



PhD-FSTC-2017-59  
The Faculty of Sciences, Technology and Communication

## DISSERTATION

Defence held on 31/08/2017 in Luxembourg  
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG  
EN INFORMATIQUE

by

SADEEQ JAN

Born on 1<sup>st</sup> September 1981 in Peshawar (Pakistan)

# AUTOMATED AND EFFECTIVE SECURITY TESTING FOR XML-BASED VULNERABILITIES

## DISSERTATION DEFENCE COMMITTEE

PROF. DR. LIONEL BRIAND, Dissertation Supervisor  
*University of Luxembourg, Luxembourg*

DR. SHIVA NEJATI, Chairman  
*University of Luxembourg, Luxembourg*

DR. FABRIZIO PASTORE, Vice Chairman  
*University of Luxembourg, Luxembourg*

PROF. DR. ALESSANDRO ARMANDO  
*Universita degli Studi di Genova, Italy*

A-PROF. DR. ALESSANDRA GORLA  
*IMDEA Software Institute, Spain*

DR. ANNIBALE PANICHELLA, Expert in an advisory capacity (Co-supervisor)  
*University of Luxembourg, Luxembourg*

Dedicated to my father  
*Muhammad Jan Khan*  
May you rest in peace.

## Abstract

Nowadays, the External Markup Language (XML) is the most commonly used technology in web services for enabling service providers and consumers to exchange data. XML is also widely used to store data and configuration files that control the operation of software systems. Nevertheless, XML suffers from several well-known vulnerabilities such as XML Injections (*XMLi*). Any exploitation of these vulnerabilities might cause serious and undesirable consequences, e.g., denial of service and accessing or modifying highly-confidential data. Fuzz testing techniques have been investigated in the literature to detect *XMLi* vulnerabilities. However, their success rate tends to be very low since they cannot generate complex test inputs required for the detection of these vulnerabilities. Furthermore, these approaches are not effective for real-world complex XML-based enterprise systems, which are composed of several components including front-end web applications, XML gateway/firewall, and back-end web services.

In this dissertation, we propose several automated security testing strategies for detecting XML-based vulnerabilities. In particular, we tackle the challenges of security testing in an industrial context. Our proposed strategies, target various and complementary aspects of security testing for XML-based systems, e.g., test case generation for XML gateway/firewall. The development and evaluation of these strategies have been done in close collaboration with a leading financial service provider in Luxembourg/Switzerland, namely *SIX Payment Services* (formerly known as *CETREL S.A.*). *SIX Payment Services* processes several thousand financial transactions daily, providing a range of financial services, e.g., online payments, issuing of credit and debit cards.

The main research contributions of this dissertation are:

- A large-scale and systematic experimental assessment for detecting vulnerabilities in numerous widely-used XML parsers and the underlying systems using them. In particular, we targeted two common XML parsers vulnerabilities: (i) *XML Billion Laughs (BIL)*, and (ii) *XML External Entities (XXE)*.

- A novel automated testing approach, that is based on constraint-solving and input mutation techniques, to detect *XMLi* vulnerabilities in XML gateway/firewall and back-end web services.
- A black-box search-based testing approach to detect *XMLi* vulnerabilities in front-end web applications. Genetic algorithms are used to search for inputs that can manipulate the application to generate malicious XML messages.
- An in-depth analysis of various search algorithms and fitness functions, to improve the search-based testing approach for front-end web applications.
- Extensive evaluations of our proposed testing strategies on numerous real-world industrial web services, XML gateway/firewall, and web applications as well as several open-source systems.

## Acknowledgements

There are a number of people who contributed to the success of my PhD in one way or the other. I would like to state my gratitude to all of them.

I would like to first thank my supervisor Prof. Dr. Lionel Briand for making me a part of his research team, and giving me an opportunity to work with and learn from the worlds renowned researchers in the field. I am very thankful for his continuous support and constructive feedback during our meetings throughout my PhD. It was a great privilege to work under his supervision.

I would be forever grateful to Dr. Cu Duy Nguyen who has been my co-supervisor for the first three years. His guidance, dedication, comments, and availability made this work possible. I have to thank him for always believing in me, encouraging, and supporting me in every possible way. For me, Cu was not only a co-supervisor but also a teacher, a mentor, and a good friend.

The best thing that happened to me in my last year of PhD is to have Dr. Annibale Panichella as my co-supervisor. I have learned a lot from his expertise on research and scientific writing skills. His counseling, encouragement, and support led to my significant progress in the last year of PhD. Working with him was not only an honor but also a pleasant experience that I really enjoyed.

I would like to thank Dr. Andrea Arcuri for his time and valuable advises. I am lucky enough to have worked with a renowned researcher like him. The numerous Skype meetings that we had in the last year of my PhD, has led to the quality research work. I am also thankful to him for making it possible to evaluate my work on a real-world industrial application.

I am thankful to the other members of my defense committee: Prof. Dr. Alessandro Armando, A-Prof. Dr. Alessandra Gorla, Dr. Fabrizio Pastore, and Dr. Shiva Nejati for their time and effort to review this dissertation. I would like to further thank Prof. Dr. Alesandro Armando and A-Prof. Dr. Alessandra Gorla for coming to Luxembourg to attend my defense.

I would like to thank SIX Payment Services (formerly CETREL S.A.) for providing me the case study system that I used for my empirical studies.

I am thankful to my colleagues and housemates who were like a family and did not make me feel any homesickness. They made these four years a pleasant and memorable experience for me.

I would like to thank my family members including: my mother, brother, wife, and kids for their love, prayers and support from far away.

Last but not least, I am really thankful to my father for his lifelong support and love. Whatever I am today, it is because of him. It was his dream to see me becoming a Doctor, but unfortunately he passed away during the last year of my PhD. I wholeheartedly dedicate this thesis to him. This thesis is for you Daji. May you rest in peace.



Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR6024200).

