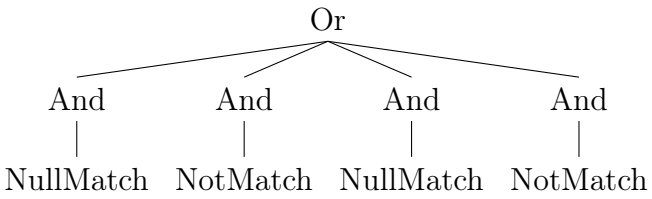

Figure 7.1: Boolean Constraints

For the sample PHP application the complete DAG's generated by the IRE module can be represented as in Figure 7.2.

7.3 Result of Vulnerability Analysis

Here we show how exactly the intersected graph generated by the poc_checker module exactly looks like. This graph is the intersection of all the edges in the complete version space graph generated by the IRE module and the version space graph generated by the vulnerability module with XSS payload as input. We can see that there is an edge present with the same theory on the intersected graph as there is on the complete version space graph. This proves our theory of graph intersection and also creates a proof of concept about the payload we used to attack on the web application.

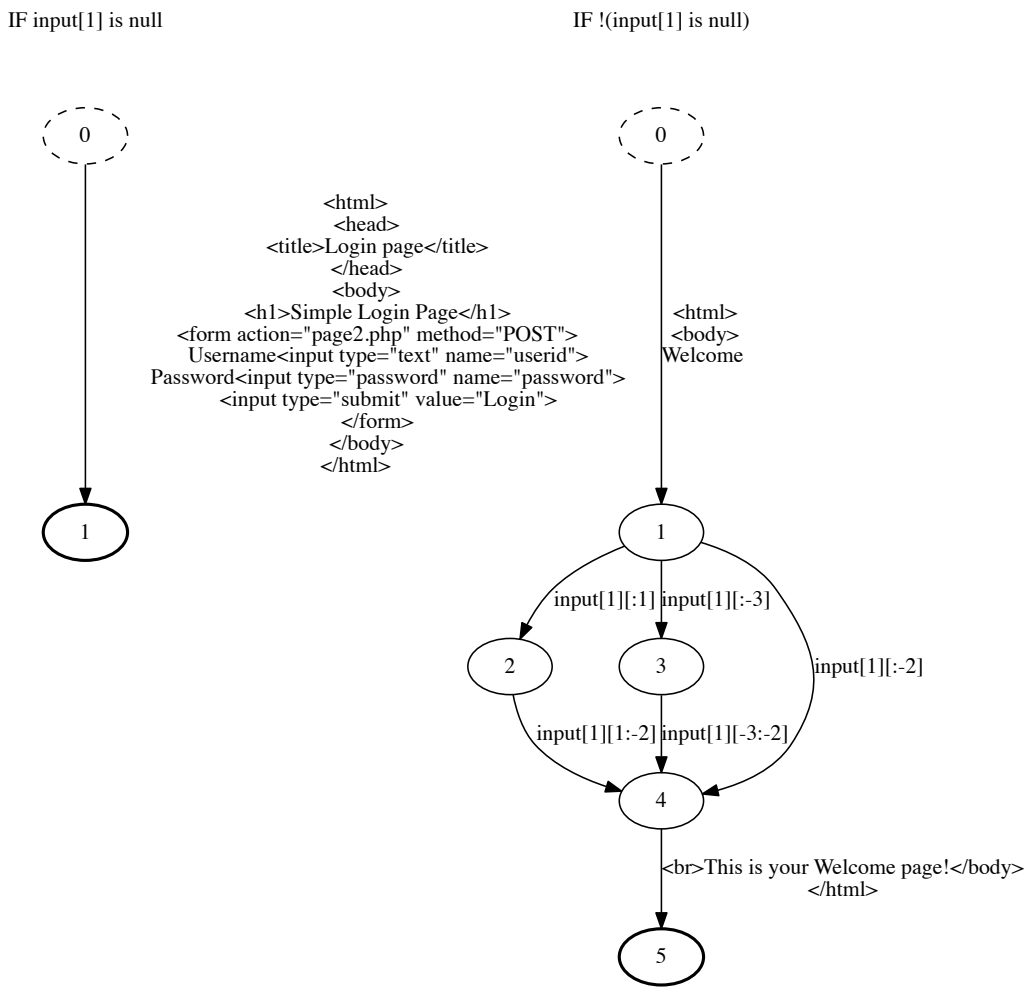

Figure 7.2: Version Space Graphs for Sample PHP Application 7.1.

