A MULTI OBJECTIVE APPROACH TO DETECT AND CORRECT THE SECURITY VULNERABILITIES IN WEB APPLICATIONS

BY

SAMRAN NAVEED

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

MAY 2016

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **SAMRAN NAVEED** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**

Jan 22, 2017

Dr. Khalid Al-Jasser

Department Chairman

Dr. Salam A. Zummo

Dean of Graduate Studies

23/1/17

Date

Thesis Committee

Sajjad Mahmood

Dr. Sajjad Mahmood (Advisor)

Dr. Mahmood Khan Niazi
(Member)

Md. Rafiul Hassan

Dr. Mohammed Rafi ul Hassan
(Member)

This work is dedicated to my beloved Mother(Late) my respected Father and my lovely sisters, whose unconditional love, tireless efforts and countless prayers fostered me to achieve this important milestone of my life.

ACKNOWLEDGMENTS

In the name of Allah, the most Beneficent, the most Merciful. All praises and gratitude are to Allah. I would have never made this journey without the countless blessings of Allah. Peace and blessings upon the final Prophet Muhammad (Peace be upon him), his family, and his companions.

I am thankful to KFUPM from the bottom of my heart for granting me the opportunity to complete my Graduate Studies with world class research facilities and under the supervision of high profiled faculty.

A heartfelt gratitude and intense appreciation to my thesis advisor Dr. Sajjad Mahmood for the precious guidance and advising. Without his help and efforts, I would have never completed this thesis. He has been always the source of great encouragement and guidance. I am thankful to respected Dr. Mahmood Khan Niazi and Dr. Mohammed Rafi ul Hassan for honoring me with their precious participation as committee members of this thesis.

And finally, I am thankful to all my friends and colleagues for their support and encouragement in every aspect. Specially I would like to thank Muhammad Sajid Iqbal for being the constant source of support, encouragement and wise advices. And thanks to Muhammad Sajid Anwar for his guidance and advices related to academics. And everyone else who is or was the part of my life and contributed in my life positively.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GA	:	Genetic Algorithm
SQLI	:	SQL Injection
XSS	:	Cross Site Scripting

ABSTRACT

Full Name : Samran Naveed

Thesis Title : A Multi-objective Approach to Detect and Correct the Security Vulnerabilities in Web Applications

Major Field : Computer Science

Date of Degree : May 2016

In recent years, web applications have become an integral part of our routine life. Numerous transactions are performed online daily including, online shopping, financial transactions information sharing and communication. Similar to other computing systems, web applications are prone to security vulnerabilities. Hence, vulnerabilities that impact the security perspective of web applications should be detected and removed from their code base. To date, researchers and industrial practitioners have identified that SQL injection and cross site scripting are main security vulnerabilities for web applications. The overarching objective of our work is to detect the SQL injection and Cross Site Scripting (XSS) vulnerabilities in the source code of web applications by generating the attack scenarios.

In this research work, we have developed an automated technique based on Genetic Algorithm (GA) to detect and correct SQL Injection (SQLI) security vulnerabilities in web applications. Our approach allows the automatic SQLI attacks generation using Genetic Algorithm to detect and correct SQLI vulnerabilities by identifying the malicious queries in the source code. Furthermore, we have developed a tool to implement and empirically evaluate our adopted approach. The tool implements two primary processes: first, the preliminary static analysis of the given source code to identify the vulnerable queries and

second is the generation of SQLI attacks for the identified vulnerable queries in first step using genetic algorithm. In order to correct the detected SQLI vulnerabilities, we used simple refactoring approach. To empirically evaluate our tool, we used several open source projects and available source code to detect SQLI vulnerabilities. the tool successfully detected and generated the SQLI attacks for the given source code and refactored it to prevent from SQLI.

Furthermore, we formulated XSS vulnerabilities detection problem as optimization problem and solved it using genetic algorithm. We combined the static code analysis with genetic algorithms to identify and generate attack scenarios for XSS vulnerabilities. The purpose of static analysis is to identify the sensitive sinks in the source code. After identification of sensitive sinks, we then converted the source code into control flow tree and developed the control paths to reach the identified sensitive sinks. The control flow paths are tested using genetic algorithm by generating the XSS attack scenarios with the help of XSS database. The experimental results indicate that our approach is useful for security penetration testing.

ملخص الرسالة

الاسم الكامل: سمران نافيد محمد رفيق

عنوان الرسالة: منهج متعدد الأهداف لكشف وتصحيح الثغرات الأمنية في تطبيقات الويب

التخصص: علم الحاسب الآلي

تاريخ الدرجة العلمية: مايو 2016

في السنوات الأخيرة، أصبحت تطبيقات الويب جزءا لا يتجزأ من حياتنا اليومية. يتم تنفيذ عدد مهول من العمليات على الانترنت يوميا بما في ذلك، التسوق عبر الانترنت، والمعاملات المالية وتبادل المعلومات والاتصالات. على غرار أنظمة الحوسبة الأخرى، تعتبر تطبيقات الإنترنت عرضة لنقاط الضعف الأمنية. بالتالي، فان نقاط الضعف التي تؤثر على النواحي الأمنية لتطبيقات الويب تتطلب أن يتم الكشف عنها وإزالتها من كودها الاساسي. حتى الآن، حدد الباحثون والممارسون في المجال الصناعي ان حقن لغة الاستعلامات المهيكلة والبرمجة عبر المواقع ومراجع الكائنات المباشرة غير المؤمنة يعتبرون من الثغرات الأمنية الرئيسية لتطبيقات الويب. الهدف الأسمى من بحثنا هذا هو الكشف عن ثغرات حقن لغة الاستعلامات المهيكلة والبرمجة عبر المواقع في مصدر التعليمات البرمجية لتطبيقات الويب عن طريق توليد سيناريوهات الهجوم.

في هذا البحث، قمنا بتطوير تقنية آلية بناء على الخوارزميات الجينية لتحديد وتصحيح ثغرات حقن لغة الاستعلامات المهيكلة في تطبيقات الويب عن طريق تحديد الاستعلامات الخبيثة في شفرة المصدر. وعلاوة على ذلك، قمنا بتطوير أداة لتنفيذ وتقييم طريقتنا المتبعة. الأداة تنفذ عمليتين أساسية :أولا، التحليل الثابت الأولي للكود المعطى للتعرف على الاستفسارات الضعيفة والثاني هو توليد هجمات حقن لغة الاستعلامات المهيكلة من الخطوة الاولى باستخدام الخوارزمية الجينية. من أجل تصحيح ثغرات حقن لغة الاستعلامات المهيكلة المكتشفة، استخدمنا نهج إعادة هيكلة بسيطة. لتقييم اداتنا تجريبيا، استخدمنا العديد من المشاريع مفتوحة المصدر وكود متاح لتحديد ثغرات حقن لغة الاستعلامات المهيكلة. الاداة نجحت في الكشف وتوليد هجمات حقن لغة الاستعلامات المهيكلة للكود المعطى واعادة هيكلته لمنع الثغرات. وعلاوة على ذلك، قمنا بصياغة مشكلة كشف ثغرات البرمجة عبر المواقع كمشكلة تحسينيه وحلها باستخدام الخوارزمية الجينية. نحن جمعنا بين التحليل الثابت مع الخوارزميات الجينية لتحديد وإعداد سيناريوهات الهجوم على نقاط ضعف البرمجة عبر المواقع. الغرض من التحليل الثابت هو تحديد المنافذ الحساسة في شفرة المصدر. بعد تحديد المنافذ الحساسة، قمنا بعد ذلك بتحويل شفرة المصدر إلى شجرة التحكم في التدفق وطورنا مسارات التحكم

للوصول إلى المنافذ الحساسة التي تم تحديدها. مسارات التحكم في التدفق بعد ذلك، تم اختبارها باستخدام الخوارزمية الجينية عن طريق توليد سيناريوهات هجوم البرمجة عبر المواقع مع مساعدة من قاعدة بيانات للبرمجة عبر المواقع التي تم تطويرها. وتشير النتائج التجريبية أن طريقتنا مفيدة لاختبار الاختراق الأمني.

CHAPTER 1

INTRODUCTION

Over the last couple of decades, web applications have become main stream and are used for numerous daily transactions such as online shopping, financial transactions, information sharing and communication. Despite widespread use of web applications, security threats or security vulnerabilities in web applications are still a big challenge. According to semantic internet security report [1], 75% of the scanned web applications were detected to have some sort of vulnerabilities. And 20% of detected vulnerabilities were considered critical. As a result, a number of organizations have lost billions of dollars due these security vulnerabilities. For example, the yearly cost of cyber crime to the worldwide economy was evaluated to be around US$400 billion [2].

To date, researchers and industrial practitioners have identified that SQL injection and cross site scripting are main security vulnerabilities for web applications. The overarching objectives of our work is to detect and correct the SQL injection and cross site scripting vulnerabilities in the source code of web applications by generating the attack scenarios.

In this research work, we have developed an automated technique, based on Genetic Algorithm (GA), to detect and correct SQL Injection (SQLI) security vulnerabilities in web applications. Our approach allows the automatic SQL injection attacks generation using Genetic Algorithm to detect and correct SQL injection vulnerabilities by identifying the malicious queries in the source code. Furthermore, we have developed a tool to implement

and empirically evaluate our adopted approach. The tool implements two primary processes: first, the preliminary static analysis of the given source code to identify the vulnerable queries and second is the generation of SQL injection attacks for the identified vulnerable queries in first step using genetic algorithm. In order to correct the detected SQL injection vulnerabilities, we used simple refactoring approach. To empirically evaluate our tool, we used several open source projects and available source code to detect SQL injection vulnerabilities. the tool successfully detected and generated the SQL injection attacks for the given source code and refactored it to prevent from SQL injection attacks.

Next, we proposed an approach to detect XSS vulnerabilities. Our proposed approach combines the static source code analysis with genetic algorithm. The primary aim of using static code analysis is to identify the sensitive sinks (vulnerable code points) in the source code. The code identified with sensitive sinks is prone to security vulnerabilities so, we generated the control flow paths of the vulnerable code to cover all sensitive sinks in the code. To illustrate the control flow paths, we also generated the graphical trees. Typically, a program may contain a large number of control flow paths if it contains loops in it. To cover all control flow paths is not possible using traditional algorithms. Hence, we used genetic algorithms to generate XSS attack scenarios to exploit the sensitive sinks identified by static code analysis in those control flow paths. We designed a database of XSS attack pattern by collecting the attack patterns from different resources on internet. The XSS detections starts with static code analysis. We generate tried to cover maximum control flow paths in the provided source code and these control flow paths are being tested using genetic algorithm. Our genetic approach generates XSS attack by using the XSS attack patterns database and applying them on the control flow paths.

1.1 Research Objectives

The objective of this research work is to detect and correct SQL injection and cross site scripting vulnerabilities in the source code of web applications by generating the attack scenarios. The sub-objectives of this thesis are described below.

- Identify common security vulnerabilities in the source code of web applications.
- Literature review of existing techniques and methods that detect and correct SQL injection and cross site scripting security vulnerabilities.
- Applying the genetic algorithm to detect SQL injection security vulnerabilities in source code.
- Correct the identified vulnerable queries using the refactoring technique to mitigate the SQL injection.
- Apply genetic approach to detect Cross Site Scripting vulnerabilities by generating the attack scenarios.

1.2 Main Contributions

The main contributions to this thesis are as follows:

- Literature review of the current vulnerabilities detection techniques.
- Genetic algorithm based detection technique to detect SQL injection vulnerabilities in source code.
- A tool implementing the described approach for detection and correction of SQL injection vulnerabilities.

- Genetic approach to detect cross site scripting vulnerabilities by generating the attack scenarios.

1.3 Organization of the Thesis

This thesis is organized into chapters; each chapter describes a specific part of the whole work. The following chapter 2 provide a detailed background about the most common security vulnerabilities in web applications and their possible threats. Along with that, the details about how users with negative intent can exploit the SQL vulnerabilities by injecting exploits and SQL Injection is illustrated with examples. And the last thing in chapter 2 is the details of Genetic Algorithm (GA). Some of the existing techniques for the detection and correction of vulnerabilities in web applications are presented in Chapter 3 as a Literature Review. The existing techniques are also classified based on their underlying methods to mitigate the problem. Chapter 4 presents the formulation and details about our adopted approach to detect and correct the SQL Injection vulnerabilities in web applications. The experimental setup, experiments details, and results are discussed in Chapter 5. Chapter 6 describes the genetic approach for cross site scripting vulnerabilities detection and Chapter 7 provided its empirical evaluation details for cross site scripting vulnerabilities detection approach. Finally, the last Chapter 8 concludes the complete work along with the discussion of possible validity threats and provides future directions.

CHAPTER 2

BACKGROUND

2.1 Security Vulnerabilities

The security vulnerabilities with most serious risks identified by OWASP are as follows. Most critical of them are briefly described.

2.1.1 Injection Vulnerabilities

Injection vulnerabilities are related to SQL, OS, and LDAP, this is the most recurring attack and it has severe impacts. Attackers can send a malicious text with Queries or input data to the system to execute illegal operations. The consequences of injection are critical it can result in sensitive data corruption, deletion or denial of access in extreme case it can result in complete host takeover. Injection attacks normally found in LDAP, NoSQL queries or XPath, OS commands; SMTP Headers, XML parsers, program arguments.

2.1.2 Cross Site Scripting

XSS is the most prevailing security vulnerability it occurs whenever user provide an untrusted data and application sends it directly to the browser without escaping or prior validation. If XSS is exploited the attacker can deface websites, take over user sessions, or redirect to malicious websites.

2.1.3 Broken Authentication and Session Management

Web applications use sessions to manage the user authentications and to preserve account login details. Users with vicious intentions can take advantage of leaks or flaws in the session management to disclose the confidential information e.g. account passwords, session ids. This vulnerability can cause the leakage of confidential information and account hacking.

2.1.4 Insecure Direct Object References

Attackers can exploit direct object referencing and can access to unauthorized data. The causes of insecure direct object references are the direct exposure of a reference to an internal object, for instance, database connections, file objects and directory.

2.1.5 Security Misconfiguration

Web applications comprise of many things, such as frameworks, web servers, platform, application server and database server. They all should be securely configured and updated to latest versions. If these things are not securely configured, then the system could be compromised completely without you knowing it. Furthermore, other significant vulnerabilities along with mentioned above are, sensitive data exposure, missing function level access control, cross site request forgery, using components with known vulnerabilities and invalidated redirects and forwards.

2.2 SQL Injection Attacks by Examples

Modern web applications mostly consist of web pages which contain the logics and views and a database system at the backend to save the records. Most of the applications do provide login forms for their users to log in to the system, these login forms usually contain

a username, password field and forget the password or email me my password link. This link can be proved as the downfall of the system if it's not handled carefully. When user enter his email while using the email me my password link, it presumably searches in the database for the user email and send something to the provided email. To test if the SQL query for searching the email from database is created with some security parameters or without any sanitization of the user input we add a single quote with input value. If we get a server failure error (500 error) it means that the vulnerable input is parsed and SQL structure will as follows:

SELECT fields_names FROM table WHERE field = '$email';

In the above query $email is the variable which will be replaced by user provided email address. We do not have any information about the fields and the tables yet, but we can make some good guesses. When we enter samran@yahoo.com', observe the single quote and the end it will yield the SQL query something like this.

SELECT fields_names FROM table WHERE field = 'samran@yahoo.com";

When the system will execute the above constructed query, the SQL parser will find and abort it due the extra single quote and will return a syntax error. This response error is a dead exposure that user provided input is not sanitized and that the system is exploitable. As the input values user provide are used in WHERE clause, so the user can manipulate it and change the behavior of the query legally. For example, if we enter anything' OR 'x'='x the resulting query will be.

SELECT fields_name FROM table WHERE field = 'anything' OR 'x'='x';

As we already have identified that the application is not concerning about the query, it's just constructing a string, so our provided input will turn this query from single component WHERE clause to two component clause. And the second clause we provided 'x'='x' will always return true no matter what is in first clause. Unlike the original query, which is intended to return only one item each time, the modified version will must return every details in the database. This was the basic example how we can guess and modify the SQL query structure to alter its behavior. Following few sections will illustrate some of the different scenarios of SQL Injection to perform various functions.

2.2.1 Schema field mapping

Now we have described the basics of SQL injection and how to alter the query structure by injecting the malicious strings into it. In this example we will illustrate how we can guess the schema mapping of the system to get the fields names. The initial steps are to guess the names of some fields, for example we are fairly sure that the login query contains an email address and password, in addition to that there may be few other fields like "address", "userid" or "phone_number". We would affectionately like to execute SHOW TABLE query, but we do not know the name of the table yet, along with that there is no other means to get the result of this query routed to us. So we will approach it in steps in each case, we will start with the email field, the following example will illustrate the query structure to guess the field name. as we have already shown that at the end of the query we can embed malicious strings in where clauses.

```
SELECT fieldlist FROM table WHERE field = 'x' AND email IS NULL; --';
```

Here our objective is not to match the email address or to crack the password, instead we want to guess the field name. so, if we get a server error it means that our query is malformed and server throw a syntax error. Otherwise if we get response like unknown email address or password it means we have correctly guessed the field name. now in the above shown query the we have used “x”’ with AND operator and at the end of the string there is -- which means the start of the SQL comment, and this is a clean and nice way to get rid of the last single quote. In this particular example we have used conjunction (AND) we can also use OR operator instead.

2.2.2 Finding the Table Name

As a user we do not know the table name, although it is already embedded in the query. There are several ways to find out the table name, in the following example we will illustrate one of them, which is by using a sub query that returns the count in of records in a table e.g. SELECT COUNT (*) FROM tab_name.

We can embed this sub query in to the original query to probe the guessing of table name, obviously it will return an error if it fails due to the wrong name of table mentioned in it.

SELECT email, passwd, login_id, full_name FROM table WHERE email = 'x' AND 1=
(SELECT COUNT (*) FROM tabname); --';

We are not concerned about the number of records in the table at the moment we are only looking to guess the table name. After several trails we eventually find out that the table name is members. To verify this, we re-write our query with the guessed name and check the response. Similarly, there are many other scenarios that can be formulated and executed by appending them with the queries.

2.3 Cross Site Scripting Consequences and Examples

Cross Site Scripting (XSS) is another type of injection vulnerabilities in web applications. The difference between SQL injection vulnerabilities and Cross Site Scripting vulnerabilities is that, SQL injection is server side vulnerability and Cross Site Scripting is client side vulnerability. XSS is one of the most flagrant vulnerability and it occurs when an attacker injects malicious script into a legitimate website which then can be executed in client's web browsers. It usually occurs when a web application uses an un-encoded and un-validated data and process it along with its output.

An attacker does not directly affect the victim using XSS, instead the attacker exploits the XSS vulnerability in another website which the victim would visits. The intention is to use the vulnerable site as a medium to deliver the malicious code into the victim's web browser. Usually attackers inject JavaScript code in vulnerable websites as JavaScript is executable in almost all modern browsers. For example,

Figure 1 is showing the simplest example of printing comments from database in PHP. The problem in the illustrated example is that, the code is assuming the comments consists of text only and not escaping or validating them prior to printing. In this example the attacker may insert a JavaScript snippet (e.g. <script>alert ('something vulnerable.') </script>) as shown in

Figure 1 and Figure 2 which will be executed every time browser will load these comments.

```php
<?php
echo "<html>";
echo "Most Recent Comment:";
echo $database_results->last_comment;
echo "</html>";
?>
```

Figure 1. Print Comments from Database in PHP

```php
<?php
echo "<html>";
echo "Most Recent Comment:";
echo "<script>alert('something vulnerable.')</script>";
echo "</html>";
?>
```

Figure 2. Simple Cross Site Scripting Example

JavaScript its self is not harmful, in fact most of the browsers executes JavaScript in a very restrict manner and it does not have access to server files and operating systems. However, it becomes malicious due to the following facts.

- JavaScript has access to user's sensitive information like Cookies
- JavaScript can send HTTP requests to arbitrary destinations with arbitrary content using XMLHttpRequest mechanism.
- JavaScript can make modifications in HTML content using DOM manipulation methods.