# Contents

<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xii</b>
<b>Acronyms</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	1
1.2 Research Problem and Motivation . . . . .	3
1.3 Research Contributions and Organization . . . . .	6
1.4 Dissemination . . . . .	7
<b>2 Background on XML-based Vulnerabilities</b>	<b>9</b>
2.1 Vulnerabilities in XML Parsers . . . . .	9
2.1.1 XML Billion Laughs (BIL) . . . . .	9
2.1.2 XML External Entities (XXE) . . . . .	11
2.2 Vulnerabilities in Web Applications and Services . . . . .	11
2.2.1 XML Injections ( <i>XMLi</i> ) . . . . .	11
<b>3 Testing XML Parsers for BIL and XXE Vulnerabilities</b>	<b>16</b>
3.1 Experimental Study . . . . .	17
3.1.1 Objectives . . . . .	17
3.1.2 Subject Selection . . . . .	18
3.1.3 Experimental Procedure . . . . .	19
3.1.4 Results on the Parsers . . . . .	20
3.1.5 Results on Open Source Systems . . . . .	25
3.1.6 Discussion and Recommendations . . . . .	29
3.2 Related Work . . . . .	30

3.3	Summary . . . . .	31
<b>4</b>	<b>Testing XML Gateway and Back-end Web Services for XML Injections</b>	<b>33</b>
4.1	Taxonomy of <i>XMLi</i> Attacks . . . . .	35
4.1.1	Type 1 - Deforming . . . . .	35
4.1.2	Type 2 - Random closing tags . . . . .	36
4.1.3	Type 3 - Replicating . . . . .	36
4.1.4	Type 4 - Replacing . . . . .	37
4.2	SOLMI: A FRAMEWORK FOR <i>XMLi</i> TESTING . . . . .	37
4.2.1	Mutation Operators . . . . .	38
4.2.2	Producing Nested Attacks . . . . .	39
4.2.2.1	Domain Constraints . . . . .	40
4.2.2.2	Attack Grammar . . . . .	42
4.2.2.3	Producing Malicious Content using Constraint Solving . . . . .	42
4.2.3	Mutation-Based Test Generation . . . . .	43
4.2.4	Test Oracle . . . . .	44
4.3	TOOL SUPPORT . . . . .	46
4.4	EXPERIMENTAL EVALUATION . . . . .	47
4.4.1	Subject Application . . . . .	48
4.4.2	Variables . . . . .	49
4.4.3	Results . . . . .	50
4.4.4	Discussion . . . . .	52
4.5	RELATED WORK . . . . .	54
4.5.1	Testing for Vulnerabilities in Web Services . . . . .	54
4.5.2	XML Test Data Generation . . . . .	55
4.6	Summary . . . . .	56
<b>5</b>	<b>Testing Front-end Web Applications for XML Injections</b>	<b>58</b>
5.1	Testing Context . . . . .	60
5.2	Approach . . . . .	60
5.2.1	Test Objectives (TOs) . . . . .	61
5.2.2	Search-based Testing . . . . .	62
5.2.2.1	GA Encoding and Reproduction . . . . .	64
5.2.2.2	Fitness Function . . . . .	64
5.2.2.3	Reducing the Search Space . . . . .	65
5.2.3	Tool Implementation . . . . .	66
5.3	Evaluation . . . . .	67



5.3.1	Research Questions . . . . .	67
5.3.2	Metrics . . . . .	68
5.3.3	Study 1: Controlled Experiments . . . . .	69
5.3.3.1	Subject Applications . . . . .	69
5.3.3.2	Test Objectives . . . . .	70
5.3.3.3	Experiment Settings . . . . .	71
5.3.3.4	Results . . . . .	73
5.3.4	Study 2: Third-party Applications . . . . .	76
5.3.4.1	Subject Applications . . . . .	76
5.3.4.2	Test Objectives . . . . .	77
5.3.4.3	Experiment Settings . . . . .	77
5.3.4.4	Results . . . . .	77
5.4	Related Work . . . . .	79
5.5	Summary . . . . .	81
<b>6</b>	<b>Improving Test Case Generation for XML Injections in Front-end Web Applications</b>	<b>82</b>
6.1	Approach . . . . .	83
6.1.1	Search-Based Testing . . . . .	83
6.1.1.1	Solution Encoding . . . . .	84
6.1.1.2	Fitness Function . . . . .	84
6.1.1.3	Solvers . . . . .	88
6.2	Empirical Studies . . . . .	93
6.2.1	Study Context . . . . .	94
6.2.2	Research Questions . . . . .	97
6.2.3	Variable Selection . . . . .	98
6.2.4	Experimental protocol . . . . .	101
6.2.5	Parameter settings . . . . .	103
6.3	Results . . . . .	104
6.3.1	RQ1: What is the best fitness function for detecting XMLi vulnerabilities? . . . . .	105
6.3.2	RQ2: What is the best solver for detecting XMLi vulnerabilities? . . . . .	112
6.3.3	RQ3: How does the proposed technique scale to industrial systems? . . . . .	116
6.4	Additional Analysis . . . . .	117
6.5	Related Work . . . . .	125
6.6	Threats to Validity . . . . .	125
6.7	Summary . . . . .	127

<b>7</b>	<b>Conclusions and Future Work</b>	<b>128</b>
7.1	Summary . . . . .	128
7.2	Future Work . . . . .	130
	<b>Bibliography</b>	<b>132</b>