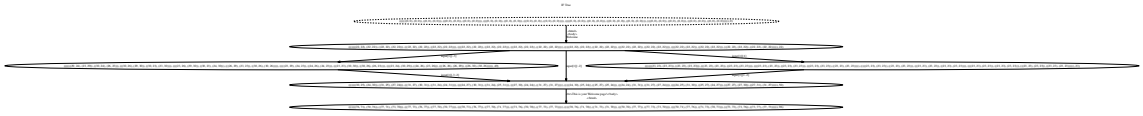

Figure 7.3: Version Space Graphs for Sample PHP Application Generated by AWLSA module.

Chapter 8

DISCUSSION

In this section we dig into the results of the scanner and discuss how these results were obtained and what are the certain limitations of our scanner.

8.1 Lessons Learned

Our results show that we can definitely scale Inductive Reverse Engineering technique for reversing source code of web applications. The more we extend our AWL language, the more we learn about the web applications. Static program analysis technique has been already been used in the security community for vulnerability analysis. Using this already proven technique we moved ahead with fusing this with lesser used constraint solving technique.

This was an implementation challenge for us because using constraint solvers to generate precise payloads required us to convert our boolean constraints into constraints which Z3-str2 will understand. Hence we converted all the boolean constraints generated by IRE module into Z3-str2 constraints and then solved them.

Z3 constraint solver does not support for regular expression and hence finding work done above Z3 which supports regular expressions was also a lesson learned. There were several other similar SMT solvers, but we chose Z3-str2 because of its efficiency.

8.2 Vulnerable Applications

test-vul is a stateless web application is created for testing purposes. This application is a simple PHP login form with two input boxes and a submit button. The source code of this PHP application on the server side does not have any kind of

input sanitization routines in place. The application simply 'echo's' the user input with some static HTML content. Because this application did not contain any kind of input sanitization or any kind of string matching checks at the server side, it was very easy for our scanner to generate the synthesized PHP code and fuzz the input with the right XSS payload.

WackoPicko [16] is a website developed by Dr. Adam Doupe that contains known vulnerabilities. Our scanner being in prototype mode for IRE technique, only works on stateless applications. Hence we run our scanner only on the page in WackoPicko which is vulnerable to reflected XSS. This page contains only one search box, where user can search for pictures uploaded by other users of the application. This search box has no input sanitization and uses the query parameter directly to search for the input string. However, this application uses HTML encoding techniques, and performs HTML encoding for the & character. It converts this character into its HTML encoded form,which is &. Our scanner fails this case becuase we fuzz the inputs with special character regular expression which contains & in it. Hence when the application generates the output for these inputs, our IRE theory treats the 'amp;' after the & as a constant string and creates new edges in the directed acyclic graph for the application. Due to this we are still not able to understand the web application completely. But, we intend to modify our scanner to an extent where it will be having HTML encoding as inputs to the fuzzer.

xvwa is a badly coded web application in PHP/MySQL and is listed on the OWASP broken web application list. The page in this application is vulnerable to reflected XSS contains a form with submit button which asks user to enter a message. The code on the server side uses the GET method to receive the message. Our scanner fuzzes this vulnerable parameter with precise XSS payload and performs the PoC generation too.

btslab is another badly coded web applicaton listed on OWASP broken application list. This application is vulnerable to various vulnerabilities starting from SQL injection to file upload vulnerability. This application has a lot of HTML content and many hyperlinks. Due to this fact our IRE module takes a little bit longer to generate all the possible DAG's and perform intersection of them. Again our vulnerability finder module, fuzzes the search parameter with XSS payload and generates DAG's for that input/output pair. Even though our approaches do not scale to a good amount for this application, we were able to detect this application as vulnerable.