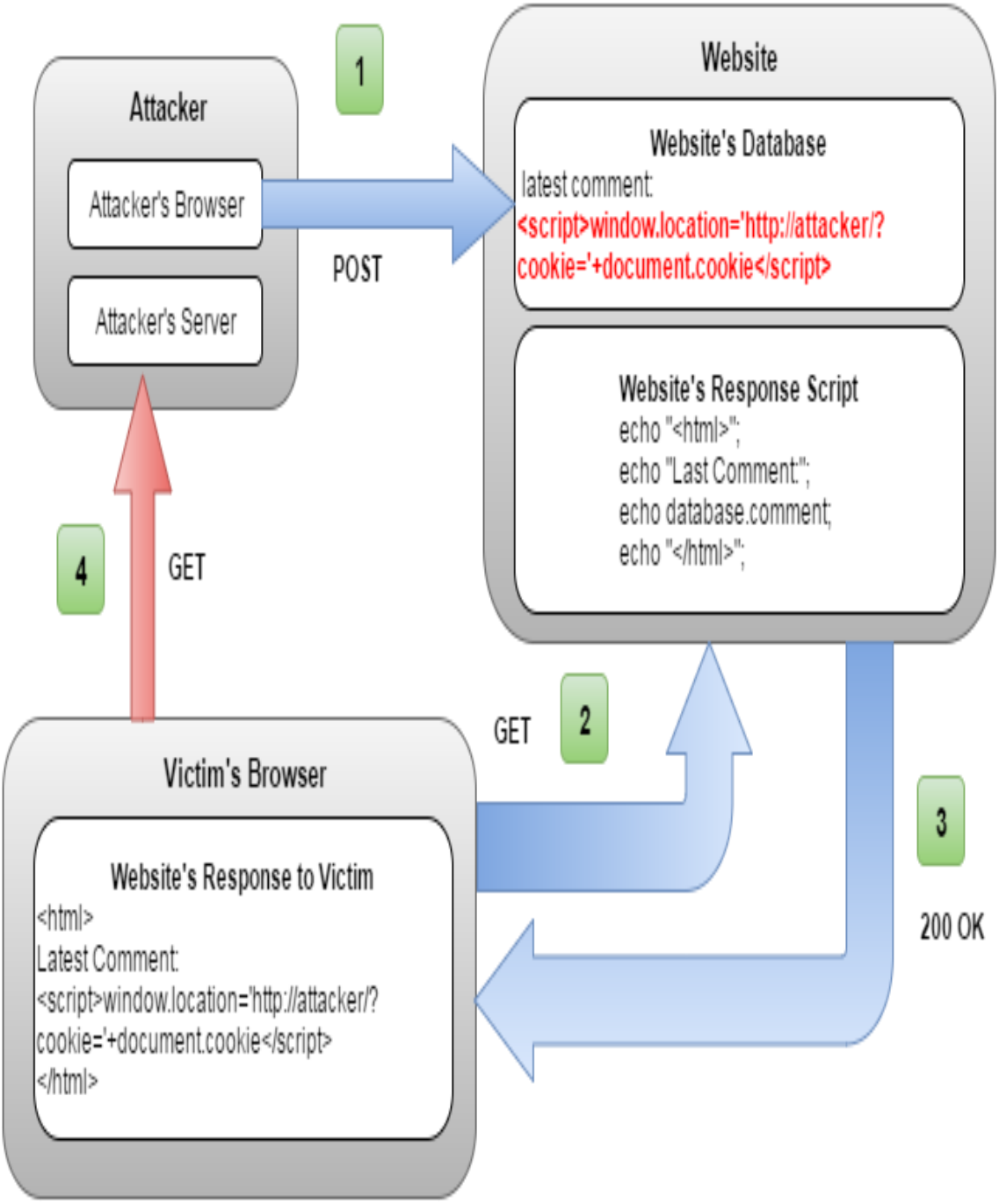

Figure 3. Cross Site Scripting Attack Scenario

2.3.1 Consequences of Malicious JavaScript Code

An attacker can perform various kind of malicious activities if he found a way to execute JavaScript in someone other's browsers. A simple attack scenario of Cross Site Scripting is shown in Figure 3 which is illustrating how an attacker can inject malicious code into a website and how that malicious code then can compromise the security of an individual. Some of the security risks of XSS are described below.

Cookie theft

Attacker can execute a JavaScript in victim's browser to send him the victim's cookie details. which he can use to extract sensitive information like Session Ids and user tokens.

Keylogging

Another security thread of Cross Site Scripting is that; the attacker can register event listeners in victim's browser. The registered event listeners can send any thing victim will type this may result in sensitive information leakage like credit card numbers, passwords and confidential conversations.

Phishing

Phishing is another kind of Cross Site Scripting attack in which the attacker can replicate a legitimate webpage and can make the victim to provide sensitive information like user name, password and credit card numbers. Another approach for phishing is that the attacker can use DOM manipulation and insert a fake form with form's action set to his server. As a result, the details provided in the form will be submitted to the attacker's server.

2.3.2 Types of Cross Site Scripting

There are different ways of executing the malicious JavaScript in victim's browser to perform Cross Site Scripting attacks. Majorly Cross Site Scripting is divided into three types, those are described below.

Persistent XSS

Persistent XSS is one of the most serious kind of XSS vulnerability. Malicious JavaScript can be stored into website's database and results into persistent XSS when the website insert user input into its database without any sanitization and checking [3].

Reflected XSS

Reflected XSS type of vulnerabilities are not loaded with the website it actually loads when the victim tries to load the injected website. They are also called non-persistent or type-one Cross Site Scripting vulnerabilities [4].

DOM Based XSS

DOM based XSS vulnerabilities are based on client side. The attackers use DOM manipulation to deface or to change the forms and their action attributes in the website, so all the data which victim will provide in that form will be sent to the attacker [5]. Figure 4. is showing a DOM based attack scenario. DOM based XSS vulnerabilities are different from Persistent and Reflected XSS vulnerabilities. The subtle difference is the traditional XSS (reflected and persistent) executed when the website is loaded in client's browser while the DOM based XSS executed after the loading of website as result of not handling the legitimate JavaScript properly.

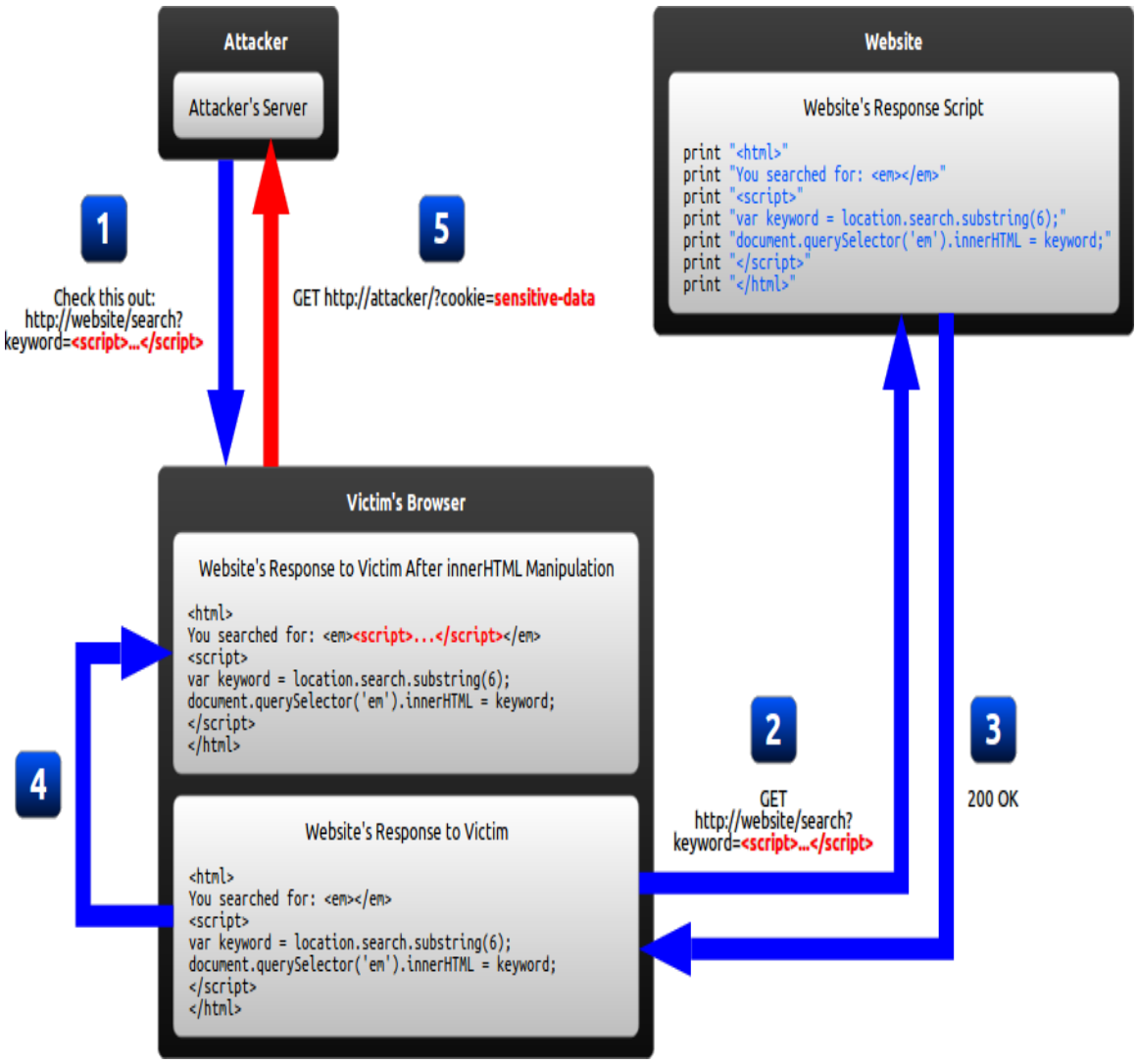

Figure 4. DOM based XSS attack Scenario

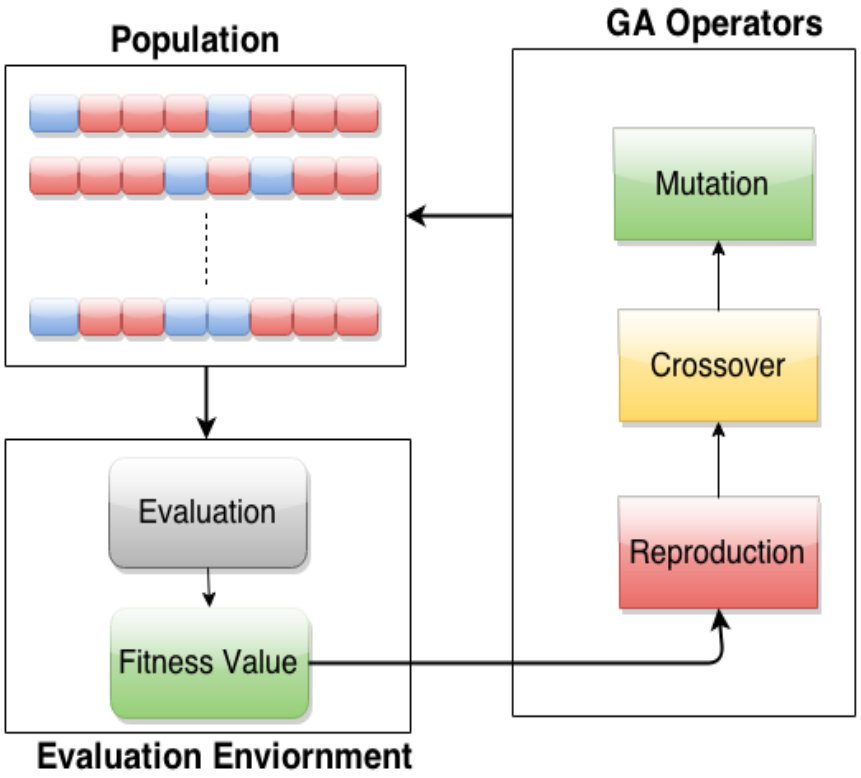

Figure 5. Genetic Algorithm Overview

2.4 Genetic Algorithm

Genetic algorithm (GA) inspired by theory of evolution is a search heuristic algorithm. Genetic algorithm often used to search solution for optimization problems. The solution to a problem solved by genetic algorithm is evolved. For large search spaces and np-complete problems genetic algorithms are suitable. Genetic algorithm evolves a population of generations. Each generation consists of a solution, and the most fit or most relevant generation is selected as the final solution. To reach the final solution multiple populations generated and the most-fit generation has higher chance to survive in next generation. In the new population the generations undergo crossover and mutation processes. Figure 5. is showing the main steps of GA which includes, population generation, evaluation of

individuals, crossover, mutation and reproduction. Following few sections will briefly describe the GA processes.

Population: population consists of different generations called chromosomes or individuals and each chromosome consist of genes. In chromosome the genes encoding is problem specific. In some problems genes could be encoded as binary bits and in some other problems genes could be characters or strings. Each population consists of a reasonable length of chromosomes. When algorithm run for the first time it generates the population randomly.

Evaluation: once the population has been generated and initialized it is evaluated to calculate its chromosome's fitness. To evaluate fitness of a chromosome a certain function used which is called fitness function. Similar to genes encoding the fitness function is problem specific for example, for one kind of problem fitness function could be simple expression while for other kind of problem it could be a complete equation. All the chromosomes undergo the evaluation process to find the most fitted chromosome in the population. The calculated fitness of a certain chromosome is compared with the max fitness, if it is equal to max fitness it means this is the most fit chromosome and also this contains the solution. If it is not equal to max fitness the algorithm will check next chromosome until it found the solution of most fit chromosome in the current population. If algorithm unable to find the solution in current population, it will reproduce new population and check the fitness of all chromosomes again.

Selection (Reproduction): We want to improve the populations overall fitness. Selection helps us to improve the fitness by rejecting the bad chromosomes and keeping the best

chromosomes in the population. There are different selection methods but the basic idea is same, that fitter chromosomes will be selected for next generation. Tournament Selection approach to select chromosomes for next generation is usually preferred. The reason is to avoid the algorithm to stuck in local optimum and to converge at global optimum solution.

Crossover: crossover is the process that create a new chromosome by partially inheriting genes from two different chromosomes. Once the chromosomes are selected for new generation they undergo the process of crossover. Crossover can be performed at different rate or probability.

Mutation: mutation is the process in which a gene of a chromosome is changed randomly. That idea is to produce a variety of new population with vast attributes.

Repeat: the process of selection, crossover and mutation is repeated until the algorithm found the solution. Or the population fitness is not improving for a certain number of generations. For example, if the fitness of the generation is stuck to a specific number and it is not improved since 100 generations then the algorithm should terminate.

CHAPTER 3

LITERATURE REVIEW

In this section we will describe the most recurring security vulnerabilities and the proposed techniques to detect these security vulnerabilities in literature. And at the end of this section a summary table is presented illustrating the focus and main idea of existing techniques.

Web applications gained popularity in the start of twentieth century and at the same era researcher started the investigation on vulnerabilities to detect and prevent from them. An online community started a project [6] and dedicated it to application's security. They have identified and ranked the web applications vulnerabilities according to their well-defined and well versed Risk Rating Methodology. Along with the identification and raking, well documented recommendations also provided for programmers, testers and organizations to avoid and fix these vulnerabilities. In the following section we will describe some of the existing techniques for detection and correction of vulnerabilities in web applications.

3.1 Existing Detection and Correction Techniques

3.1.1 Taint Analysis Based Techniques

Yau-Wen Huang in [7] considered the web applications vulnerabilities as secure information flow problem and developed a tool to prevent the exploitation of vulnerabilities on runtime. The tool, they developed named WebSSARI (Web application Security by Static Analysis and Runtime Inspection). In the tool they used lattice based

static code analysis algorithm. The problem in their approach and tool is that they only guarding the vulnerabilities not actually removing them from the code.

Another static code analysis approach is described in [8] by N. Jovanovic et al. the approach presented in this research work used alias analysis, and they integrated their concept into a static analysis tool to test its credibility. But again, their approach is limited to specific vulnerability and the identified vulnerabilities have to be fixed manually by the programmer.

W.G.J. Halfond [9], proposed a technique to protect the web applications against SQL injection vulnerabilities, their technique is based of taint analysis and syntax aware evaluation. The proposed technique used positive taint analysis which if different from traditional taint analysis. Positive taint analysis refers to identification and marking of trusted input data, while on the other hand traditional taint analysis identify and mark the untrusted data and prevent it from being used in exploitable way. To verify the data marked as trusted by positive tainting, syntax aware evaluation is performed on it before being sent to database.

N. Jovanovic et al. [10] presented static code analysis approach and implemented their approach in tool called Pixy. In their approach they used inter-procedural, context-sensitive data flow and flow-sensitive analysis to discover vulnerable points in a program. In addition to above mentioned methods, literal and alias analysis are utilized to improve the precision and correctness of the results. The methodology can be applied to the identification of SQL injection and cross site scripting, because the presented concepts are targeted at the general class of taint-style vulnerabilities.

S. Son [11] proposed a new classification of security vulnerabilities in PHP web applications and as a solution a framework has been presented, which comprises of several new algorithms to identify potential security vulnerabilities that existing techniques can easily miss. The proposed approach implemented as a tool named SaferPHP that can identify different kind of potential vulnerabilities in PHP applications. SaferPHP comprises of six modules; computing the control-flow graph, computing the call graph, Taint analysis, Tainted loops and symbolic execution, collecting critical variables, Security analysis modules. Security analysis modules comprises further on three sub models, finding denial of service vulnerabilities, finding missing authorization checks and Finding other vulnerabilities.

Another framework for detection of security vulnerabilities in PHP applications is described by J. Zhao and R. Gong [12]. In this framework, they combined two existing analysis techniques those are static analysis and dynamic analysis this combination of two techniques helped in improving the efficiency of detection. A tool based on HHVM also developed to empirically evaluate their framework. HHVM is an open source virtual machine it is used to compile PHP or Hack script to speed up service delivery and it is designed by Facebook.

M. K. Gupta [13] proposed a technique which is based on pattern matching and static taint analysis. The proposed technique is context-sensitive and designed to detect and mitigate the cross site scripting vulnerabilities in the source code of web applications. A prototype of the proposed approach in shape of a tool is implemented to empirically evaluate the credibility of the technique.

3.1.2 Runtime Enforcement Techniques

Tadeusz Pietraszek [14] proposed CCSE approach to serialize the user input on runtime. This approach does not require any programmer interaction or code modifications to prevent various kind of Injection vulnerabilities. This is a preventive approach, which avoid exploitations of Injection vulnerabilities by serializing the user inputs.

O. Hallaraker and G. Vigna [15] proposed an approach to avoid cross site scripting vulnerabilities. In their approach they introduced an auditing system for the JavaScript's interpreter for Mozilla web browser. In their auditing system they used and intrusion detection system (IDS) which detect exploitable execution of JavaScript operations, and the same time take appropriate counter-measures to avoid from infringement against the browser's security. The key idea behind the proposed approach is to identify the scenarios of executions of a script written in JavaScript, in which the browser resources are used abusively.

S. Gupta et al. [16] analyzed issues in the performance of existing cross site scripting filters approach, based on that, the author proposed a framework named XSS immune. The proposed framework based on string comparison and context aware sanitization of JavaScript. The framework is integrated inside the browser and compares the scripts embedded in the hypertext transfer protocol (HTTP) request and response to identify any malicious or untrusted JavaScript code. The proposed framework can also determine the context of detected XSS vulnerabilities and then it can sanitize them accordingly to alleviate their effects on the real time web applications. Furthermore, the framework is also capable of detecting the partial script injections, which can inject the malicious parameter values in the exiting JavaScript.

M. Alenezi et al. [17] analyzed several open source projects to identify different kind of vulnerabilities and their causes. They identified SQL inject, CSRF and cross site scripting vulnerabilities and find out that, these are the results of programmer's negligence, bad programming practices and the maintenance and enhancement activities. Author also claimed that the open source projects are widely used by organizations to save cost but it also leads them vulnerable to security attacks. Furthermore, they made two suggestions to cope with security vulnerabilities, first before selecting any open source platform the system analyst should analyze the size, type and the attacks those can be done on the under consideration web application. Before finalizing the considered platform, the analyst should also consider the expertise of the programmers to avoid bigger problems. Finally, the organization should develop a secure development framework to urge developers to follow best and secure programming practices to avoid leaving loop holes in the code. The framework should be flexible enough to select previous projects and remove their vulnerabilities. Last but not least, the framework should be dynamic and should be easily updatable to accommodate the latest threats and vulnerabilities as they emerge with time.

S. Cho et al. [18] presented a runtime validation and security enforcement approach to prevent from security vulnerabilities. The proposed approach can work with the web applications developed in Java server pages (JSP). The approach enforces the verification of input values using static bytecode instrumentation at runtime. The underlying technique searches for object constructors or target methods in compiled Java classes, and inserts bytecode modules statically. Their approach successfully identified and mitigated the SQL injection vulnerabilities in WebGoat project.

3.1.3 Testing Based Techniques

Yau-Wen Huang et al. [19] described mechanisms to apply different testing techniques like dynamic analysis, black box testing, behavior monitoring and fault injection to identify SQL injection and Cross Site Scripting vulnerabilities in web applications.