# List of Figures

1.1	Vulnerabilities in XML-based System . . . . .	2
1.2	Security Testing of XML-based System in Industrial Context . . . . .	4
2.1	Example of an XML Bomb, an attack that uses the reference mechanism in XML. . . . .	10
2.2	The typical service communication scenario. . . . .	12
2.3	The user registration web form having three input fields: User Name, Password, and Email. . . . .	13
2.4	An example of an injected SOAP message. . . . .	14
3.1	An output example of Microsoft Internet Explorer that uses a XXE-vulnerable parser. The browser expands the content of a text file referred to by the XXE attack and displays it. . . . .	21
3.2	An output example of Google Chrome that recognises an input XML Bomb and raises an exception. . . . .	22
3.3	An output example of Microsoft Internet Explorer that uses a BIL-vulnerable parser. The browser expands recursively the content of an XML Bomb, occupies the system's CPU and memory, and renders the system irresponsive. . . . .	22
3.4	Memory consumption of the parsers when parsing XML Bomb files of different sizes (specified in $M_xN$ , $M$ is the number of recursive reference loops; $N$ is the number of references per loop). . . . .	23
3.5	CPU time required for the parsers to parse XML Bomb files of different sizes (specified in $M_xN$ , $M$ is the number of recursive reference loops; $N$ is the number of references per loop). . . . .	24
4.1	Workflow for producing content using constraint solvers. . . . .	43
4.2	The components of SOLMI. . . . .	46
4.3	XML Gateway and Web services . . . . .	48

## LIST OF FIGURES

---

4.4	An example of a test case generated by ReadyAPI, sensitive information was pixelated. . . . .	52
4.5	An example of a test case generated by SOLMI, sensitive information was pixelated. . . . .	54
5.1	Testing Context . . . . .	61
5.2	The overall search-based approach to generating tests for reaching the TOs. .	63
5.3	An example of output XML message created by <i>SBank</i> . . . . .	70
5.4	An example of test objective containing a malicious attack. . . . .	71
6.1	Fitness landscapes for the <i>edit distance</i> and the <i>real-coded edit distance</i> for the target string $TO = \langle t \rangle$ . . . . .	87
6.2	An example of output XML message created by <i>SBank</i> . . . . .	95
6.3	Comparison of the average success rates for SBANK (without input validation) and SSBANK (with input validation). . . . .	118
6.4	Comparison of the average success rates for SBANK and SSBANK with 1, 2 and 3 input parameters. . . . .	119
6.5	Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length. . . . .	121
6.6	Comparison of the average success rates (SR) with complete (Full) and restricted (Restricted) alphabet size. . . . .	122
6.7	Comparison of the average execution time with complete (Full) and restricted (Restricted) alphabet size. . . . .	123
6.8	Convergence rate for RGA with <i>Rd</i> and <i>Ed</i> for SSBANK with <i>SS.1.F.N</i> configuration. . . . .	124

# List of Tables

3.1	List of 13 popular XML parsers selected for our assessment. . . . .	18
3.2	The use of the parsers in open source systems, data collected from GitHub and Google Code as of August 12th, 2014. . . . .	19
3.3	Summary of <i>BIL</i> and <i>XXE</i> vulnerabilities in the parsers. We report which parsers are vulnerable to <i>BIL</i> and <i>XXE</i> . . . . .	25
3.4	A sample of 99 open sources projects among those selected in our study. The projects are accessible by appending these names to github.com, as of August 2014. . . . .	26
3.5	Tested applications that are vulnerable to BIL and XXE. . . . .	28
4.1	XML Meta-characters . . . . .	35
4.2	Summary of the proposed mutation operators for manipulating XMLs . . . .	40
4.3	Summary of the results with ReadyAPI and SOLMI . . . . .	51
5.1	Experiment Settings for Study 1: SSBank stands for <i>SecureSBank</i> , Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), the input length setting (Len), and whether the alphabet is restricted (Res. Alph.). . . . .	72
5.2	Results for <i>SBank</i> for the GA and Random Search in terms of coverage rate $C_{ga}$ , $C_{rn}$ and average execution time $T_{ga}$ , $T_{rn}$ in minutes. . . . .	73
5.3	Results for <i>SecureSBank</i> for the GA and Random Search in terms of coverage rate $C_{ga}$ , $C_{rn}$ and average execution time $T_{ga}$ , $T_{rn}$ in minutes. . . . .	74
5.4	Experiment Settings for Study 1: Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), the input length setting (Len), and whether the alphabet is restricted (Res. Alph.).	78
5.5	Results for XMLMao and <i>M</i> in terms of coverage rate $C_{ga}$ and average execution time $T_{ga}$ in minutes. . . . .	78

6.1	Description of Test Objectives . . . . .	96
6.2	Experiment Settings: Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), length of input strings in the initial population (PopLen), and whether the alphabet is restricted (Res. Alph.). . . . .	102
6.3	Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SBANK . . . . .	105
6.4	Average Odds Ratios (OR) of the Success Rate for SBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd > Ed) or worse (# Rd < Ed) than the edit distance. . . . .	106
6.5	Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SSBANK . . . . .	106
6.6	Average Odds Ratios (OR) of the Success Rate for SSBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd > Ed) or worse (# Rd < Ed) than the edit distance. . . . .	107
6.7	Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for XMLMao . . . . .	108
6.8	Average Odds Ratios (OR) of the Success Rate for XMLMao application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better (# Rd > Ed) or worse (# Rd < Ed) than the edit distance. . . . .	108
6.9	Average execution time (in minutes) results for SBANK . . . . .	109
6.10	Average execution time (in minutes) results for SSBANK . . . . .	110
6.11	Average execution time (in minutes) results for XMLMao . . . . .	111
6.12	Average $A_{12}$ statistics of the execution time for SBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better (# Rd > Ed) or worse (# Rd < Ed) than the edit distance. . . . .	112
6.13	Average $A_{12}$ statistics of the execution time for SSBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better (# Rd > Ed) or worse (# Rd < Ed) than the edit distance. . . . .	113

6.14	Average $A_{12}$ statistics of the execution time for XMLMao application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better ( $\# Rd > Ed$ ) or worse ( $\# Rd < Ed$ ) than the edit distance. . . . .	114
6.15	Ranking produced by the Friedman’s (smaller values of Rank indicate more effectiveness) when using $Rd$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.	114
6.16	Ranking produced by the Friedman’s (smaller values of Rank indicate more effectiveness) when using $Ed$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.	114
6.17	Ranking produced by the Friedman’s (larger values of Rank indicate more efficiency) when using $Rd$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.	115
6.18	Ranking produced by the Friedman’s (larger values of Rank indicate more efficiency) when using $Ed$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.	116
6.19	Results on the industrial systems . . . . .	116
6.20	Comparison of the average Success Rates (SR) of the experiments involving apps. with 1, 2 and 3 Inputs . . . . .	120
6.21	Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length . . . . .	120