8.3 Limitations

Our IRE module was able to scale the inductive programming techinques to real web applications, the type of web application we are able to reverse engineer and size of the applications are the areas for improvement. Our scanner right now only works on stateless web application. In our theories we plan to consider state of the web application too. Once, we consider the state then the vulnerability module can be extended to find vulnerabilities like SQL Injection and stored XSS. Our IRE module fuzzes the web application in order to learn its behaviors with certain number of randomly generated strings. The more we increase this number the more we can learn about the behaviour of the web application. Finding a technique which will fuzz the web application with minimum number of inputs and learning most of it is another direction for research.

Chapter 9

RELATED WORK

Number of previous efforts in the area of static program analysis focus on finding vulnerabilities in PHP, JAVA applications, including tools such as Pixy [28], NoTamper [9], WAPTECH [10]. Pixy [28] employed static analysis with taint propagation to identify SQL injection and XSS vulnerabilities. Saner [6] also uses static analysis techniques to detect flaws in the input sanitization routines. WebSSARI [25] also utilizes static analysis techniques for detecting SQL injection vulnerabilities but it also proposes a techniques to insert proper input sanitization routine during runtime. These tools constructively use static program analysis to analyze program flows and find sink points where the vulnerability lie. These techniques can be further extended to find vulnerabilities over abstracted web language programs. The concept of generating abstracted web language program using the technique of Inductive Reverse Engineering is very novel and hence till now no work is done on the same. Even though, there are some approaches [14] that perform static analysis on the code to create UML diagrams of the application. Bisht et al. [9], propose an approach for automatically finding parameter tampering vulnerabilities using the idea of constraint solving. This approach automatically identifies various constraints on the server side for a web application and solve these constraints to trigger parameter tampering vulnerability. They use HAMPI [30], a solver for string constraints in order to solve the boolean constraints their tool generate. Our work uses a combination of these two profound techniques, static program analysis and constraint solving, in order to find vulnerabilities in abstracted web language programs. Also, there are numerous tools which target on finding XSS vulnerability in web application. There are mainly

tow approaches in order to do so. One is the Black-Box approach and other is the White-Box approach. Various tools like Enemy of the state [15], Secubat [29] describe their black-box approach in order to understand the state of the web application and finding vulnerabilities in the web application. Enemy of the state [15], infers the state of the web application from the outside, by observing differences in the output and generating state of the web application using the same. Our tool currently is not considering the state of the web application and hence this research can help us in inculcating the state in our tool. To our knowledge, no other research has been conducted in finding vulnerabilities over abstracted web language programs. Combining our study from above research we present a novel approach built on the baseline of static program analysis and constraint solving to automatically find XSS vulnerability over this AWL programs.

Chapter 10

FUTURE WORK

Our work is currently very precise and scales over only stateless web applications built in PHP language. In the future we plan to consider the state of the web application as an input in our IRE module. This will enable us to understand when the web application changes its behavior according to the input-output pairs summited to it. Once we have the state as one of our theories, we can extend our vulnerability analysis module to finding SQL injection vulnerabilities in web applications. The constraint solving technique is also widely used previously by researchers in order to detect parameter tampering attacks. Our scanner will also be able to cover those vulnerabilities once the milestone of considering state is achieved. Because we are working on this novel approach of Reverse Engineering the source code of the web application, there are many paths we can take in the future to refine our scanner.

Chapter 11

CONCLUSION

In this thesis we present a technique to perform vulnerability analysis over abstracted web language programs which are generated by a novel approach of Inductive Reverse Engineering. We show that how we can successfully find vulnerabilities over these programs and make our vulnerability scanner more intelligent and precise as compared to other Black-Box vulnerability scanner which fuzz the web application without knowing the server side code. We believe our approach of fusing static program analysis and constraint solving to detect vulnerabilities in abstracted web language programs is valid. Further, these techniques can be adopted by other black-box scanners to generate precise payloads.

REFERENCES

[1] Akers, S. B., "Binary Decision Diagrams", IEEE Transactions on computers (1978).

[2] Amarel, S., "On the automatic formation of a computer program which represents a theory", in "Self-Organizing Systems", (1962).

[3] Amarel, S., "Representations and Modeling in Problems of Program Formation", in "Machiene Intelligence 6", (1972).

[4] Artzi, S., A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking", IEEE Transactions on Software Engineering **36**, 4, 474–494 (2010).

[5] Balzarotti, D., M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel and G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications", (2008).