R. Akrout [20] presented a new methodology, the proposed methodology based on black box analysis of the target application. The mechanism of their approach is they actually exploit the identified vulnerability in order to make it more accurate. An additional aspect of the proposed approach is that it can devise different potential attack scenarios including the exploitation of several successive vulnerabilities, taking into account explicitly the dependencies between these vulnerabilities. The main focus is to identify on the code security vulnerabilities, such as SQL injections. An experimental evaluation is also provided of the proposed approach by developing the prototype of the approach in a tool.

N. M. Vithanage et al. [21] presented a tool based on their proposed approach named WebGaurdia, which is capable of detecting top five web application vulnerabilities among top ten. Two new approaches have been proposed in this paper to detect two types of vulnerabilities and already existing approaches have been highly modified in order to enhance the performance of the system. On completion of the tests, a report is generated along with a user friendly graphical overview of the detected vulnerabilities which can accordingly be used to find in detailed information regarding the executed attacks. The results indicate that, even though it is technically infeasible to completely avoid generation of false positives and false negatives, by using WebGuardia and proposed approaches, generation of false positives and negatives can be kept at a minimum level.

M. R. Reddy et al. [22] presented another penetration testing based approach to detect vulnerabilities in web applications. They illustrated the importance of mathematical constructs in the development of efficient test data generators. They proposed penetration testing approach for web applications security testing using several tools in which the tools scan for vulnerabilities and security loop holes in the web applications and finally they performed security audit based on these scanning results.

3.1.4 Prevention Based Techniques

D. Scott and R. Sharp [23] suggested to use a proxy server at web applications site to monitor and filter the incoming and outgoing data. But there are limitations in this approach, the deployment of monitoring and filtering proxy server would create scalability and performance limitations. Another filtering technique is proposed by Z. Su and G. Wassermann in [24], the proposed approach filter out the malicious data by using syntactic criterion. The solution is quite efficient in analyzing queries to detect misuses, by wrapping the malicious statement to avoid the final stage of an attack. The lack in this approach is that, it is language dependent and for the time being the organization does not seem a trivial task.

B.Hanmanthu et al. [25] presented an approach to prevent from SQL injection vulnerabilities. Their proposed approach sends different specially formulated SQL attacks to an application. After that, they construct a decision tree model to create a database of the responses of SQL injection attacks.

H. Zhang et al. [26] presented an accurate and practical prevention approach for SQL injection vulnerabilities. Their approach based on taint analysis and mark the trusted sensitive data into extended utf-8 encodings. Unlike typical positive taint analysis solutions

that taint all characters in hard-coded strings written by the developer, we only taint the trusted sensitive characters in these hard-coded strings. Furthermore, rather than modifying Web application interpreter to track taint information in extra memories, we encode the taint metadata into the bytes of trusted sensitive characters, by utilizing the characteristics of UTF-8 encoding. Lastly, we identify and escape untrusted sensitive characters in SQL statements to prevent SQL injection attacks, without parsing the SQL statements. A prototype called PHPGate is implemented as an extension on the PHP Zend engine. The evaluation results show that PHPGate can protect Web applications from real world SQL injection attacks and introduce a low performance overhead

3.1.5 Other Techniques

I. Medeiros [27] proposed a technique which is basically a combination of two methods to detect security vulnerabilities in web applications with minimum number of false positives. The proposed approach combined taint analysis which identify the potential security vulnerabilities, with data mining technique, which predicts the existence of false positives in the identified vulnerabilities. The presented approach unites two approaches that are seemingly orthogonal; taint analysis and machine learning. Along with the combination of two important methods for detection of vulnerabilities, the author proposed automatic corrections of the detected vulnerabilities by embedding solutions to source code.

D. G. Kumar and M. Chatterjee [28] described a model for SQL injection attack detection. To detect SQL injections, they used the concept of information theory. The proposed framework works on both ends, client end and server end. The key idea is to implement a filter algorithm on Client side that checks the data type and length of the submitted variables, and issue warnings of the keywords and injection sensitive characters on Client

side after preliminary examination. And on the Server side the proposed approach works in two phases first training and then detection. The complexity of the query is calculated in training phase statically and when it is submitted dynamically which is called entropy of the. Once the entropy is calculated an algorithm called Message authentication algorithm (MAC) is applied on both entropies. If the query is altered with attack inputs it means its structure has been changed, therefore the entropy of the query has changed significantly which will change the corresponding MAC value. Change in values of MAC indicates SQL injection.

Another recent research work presented by Aziz et al. [29] proposed a genetic algorithm approach to detect SQL injection vulnerabilities. The core concept of their approach is to generate test cases using genetic algorithm to inject in SQL queries. They defined two kinds of functions to generate test cases for different scenarios of SQL injection. But, their presented work has many shortcomings. First of all, the grammar used to generate the test cases is too limited which can only generate specific type and limited number of test cases and can easily miss the critical injection exploits. Secondly, the underlying important details of the genetic adaptation is missing or too brief, for example how the individuals are represented and population is generated, and how the genetic operators (crossover and mutation) are carried out. Finally, the empirical evaluation is conducted on only a single system which is not enough to conclude the credibility of the presented approach.

Table 1: Summary of Literature Survey

Paper	Focus	Main Idea
Y. W. Huang [7, 19]	Detection and Prevention	Automatic detection of vulnerabilities and employing runtime guards to vulnerable code in order to avoid exploitation of vulnerabilities.
N. Jovanovic [8]	Automated Detection of XSS	Automatic detection of XSS vulnerabilities using taint analysis, and generate the exploit scenarios
T. Pietraszek [14]	Detection and Prevention of SQL Injection	Uses ad-hoc serialization of user input to avoid SQL Injection attacks
W.G.J. Halfond [9]	Automated Detection of SQL Injection	Automated technique to protect the web applications against SQL injection vulnerabilities, the technique is based on positive taint analysis and syntax aware evaluation.
D. Scott and R. Sharp [23]	Preventive Approach to Avoid Vulnerabilities	Suggested to use proxy servers and data filters to monitor the information exchanged between client and server and by pass the suspicious information.
Z. Su and G. Wassermann [24]	Preventive Approach to Avoid Vulnerabilities	Proposed an approach to filter out the malicious data by using syntactic criterion; context free grammars and compiler parsing techniques
O. Hallaraker and G. Vigna [15]	Detection and Prevention of XSS	Introduced an auditing system for the JavaScript's interpreter for Mozilla web browser. In which they used and intrusion detection system (IDS) which detect exploitable execution of JavaScript operations, and the same time take appropriate counter-measures to avoid from infringement against the browser's security
R. Akrout et al. [20]	Detection of SQL Injection	The proposed methodology based on black box analysis of the target application. The mechanism of their approach is they actually exploit the identified vulnerability in order to make it more accurate.
I. Medeiros et al. [27]	Automated Detection and Correction	The proposed approach combined taint analysis which identify the potential security vulnerabilities, with data mining technique, which predicts

		the existence of false positives in the identified vulnerabilities.
Aziz et al. [29]	Automated Detection of SQLI	The core idea is to generate the SQL injection test cases using Genetic Algorithm. And inject the generated test cases in the SQL queries of application to detect SQL injection.
S. Gupta et al. [16]	XSS Detection and Prevention	The proposed framework based on string comparison and context aware sanitization of JavaScript. The framework is integrated inside the browser and compares the scripts embedded in the hypertext transfer protocol (HTTP) request and response to identify any malicious or untrusted JavaScript code.
M. Alenezi et al. [17]	SQL injection Detection	They applied different existing approaches to identify SQL injection vulnerabilities and figure out the primary reasons of SQL injection vulnerabilities. On the basis of their findings they proposed two suggestions, first to select suitable open source application according to their give guidelines. Second the development of a secure programming framework within the organization.
N. M. Vithanage et al. [21]	Multiple Vulnerabilities Detection	Two new approaches have been proposed in this paper to detect different types of vulnerabilities and already existing approaches have been highly modified in order to enhance the performance of the system. On completion of the tests, a report is generated along with a user friendly graphical overview of the detected vulnerabilities which can accordingly be used to find in detailed information regarding the executed attacks
S. Cho et al. [18]	SQL injection Detection	The author presented a runtime validation and security enforcement approach to prevent from security vulnerabilities. The proposed approach can work with the web

		applications developed in Java server pages (JSP). The approach enforces the verification of input values using static bytecode instrumentation at runtime
H. Zhang et al. [26]	SQL injection Detection	Author presented an accurate and practical prevention approach for SQL injection vulnerabilities. Their approach based on taint analysis and mark the trusted sensitive data into extended utf-8 encodings
The Proposed Approach	Automated Detection and Correction of Security Vulnerabilities	Automated detection and correction of security vulnerabilities in web applications. Genetic approach would be used to automatically generate exploits for vulnerable queries to detect SQLI. And the detected vulnerable queries are corrected using refactoring technique.

CHAPTER 4

GENETIC APPROACH TO DETECT SQL INJECTION

4.1 Approach Overview

To tackle the SQL injection problem or at least circumvent it we proposed genetic approach, which comprises of two following process.

- Preliminary scanning to identify malicious queries in the given source code.
- SQL Attack/Scenarios generation: we used genetic algorithm to generate SQL injection attack for the malicious queries identified in previous step.

Figure 2. illustrate the general structure of our adopted approach and more details are provided in the following sections.

4.2 Preliminary Scanning Process

In this step, malicious queries those seems vulnerable to SQL injection attacks are identified and stored in an array data structure. The identification step takes the path of file or directory of sources code to scan for malicious queries that exists on the system. If the given file or directory path is an invalid path then, the process will terminate with an error message. If the give path is a valid path then, the algorithm will check if it's a file or directory. If it's a file, then the algorithm will read the file line by line and look for malicious queries. To mark a query as malicious we have used certain rules those are as follow.

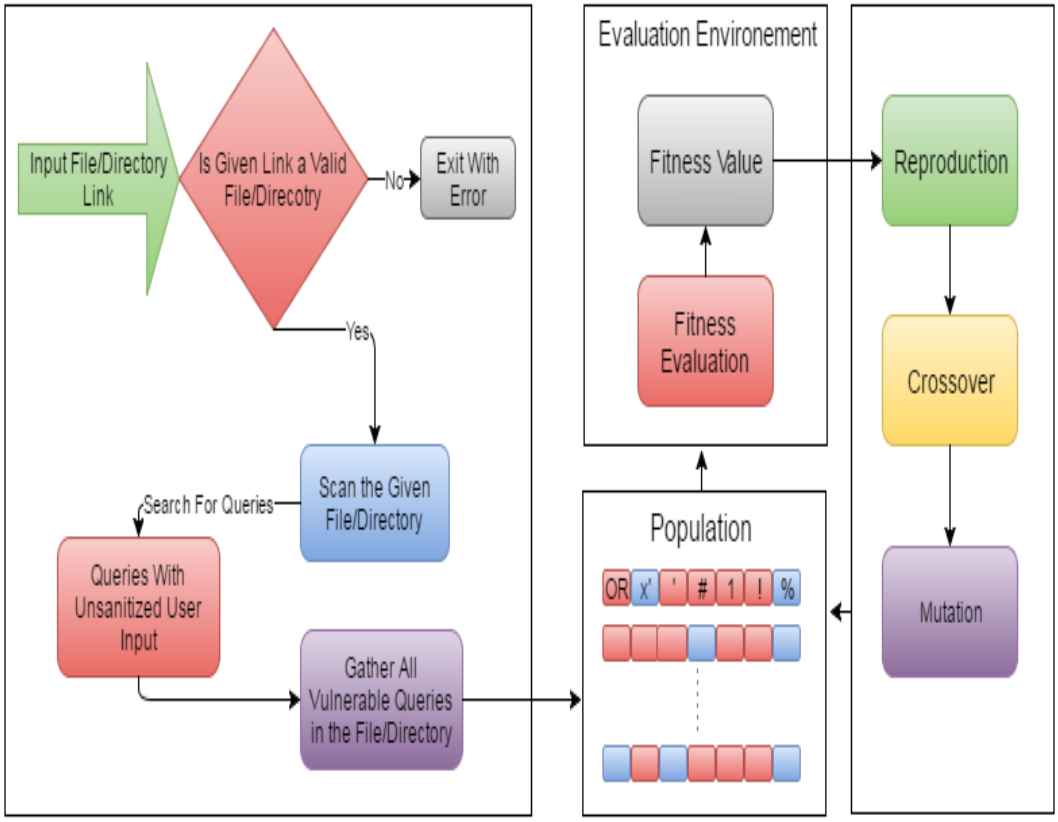

Figure 6. Approach Overview

```
SELECT * FROM table_name WHERE id='.$id.'

DELETE FROM table_name WHERE id='.$id.'

UPDATE table_name SET active='yes' WHERE id='.$id.'
```

Figure 7. Queries with where clauses

```
INSERT INTO members ('email', 'passwd', 'login_id', 'full_name') VALUES ('.$email.', '.$password.', '.$login_id.', '.$full_name.')
```

Figure 8. Insert query with un-sanitized input

- The query should be one of these: SELECT, DELETE, UPDATE, INSERT.
- The query must have a WHERE clause in it, except in INSERT case as shown in Figure 7.

- The query must be using un-sanitized user input to store the details in INSERT case and in WHERE clause for other cases. As shown in Figure 8.

If any query fulfills the above rule it will be stored in an array for next process. In the case of a directory path, the algorithm will loop through all valid source code files in the given directory and will apply the same process as described for file to collect all malicious queries from that directory, as shown in Figure 6.

4.3 Genetic Approach to Generate SQL Attack/Scenarios

After collecting all malicious queries in the given directory or file the next process is to verify the maliciousness of the collected queries by generating the SQL injection attacks using genetic algorithm. The following section describes the adaptation of genetic algorithm (GA), used to generate SQL injection attacks. To apply GA to a specific problem, its following main steps have to be described according to that particular problem:

- Individual Encoding/Representation,
- Generation of population from individuals,
- Fitness function definition, to evaluate the individuals in population for their ability to solve the problem,
- Individuals selection to transmit to new generation,
- Production of new individuals using genetic operators (mutation and crossover),
- Production of new population.

The next sections comprehensively explain the adaptation of the design of these elements for the automatic generation of SQL injection attacks using GA.

Algorithm: SQLInjectionDetection

Input:

Q = Set of malicious queries

Process:

```
1.  I = exploits_set (individual_size)
2.  P = set_of (I)
3.  Initial_population (P, population_size)
4.  for all queries in Q
5.      While (Fitness < MaxFitness OR NumOfIterations = threshold)
6.      For all I in P do
7.          generate_query = decode_exploits(I)
8.          fitness (I) = execute_query(generated_query)
9.      end for
10.     P = new_population(P)
11.     end while
12.     attack_scnarios = array (query => best_fitness(I))
13. end for
14. return attack_scnarios
```

Output:

Set of SQL injection attack/scenario

Figure 9. High Level Pseudo Code of Adopted Genetic Algorithm

4.3.1 Genetic Algorithm Overview and Adaptation

A high level pseudo code of genetic algorithm (GA) to generate SQL injection attack is illustrated in Figure 9. As the pseudo code describes the algorithm takes as input a set of malicious queries vulnerable to SQL injection attacks, and returns a generated attack for each corresponding query. Line 1 generates the individual of given size by randomly selecting the exploits from exploit set. From line 2-3 a population is generated and initialized with the give size of population from the set of individuals generated in previous step. Line 4 applies a for loop on all queries in the array given to the algorithm as input to generate SQL injection attacks. From line 5 to 11 encode a while loop with termination conditions, the while loop will carry out the actual genetic functions: fitness calculation, crossover, mutation and new population generation until it found the most fitted solution or certain number of iterations are completed. Inside the while loop on line 6 there is a for

loop which will iterates all individuals in the current population, and during each iteration the algorithm will decode the individual and generate an exploitable query by appending the decoded individual with the query. And once the query is generated, it will be executed to evaluate its fitness. Once the fitness is being calculated for all the individuals in current population the algorithm will perform the genetic operators: crossover and mutation and will generate new population. by the end of the execution of algorithm we will have a SQL Injection attack generated for the identified vulnerable queries.

In the following few sections we will describe our adaptation of Genetic Algorithm (GA) to SQL Injection detection problem more precisely.

4.3.1.1. Individual representation

In our adaptation of GA, the individuals are derived from a set of exploit strings, which is constructed by taking the examples from [30] and [31], as discussed in sub section 2.2. SQL Injection attacks by examples of chapter 2. Each gene can either be an operator, operand or a string value. In our algorithm genes are represented as array data structure as shown in Figure 10. Next, algorithm will randomly choose an element from the exploit string's set (e.g. OR, AND, 1 etc.) and will add it to individual gene.

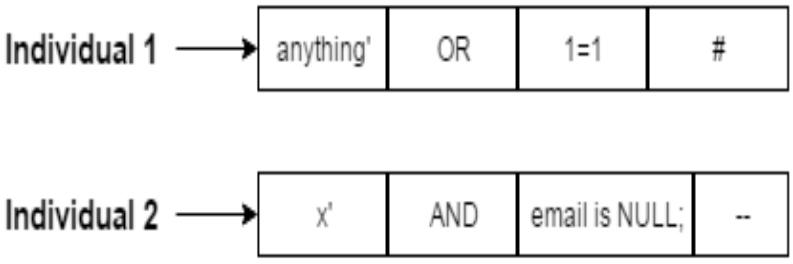

Figure 10. Individual Representation

The key idea behind encoding the exploit strings as array is that, when we will be calculating the fitness of an individual we will convert them to string and will append with the query as query string. The size of individual is dynamic and we can set it to an appropriate length.

4.3.1.2. Generation of Initial Population

To generate initial population, we start by defining the size of the population. The algorithm will take the population size and generate the population of individuals. The individuals will be generated by picking genes from exploit strings set randomly and will assigned to population. Each individual in the population will be representing an attack scenario for SQLI. Furthermore, it's important to note that all individuals in the population initially have zero fitness value. Figure 11 is showing the initial population of five individuals all containing different combinations of exploit strings.

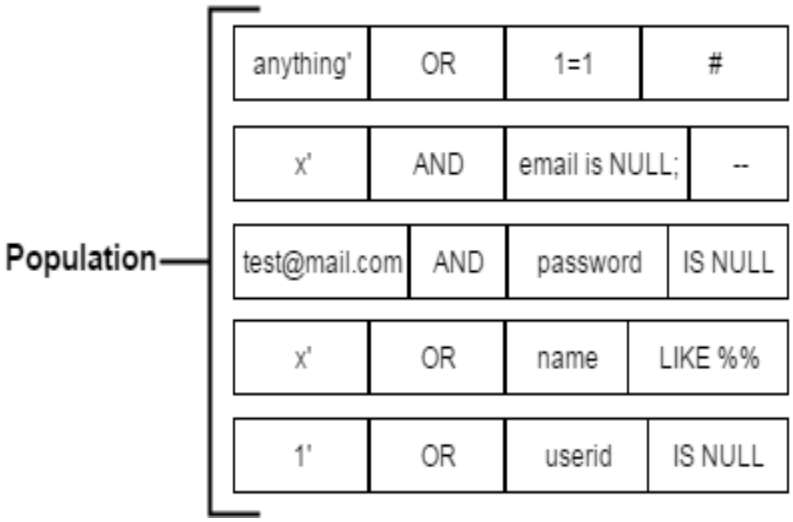

Figure 11. Initial Population

4.3.1.3. Genetic Operators

Selection: To select the individuals for genetic operators, there are different selection approaches available e.g. Roulette Wheel Selection, Elitism Selection, Rank Selection, Stochastic Universal Sampling and Tournament Selection etc. In Roulette Wheel Selection the probability of an individual to be selected in next population is proportional to its fitness, high fitness means high chances of being selected [32]. The drawback of this method is that, the chances of algorithm being stuck to local optimum solution are high. In other words, Roulette Wheel Selection will reduce the mutation properties and limit the selection pool for the algorithm, hence we are negligently limiting its potential to optimal solution. Other famous selection method is Tournament selection were we select x random individuals and pick the best one among them. This approach provides diversity in the population and the chances of algorithm being stuck in local optimum solution are low. That is why we have selected this selection method in our adaptation.

Crossover: crossover on individuals of population is performed by using the selection criteria as described in section 4.3.1.3. Next, the algorithm will apply the crossover operation on the selected individuals and will swap their genes according to the given crossover rate. Thus, by applying the crossover function will have two new offspring inheriting the genes from two parent individuals as represented in Figure 12. Varying the crossover rate can alter the performance of genetic algorithm further details about the crossover rate we have used is provided in Table 5.