# Acronyms

**XML** Extensible Markup Language

**XMLi** XML Injection

**SQLi** SQL Injection

**BIL** Billion Laughs

**XXE** XML External Entities

**XPATH** Extensible Markup Language (XML) Path Language

**SBANK** A Front-end web application for bank card processing system

**SSBANK** Secured version of Sbank

**TO** Test Objective

**HPC** High Performance Cluster

**HTTPS** HTTP over TLS

**OWASP** Open Web Application Security Project

**GA** Genetic Algorithm

**SOAP** Simple Object Access Protocol

**WSDL** Web Service Description Language



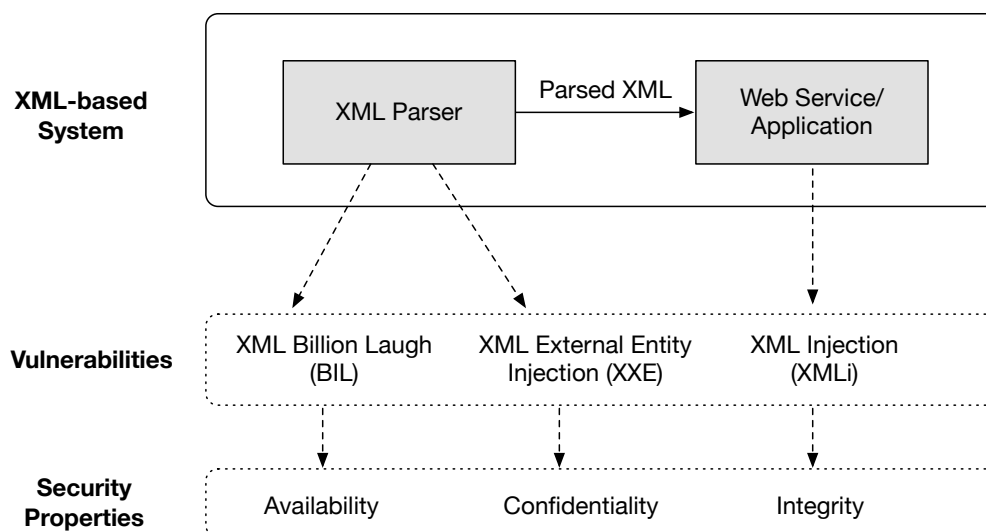
# Chapter 1

## Introduction

### 1.1. Context

Nowadays, web applications have a significant impact on our daily life, providing various services (e.g., e-commerce, mobile banking, social networking) and serving millions of users. On the downside, the growing usage of web technologies makes web services an attractive target for attackers. In 2012, the Web Application Vulnerability Statistics Report [1] revealed that 99% of web applications (within a sample of 5000 applications) had, at least, one vulnerability while 82% had, at least, one critical vulnerability. A later study by Berry and Niv [2] revealed that web applications experience, on average, 27 attacks per hour. The WhiteHat Security Report [3] found an average of 79 serious vulnerabilities per web site per year. The Open Web Application Security Project (OWASP) lists the top 10 most frequent web application security vulnerabilities. According to the ranking, injection attacks (e.g., XML, SQL, LDAP) are the most dangerous web attacks and their impact is severe. Injection attacks exploit the validation procedures for user inputs to inject malicious code that can cause the application to disclose sensitive information or behave in an unintended way.

This dissertation focuses on XML-based vulnerabilities that typically affect web services and applications as well as XML parsers. An XML-based web service or application relies on XML parsers to correctly parse the XML messages that it receives. Attacks on such web services or applications may target the application/service itself or the associated XML parser. As depicted in Figure 1.1, we identified and investigated three well-known vulnerabilities related to XML-based systems. The first vulnerability, *XML Billion Laugh (BIL)*, is used to launch a denial of service attack on the parser with the aim of undermining the



**Figure 1.1:** Vulnerabilities in XML-based System

availability of the system. The second vulnerability, called *XML External Entity (XXE)*, also uses the XML parser to disclose sensitive information affecting the confidentiality of the system. The third XML-based vulnerability, namely *XML Injection (XMLi)*, is aimed at compromising the integrity of the web service or application by injecting malicious code into XML messages. Among these three XML-based vulnerabilities, the dissertation mainly focuses on *XMLi*, which is the most critical one according to the OWASP ranking.

In particular, we investigate the challenges of security testing for XML-based vulnerabilities in an industrial context. We explore the prevalence of these vulnerabilities in web services and applications, and develop several security testing strategies to automatically and effectively detect them. The development and the evaluation of such strategies have been done in close collaboration with *SIX Payment Services* (formerly known as *CETREL S.A.*), which is a leading financial service provider in Luxembourg and Switzerland. The company provides services to its clients (e.g., banks/merchants) through a range of web services accessible via the Internet. Such financial data (e.g., credit card details) have to be properly protected to avoid attackers disclosing and manipulating them. Therefore, *SIX Payment Services* has a strong interest in thoroughly testing their web services. XML is the core part of SIX's IT infrastructure since it is used for both (i) providing services to their clients and (ii) for internal communication between sub-systems. Although the testing strategies developed in this dissertation mainly target the *SIX Payment Services*, they are meant to be general and can be applied to other systems. Indeed, our strategies have been

also evaluated with several open-source systems as well as a real-world web application<sup>1</sup> having millions of registered users.