[6] Balzarotti, D., M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications", in "Security and Privacy, 2008. SP 2008. IEEE Symposium on", pp. 387–401 (IEEE, 2008).

[7] Biermann, A., "On the Inference of Turing Machines from Sample Computations", Artificial Intelligence **3** (1972).

[8] Biermann, A., R. Baum and F. Petry, "Speeding up the Synthesis of Programs from Traces", IEEE Transactions on Computers (1975).

[9] Bisht, P., T. Hinrichs, N. Skrupsky, R. Bobrowicz and V. Venkatakrishnan, "Notamper: automatic blackbox detection of parameter tampering opportunities in web applications", in "Proceedings of the 17th ACM conference on Computer and communications security", pp. 607–618 (ACM, 2010).

[10] Bisht, P., T. Hinrichs, N. Skrupsky and V. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction", in "Proceedings of the 18th ACM conference on Computer and communications security", pp. 575–586 (ACM, 2011).

[11] Bisht, P. and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks", (2008).

[12] Chaudhuri, A. and J. Foster, "Symbolic Security Analysis of Ruby-on-Rails Web Applications", (2010).

[13] De Moura, L. and N. Bjørner, "Z3: An efficient smt solver", in "International conference on Tools and Algorithms for the Construction and Analysis of Systems", pp. 337–340 (Springer, 2008).

[14] Di Lucca, G. A., A. R. Fasolino, F. Pace, P. Tramontana and U. De Carlini, "Ware: A tool for the reverse engineering of web applications", in "Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on", pp. 241–250 (IEEE, 2002).

[15] Doupé, A., L. Cavedon, C. Kruegel and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner.", in "USENIX Security Symposium", vol. 14 (2012).

[16] Doupé, A., M. Cova and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners", (2010).

[17] Ganesh, V., "Stp constraint solver", URL `https://stp.github.io/` (2017).

[18] Green, C. and D. Barstow, "On program synthesis knowledge", Artificial Intelligence (1978).

[19] Gulwani, S., "Automating string processing in spreadsheets using input-output examples", (2011).

[20] Gulwani, S., W. R. Harris and R. Singh, "Spreadsheet Data Manipulation Using Examples", Communications of the ACM **55**, 8, 97–105 (2012).

[21] Gulwani, S., J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid and B. Zorn, "Inductive programming meets the real world", Communications of the ACM **58**, 11, 90–99 (2015).

[22] Harris, W. R. and S. Gulwani, "Spreadsheet table transformations from examples", in "ACM SIGPLAN Notices", vol. 46, pp. 317–328 (ACM, 2011).

[23] Hooimeijer, P., B. Livshits, D. Molnar, P. Saxena and M. Veanes, "Fast and Precise Sanitizer Analysis with Bek", (2011).

[24] hoon An, J., A. Chaudhuri and J. Foster, "Static Typing for Ruby on Rails", (2009).

[25] Huang, Y.-W., F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection", in "Proceedings of the 13th international conference on World Wide Web", pp. 40–52 (ACM, 2004).

[26] Jensen, S. H., A. Møller and P. Thiemann, "Interprocedural Analysis with Lazy Propagation", (2011).

[27] Jim, T., N. Swamy and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies", (2007).

[28] Jovanovic, N., C. Kruegel and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities", in "Security and Privacy, 2006 IEEE Symposium on", pp. 6–pp (IEEE, 2006).

[29] Kals, S., E. Kirda, C. Kruegel and N. Jovanovic, "Secubat: a web vulnerability scanner", in "Proceedings of the 15th international conference on World Wide Web", pp. 247–256 (ACM, 2006).

[30] Kiezun, A., V. Ganesh, P. J. Guo, P. Hooimeijer and M. D. Ernst, "Hampi: a solver for string constraints", in "Proceedings of the eighteenth international symposium on Software testing and analysis", pp. 105–116 (ACM, 2009).

[31] Kitzelmann, E., "Analytical inductive functional programming", in "International Symposium on Logic-Based Program Synthesis and Transformation", pp. 87–102 (Springer, 2008).

[32] Kitzelmann, E., "Inductive programming: A survey of program synthesis techniques", in "Approaches and Applications of Inductive Programming", pp. 50–73 (Springer, 2010).