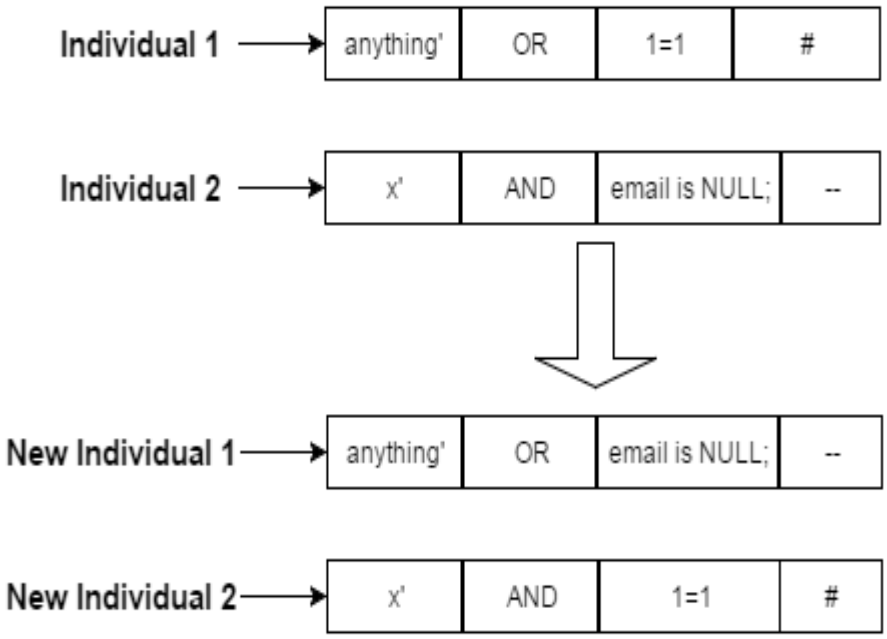

Figure 12. Crossover Operation

Mutation: Unlike to crossover operation, mutation operation is performed on a single individual. The algorithm will pick a gene from the given individual and will change it randomly with any element taken from exploit strings set according to the given mutation rate. Figure 13 is showing that, how the mutation operation changes the gene of an individual which can have different behavior. Just like crossover rate mutation rate can also vary the performance of genetic algorithm and more details about the mutation rate we used in our approach are presented in Table 5.

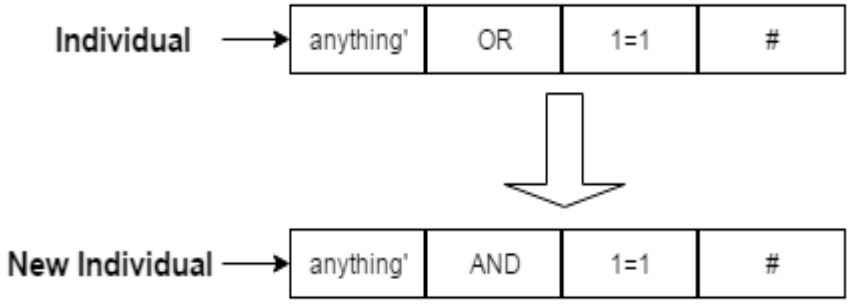

Figure 13. Mutation Operation

4.3.1.4. Fitness Function

The fitness function is the most crucial part of the GA, also considered as the heart of GA. Fitness function actually guides the GA towards solution. The quality of the solution generated by GA is very dependent on its evaluation (fitness) function. The development of an efficient fitness function is very much dependent on the type of individuals and the problem we are dealing with. To evaluate the fitness of an individual, we have formulated the following fitness function.

$$\boldsymbol{Fitness} = \frac{\boldsymbol{genes\ fitness}}{\boldsymbol{size\ of\ the\ individual} + \mathbf{1}}$$

Each gene of an individual will be decoded into a string which will formulate an exploit and we will append it with the given query to execute. And at the last all the genes of the current individual will be combined by appending blank spaces in between their genes to formulate another kind of exploit and it will also be injected to the query. Following image will explain the idea.

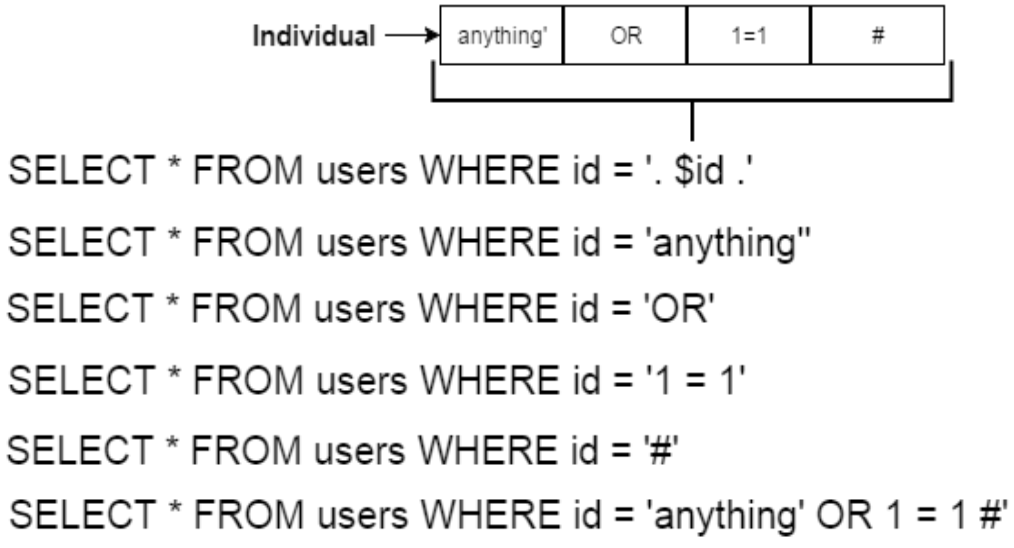

Figure 14 Generated Exploits from and Individual

For example, we have an individual as represented in Figure 10, we will create 5 different exploits from it as represented in Figure 14 and will add them to the user input variable which $id in the above example. If any of the created exploit work we will add 1 to the current fitness of the individual, after evaluating all the 5 exploits we will take the ratio and check the fitness. Maximum fitness could be 1 which means all the generated exploits worked and we have generated maximum exploits for a single query in an individual and minimum fitness could be zero which means none of the created exploit worked.

4.4 Correction of Detected SQL Injections

To avoid SQL injection in web applications many recommendations are provided by the industry professionals and in the literature [6, 31, 33, 34]. One of the most recommended approach for specifically PHP web applications is to use PDO classes instead of native SQL functions and sanitize the user input by binding them in PDO bind parameters. But most of the recommendations can be followed by the time of initial development of the application. In order to mitigate the SQL injection vulnerabilities in existing source code refactoring is a popular approach. Refactoring is the art of improving the existing code. Refactoring provides us with ways to recognize problematic code and gives us recipes for improving it [35]. We will describe the SQL injection prevention recommendations briefly in the following section and then we will describe the details how we utilized them to refactor the vulnerable code.

To mitigate from SQL injection attacks, the software tester and developers should handle the user inputs carefully because the user inputs are the potential weak links which could be exploited by attackers to harm the application. Not only precautions should be

implemented but also the user input should be validated according to the following recommendations [33] given in Table 2.

Table 2. User input validation recommendations

No.	Validation Recommendation
1	Always validate the user input's type, size along with contents and do not leave it on assumptions.
2	Verify the string variable's content.
3	Only accept the expected input values, and reject all vulnerable strings like binary value and comment characters.
4	Enforce suitable limits on the size and value type of user input.
5	The programmers should avoid the build transactions SQL queries from user input directly.
7	Use stored procedures to validate user input.
8	Without validation do not concatenate user input with queries.

In addition to the above mentioned recommendations there are few more provided by OWASP [6] and are described in Table 3.

Table 3. OWASP Recommendations for SQLI prevention.

No.	Validation Recommendation
1	Use Parameterized Queries instead of plain and simple Queries.
2	Validate user input
3	Escape all user input.
4	Enforce least privileges to database users
5	White list input validation
6	Configure error reporting

As described before in this section that most of the suggestions can be practiced by the time of initial design and development of the applications. Some of the suggestions are related to the configurations for example, suggestion number 4 in Table 3 “Enforce least privileges to database users” is a configuration setting. It means that, the regular users should be given minimum permissions to perform database operations. And suggestion number 6 in Table 3 “Configure error reporting” is also a configuration setting which means the database administrators should enable the error reporting and should log all the activities performed on database in order to track any suspicious activity or user which may cause harm to the database. Especially the database logs provide a great help and insights about the user accessing and performing operations on database. In the following section, we will describe some of the implementable suggestions with refactoring approach to mitigate SQL injection vulnerabilities in existing source code.

4.4.1 Validate User Input

By applying certain validation and filtering rules on the user input we can avoid SQL injection attacks for example filtering the SQL keywords (insert, select, and, or, etc.) from the user supplied input can prevent SQL injection attacks. We developed a function to filter and validate the user input which is described below in Figure 15. The main objective of the function is to take the user input and purify if from malicious characters. we have stored almost all possible keywords in an array, the function will iteratively check those keywords in user input and it will return false if user input will contain any of them.

Algorithm: FilterUserInput

Input:

I = User Input

Process:

```
Function FilterUserInput ($input)
{
        filter_words = array (insert, delete, update, select, and, exec, mid,
        master, or, truncate, declare, union, join, char)
        if ($input == null)
        {
                return false;
        }
        foreach ($input as $keyword)
        {
                If (strops(strtolower($input), strtolower($keyword)))
                {
                        return true;
                }
        }
        return false;
}
```

Output:

Boolean value, either the input contains any reserve word or not.

Figure 15. Algorithm to filter user input

4.4.2 Escape all User Input

It's very important to just not filter all the user supplied input. There are few characters those are legitimate to use but can be used with wrong intentions and can help in SQL injection, for example single quote (‘), double quote (“”) etc. These characters must be escaped before appending them with SQL queries. Some languages also provide built in functions for escaping the user input for example in PHP the function “mysql_real_escape_string()” is used to escape invalid characters. We define a custom

function for escaping the user input, a complete list of escape characters and the function definition is give in Figure 16.

Algorithm: EscapeUserInput

Input:

I = User Input

Process:

```
Function EscapeUserInput ($input)
{
        if (! empty($input) && is_string($input))
        {
                return str_replace (array ('\\', "\0", "\n", "\r", "'", '"', "\x1a"),
                array ('\\\\', '\\0', '\\n', '\\r', "\\'", '\\"', '\\Z'), $input);
        }
        return $input;
}
```

Output:

Input string after replacing the characters.

Figure 16. Escape User Input Function

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1 Design of Experimental Setup

To implement our described approach in section 4 and to evaluate it empirically on real time source code we developed a tool using PHP language. We used object oriented programing paradigm to develop our tool and developed different classes for all different GA processes. To generate and initialize individuals we developed an individual class which performs all individuals related operations. General functions of individual class are, generate individual, get specific gene and get fitness of the specified individual. After that, we developed a population class whose general functions are: generate population, get fittest individual, sort population, get individual and save individual. And then we developed the fitness class whose sole function is to calculate the fitness of the individuals using our designed fitness function which is described in section 4.3.1.4. And finally created an algorithm class which has the genetic operation, crossover, mutation and evolve population functions. To conduct the experiments, we collected open source projects from internet, these projects are publically available and are provided with known vulnerabilities for testing purposes. Along with these open source projects we also used some other custom developed source code for testing. All the experiments are being performed on a core i5 machine with 4 GB of RAM and running windows operating system. In following section, we will further describe about the systems used to carry out experiments and the research questions designed to answer with the help of experimental results.

5.1.1 Research Questions

In order to access the performance, applicability, usefulness and comparison of our genetic approach with existing SQLI detection techniques we defined following research questions.

RQ1. What percentage of SQL Injection vulnerabilities the proposed approach can detect, that would otherwise go undetected?

RQ2. What percentage of legitimate queries are incorrectly identified by genetic approach as vulnerable?

RQ3. How does the proposed approach performs as compared to existing SQL Injection detection techniques/tools?

RQ4. How successfully our approach refactored the vulnerable code to mitigate the SQL injection vulnerabilities.

Table 4. Vulnerable Applications Used for Empirical Evaluation

Systems	Number of Files	Number of Vulnerabilities
Bricks	61	9
bWAPP	285	20
Damn Vulnerable Web Application (DVWA)	550	33
XVWA	704	27
Twitterlike	60	9
Peruggia	15	27
WackoPicko	129	114

5.1.2 Studied Systems

To empirically evaluate our presented approach, we considered extensive open source projects those are available at [36] and are known to have certain types of vulnerabilities. Table 4 is presenting the systems used for experimental evaluations, the first system is Bricks[1] which is a web application developed in PHP and MySQL for learning and practice the security vulnerabilities in web applications. The system concentrates on varieties of usually observed application security vulnerabilities. Every "Block" has some kind of security issue which can be exploit manually or by using some automated program. Similarly, the second system presented in the table is buggy web application (bWAPP)[2] is also an application with vulnerabilities deliberately embedded in it. The key objective is to aid the systems engineers, security enthusiast, students and developers to identify and secure the web applications. In the same way, all other applications mentioned in the table: Damn Vulnerable Web Application (DVWA)[3], Hackademic Challenges Project[4], Mutillidae[5], Peruggia[6], and WackoPicko[7] all these applications are developed with known vulnerabilities and free and open source for testing the detection of vulnerabilities. one important thing to mentions here is, there is no standard benchmark available for SQL injection testing so we selected the best available options. Other thing is the application's details presented in Table 4 are known to have various SQL injection vulnerabilities but the firm number about how many SQL injection vulnerabilities are present in those

[1] http://sechow.com/bricks/index.html
[2] http://itsecgames.blogspot.be/
[3] http://www.dvwa.co.uk/
[4] https://www.owasp.org/index.php/OWASP_Hackademic_Challenges_Project
[5] http://www.irongeek.com/i.php?page=mutillidae/mutillidae-deliberately-vulnerable-php-owasp-top-10
[6] https://sourceforge.net/projects/peruggia/
[7] https://github.com/adamdoupe/WackoPicko

applications are not available. So we first manually identified the SQL injection vulnerabilities in those applications to get some estimated numbers.

5.1.3 Genetic Parameters setting and tuning

Table 5. Genetic parameters with their values

Parameter	Value
Population Size	50
Mutation Rate	10%
Crossover Rate	90%
Selection Method	Tournament Selection
Pool Size	25
Number of Iterations	30

5.2 Results and Analysis

In this section we will present the experiments conducted on source code of different files and projects. Each experiment is repeated at least three times to ensure the output and to reduce the error.

5.2.1 Results for RQ1

Table 6 is presenting the RQ1's answer which is, what percentage of SQL Injection vulnerabilities the proposed approach can detect, that would otherwise go undetected?

Other research questions will describe the comparison and usefulness of our approach and this research question is showing the efficiency of our genetic approach. The Table 6 is showing the system names in first column and their respective total SQL injection vulnerabilities in second column and finally the third column is representing the SQL

injection vulnerabilities detected by our genetic approach. It is clear from the Table 6 that our genetic approach detected most of the SQL injection vulnerabilities. Along with detection results some of the examples of vulnerable queries generated by our genetic approach are presented in Figure 17.

Table 6. Detected number of SQL injections by genetic approach

System	Total SQLI	Detected
Bricks	9	10
bWapp	20	19
DVWA	33	35
XVWA	27	30
Twitterlike	9	9
Peruggia	27	25
WackoPicko	114	118

```
sqli_1.php | SELECT * FROM movies WHERE title LIKE '% all %'

sqli_16.php | SELECT * FROM users WHERE login = 'email@mail.com' OR id=1

SELECT * FROM heroes WHERE login = '1' OR 1 # AND password =
```

Figure 17. Examples of Malicious queries detected by genetic approach

5.2.2 Results for RQ2

Our second research question which "what percentage of legitimate queries are incorrectly identified by genetic approach as vulnerable?" is aim to validate the correctness and accuracy of our genitive approach. We have presented false positives and false negatives analysis generated by our genetic approach. In Table 7 the results for false positive and false negatives are shown, first three columns are same as Table 6 which are the systems used, total SQL injection vulnerabilities in that system and the detected number of vulnerable while the fourth column is showing the false positives which means the vulnerabilities those were wrongly detected as vulnerabilities and fifth columns is showing the false negatives which means our approach missed few vulnerabilities.

Table 7. False positives and false negatives in genetic approach

System	Total SQLI	Detected	False Positive	False Negative
Bricks	9	10	1	0
bWapp	20	19	0	1
DVWA	33	35	2	0

XVWA	27	30	3	0
Twitterlike	9	9	0	0
Peruggia	27	25	0	2
WackoPicko	114	118	4	0

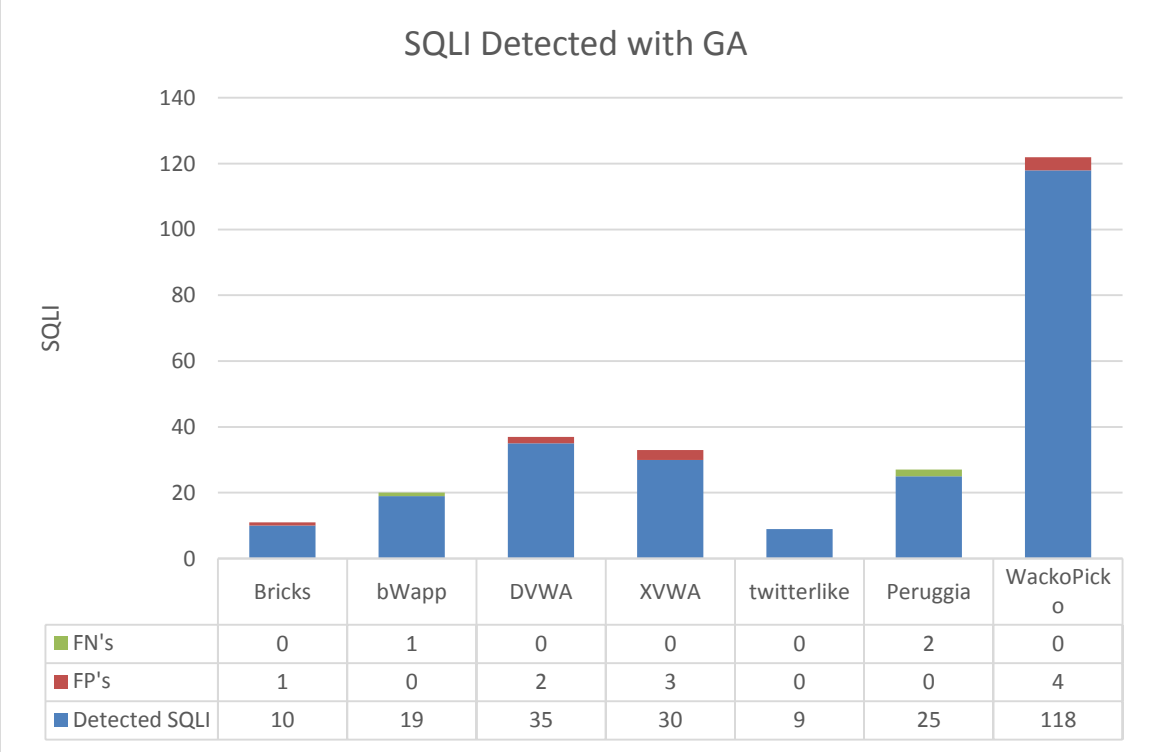

Figure 18. SQLI detected by genetic approach with false positives and false negatives

Along with tabular representation of results for second research question we have also presented them graphically in Figure 18.

5.2.3 Results for RQ3

In this section we will provide a comparative analysis of our genetic approach against RIPs which will answer our third research question "how does the proposed approach performs

as compared to existing SQL Injection detection techniques/tools?". RIPs is a famous static code analysis tool which can also detect SQL injection vulnerabilities in source code.

In this comparison analysis we did not just presented the total number of vulnerabilities detected by each approach but we also provided the false positives and false negatives analysis to better understand the overall accuracy. Table 8 is presenting the comparative analysis results first two columns are same as we shown in previous results, systems and their total vulnerabilities. the third column, is the total detected vulnerabilities by genetic approach fourth and fifth columns are showing the false positives and false negatives respectively. Sixth column is showing the total SQL injection vulnerabilities detected by RIPs. Seventh and eighth columns are showing the false positives and false negatives respectively generated by RIPs. False positives and false negatives for both our genetic approach and RIPs are graphically illustrated in graphs in Figure 19 and Figure 20. From both the tabular and graphical results, we can conclude that the genetic approach performed more efficiently as compared to RIPs. Genetic approach detected the SQL injection vulnerabilities more accurately and produced lesser false positives and false negatives as compared to RIPs.