## 1.2. Research Problem and Motivation

Several XML-based vulnerabilities have been discovered and reported over the years [4, 5]. They provide opportunities for denial of service attacks or malicious data access and manipulation. As a consequence, systems that rely on XML are at risk if they are not designed and tested properly against such attacks. Such systems include (i) XML parsers, (ii) web services/applications, and (iii) other systems that read XML input data or configurations. Despite the research and development efforts devoted to secure software and systems, XML-based vulnerabilities are still widely common [6, 7]. The presence of such vulnerabilities and their successful exploitation can be due to the lack of secure coding practices, incompetence or unawareness of developers, and incomplete/inappropriate security testing due to time and resource constraints.

In the following, we list the key challenges to face when dealing with XML-based vulnerabilities and the solutions we propose to address them.

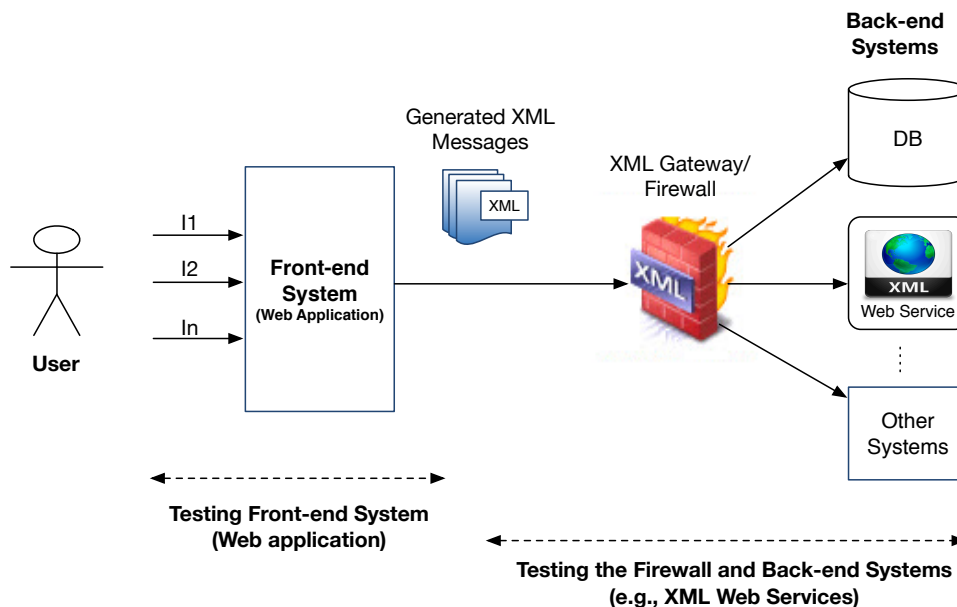
**Insecure use and configuration of XML parsers:** Due to the lack of security expertise and development time pressures, many XML parsers are not configured securely before they are deployed within web services and applications. As a result, such systems become susceptible to vulnerabilities like BIL and XXE that may result in denial of service and information leakage. While these types of vulnerabilities have been identified and discussed in the literature [4, 5, 8], no prior work assessed, in a systematic and rigorous manner, to study *BIL* and *XXE* vulnerabilities in XML parsers and the associated open-source systems.

In the first part of this dissertation, we perform a systematic and large-scale exploratory study to test most popular XML parsers that are widely used in open-source systems. We also propose a testing approach that detects attacks affecting XML parsers by measuring their impact on CPU time and memory consumption. This approach will be discussed in Chapter 3.

**Security Testing for XML Injections in Industrial Context:** Enterprise systems used in industry are composed of several components (e.g., SOAP web services, web applications). Figure 1.2 depicts a typical three-tiered XML-based business application [9]. It consists of

---

<sup>1</sup>The name of the application/company cannot be disclosed to preserve confidentiality.



**Figure 1.2:** Security Testing of XML-based System in Industrial Context

different components that work in a harmonized fashion: front-end systems (typically web applications), an XML gateway/firewall, and the back-end web services or databases. In a typical scenario, the front-ends receive user inputs and generate XML messages, which are forwarded to the XML gateway/firewall. At this stage, malicious XML messages are filtered out while the benign ones are sent to the back-end web services (or databases). Attackers may exploit XML-based vulnerabilities at any tier, e.g., targeting the front-end system or the XML gateway/firewall. Therefore, it is important to effectively test all components of such a multi-tiered enterprise system, for vulnerabilities.

### 1. Testing the front-end systems (web applications):

The security of front-end web applications is paramount as an attacker can directly interact with them via web forms. As depicted in Figure 1.2, these applications receive user inputs, produce XML messages, and send them to back-end web services for processing (e.g., as part of communications with SOAP and RESTful web services [10, 11]). If such user inputs are not properly validated, malicious XML messages can be generated that can further compromise the back-end web services where these messages are consumed. In practice, there exist approaches based on fuzz testing (e.g., ReadyAPI [12], WSFuzzer [13]), that try to send some XML *meta-characters* (e.g.,  $<$ ) and seek for abnormal responses from the systems under test (SUTs), i.e., front-end web applications in this case. These approaches might be able to detect simple

*XMLi* vulnerabilities for small applications where the user inputs are used in the generated XML messages without any modifications. However, they typically fail to detect subtler vulnerabilities, especially for complex web applications where user inputs go through various transformation procedures (e.g., encoding, sanitization) before they are utilized in generating XML messages.

To address the challenges described above, we propose an automated testing approach based on Genetic Algorithm (GA) [14] to search for effective test cases (user inputs/attacks patterns) that can circumvent the security of the front-end web applications. We first identify a set of malicious XML messages, that are known to affect the back-end. Then, we develop a strategy to determine whether such XML messages can be generated starting from the input forms of the web application in the front-end. In other words, we test the input sanitization of the front-end, as malicious input should be sanitized before generating the complete XML message. This approach will be discussed in Chapters 5 and 6.

### 2. Testing the XML gateway/firewall and back-end web services:

Testing back-end web services or databases that are protected by an XML gateway/firewall is another security challenge that state-of-the-art tools [12, 13] fail to address. The XML gateway/firewall is configured using XML schema and security policies defined by the organization to ensure that the received messages are well-formed and valid. Since state-of-the-art tools only generate simple test cases by inserting XML meta-characters (e.g.,  $<$ ) in the messages, they can be easily detected and blocked by the XML gateway/firewall when checking for validity of these messages. Therefore, techniques that generate more complex attacks able to bypass the XML gateway/firewall are needed.