[33] Lau, T., S. A. Wolfman, P. Domingos and D. S. Weld, "Programming by demonstration using version space algebra", Machine Learning **53**, 1-2, 111–156 (2003).

[34] Liao, K., "Toward Inductive Reverse Engineering of Web Applications", in "Kevin's Thesis", (2017).

[35] Livshits, B. and S. Chong, "Towards Fully Automatic Placement of Security Sanitizers and Declassifiers", (2013).

[36] Livshits, V. B. and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis", (2005).

[37] Livshits, V. B. and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.", vol. 2013 (2005).

[38] Louw, M. T. and V. Venkatakrishnan, "BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers", (2009).

[39] Madsen, M., B. Livshits and M. Fanning, "Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries", (2013).

[40] Manna, Z. and R. Waldinger, "A deductive approach to program synthesis", ACM Transactions on Programming Languages and Systems (TOPLAS) (1980).

[41] Manna, Z. and R. J. Waldinger, "Toward automatic program synthesis", Communications of the ACM (1971).

[42] Mitchell, T. M., "Generalization as search", Artificial intelligence **18**, 2, 203–226 (1982).

[43] Muggleton, S. and L. De Raedt, "Inductive logic programming: Theory and methods", The Journal of Logic Programming (1994).

[44] Muggleton, S., L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue and A. Srinivasan, "ILP turns 20", Machine Learning (2012).

[45] Nguyen-tuong, A., S. Guarnieri, D. Greene and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting", (2005).

[46] Olsson, R., "Inductive functional programming using incremental program transformation", Artificial intelligence (1995).

[47] Open Web Application Security Project (OWASP), "OWASP Top Ten Project", `http://www.owasp.org/index.php/Top_10` (2010).

[48] OWASP, "Owasp filter evasion xss cheat sheet", `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet` (2017).

[49] Raghavan, S. and H. Garcia-Molina, "Crawling the hidden web", Tech. rep., Stanford (2000).

[50] Samuel, M., P. Saxena and D. Song, "Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers", (2011).

[51] Saxena, P., D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song, "A symbolic execution framework for javascript", in "Security and Privacy (SP), 2010 IEEE Symposium on", pp. 513–528 (IEEE, 2010).

[52] Saxena, P., D. Molnar and B. Livshits, "ScriptGard: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications", (2011).

[53] Singh, R. and S. Gulwani, "Learning semantic string transformations from examples", Proceedings of the VLDB Endowment **5**, 8, 740–751 (2012).

[54] Smith, D. R., "Top-down synthesis of divide-and-conquer algorithms", Artificial Intelligence (1985).

[55] Srivastava, S., S. Gulwani and J. S. Foster, "From program verification to program synthesis", in "ACM Sigplan Notices", vol. 45, pp. 313–326 (ACM, 2010).

[56] Stamm, S., B. Sterne and G. Markham, "Reining in the Web with Content Security Policy", (2010).

[57] Sun, F., L. Xu and Z. Su, "Static Detection of Access Control Vulnerabilities in Web Applications", (2011).

[58] Tripp, O., M. Pistoia, S. J. Fink, M. Sridharan and O. Weisman, "TAJ: Effective Taint Analysis of Web Applications", (2009).

[59] W3techs, "Usage Statistics and Market Share of PHP for Websites, February 2017", URL `http://w3techs.com/technologies/details/pl-php/all/all` (2017).

[60] Wang, R., S. Chen, X. Wang and S. Qadeer, "How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores", (2011).

[61] Wassermann, G. and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities", (2007).

[62] Weinberger, J., P. Saxena, D. Akhawe, M. Finifter, R. Shin and D. Song, "A Systematic Analysis of XSS Sanitization in Web Application Frameworks", (2011).

[63] Xie, Y. and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages", (2006).

[64] Yu, F., M. Alkhalaf and T. Bultan, "STRANGER: An Automata-based String Analysis Tool for PHP", (2010).

[65] Zheng, Y. and X. Zhang, "Static detection of resource contention problems in server-side scripts", in "Proceedings of the 34th International Conference on Software Engineering", pp. 584–594 (IEEE Press, 2012).

[66] Zheng, Y., X. Zhang and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis", in "Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering", pp. 114–124 (ACM, 2013).