Table 8. Comparison between Genetic Approach and RIPs

System	Total SQLI	Detected by GA	GA FP's	GA FN's	RIPS	RIPS FP's	RIPS FN's
Bricks	9	10	1	0	9	0	2
bWapp	20	19	0	1	22	2	0
DVWA	33	35	2	0	35	2	3
XVWA	27	30	3	0	33	6	1
Twitterlike	9	9	0	0	9	0	0
Peruggia	27	25	0	2	30	3	1
WackoPicko	114	118	4	0	120	6	9

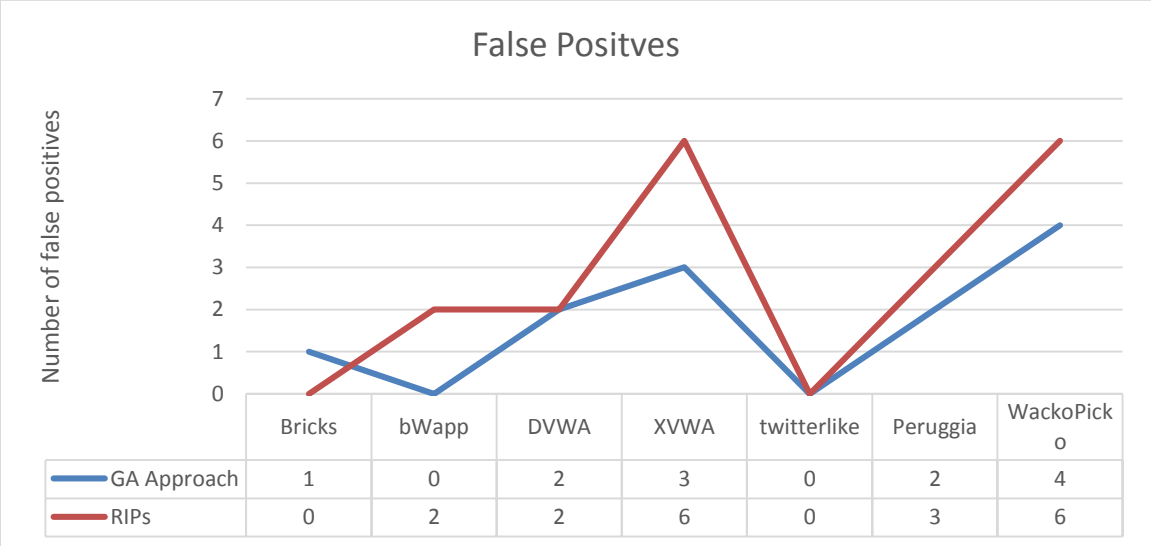

Figure 19. False positives comparison between Genetic Approach and RIPs

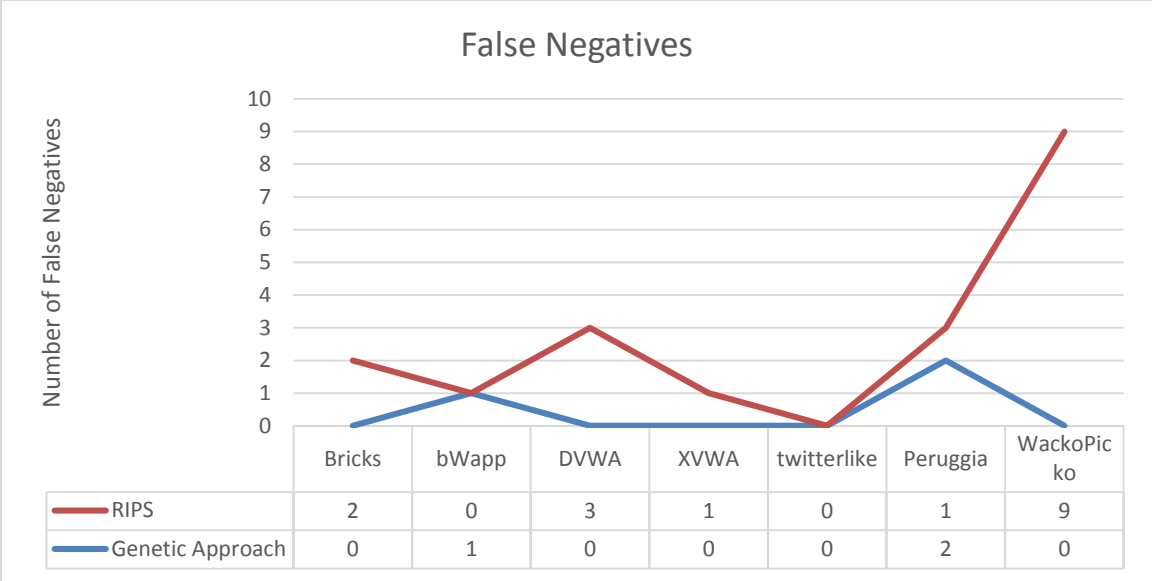

Figure 20. False negatives comparison between genetic approach and RIPs

5.2.4 Results for RQ4

After successful and efficient detection of SQL injection vulnerabilities the next challenge is to fix them, in this section we will answer our fourth research question "how successfully our approach refactored the vulnerable code to mitigate the SQL injection vulnerabilities?" to illustrate how easily and correctly we have fixed the SQL injection vulnerabilities using refactoring approach.

Automated Refactoring is very useful to correct problematic code automatically which is a laborious and time consuming job otherwise. We have defined two function to filter and escape the user supplied input and the details are provided in section 4.4.1 and 4.4.2. we then applied those functions to the detected SQL injection queries and the results are given in Table 9. The fourth column in Table 9 is showing the number of refactored SQL injection vulnerabilities. we achieved these good results for refactoring because we have done the hard part in previous step in identifying the injection vulnerabilities. so in mitigation step we just have to apply the described filtering functions.

Table 9. Refactored SQL injection Vulnerabilities

System	Total SQLI	Detected	Refactored
Bricks	9	10	10
bWapp	20	19	19
DVWA	33	35	35
XVWA	27	30	30
Twitterlike	9	9	9
Peruggia	27	25	25
WackoPicko	114	118	118

CHAPTER 6

GENETIC APPROACH TO DETECT CROSS SITE SCRIPTING (XSS)

6.1 Approach Overview

Intended behavior of any software application is the most important objective of traditional software testing approaches [37]. The attain this essential objective different scenarios and use cases are applied on the software. In this research, we extended this concept and applied it to web application's security testing. By forcing the source code of an application through different control flow paths we identified that either if any path in the given source code is vulnerable to cross site scripting. There is possibility that a certain control flow path is vulnerable to cross site scripting while the other one is safe. The control flow path will be considered vulnerable if it will be executed in response to user input without any prior validation and sanitization. To tackle the Cross Site Scripting vulnerabilities or at least circumvent them we proposed path coverage approach by generating test data using genetic algorithm to validate control flow paths in the source code. The core idea of our approach is to first identify the sensitive sinks also called vulnerable paths in the source code and generate test data for the identified vulnerable control flow paths using genetic algorithms. The primary aim of our proposed approach along with cross site scripting vulnerabilities detection is that, to leverage the test data generation by automatically generating the test data for multiple path coverage using genetic algorithm. We formulized the test data

generation problem as a search optimization problem and devised genetic algorithm accordingly to generate multiple test data for multiple path coverage in one run. Typically, it's not possible to cover all control flow paths for various reasons [38], our aim is to identify maximum cross site scripting vulnerabilities by minimal number of test data. The two principle reasons that contribute to infeasibility of covering all paths in a source code are.

- If a program has loops it may contain an infinite number of control flow paths.
- In a program number of branches produces multiple paths, and the paths are exponential to the number of branches. There is possibility that many of the paths are infeasible.

On these grounds, the path testing problem become an NP complete problem and making the coverage of all potential paths challenging [39]. Generally, a subset of paths of interest is selected by testers to cover with test data. Our interest here is to cover a subset of paths and vulnerable paths whose execution may lead to potential cross site scripting attacks. It is important to mention here that; the cross site scripting will be accounted whenever a variable or user input is used in a sensitive sink.

We selected genetic algorithms to generate test data due its proven capability of generating test data for conventional programs [38]. Genetic algorithm is very limitedly used for web security testing, an application of GA is provided by Avancini and Ceccato [40] but their work has some shortcomings. They only considered reflected cross site scripting type of vulnerabilities and not others. Their work also lacks in extensive experimentation, they presented very limited number of experiments with limited settings of genetic algorithm

used specially the mutation. To address these shortcomings, we proposed the genetic approach to generate test data for path flow problem to detect cross site scripting vulnerabilities. We actually combined two approaches; static code analysis and genetic approach for test data generation to make the detection process more efficient. The proposed approach is comprising of two steps.

- Preliminary static code analysis to identify the sensitive sinks in source code.
- Genetic approach to generate test data for the verification of path flows to exploits sensitive sink identified in previous step.

The complete overview of our two step approach is shown in Figure 21.

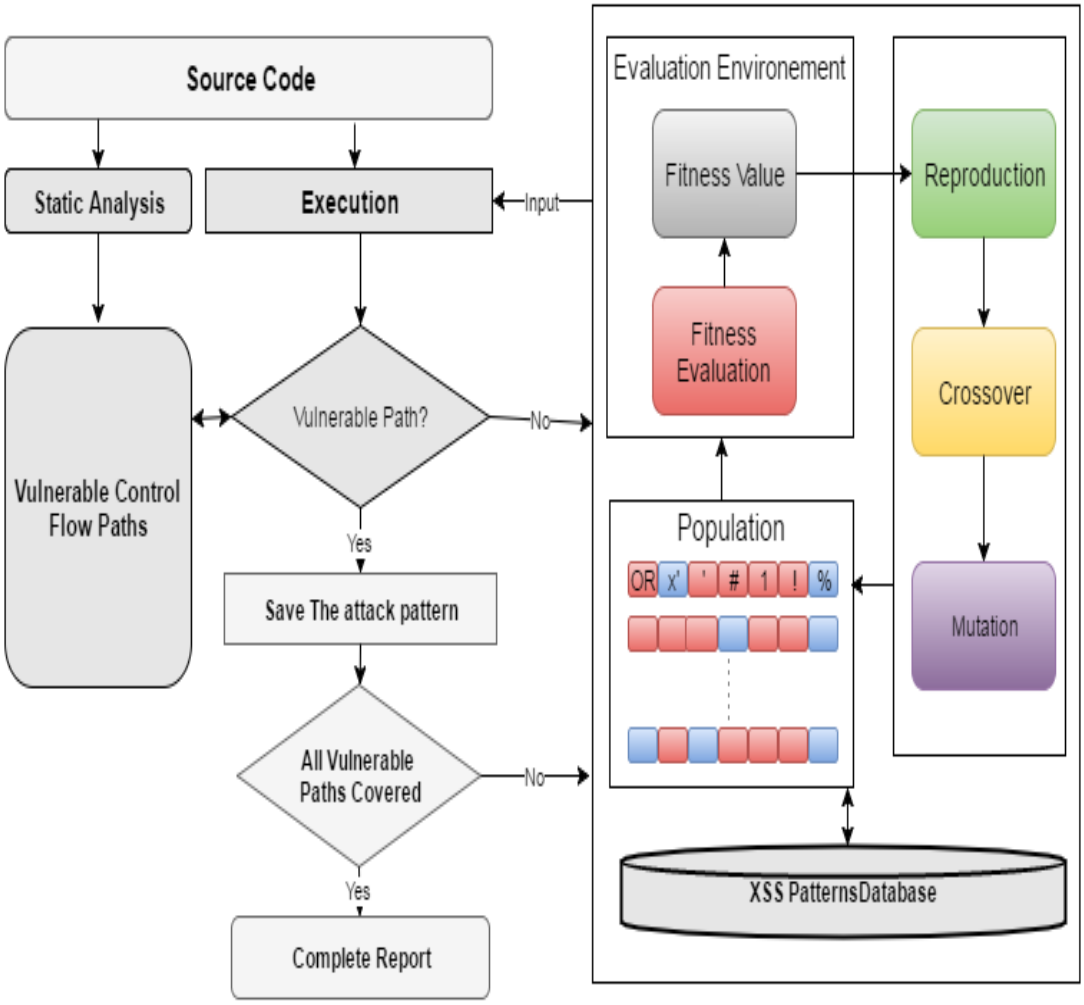

Figure 21. Proposed approach for XSS vulnerabilities detection.

6.2 Preliminary Scanning Process

The process of cross site scripting vulnerabilities detection starts with static analysis also called taint analysis of source code. The concept of static analysis was initially proposed by B. Chess and J. West [41] to leverage the manual inspection for vulnerabilities. Taint analysis tracks the tainted or untainted status of variables throughout the source code control flow. Whenever a variable is used in a sensitive sink its reported as vulnerable taint analysis has been mostly used for missing or inadequate user input validation which results in cross site scripting [10, 42]. To perform taint analysis of source code we used RIPs [43]

in our work. RIPs is a static code analysis tool which is capable of detection of various kinds of vulnerabilities including cross site scripting vulnerabilities. RIPs takes a directory or an individual files path to perform static analysis and provides statistical report about the vulnerable points in the code. If the give input is a directory not a single file it can recursively scan all the files and sub directories in the given input directory. It is worth mentioning here that the RIPs can generate false positives, that means it can indicate a flow path as vulnerable but in fact that path is not vulnerable. So we use genetic algorithm to generate test data and enforce the program to go through all the possible flow paths to verify if they are indeed vulnerable or not.

6.3 Genetic Algorithm for XSS Detection

The general details about genetic algorithm has already been discussed in earlier chapter 2.4. We will describe here the specific details and formulation of the cross site scripting test data generation using genetic algorithm and high level pseudo code is shown in Figure 22. The underlying stages and steps will remain same and those are:

- Individual Encoding/Representation,
- Generation of population from individuals,
- Fitness function definition, to evaluate the individuals in population for their ability to solve the problem,
- Individuals selection to transmit to new generation,
- Production of new individuals using genetic operators (mutation and crossover),
- Production of new population.

Before going into details of each step it's important to mention here that we made a database of cross site scripting attack patters by collecting different patterns from different resources available on internet[44, 45]. The database can be updated with new cross site scripting attack patters as they emerged with time, they can be simply added to the database. The purpose to use these attack patterns is to assist the genetic algorithm in the generation of adequate test data to reveal cross site scripting vulnerabilities. Genetic algorithm produces permutations and combinations of these attack patterns to generate inputs for the source code to enforce it to execute certain control flow paths.

```
1   Population = random_population();
2   foreach(vulnerable_paths as P)
3   {
4       while (P not executed AND attempt < max_Try)
5       {
6           selection = TournamentSelect( population ) ;
7           offspring = crossover( selection ) ;
8           population = mutate ( offspring ) ;
9           attempt = attempt + 1 ;
10      }
11  }
```

Figure 22. Genetic algorithm for path testing.

6.3.1. Individual representation

The common practice to represent individuals is using the binary strings, but it is sometimes problem specific and should be used accordingly. In our adaptation of GA, the individuals are derived from cross site scripting database, which is constructed by collecting attack patterns from different resources on internet [44, 45]. Individuals are formulated to represent the parameter values; each gene would represent a value or an attack pattern. In our algorithm genes are represented as array data structure as shown in

Figure 23. Next, algorithm will randomly choose an element from the XSS attack database and will add it to individual gene.

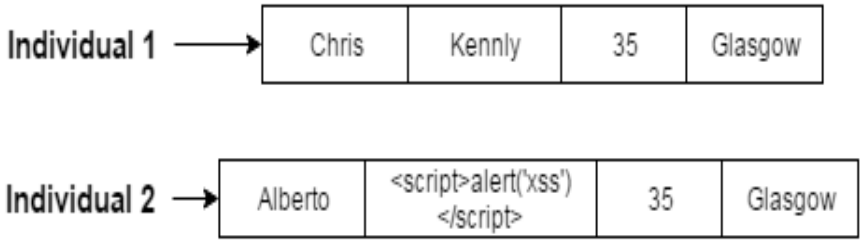

Figure 23. XSS individuals Representation

The key idea behind encoding the XSS attack patterns as array is that, when we will be calculating the fitness of an individual we will provide them as parameter values to the source code under XSS vulnerabilities testing. The size of individual is dynamic and we can set it to an appropriate length.

6.3.2. Generation of Initial Population

To generate initial population, we start by defining the size of the population. The algorithm will take the population size and generate the population of individuals. The individuals will be generated by picking genes from XSS attack database randomly and will assigned to population. Each individual in the population will be representing some parameter values and an attack scenario. Furthermore, it's important to note that all individuals in the population initially have zero fitness value. In initial population all individuals will be containing different combinations of parameter values and attack scenario.

6.3.3. Genetic Operators

Selection: To select the individuals for genetic operators, there are different selection approaches available e.g. Roulette Wheel Selection, Elitism Selection, Rank Selection,

Stochastic Universal Sampling and Tournament Selection etc. In Roulette Wheel Selection the probability of an individual to be selected in next population is proportional to its fitness, high fitness means high chances of being selected [32]. The drawback of this method is that, the chances of algorithm being stuck to local optimum solution are high. In other words, Roulette Wheel Selection will reduce the mutation properties and limit the selection pool for the algorithm, hence we are negligently limiting its potential to optimal solution. Other famous selection method is Tournament selection were we select x random individuals and pick the best one among them. This approach provides diversity in the population and the chances of algorithm being stuck in local optimum solution are low. That is why we have selected this selection method in our adaptation.

Crossover: crossover on individuals of population is performed by using the selection criteria as described in previous section4.3.1.3. Next, the algorithm will apply the crossover operation on the selected individuals and will swap their genes according to the given crossover rate. Thus, by applying the crossover function will have two new offspring inheriting the genes from two parent individuals as represented in Figure 24. Varying the crossover rate can alter the performance of genetic algorithm further details about the crossover rate we have used is provided in Table 5.

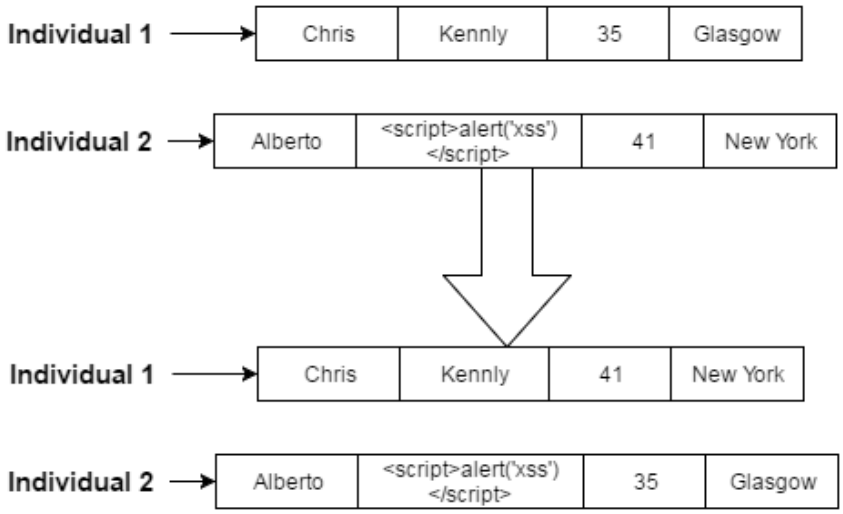

Figure 24. Crossover Operation

Mutation: Unlike to crossover operation, mutation operation is performed on a single individual. The algorithm will pick a gene from the given individual and will change it randomly with any element taken from XSS attack database according to the given mutation rate. Figure 25 is showing that, how the mutation operation changes the gene of an individual which can have different behavior. Just like crossover rate mutation rate can also vary the performance of genetic algorithm and more details about the mutation rate we used in our approach are presented in Table 5.

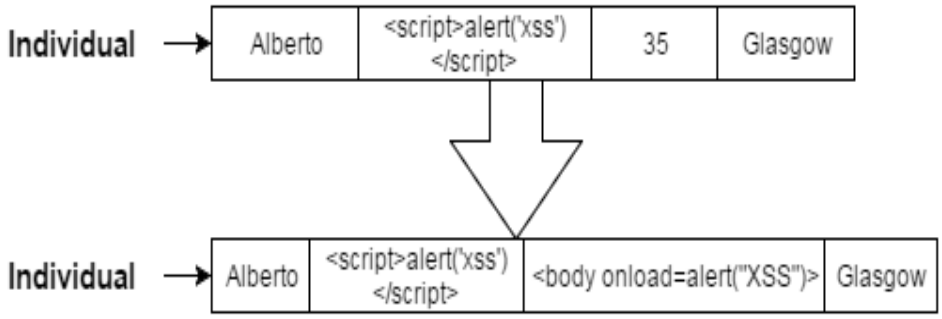

Figure 25. Mutation Operation

6.3.4. Fitness Function

The fitness function is the most crucial part of the GA, also considered as the heart of GA. Fitness function actually guides the GA towards solution. The quality of the solution generated by GA is very dependent on its evaluation (fitness) function. The development of an efficient fitness function is very much dependent on the type of individuals and the problem we are dealing with. To evaluate the fitness of an individual, we considered multiple objectives and consider it as a maximization problem. The first objective we consider to maximize for control flow path testing to detect cross site scripting vulnerabilities is, number of branches covered in a vulnerable path. Our objective is to uncover maximum number of branches in a path that leads to a sensitive sink. For example, if a path has seven branches in it our aim is to discover them all and if an individual discover them all we will assign it the maximum fitness which is 1. If the individual only able to discover 3 of them, we will assign it the fitness equals to 0.43. We formulated the following equation for branches discovery.

$$\boldsymbol{Fitness} = \frac{\boldsymbol{Discovered\ Branches}}{\boldsymbol{total\ branches\ in\ a\ path}}$$

Other objective we considered for path coverage is to generate different attack scenarios available in attack patterns database for a single path. We will keep track of each attack that is successful for a particular path. Along with that the importance of an attack scenario will be increased for that particular path for which it has successfully uncovered the sensitive sink.