To this aim, we propose a novel automated testing approach that covers a wide range of *XMLi* attacks. Our approach makes use of a constraint solver to automatically generate well-formed and valid XML messages with respect to the domain constraint (e.g., security policies used in the XML gateway/firewall), that also contain malicious content. This ensures that the generated XML messages are not easily detected by the XML gateway/firewall and have higher chances to access and compromise the back-end web services. This approach will be presented in Chapter 4.

The testing strategies, that will be presented in this dissertation, together provide a holistic approach for the automatic and effective detection of XML-based vulnerabilities.

## 1.3. Research Contributions and Organization

This dissertation presents several complementary security testing approaches to automatically and effectively detect XML-based vulnerabilities in web services and applications.

Specifically, we make the following contributions:

- A large-scale systematic experimental assessment of widely-used XML parsers and a large number of underlying systems using those parsers with respect to two common XML-based vulnerabilities: (i) *BIL* and (ii) *XXE*. We develop a testing approach based on various performance measurements (e.g., memory consumption, CPU time) to detect these two vulnerabilities. Our results provide a clear and solid scientific evidence about the extent of the threat associated with these vulnerabilities. In turn, this can help raise awareness among software developers regarding security measures for XML parsers. This contribution has been published in a conference paper [6] and is discussed in Chapter 3.
- A novel automated testing approach and tool, namely *SOLMI*, to detect *XMLi* vulnerabilities in XML gateways/firewalls and back-end web services. The approach uses constraint-solving and input-mutation techniques to generate valid but malicious XML messages (test cases) that correspond to potential injection attacks. Test cases (attacks) generated with this approach, have a higher chance of bypassing XML gateways/firewalls to reach the back-end web services. We also present a taxonomy of *XMLi* attacks, leading to the definition of XML mutation operators to be used for testing purposes. This contribution has been published in a conference paper [7] and is discussed in Chapter 4.
- A black-box testing technique for front-end systems (web applications), based on Search-Based Testing (SBT) [15], to search for sophisticated and effective test cases (attacks) to detect *XMLi* vulnerabilities. The approach first utilizes *SOLMI* to generate malicious XML messages that have a higher chance of bypassing the firewall. A Genetic Algorithm (GA) is then used to search for inputs for the front-end web application, in an attempt to generate XML messages matching the malicious ones, i.e., previously created using *SOLMI*. A fitness function based on string edit distance is used to guide the search towards the generation of malicious XML messages. This contribution has been published in a conference paper [16] and is discussed in Chapter 5.
- A technique to improve the search-based testing approach targeting front-end systems (web applications). We investigate four different search algorithms and two fitness

functions and provide an in-depth analysis by comparing all possible combinations of these fitness functions and search algorithms to determine the most effective and efficient combination for detecting *XMLi* vulnerabilities. This contribution has been submitted to *IEEE Transactions on Software Engineering* journal and is currently under review. Chapter 6 presents this approach.

- Extensive evaluation on real-world industrial web services, XML gateway/firewall, and web applications. The presented approaches in this dissertation have been evaluated on SIX's Payment Services and also on one real-world web application having millions of registered users. For generalizability, we have also carried out the evaluation of our approaches on many popular XML parsers and open source systems.

## 1.4. Dissemination

The research work presented in this dissertation has led to the following publications:

### Published papers

- Jan, Sadeeq; Nguyen, Duy Cu; Briand, Lionel (2015). "Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems." *In Proceedings of IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Vancouver, BC, 2015, pp. 233-241.

This paper is the basis for Chapter 3.

- Jan, Sadeeq and Nguyen, Cu D. and Briand, Lionel C. "Automated and Effective Testing of Web Services for XML Injection Attacks." *In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken, Germany, 2016, pp. 12-23.

This paper is the basis for Chapter 4.

- Jan, Sadeeq; Nguyen, Duy Cu; Briand, Lionel. "A Search-Based Testing Approach for XML Injection Vulnerabilities in Web Applications." *In Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, 2017, pp. 356-366.

This paper is the basis for Chapter 5.

### Submitted papers (under review)

- Jan, Sadeeq; Annibale, Panichella; Briand, Lionel. “Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications”. This paper has been submitted to *IEEE Transactions on Software Engineering* and is currently under review.

This paper is the basis for Chapter 6.



# Chapter 2

## Background on XML-based Vulnerabilities

This section describes the three well-known XML-based vulnerabilities that we target in this dissertation. First, we introduce the two vulnerabilities related to XML parsers: (i) *XML Billion Laughs (BIL)* and (ii) *XML External Entities (XXE)*. Next, we describe *XML Injection (XMLi)* vulnerabilities, which target XML-based web applications and services.

### 2.1. Vulnerabilities in XML Parsers

Standardised by the W3C [17], Document Type Definition (DTD) is a mechanism to define legal building blocks (e.g., elements, types, or content) of XML documents [18]. Many XML parsers support DTD. However, when these parsers are used improperly or the developers are unaware of such a DTD support feature, the resulting software systems might be vulnerable to DTD-based attacks. *XML Billion Laughs (BIL)* and *XML External Entities (XXE)* are two such attacks to exploit vulnerabilities in XML parsers.