CHAPTER 7

EXPERIMENTS AND RESULTS

7.1 Design of Experimental Setup and Research Questions

To implement our test data generator for XSS vulnerabilities detection described in section 6 and to evaluate it empirically on real time source code we implemented it in PHP language. We are interested in detection of cross site scripting in PHP applications the reason is that, PHP has a large share in currently active web applications over internet and is more prone to vulnerabilities. The reason of implementing our approach in PHP is, so we can execute the source code under testing in the same environment. We used object oriented programing paradigm to implement our genetic test data generator and developed different classes for all different GA processes. The architecture and underlying process is same as we described earlier in section 5.1. In addition to that, we developed some vulnerable source code to perform experiments. Finally, we have designed following research questions to answer with the results of our experiments.

RQ1. Is the genetic approach capable of covering all the vulnerable paths to reach sensitive sinks?

RQ2. How efficiently the genetic approach covered the branches in vulnerable paths?

RQ3. Is the genetic approach capable of generating multiple attack scenarios for a single vulnerable path?

7.2 Login Page Experiment

Login forms are very common in most of the modern web applications they require users to input some credentials and the application validates the user input. We designed a special kind of login form which asks the user to input his first name and last name then check if the user input contains any cross site scripting vulnerability, despite of basic check it's still vulnerable to cross site scripting the login form is shown in Figure 26.

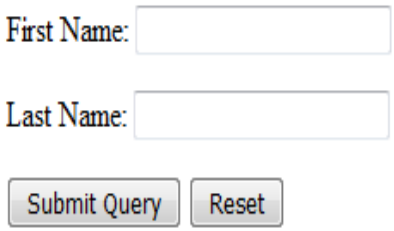

Figure 26. Simple Login Form

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Login</title>
</head>

<body >
<form action="action.php" method="post" style="margin:0 auto;">
  <p>
    <label for="fname">First Name:</label>
    <input type="text" name="fname" />
  </p>
  <p>
    <label for="lname">Last Name:</label>
    <input type="text" name="lname" />
  </p>
  <p>
    <input type="submit" name="submit" />
    <input type="reset" name="reset" />
  </p>
</form>
</body>
</html>
```

Figure 27. Login Form HTML

The form will submit the user input details to another PHP script to process further which is shown in Figure 28. The method used in this form is POST method so the form handler will be able to get all the posted values in POST global array of PHP.

```php
<?php
$fname = $_POST['fname'] ;
$lname = $POST['lname'] ;
if (substr($fname, 0, strlen("<script"))=== "<script" )
{
    $fname=htmlspecialchars($fname ) ;
}
if(isset($lname))
{
    $vulnerable = true;
}
else
{
    $vulnerable = false;
}
if ($vulnerable)
{
    $lname=htmlspecialchars ( $lname ) ;
}
echo $fname ;
if ( $vulnerable )
{
    echo $lname ;
}
?>
```

Figure 28. Action Script

To apply genetic algorithm on this script to validate its path flows we first need to perform taint analysis on it and as we mentioned in earlier section that we have used RIPs to scan source code to identify the sensitive sinks in it.

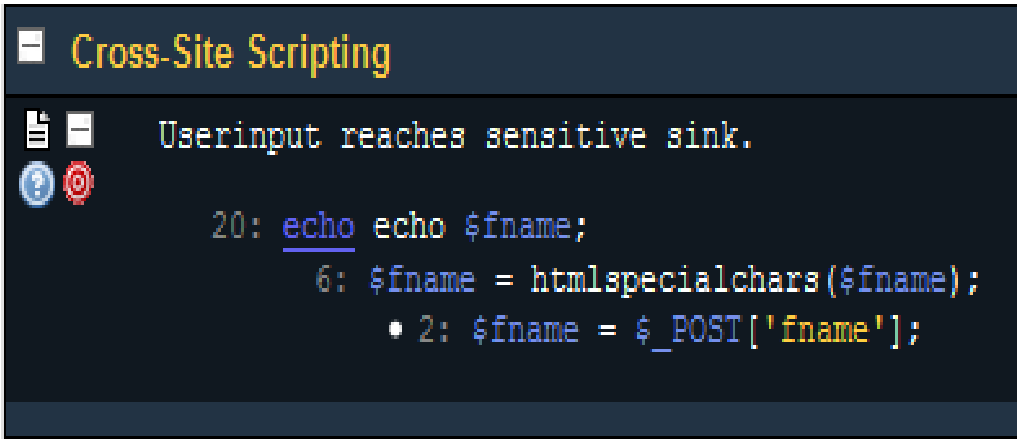

Figure 29. Sensitive Sink No. 1 in Login Script.

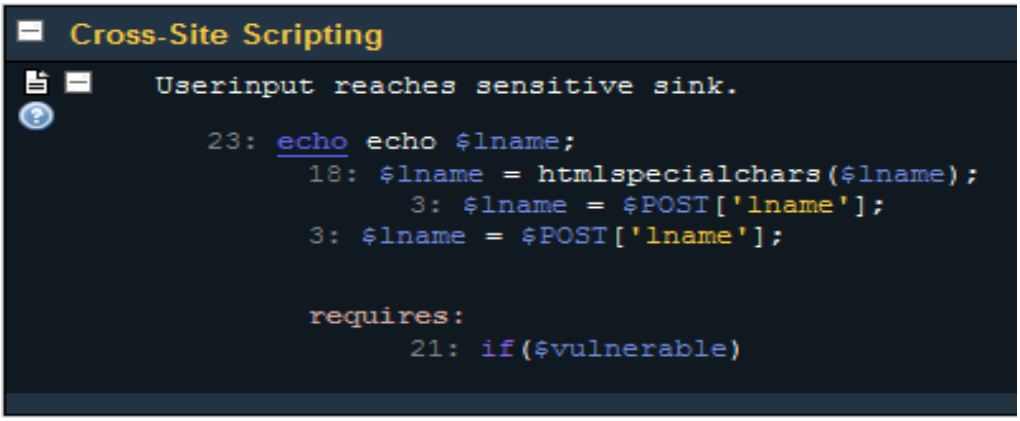

Figure 30. Sensitive Sink No. 2 in Login Script.

RIPs identified two sensitive sinks in the given script as shown in Figure 29 and Figure 30 respectively. The reason is because script is only checking for the patter “<script” in it and there are other patters as well those can successfully penetrate from this script for example “<img dynsrc="javascript:alert('XSS')">”. We converted this source code to control flow tree and formulated different paths of the code to reach sensitive sinks. The control flow tree is shown in Figure 31.

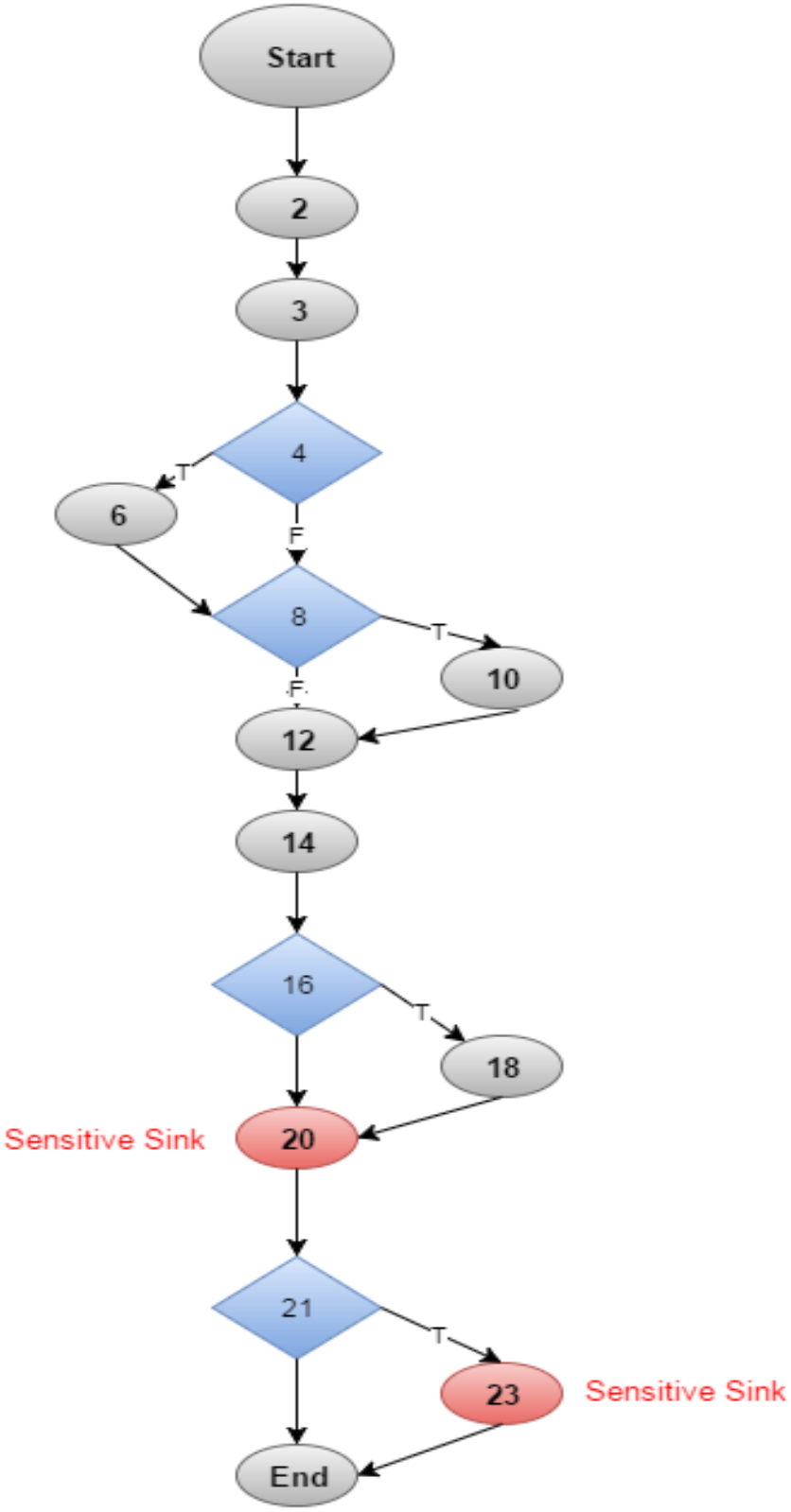

Figure 31. Control flow tree of login script.

By following the control flow tree shown in Figure 31. 12 paths generated those leads to sensitive sinks in the code and we will exploit them using attack scenarios generated by our proposed genetic approach. The generated vulnerable paths are shown in Figure 32.

```
1   4 - 8 - 12 - 14 - 16 - 20
2   4 - 8 - 12 - 14 - 16 - 20 - 21 - 23
3   4 - 6 - 8 - 12 - 14 - 16 - 20
4   4 - 6 - 8 - 12 - 14- 16 - 20 - 21 - 23
5   4 - 8 - 10 - 12 - 14 - 16 - 20
6   4 - 8 - 10 - 12 - 14 - 16 - 20 - 21 - 23
7   4 - 6 - 8 - 12 - 14 - 16 - 18 - 20
8   4 - 6 - 8 - 12 - 14 - 16 - 18 - 20 - 21 - 23
9   4 - 6 - 8 - 10 - 12 - 14 - 16 - 20
10  4 - 6 - 8 - 10 - 12 - 14 - 16 - 20 - 21 - 23
11  4 - 6 - 8 - 10 - 12 - 14 - 16 - 18 - 20
12  4 - 6 - 8 - 10 - 12 - 14 - 16 - 18 - 20 - 21 - 23
```

Figure 32. Vulnerable Paths in login script

7.2.1 Results for Login Page Experiment

In login page experiment we considered one control flow path at a time to apply attack scenarios on and to reach the sensitive sink. So, we loop through all the twelve control flow paths shown in Figure 32 one by one and tested them for cross site scripting vulnerabilities using attack scenarios generated by our genetic approach. We will explain the results and will try to answer our research questions we formulated in previous section.

7.2.2 Results for RQ1 and RQ2

The first two research questions are related to the performance of our genetic approach which concerns with, is the genetic approach capable of covering all or maximum number of paths and is the genetic approach capable of covering all or maximum number of branches in a given control flow path.

Table 10. Genetic Parameters and Their Values Used in Experiments.

Parameter	Value
Population Size	40
Mutation Rate	40%
Crossover Rate	60%
Selection Method	Tournament Selection
Pool Size	11
Number of Iterations	30

Table 11. Results for XSS RQ1 and RQ2.

Paths	No. of Branches Covered.	No. of Generations to reach sensitive sink.
1	6	4
2	8	7
3	7	10
4	9	6
5	7	8
6	9	7
7	8	5
8	10	6
9	8	11
10	10	8
11	9	10
12	11	9

Table 11 is showing the results for control flow path testing using genetic algorithm. Column one in Table 11 is showing the path number and column is showing that either the genetic algorithm covered all the branches of a path under testing or not. And the third column is showing the number of generations genetic algorithm took to cover all the branches and to reach the sensitive sink in a given path. The results are showing that the genetic algorithm was able to traverse all the paths and have covered all their branches by successfully reaching to the sensitive sink by following that path. Finally, the results are also illustrating that the genetic algorithm did not took very large number of generations to cover all the branches in a path instead it emerged in a reasonable number of generations.

7.2.3 Results for RQ3

In our third research question we aim to show the ability of our genetic approach to generate multiple attack scenarios for a single control flow path.

Table 12. Results for XSS RQ3.

Paths	No. of Branches Covered.	No. of Generations to reach sensitive sink.	No. of Attacks Generated.
1	6	4	3
2	8	7	5
3	7	10	4
4	9	6	5
5	7	8	3
6	9	7	5
7	8	5	4
8	10	6	6

9	8	11	3
10	10	8	4
11	9	10	5
12	11	9	6

7.3 Blog Comment Experiment

Blog comments is another common example in modern web applications where different users can post their views and comments. Blog comments usually ask the users to provide their names and the comment text. We designed a special kind of blog comments form which asks the user to input his name and the comment text then check if the user input contains any cross site scripting vulnerability, despite of basic check it's still vulnerable to cross site scripting the blog comment form is shown in Figure 33.

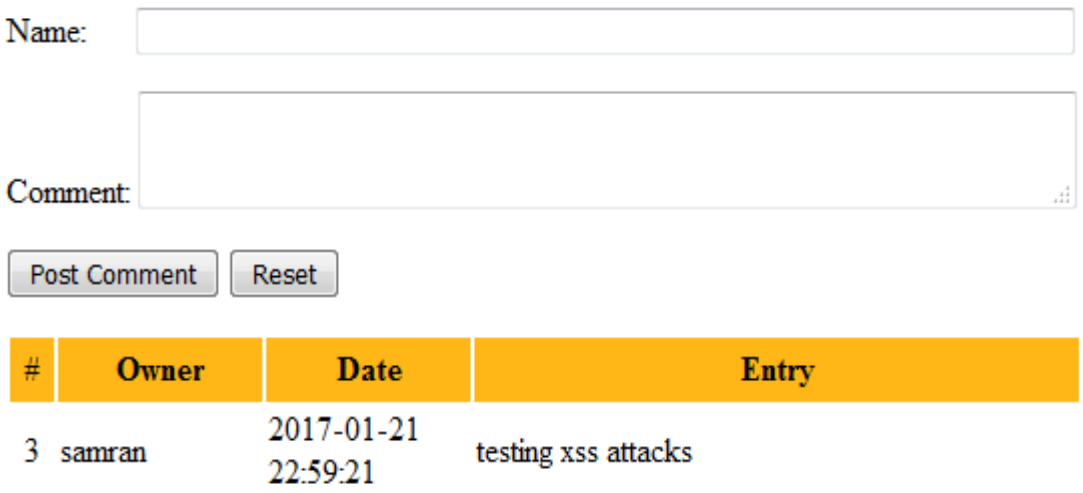

Figure 33. Comments Form

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Blog Comments</title>
</head>
<body >
<form action="blog_action.php" method="post" style="margin:0 auto;">
  <p>
    <label for="name" style="margin-right:21px;">Name:</label>
    <input type="text" name="name" style="width:463px;" />
  </p>
  <p>
    <label for="lname">Comment:</label>
    <textarea name="comment" style="width:463px;"></textarea>
  </p>
  <p>
    <input type="submit" name="submit" value="Post Comment" />
    <input type="reset" name="reset" />
  </p>
</form>
<table id="table_yellow">
<tr height="30" bgcolor="#ffb717" align="center">
    <td width="20">#</td>
    <td width="100"><b>Owner</b></td>
    <td width="100"><b>Date</b></td>
    <td width="300"><b>Entry</b></td>
</tr>
```

Figure 34. Comments Form HTML

Figure 34. is showing the html code for blog comments example. The code is is displaying the comments form which contains two fields one for name and other for comments text. The comments html form will get the user input for name and comment text and it will post those values to another script to validate and store in the database. As we mentioned this is a special example so it is still vulnerable to XSS attacks and we can illustrate the stored XSS attack here. The action script to which the comment form will post the values is shown in the figure. Once the comments are saved into database they need to be shown on the page, we have developed the script to fetch and display the comments from database which is shown in Figure 35.

```
$sql = "SELECT * FROM blog";
$recordset = mysql_query($sql) or die(mysql_error());
if(!$recordset)
{
    die("Error: " . $link->connect_error . "<br /><br />");
}
else
{
    while($row = mysql_fetch_object($recordset))
    {
        $comments = $row->entry;
        $name = $row->owner;
        if (substr($comments, 0, strlen("<script"))=== "<script" )
        {
            $comments = htmlspecialchars($comments) ;
        }
        if(isset($name))
        {
            $vulnerable = true;
        }
        else
        {
            $vulnerable = false;
        }
        if ($vulnerable)
        {
            $name = htmlspecialchars($name) ;
        }
        echo '<tr height="40">';
        echo '<td align="center">'.$row->id.'</td>';
        echo '<td>'.$name.'</td>';
        echo '<td>'.$row->date.'</td>';
        echo '<td>'.$comments.'</td>';
        echo '</tr>';
    }
}
```

Figure 35. Comments Display

The display comments script will bring all comments from the database and will show them on the html page. The script checks for only script keyword in only comments variable and which means it will allow other XSS patterns to penetrate through the script and they will become the part of the out when script will print user's name and comment text.

```php
<?php
include('config.php');
$comment = "";
$name = "";
if(isset($_POST["submit"]))
{
    $comment = $_POST["comment"];
    $name = $_POST["name"];
    if($comment == "")
    {
        $message =  "Please enter some text...";
    }
    else
    {
        $sql = "INSERT INTO blog (date, entry, owner) VALUES (now(), '".$comment."','".$name."')";
        $recordset = mysql_query($sql) or die();;
        if(!$recordset)
        {
            die("Error: ".mysql_error()."<br />");
        }
        $message = "Your entry was added to our blog!";
    }
    header('location:blog_form.php');
}
?>
```

Figure 36. Comment Action

Figure 36. is showing the comments action script, the primary work on the action script here is to get the values posted from user for name and comments and save them in the database. The script is vulnerable for both XSS and SQL injection scripts because it is not sanitizing or validating the user provided inputs at all and directly embedding them in the SQL query. User could supply and SQL injection sequence which will be executed along with the original SQL query or user can supply an XSS attack patter which will be stored in the database and will be called whenever the restore methods is called. The XSS attack patter will reside in the database until the administrator delete it, which will result in Stored or Persistent type of XSS attack. Now, we need to identify the sensitive sinks in the given code for that purpose used RIPs and the identified sensitive sinks are shown in

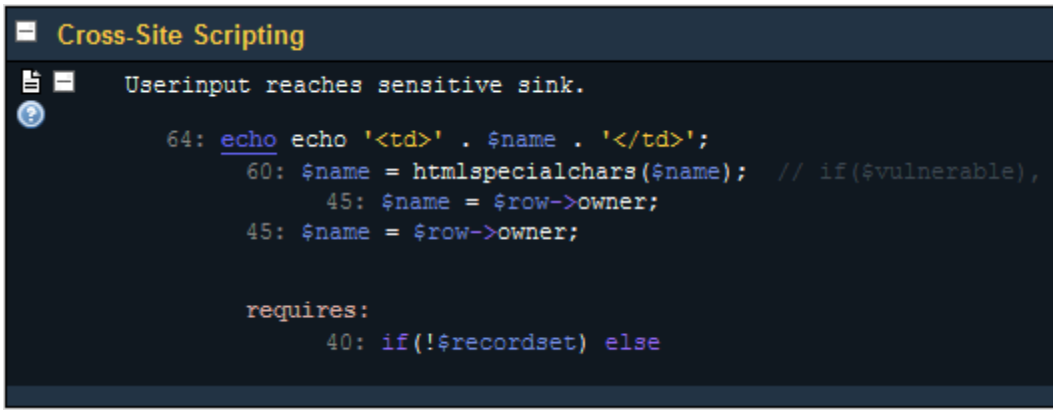

Figure 37. Sensitive Sink 1 in Comments Script

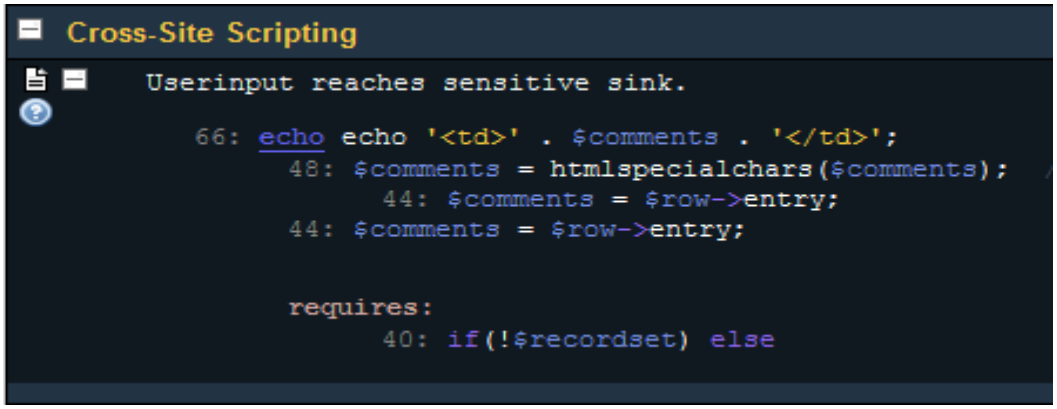

Figure 38. Sensitive Sink 2 in Comments Script

RIPs identified two sensitive sinks in the given script as shown in Figure 37 and Figure 38 respectively. The reason is because script is only checking for the pattern "<script" in the comments variable and there are other patterns as well those can successfully penetrate from this script for example "<body onload="javascript:alert('XSS')">". We converted this source code to control flow tree and formulated different paths of the code to reach sensitive sinks. The control flow tree is shown in Figure 31.

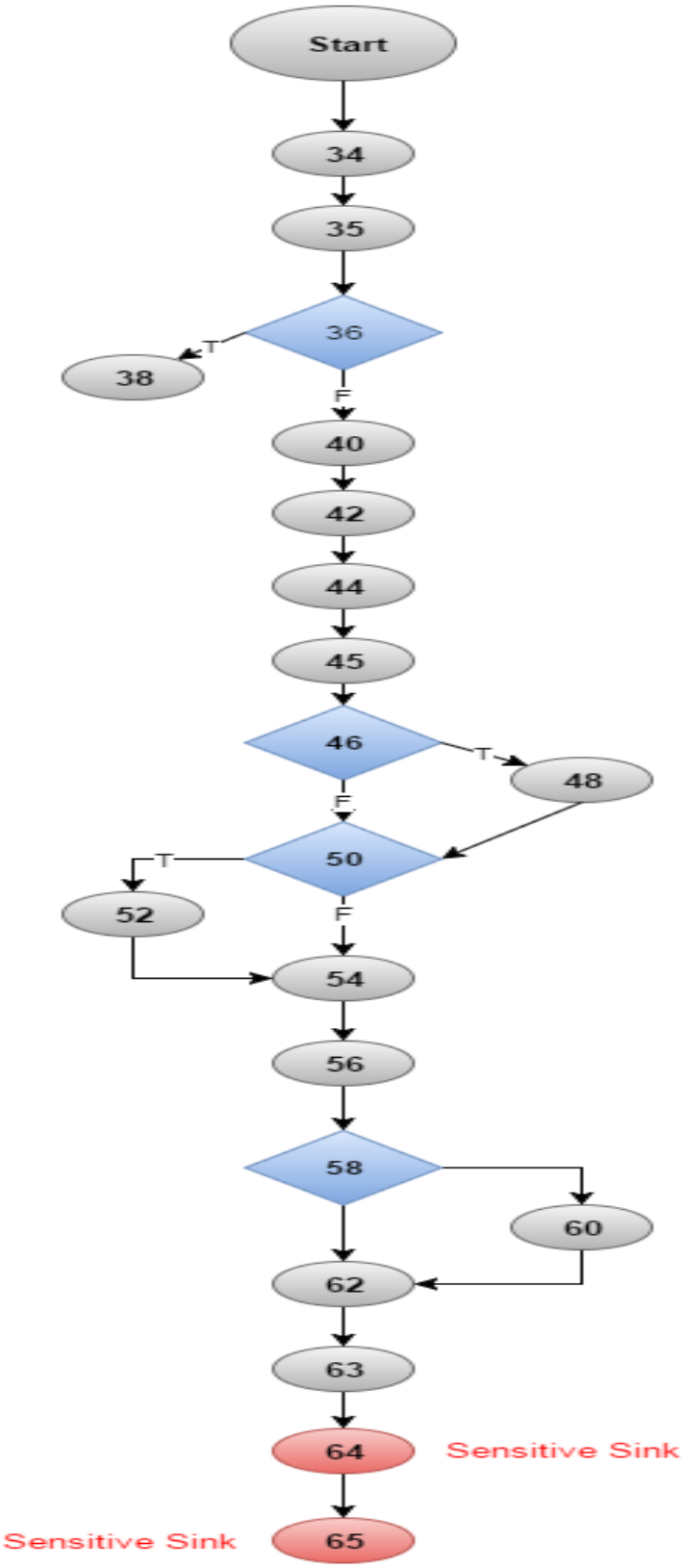

Figure 39. Comments Controll Flow

By following the control flow tree shown in Figure 39. 13 control flow paths generated those leads to sensitive sinks in the code and we will exploit them using attack scenarios generated by our proposed genetic approach. The generated vulnerable paths are shown in Figure 40.

```
1  34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 50 - 54 - 56 - 58 - 62 - 63 - 64 - 65
2  34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 50 - 54 - 56 - 58 - 62 - 63 - 64 - 65
3  34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 54 - 56 - 58 - 62 - 63 - 64 - 65
4  34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 54 - 56 - 58 - 62 - 63 - 64 - 65
5  34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 50 - 52 - 54 - 56 - 58 - 62 - 63 - 64 - 65
6  34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 50 - 52 - 54 - 56 - 58 - 62 - 63 - 64 - 65
7  34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 52 - 54 - 56 - 58 - 62 - 63 - 64 - 65
8  34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 50 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
9  34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 50 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
10 34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
11 34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
12 34 - 35 - 36 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 52 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
13 34 - 35 - 36 - 38 - 40 - 42 - 44 - 45 - 46 - 48 - 50 - 52 - 54 - 56 - 58 - 60 - 62 - 63 - 64 - 65
```

Figure 40. Comments Script Control Flow Paths

7.3.1 Results for Blog Comment Experiment

In blog comment experiment we considered one control flow path at a time to apply attack scenarios on and to reach the sensitive sink. So, we loop through all the thirteen control flow paths shown in Figure 40 one by one and tested them for cross site scripting vulnerabilities using attack scenarios generated by our genetic approach. We will explain the results and will try to answer our research questions we formulated in previous section.

7.3.2 Results for RQ1 and RQ2

The first two research questions are related to the performance of our genetic approach which concerns with, is the genetic approach capable of covering all or maximum number of paths and is the genetic approach capable of covering all or maximum number of branches in a given control flow path.

Table 13. Results for XSS RQ1 and RQ2 for Comments Experiment

Paths	No. of Branches Covered.	No. of Generations to reach sensitive sink.
1	15	7
2	16	9
3	16	8
4	17	9
5	16	7
6	17	10
7	17	11
8	16	7
9	17	8
10	17	7
11	18	10
12	18	9
13	19	8

Table 13 is showing the results for control flow path testing using genetic algorithm. Column one in Table 13 is showing the path number and column two is showing that either the genetic algorithm covered all the branches of a path under testing or not. And the third column is showing the number of generations genetic algorithm took to cover all the branches and to reach the sensitive sink in a given path. The results are showing that the genetic algorithm was able to traverse all the paths and have covered all their branches by successfully reaching to the sensitive sink by following that path. Finally, the results are

also illustrating that the genetic algorithm did not took very large number of generations to cover all the branches in a path instead it emerged in a reasonable number of generations.

7.3.3 Results for RQ3

In our third research question we aim to show the ability of our genetic approach to generate multiple attack scenarios for a single control flow path.

Table 14. Results for XSS RQ3 for Comments Experiment.

Paths	No. of Branches Covered.	No. of Generations to reach sensitive sink.	No. of Attacks Generated.
1	15	7	5
2	16	9	7
3	16	8	6
4	17	9	9
5	16	7	9
6	17	10	11
7	17	11	8
8	16	7	7
9	17	8	12
10	17	7	6
11	18	10	5
12	18	9	10
13	19	8	12

Table 14 is basically showing the capability of genetic algorithm for generation of diverse and multiple XSS attack for a single path. The third column in Table 14 is showing that, how many XSS attacks the genetic algorithm generated for a particular control flow path. It is possible that a particular attack pattern is successful for a give control flow path but the other one is not so, it is important to verify a control flow path against multiple attack patterns so insure robust security against XSS vulnerabilities. furthermore, the multiple attacks generations can point out the weaknesses and the loop holes in the script which can be fixed to avoid attacks.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Introduction

This chapter concludes the thesis by summarizing our research contributions, strengths and limitations of the study, threats to validity and identifies possible areas for future research.

8.2 Summary

In this research work, we presented literature review to identify the most recurring and most dangerous security vulnerabilities in web applications. In our work, we consider two web security vulnerabilities, namely, SQL injection and cross site scripting. SQL injection and cross site scripting security threats are considered as they are among top ten most recurring and dangerous vulnerabilities [6] for web applications. Furthermore, both security vulnerabilities have an economic impact to web applications especially application that are developed using loosely typed languages like PHP.

We also presented an indicative literature review of existing approaches and techniques to detect and correct security vulnerabilities in web applications. Majority of existing approaches focus on detecting a single security vulnerability and do not provide auto correction mechanism. The summary of existing techniques and their short falls are presented in Section 3.

8.2.1 SQL Injection Detection and Correction

In this thesis, we presented a formulation of web applications security testing as search optimization problem and used genetic algorithm to solve the resultant search optimization problem.

First, to detect SQL injection vulnerabilities, we used static source code analysis with genetic algorithm. Static source code analysis is the first phase of our approach, which will identify the potential vulnerable SQL queries in the source code. Once the static analysis has complete its process and identified all the vulnerable queries in the source code, we apply genetic algorithm to generate attack scenarios for those queries.

To start with genetic algorithm to generate attack scenario, we designed a set of exploit strings similar to other researchers [6, 31, 44]. The exploits consist of terminals, functions, behavior changing function and syntax repairing functions. We used these exploits to seed individual's population in genetic algorithm. Genetic algorithm take randomly exploits string from the set and form an SQL injection attack. We formalized a fitness function to maximize the attacks generations. The randomly generated attacks scenarios are applied on the software under test and are evaluated using the fitness function. Our experiments show promising results in detection of SQL injection vulnerabilities by generating attack scenarios.

Furthermore, we compared our results with a well-established SQL injection vulnerabilities detection tool [43]. Our approach successfully detected more SQL injection vulnerabilities with less false positive and false negative ratio. Finally, we presented refactoring approach to correct the detected SQL injection vulnerabilities. Using

refactoring approach, we applied the remedies to vulnerable queries automatically and efficiently.

8.2.2 XSS Detection and Correction

Similar to SQL injection vulnerability detection and correction, we also presented a genetic based approach to detect Cross Site Scripting (XSS) vulnerabilities. Cross site scripting vulnerabilities are classified into three types, namely, reflected, persistent and DOM based. Our presented approach can detect all three types of XSS. To start with the detection process, we use a state of the art vulnerability detection tool named RIPs [43]. The purpose of using the tool is to identify the vulnerable points in the source code; the tool we used can generate false positive and false negatives so we will make sure the authenticity of the detected vulnerabilities using test data generated by our genetic approach.

We formulated an attack patterns database by collecting the XSS attack patterns from different resources [5, 44] which can be easily extended with new attack patterns as they emerge in future. The attack patterns database is used as a seeding source for genetic algorithm's population, which is further evolved to generate XSS attack patters for source code under testing. Every source code has different control flow paths in it, which carry out different logical operations. Different user input can make the source code to follow a different control flow path, which may or may not be vulnerable to XSS vulnerabilities. To make the source code XSS free and to verify every possible control flow path is secure to XSS attacks we converted the source code into control flow paths and generated tree graphs to illustrate them.

Once we have obtained the control flow paths, we used genetic algorithm to generate test data to flow through each possible control flow path to uncover maximum vulnerable paths in the source code. Our genetic test data generation is developed in the same environment and language as the source code under testing so we can execute the vulnerable paths without any compatibility or integrity issues. We designed genetic algorithm formulation in a way that it will generate random population from XSS attack patterns database, which will undergo further genetic operations. We formulated a fitness function to minimize the XSS vulnerabilities in source code and to maximize the attack patterns generation. Our empirical evaluation shows that the proposed genetic approach generated multiple attack scenarios for a single vulnerable path and successfully uncovered the maximum XSS vulnerabilities in the source code.

8.2.3 Implications

The problem of test data generation and automated testing for uncovering the security vulnerabilities in web applications is still a challenging job. Our presented approach can leverage the laborious and difficult job of test data generation for security vulnerabilities testing in web applications. Furthermore, our approach has important and useful implications for both researchers and practitioners. Genetic algorithm is useful tool for optimizing and searching the large search space problems, but it has not been used for web security threats detection. As result of our work, there is further possibilities for the research community to apply the genetic approach to detect the security misconfiguration vulnerabilities or to detect the Broken Authentication and Session Management vulnerabilities.

For security testing practitioners our approach provides an automated way of test data generation by generating the real time SQL injection and XSS vulnerabilities attack scenario. This can help in reducing the efforts and human resources required to detect the SQL injection and XSS vulnerabilities in web applications with good efficiency. Furthermore, our approach is capable of generating multiple attack scenarios for both SQL injection XSS vulnerabilities so that, the maximum vulnerabilities can be uncovered in the source code with minimum number of false positives and false negatives.

8.3 Threats to Validity

In this section, we discuss potential threats to validity for our work. To implement the proposed approach, we used Object Oriented Programming (OOP) paradigm. To encode the solutions in genetic algorithm's chromosomes we used arrays as data structure. The performance may be improved if some other data structures are used like heaps or binary trees, it may also improve the overall execution time.

For both SQL injection and XSS vulnerabilities, we collected the attack patters/exploit strings from both published literature and well established industry de-facto security web sites. However, it is possible that we have missed some attack patters for SQL injection or XSS or for both vulnerabilities. The attack patterns may be different or evolved in future but our approach is flexible enough to add them in our database.

Finally, for XSS detection we relied on a third party scanning tool for preliminary scanning which is expected to have false positives and false negatives which may affect the results. Furthermore, we carried out limited number of experiments with medium sized control

flow paths for XSS attack generations. More experiments need to be performed by considered large size source code.

8.4 Limitations

In this section, we will describe the limitations in our research work. Our approach considered two most recurring vulnerabilities those are SQL injection and XSS. There exist more vulnerabilities those are dangerous but due to different nature of each vulnerability we did not formalized a generic approach. So the first limitation of our approach is that, it can only detect SQL injection and XSS vulnerabilities in web applications. Due to diverse nature and different modes of attack of each vulnerability, we can generalize an approach to work for all vulnerabilities but our approach can easily be extended for other vulnerabilities. In our research work, we have considered the source code analysis of only one specific language for SQL injection and XSS vulnerabilities detection that is PHP. The implemented tool is not being tested with other language's source code like classic asp and jsp.

The detection process depends on the attack patterns we have collected from different resources for both SQL injection and XSS vulnerabilities. New attack patterns may emerge in future and there is a possibility that we have missed some of the patterns. The scanner we developed for SQL injections preliminary scanning has limitations as well. The scanner can easily detect an un-sanitized user input if it is directly embedded to the SQL query. It can also even detect the un-sanitized user input if the input value is first assigned to another variable and then supplied in SQL query. But the scanner will not be able to detect the un-sanitized user input if the input is coming from different scope for example if the input is coming from a different file, class or function the scanner will not be able to backtrack it.

Finally, due to the unavailability of any standard bench mark we performed the experiments on best available resources. Performing further experiments on more diverse and large scale applications can reveals further limitations and qualities or our approach.

8.5 Future Work

As a part of future work, we are planning to expand our approach for other types of vulnerabilities among top ten specifically for, broken authentication, security misconfiguration and insecure direct object reference. Our approach is flexible to be extended for the above mentioned vulnerabilities. furthermore, our future plan also includes to enhance our static code scanner so it can back track and identify unsecure and un-sanitized user inputs in different scopes. we also planning to enhance the overall performance of our implemented tool by enhancing the attack patterns database and the fitness function along with that we are planning to try other encoding schemes for individual representations. We want to make our tool generic so we can use it with the source code of other web application languages like asp and jsp. We plan to bundle our approach in a well-designed tool, which can detect multiple security vulnerabilities in web applications with minimum false positives and false negatives along with the generation of attack scenarios so the practitioners can benefit from it to remove security vulnerabilities from their applications. Finally, our experimental analysis includes false positives and false negatives results, in future we also planning to include the true positive and true negatives analysis in experimental results.

References

[1] Symantec, "INTERNET SECURITY THREAT REPORT," April 2015.

[2] M. Labs, "McAfee Labs Threats Report," August 2015.

[3] I. Yusof and A. S. K. Pathan, "Mitigating Cross-Site Scripting Attacks with a Content Security Policy," *Computer*, vol. 49, pp. 56-63, 2016.

[4] H. Shahriar and M. Zulkernine, "MUTEC: Mutation-based testing of Cross Site Scripting," in *2009 ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 47-53.

[5] OWASP. (2016). *Cross-site Scripting (XSS)*. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[6] OWASP. (December 2, 2001). *The Open Web Application Security Project*. Available: https://www.owasp.org/

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," presented at the Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, 2004.

[8] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," presented at the Proceedings of the 2006 workshop on Programming languages and analysis for security, Ottawa, Ontario, Canada, 2006.

[9] W. G. J. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 65-81, 2008.

[10] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, 2006, pp. 6 pp.-263.

[11] S. Son and V. Shmatikov, "SAFERPHP: finding semantic vulnerabilities in PHP applications," presented at the Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security, San Jose, California, 2011.

[12] Z. Jingling and G. Rulin, "A New Framework of Security Vulnerabilities Detection in PHP Web Application," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2015 9th International Conference on*, 2015, pp. 271-276.

[13] M. K. Gupta, M. C. Govil, G. Singh, and P. Sharma, "XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications," in *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on*, 2015, pp. 2010-2015.

[14] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," presented at the Proceedings of the 8th international conference on Recent Advances in Intrusion Detection, Seattle, WA, 2006.

[15] O. Hallaraker and G. Vigna, "Detecting malicious JavaScript code in Mozilla," in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, 2005, pp. 85-94.

[16] S. Gupta and B. B. Gupta, "XSS-immune: a Google chrome extension-based XSS defensive framework for contemporary platforms of web applications," *Security and Communication Networks,* vol. 9, pp. 3966-3986, 2016.

[17] M. Alenezi and Y. Javed, "Open source web application security: A static analysis approach," in *Proceedings - 2016 International Conference on Engineering and MIS, ICEMIS 2016*, 2016.

[18] S. Cho, G. Kim, S. J. Cho, J. Choi, M. Park, and S. Han, "Runtime input validation for Java web applications using static bytecode instrumentation," in *Proceedings of the 2016 Research in Adaptive and Convergent Systems, RACS 2016*, 2016, pp. 148-152.

[19] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," presented at the Proceedings of the 12th international conference on World Wide Web, Budapest, Hungary, 2003.