#### 2.1.1 XML Billion Laughs (BIL)

BIL, also known as ‘XML Bomb’, uses the concept of XML entity reference to launch denial of service attacks. An XML entity is a variable defined for creating a reference to some content in the document or external data. In this type of attack, a block of XML is created

```
<?XML version="1.0"?>
<!DOCTYPE lolz [
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

**Figure 2.1:** Example of an XML Bomb, an attack that uses the reference mechanism in XML.

that is well-formed (conforms to W3C syntax rules for XML) [17] but forces a target parser to consume a large amount of resources (Memory or CPU).

Figure 2.1 depicts an example that can be used for this kind of attack. In this example, 10 entities ('lol'-'lol9') are created where each entity contains 10 references to the previous entity. The XML parser has to parse the entity '&lol9;' that is encountered in the root element 'lolz', which further contains several entity references '&lol8;'. Because of this recursion, when the parser resolves all entity references found in those 10 entities in this XML document, it expands to one billion copies of the first entity, occupying a very large amount of memory.

The target of BIL is a denial of service attack on XML parsers that may lead to the unavailability of systems that use such parsers. The impact of this vulnerability can be very dramatic. With a very small XML file (a few hundred bytes) an attacker can occupy several gigabytes of memory and can also keep the CPU busy for a long time, effectively preventing legitimate traffic from being processed. In order to ensure the availability of web services and systems using XML Parsers, it is very important to test the parsers for the presence of BIL vulnerabilities.

### 2.1.2 XML External Entities (XXE)

XML External Entities allow the inclusion of data dynamically from a given resource (local or remote) at the time of parsing. This feature can be exploited by attackers to include malicious data from external URIs or confidential data residing on the local system. If XML parsers are not configured to prevent or limit external entities, they are forced to access the resources specified by the URI [19].

Consider the following simple XML:

```
<?XML version="1.0"?>
<!DOCTYPE myFile [
  <!ELEMENT myFile ANY >
  <!ENTITY xe SYSTEM "file:///etc/passwd">
]>
<myFile>&xe;</myFile>
```

This is a well-formed XML document. During parsing, the parser will replace the external entity ‘&xe;’ with the content of the system file ‘/etc/passwd’, which contains confidential information and might be disclosed. Another example, if the URI ‘file:///etc/passwd’ is replaced by a link to a malicious server that never responds, the parser might end up waiting, thus causing delays in the subsequent processes.

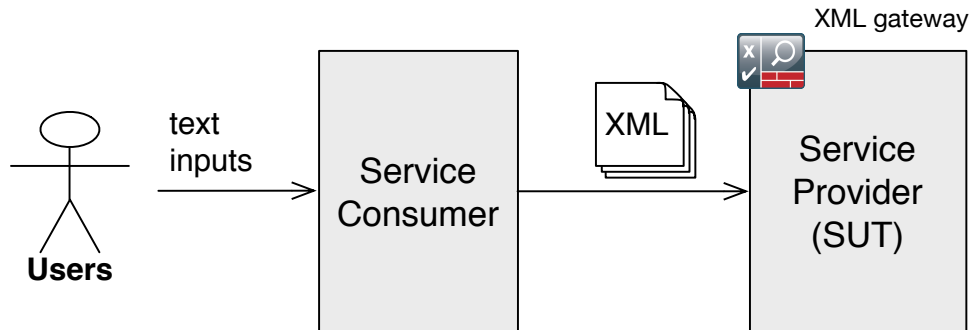
Successful exploitation of this vulnerability may result in sensitive data disclosure, denial of service, or gaining unauthorized access to the system resources. If an XML parser does not block external entity expansion and is able to access the referred content, one user may be able to gain unauthorized access the data of other users, leading to a breach of confidentiality.

## 2.2. Vulnerabilities in Web Applications and Services

In this section, we give an overview of XML injection vulnerabilities and their impact on the web services and applications.

### 2.2.1 XML Injections (*XMLi*)

XML Injection (*XMLi*) attacks are carried out by injecting pieces of XML code along with malicious content into user inputs in order to produce harmful XML messages. The aim



**Figure 2.2:** The typical service communication scenario.

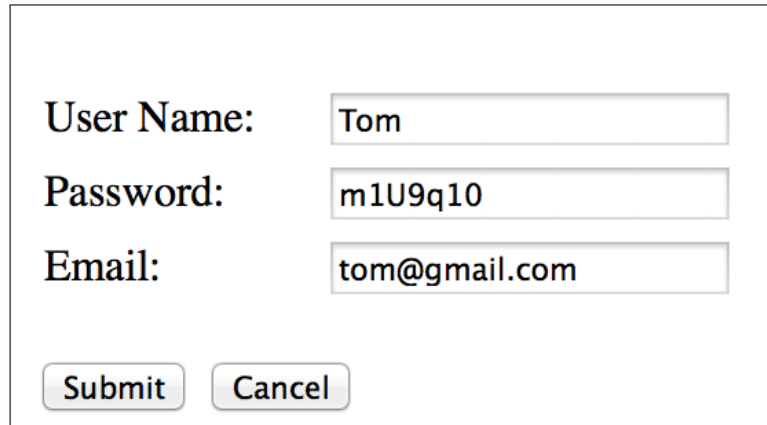
of this type of attacks is to compromise the system or system component that receives user inputs, making it malfunction (e.g. crash), or to attack other systems or subsequent components that process those injected XML messages.

The malicious content embedded in *XMLi* are called “nested” attacks. They can be SQL injection, Cross-site Scripting, or Privilege Escalation [20]. Because *XMLi* is often a vehicle to carry other attacks, the direct impact caused by *XMLi*, apart from its potential to crash a target, is negligible but the secondary impact caused by nested attacks can be very serious (e.g. *SQLi* can lead to data breaches). Therefore, *XMLi* is often discussed in combination with other types of attacks.

Figure 2.2 illustrates the typical information flow between a service consumer and a service provider. The service consumer takes user inputs (strings), generates XML messages, and sends them to the service provider. At the provider’s end, XML messages are often checked by a so-called XML gateway<sup>1</sup> for validity and security concerns before being forwarded to other service components. Following this scenario, an attacker can inject XML code and malicious content via inputs of the service consumer, which can then be injected into XML messages (*XMLi* attacks) produced by the service consumer. Such messages, if not identified and blocked by the XML gateway or the service provider, can cause a security breach.

---

<sup>1</sup>E.g., Axway’s API Gateway



The image shows a web form for user registration. It contains three input fields: 'User Name' with the value 'Tom', 'Password' with the value 'm1U9q10', and 'Email' with the value 'tom@gmail.com'. Below these fields are two buttons: 'Submit' and 'Cancel'.

**Figure 2.3:** The user registration web form having three input fields: User Name, Password, and Email.