[20] R. Akrout, E. Alata, M. Kaaniche, and V. Nicomette, "An automated black box approach for web vulnerability identification and attack scenario generation," *Journal of the Brazilian Computer Society,* vol. 20, pp. 1-16, 2014/01/23 2014.

[21] N. M. Vithanage and N. Jeyamohan, "WebGuardia - An integrated penetration testing system to detect web application vulnerabilities," in *Proceedings of the 2016 IEEE International Conference on Wireless Communications, Signal Processing and Networking, WiSPNET 2016*, 2016, pp. 221-227.

[22] M. R. Reddy and P. Yalla, "Mathematical analysis of penetration testing and vulnerability countermeasures," in *Proceedings of 2nd IEEE International Conference on Engineering and Technology, ICETECH 2016*, 2016, pp. 26-30.

[23] D. Scott and R. Sharp, "Abstracting application-level web security," presented at the Proceedings of the 11th international conference on World Wide Web, Honolulu, Hawaii, USA, 2002.

[24] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *SIGPLAN Not.*, vol. 41, pp. 372-382, 2006.

[25] B. Hanmanthu, B. R. Ram, and P. Niranjan, "SQL Injection Attack prevention based on decision tree classification," in *Intelligent Systems and Control (ISCO), 2015 IEEE 9th International Conference on*, 2015, pp. 1-5.

[26] H. Zhang, Y. Ding, L. Zhang, L. Duan, C. Zhang, T. Wei, G. Li, and X. Han, "SQL injection prevention based on sensitive characters," *Jisuanji Yanjiu yu Fazhan/Computer Research and Development*, vol. 53, pp. 2262-2276, 2016.

[27] N. F. N. Ibéria Medeiros, Miguel Correia, "Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives," presented at the Proceedings of the 23rd international conference on World wide web, Seoul, Korea, 2014.

[28] D. Kumar and M. Chatterjee, "MAC based solution for SQL injection," *Journal of Computer Virology and Hacking Techniques*, vol. 11, pp. 1-7, 2015/02/01 2015.

[29] Benjamin Aziz, Mohamed Bader, and C. Hippolyte, "Search-Based SQL Injection Attacks Testing using Genetic Programming," presented at the EuroGP 2016:

Proceedings of the 19th European Conference on Genetic Programming, Porto, Portugal, 2016.

[30] OWASP. (2016). *SQL Injection Bypassing WAF*. Available: https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF

[31] S. Friedl. *SQL Injection Attacks by Example*. Available: http://www.unixwiz.net/techtips/sql-injection.html

[32] F. Alabsi and R. Naoum, "Comparison of Selection Methods and Crossover Operations using Steady State Genetic Based Intrusion Detection System," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, 2012.

[33] Q. Li, Z. Zhenyuan, H. Jun, and L. Shuying, "Research of SQL injection attack and prevention technology," in *Estimation, Detection and Information Fusion (ICEDIF), 2015 International Conference on*, 2015, pp. 303-306.

[34] A. Sadeghian, M. Zamani, and A. A. Manaf, "A Taxonomy of SQL Injection Detection and Prevention Techniques," in *Informatics and Creative Multimedia (ICICM), 2013 International Conference on*, 2013, pp. 53-56.

[35] M. Fowler, *Refactoring: improving the design of existing code*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[36] OWASP. *OWASP Vulnerable Web Applications Directory Project/Pages/Offline*. Available: https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project/Pages/Offline

[37] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*: Wiley Publishing, 2011.

[38] M. A. Ahmed and I. Hermadi, "GA-based multiple paths test data generator," *Computers & Operations Research*, vol. 35, pp. 3107-3124, 2008.

[39] A. Rathore, A. Bohara, R. G. Prashil, T. S. L. Prashanth, and P. R. Srivastava, "Application of genetic algorithm and tabu search in software testing," presented at the Proceedings of the Fourth Annual ACM Bangalore Conference, Bangalore, India, 2011.

[40] A. Avancini and M. Ceccato, "Towards security testing with taint analysis and genetic algorithms," presented at the Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, Cape Town, South Africa, 2010.

[41] B. Chess and J. West, *Secure programming with static analysis*: Addison-Wesley Professional, 2007.

[42] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," presented at the Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, 2008.

[43] R. Technologies. (2016). Available: https://www.ripstech.com/

[44] Acunetix. Available: http://www.acunetix.com/websitesecurity/sql-injection/

[45] OWASP XSS. Available: https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet

Appendix

Preliminary Scanning Code for Vulnerable Queries Detection

```
<?php
function ReadDirectory($Dir)
{
    $FileInfo = new SplFileInfo($Dir);
    if($FileInfo->isFile() || $FileInfo->isDir())
    {
        if($FileInfo->isFile())
        {
            if($FileInfo->getExtension() == 'php')
            {
                $handle = @fopen($Dir, "r");
                if ($handle)
                {
                    unset($vars_array);
                    while (($buffer = fgets($handle, 4096)) !== false)
                    {
                        if(preg_match('/(SELECT|UPDATE|DELETE)/',
strtoupper($buffer)) && strstr(strtoupper($buffer), 'WHERE') &&
!strstr($buffer, '$_SESSION'))

                        {
                            $buffer = substr($buffer, strpos($buffer,
'=')+1, strlen($buffer));
                            $buffer = str_replace('"', '', $buffer);
                            $buffer = str_replace(';', '', $buffer);
                            $parser = new PHPSQLParser($buffer, true);
                            $string = '';
                            foreach($parser->parsed['WHERE'] as
$wherearray)
                            {
                                $string .= implode(" ", $wherearray);
                            }
                            if(strstr($string
,'mysql_real_escape_string') || strstr($string,
'mysqli_real_escape_string'))
                            {

                            }
                            else
                            {
                                fitnesscalc::$solution[] = $FileInfo-
>getFilename().' | '.trim($buffer);
                            }
                        }
                    }
```

```
                    if (!feof($handle))
                    {
                        echo "Error: unexpected fgets() fail\n";
                    }
                    fclose($handle);
                }
            }
        }
        else if($FileInfo->isDir())
        {
            $scan = glob(rtrim($Dir,'/').'/*');
            foreach($scan as $index=>$path)
            {
                ReadDirectory($path);
            }
        }
    }
    else
    {
        header('Location:index.php?show=file');
    }
}
?>
```

GA Process

```
<?php
error_reporting(0);session_start();
require_once('Vulnerabel_Applications/customfiles/config.php');
require_once('classes/PHP_SQL_Parser/src/PHPSQLParser.php');
require_once('classes/PHP_SQL_Parser/src/PHPSQLCreator.php');
require_once('classes/individual.php');
require_once('classes/population.php');
require_once('classes/fitnesscalc.php');
require_once('classes/algorithm.php');
require_once('includes/functions.php');
if(isset($_POST['FindSolution']) && $_POST['FindSolution'] ==
'FindSolution')
{
    $Dir = $_POST['DirName'];
    ReadDirectory($Dir);
    fitnesscalc::$solution = array_unique(fitnesscalc::$solution);

    algorithm::$uniformRate             = 0.90;
    algorithm::$mutationRate            = 0.1;
    algorithm::$poolSize                = 10;       /* crossover how
many to select in each pool to breed from */
    algorithm::$max_generation_stagnant = 50;   //maximum number of
unchanged generations terminate loop
    algorithm::$elitism                 = true;     //keep fittest
individual  for next gen
    $population_size            = 10;       //how many random
individuals are in initial population (generation 0)
    $lowest_time_s                      = 100.00;   //keeps track of
lowest time in seconds
    $gen_count                    = 0;
    $generation_stagnant                = 0;
    $most_fit                           = 0;
    $most_fit_last                      = 400;

    if(count(fitnesscalc::$solution) > 0)
    {
        $QueryCount = 0;
        foreach (fitnesscalc::$solution as $Queries)
        {
            $QueryCount++;
            fitnesscalc::$Query = $Queries;
            echo "<br>Max Fitness is : ".fitnesscalc::getMaxFitness();
            echo "<br>-------------------------------------------------
------<br>";
            // generate initial population
            $start_time = microtime(true);
            $Population = new population($population_size, true);
```

```
            // Evolve the population until optimum solution found
            while ($Population->getFittest()->getFitness() <
fitnesscalc::getMaxFitness())
            {
                $gen_count++;
                $most_fit = $Population->getFittest()->getFitness();
                $Population = algorithm::evolvePopulation($Population);

                if ($most_fit < $most_fit_last)
                {

                    echo "<br> Generation: " .$gen_count."
(Stagnant:".$generation_stagnant.") Fittest: ".
$most_fit."/".fitnesscalc::getMaxFitness() ;
                    echo "<br>  Best: ". $Population->getFittest();
                    $most_fit_last=$most_fit;
                    $generation_stagnant=0; //reset stagnant generation
counter
                }
                else
                {
                    $generation_stagnant++; //no improvement increment
may want to end early
                }
                //if( $generation_stagnant >
algorithm::$max_generation_stagnant)
                if( $generationCount > 5)
                {
                    echo "<br>-- Ending TOO MANY
(".algorithm::$max_generation_stagnant.") stagnant generations
unchanged. Ending APPROX solution below--<br>";
                    break;
                }
            }  //end of while loop

            //we're done
            $finish_time = microtime(true);

            echo "<br>Solution at generation: ".$gen_count. " time:
".round($finish_time-$start_time,2)."s";
            echo "<br>-------------------------------------------------
--------<br>";
            //echo "<br>Genes   : ".$Population->getFittest() ."<br>";
            //echo "<br>Solution: ".implode("",fitnesscalc::$solution);
//convert array to string
            echo fitnesscalc::$QueryWithAttach;
            echo "<br>-------------------------------------------------
--------<br>";
```

```
            if($QueryCount > 5)
            {
                exit;
            }
        }
    }
    else
    {
        header('Location:index.php?show=nothing');
        echo 'no vulnerable query found';
    }
}
?>
```

Individual Generation Class

```
<?php
/*************************************************************************
**
/ Class Individual : Genetic Algorithms
/*************************************************************************
**/

require_once('fitnesscalc.php');  //supporting class file

 class individual
 {
    public static $logics = array("OR", "AND");
    public static $rules = array("LIKE", "'",  "anything'", "x'='x",
"x%'", "_%", "1=(SELECT COUNT(*) FROM tabname);", "DROP TABLE
tabname;", "#", "1", "%%", "1=1", "--");
    public $defaultGeneLength = 3;
    public  $genes = array();  //defines an empty  array of genes
arbitrary length

    // Cache
    public $fitness = 0;

    public function random()
    {
        return (float)rand()/(float)getrandmax();
    }

    // Create a random individual
    public function generateIndividual($size)
    {
        //now lets randomly load the genes to the size of the array
        $mid = floor($size/2);
        for ($i=0; $i < $size; $i++ )
        {
            $this->genes[$i] = individual::$rules[rand(0,
count(individual::$rules) - 1)];
        }

        $this->genes[$mid] = individual::$logics[rand(0,
count(individual::$logics) - 1)];
    }

    /* Getters and setters */
    // Use this if you want to create individuals with different gene
lengths
    public function setDefaultGeneLength($length)
    {
```

```
        $this->defaultGeneLength = $length;
    }

    public function getGene($index)
    {
        return $this->genes[$index];
    }

    public function setGene($index,$value) {
        $this->genes[$index] = $value;
        $this->fitness = 0;
    }

    /* Public methods */
    public function size()
    {
        return count($this->genes);
    }

    public function getFitness()
    {
        if ($this->fitness == 0)
        {
            $this->fitness = FitnessCalc::getFitness($this);  //call
static method to calculate fitness
        }
        return $this->fitness;
    }

    public function __toString()
    {
        $geneString = null;
        for ($i = 0; $i <  count($this->genes); $i++)
        {
            $geneString .= $this->getGene($i);
        }

        return $geneString;
    }
}
?>
```

Population Generation Class

```
<?php
/*************************************************************************
**
/ Class Population : Genetic Algorithms
/
/*************************************************************************
**/

require_once('individual.php');  //supporting class file

class Population
{
    //class properties
    public $people=array();
    /** Constructors*/
    // Create a population
     function __construct($populationSize, $initialise=false)
     {
         if (!isset($populationSize) || $populationSize==0)
         {
             die("Must specify a populationsize > 0");
         }

        for ($i=0;$i<$populationSize; $i++)
        {
            $this->people[$i] = new individual();  //instantiate a new
object
        }
        // Initialise population
        if ($initialise)
        {
            // Loop and create individuals
            for ($i = 0; $i < count($this->people); $i++)
            {
                $new_person = new individual();
                $new_person->generateIndividual(3);
                $this->saveIndividual($i, $new_person );
            }
        }
    }

    /* Getters */

    /* find the fittest individual in this population */
    public function getFittest()
    {
        $fittest = $this->people[0];  //create a starting point for
fitness person0

        // Loop through individuals to find fittest
        for ($i = 0; $i < $this->size(); $i++)
        {
            if ($fittest->getFitness() <= $this->people[$i]-
>getFitness() )
```

```
            {
                $fittest = $this->people[$i];
                //echo "<br>Population:getFittest() is now: ".$this-
>people[$i]->getFitness();
            }

        }
        //exit;
        return $fittest;
    }
    /* Public methods */
       // get individual
    public function getIndividual($index)
    {
      return  $this->people[$index];
    }
    // Get population size
    public function size()
    {
        return count($this->people);
    }
    // Save individual
    public function saveIndividual($index, $indiv)
    {
        $this->people[$index] = $indiv;
    }
    // Sort the pool based on fitness ascending form 0...max_fitness
    // Fitness here is a cost function so lower is better fitness
    function  compareFitness($a, $b)
    {
        if($a->getFitness() == $b->getFitness() )
        {
            return 0;
        }
        return ($a->getFitness() < $b->getFitness()) ? -1 : 1;
    }
   //sort Population by fitness  , most fit (lowest cost first)
    function sortPopulation()
    {
        return usort($this-
>people,array('population',"compareFitness"));
    }
    //print population and fitness for debugging uses
     public function __toString()
     {
         $population_string = null;
         for ($i = 0; $i <  count($this->people); $i++)
         {
             $population_string.="\n Individual: ".$this->people[$i]."
Fitness:".$this->people[$i]->getFitness();
         }
         return $population_string;
     }
}//end class
?>
```

Fitness Calculation Class

```
<?php
/*************************************************************************
**
/ Class fitnesscalc : Genetic Algorithms
/
/*************************************************************************
**/
ob_flush();
require_once('individual.php');  //supporting class file

class fitnesscalc
{
    public static $solution =  array();  //empty array of arbitrary
length
    public static $Query;
    public static $QueryWithAttach;
    //public static $Dir;

    /* Public methods */
    // Set a candidate solution as a byte array

    // To make it easier we can use this method to set our candidate
solution with string of 0s and 1s
    static function setSolution($newSolution)
    {
         // Loop through each character of our string and save it in
our string  array
         fitnesscalc::$solution = str_split($newSolution);
    }

    // Calculate individuals fitness by comparing it to our candidate
solution
    // low fitness values are better,0=goal fitness is really a cost
function in this instance
    static function  getFitness($individual)
    {
        $fitness = 0;
        // Loop through our individuals genes and compare them to our
candidates
        $Queries = explode('|', fitnesscalc::$Query);
        $QuerryAppend = '';
        for ($i=0; $i < $individual->size(); $i++ )
        {
            $QuerryAppend .= $individual->genes[$i].' ';
        }

        if(preg_match('/(SELECT|UPDATE|DELETE)/', $Queries[1]))
```

```
        {
            $parser = new PHPSQLParser($Queries[1], true);
            $QuerryAppend = str_replace('tabname', $parser-
>parsed['FROM'][0]['base_expr'], $QuerryAppend);
            $parser->parsed['WHERE'][2]['base_expr'] =
"'".trim($QuerryAppend)."'";
            $creater = new PHPSQLCreator($parser->parsed, true);
            $Result = mysql_query($creater->created);
            if(mysql_errno())
            {
                $fitness = $fitness + 0;
            }
            else
            {
                if(strstr(strtoupper($Queries[1]), 'SELECT'))
                {
                    if(mysql_num_rows($Result) > 0)
                    {
                        $fitness = $fitness + 1;
                        fitnesscalc::$QueryWithAttach = $creater-
>created;
                    }
                    else
                    {
                        $fitness = $fitness + 0;
                    }
                }
                else
                {
                    $fitness = $fitness + 1;
                    fitnesscalc::$QueryWithAttach = $creater->created;
                }
            }
        }
        flush();
        return $fitness;  //inverse of cost function
    }

    // Get optimum fitness
    static function getMaxFitness()
    {
        $maxFitness = 1; //maximum matches assume each exact charaters
yields fitness 1
        return $maxFitness;
    }
}  //end class
?>
```

Genetic Operators

```
<?php
/*************************************************************************
**
/ Class geneticAlgorithm : Genetic Algorithms
/
/*************************************************************************
**/

require_once('individual.php');  //supporting class file
require_once('population.php');  //supporting class file

class algorithm
{
    /* GA parameters */
    public static $poolSize = 10;  /* When selecting for crossover how
large each pool should be */
    public static $max_generation_stagnant = 50;  /*how many unchanged
generations before we end */
    public static $elitism = true;

    /* Public methods */

    // Convenience random function
    private static function random()
    {
        return (float)rand()/(float)getrandmax();  /* return number
from 0 .. 1 as a decimal */
    }

    public static function evolvePopulation( $pop)
    {
        $newPopulation = new population($pop->size(), false);
        // Keep our best individual
        if (algorithm::$elitism)
        {
            $newPopulation->saveIndividual(0, $pop->getFittest());
        }

        // Crossover population
        $elitismOffset=0;
        if (algorithm::$elitism)
        {
            $elitismOffset = 1;
        }
        else
        {
            $elitismOffset = 0;
```

```
        }

        // Loop over the population size and create new individuals
with
        // crossover
        for ($i = $elitismOffset; $i < $pop->size(); $i++)
        {
            $indiv1 = algorithm::poolSelection($pop);
            $indiv2 = algorithm::poolSelection($pop);
            $newIndiv =  algorithm::crossover($indiv1, $indiv2);
            $newPopulation->saveIndividual($i, $newIndiv);
        }

        // Mutate population

        for ($i= $elitismOffset; $i < $newPopulation->size(); $i++)
        {
            algorithm::mutate($newPopulation->getIndividual($i));
        }
        return $newPopulation;
    }

    // Crossover individuals (aka reproduction)
    private static function  crossover($indiv1, $indiv2)
     {
       $newSol = new individual();  //create a offspring
        // Loop through genes
        for ($i=0; $i < $indiv1->size(); $i++)
        {
            // Crossover at which point 0..1 , .50 50% of time
            if (  algorithm::random() <= algorithm::$uniformRate)
            {
                $newSol->setGene($i, $indiv1->getGene($i) );
            }
            else
            {
                $newSol->setGene($i, $indiv2->getGene($i));
            }
        }
        //exit;
        return $newSol;
    }

    // Mutate an individual
    private static function mutate( $indiv)
    {
        // Loop through genes
        for ($i=0; $i < $indiv->size(); $i++)
        {
```

```
            if (algorithm::random() <= algorithm::$mutationRate)
            {
                $gene = individual::$rules[rand(0,
count(individual::$rules) - 1)];
                // Create random gene
                $indiv->setGene($i, $gene); //substitute the gene into
the individual
            }
        }//exit;
    }

    // Select a pool of individuals for crossover
    private static function poolSelection($pop)
    {
        // Create a pool population
        $pool = new population(algorithm::$poolSize, false);

        for ($i=0; $i < algorithm::$poolSize; $i++)
        {
            $randomId = rand(0, $pop->size()-1 ); //Get a random
individual from anywhere in the population
            $pool->saveIndividual($i, $pop->getIndividual( $randomId));
        }
        // Get the fittest
        $fittest = $pool->getFittest();
        return $fittest;
    }
}  //class
?>
```

Vitae

Name: Samran Naveed

Nationality: Pakistani

Date of Birth: 1/25/1990

Email: samran.naveed@live.com

Address: Sabir Colony, Okara, Pakistan

Academic Background:

- Master of Science in Computer Science, King Fahd University of Petroleum and Minerals, Saudi Arabia, May 2016
- Bachelor of Science in Computer Science, University of Engineering and Technology Lahore, Pakistan, September 2011

Work Experience:

- Software Engineer, SofDigital, Al Ahsa, Saudi Arabia 2016 to Present.
- Software Engineer, Virtual Base, Lahore, Pakistan from 2011 to 2014.

Publications:

Genetic Approach to Solve Tower of Hanoi Problem (11/2016), International Conference of Computing & Information Sciences [IC2IS 2016]

Research Interests:

- Automated Software Engineering
- Artificial Intelligence
- Machine Learning
- Web Development