```
<?xml version="1.0"?>
<users>
  <user>
    <username>David</username>
    <password>Lux-230</password>
    <userid>300</userid>
    <mail>david@uni.lu</mail>
  </user>
  ....
  ....
</users>
```

**Listing 2.1:** An example of an XML database for storing user registration.

Consider a concrete example in which users can register themselves through a web portal to a central service<sup>1</sup>. Once registered, a user can access different functionalities offered by the service. User registration data are stored in the XML registration database depicted in Listing 2.1. Notice that inside the XML message, each *user* element has a single child element, called *userid*, that is inserted by the application to assign privileges, users are not allowed to modify it.

The web portal has a web form (shown in Figure 2.3) with three user input fields *user-*

---

<sup>1</sup>This example is inspired by the example given by the Open Web Application Security Project (OWASP).

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <user>
      <username>Tom</username>
      <password>Un6Rkb!e</password>
      <!--
        </password>
        <userid>500</userid>
        <mail>
      -->
      <userid>0</userid>
      <mail>admin@uni.lu</mail>
    </user>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2.4: An example of an injected SOAP message.

*name*, *password*, and *email*. Each time a user submits a registration request, the application invokes the following piece of JAVA code to create a XML SOAP message and sends it to the central service. Notice that the *getNewUserId()* method is invoked to create a new user identifier and no user modification of *userid* is expected.

```
1 soapMessage = "<soap:Envelope><soap:Body>"
2 + "<user>"
3 + "<username>"+r.getParameter("username")+"</username>"
4 + "<password>"+r.getParameter("password")+"</password>"
5 + "<userid>"+getNewUserId() + "</userid>"
6 + "<mail>"+r.getParameter("mail")+"</mail>"
7 + "</user>"
8 + "</soap:Body></soap:Envelope>";
9 validate(soapMessage);
```

## 2.2 Vulnerabilities in Web Applications and Services

---

Even though there is a validation procedure at line 9, the piece of code remains vulnerable to XML injection attacks because user inputs are concatenated directly into the variable *soapMessage* without validation. Let us consider the following malicious inputs:

```
Username = Tom
Password = Un6Rkb!e</password><!--
E-mail = --><userid>0</userid><mail>admin@uni.lu
```

These inputs result in the XML message in Figure 2.4. The *userid* element is replaced with a new element having the value of “0”, which we assume is reserved to the Administrator. In this way, the malicious user *Tom* can gain administration privilege to access all functionalities of the central service. The message is well-formed and valid according to the associated XML schema (i.e., the XSD). Therefore, the validation procedure does not mitigate this vulnerability.

Similarly, by manipulating XML to exploit *XMLi* vulnerabilities, attackers can inject malicious content that can carry other types of attacks. For instance, they can replace the value “0” above with “0 OR 1=1” for an SQLi attack. If the application directly concatenates the received parameter values into a SQL Select query, the resulting query is malicious and can result in the disclosure of confidential information when executed:

```
Select * from Users where userid = 0 OR 1=1
```

## Chapter 3

# Testing XML Parsers for BIL and XXE Vulnerabilities

XML parsers are used to parse the XML messages before they can be processed by a web service or application. Various XML-based attacks on parsers have been discovered and published [4, 5, 8], however they are often overlooked in practise. Successful exploitation of such attacks can result in sensitive information leakage or denial of services.

In this chapter, we test popular XML parsers for two of the most common XML-based vulnerabilities, *XML Billion Laughs (BIL)* and *XML External Entities (XXE)*. First, among publicly available parsers, 13 of them were picked that are widely used by projects hosted at *GitHub* and *Google Code*, the two most popular open source repositories. We then submitted to each parser a set of XML files carefully selected according to a systematic test strategy. These test files can detect if the parser is vulnerable to the two XML-based attacks. Finally, we observed the behaviour of the parsers in terms of memory consumption, CPU time, and parsing results in order to assess their vulnerability. Moreover, we also investigated, based on 628 open source projects that use a vulnerable parser, whether developers properly configured the parser to thwart these XML-based attacks or adopted other mitigation measures. The obtained results are very alarming: most of the selected parsers are vulnerable to BIL and XXE attacks, and no measures are taken to prevent such attacks to harm the systems using these parsers.

The key contribution of this chapter includes a large-scale, systematic experimental assessment of widely-used and well-known XML parsers and a large number of systems that



use those parsers, with respect to two common XML-based attacks. The obtained experimental results provide an unbiased and extensive evidence of the lack of mitigation for such attacks. In turn, this can help raise awareness among software developers that appropriate security measures are required for using such vulnerable XML parsers.

The remainder of the chapter is structured as follows. Section 3.1 describes our study: procedure, results, and discussion. Section 3.2 discusses the related work and Section 3.3 concludes the work.

### 3.1. Experimental Study

This section describes our study about the security of the most popular parsers and open source software systems that use them, with respect to XML-based attacks. First, we introduce our research questions and justify our selection of subject parsers and systems. Second, we discuss the procedure that we follow to conduct the experiments. Finally, we discuss the implications of the findings and provide recommendations for developers.

#### 3.1.1 Objectives

We investigate two research questions:

- **RQ1:** To which extent are BIL and XXE attacks successful in modern XML parsers?
- **RQ2:** Do software systems, which use one of the vulnerable parsers, apply mitigation techniques for BIL and XXE attacks?

We scoped our research by focusing on parsers that are integrated with modern programming languages and are popular in open source systems. We expect such parsers to be widely used in practice. We focus on parsers since, when XML inputs are submitted to a system, an XML parser used in the system needs to treat those inputs first. And if the parser is vulnerable, the impact can be escalated to its encompassing system. In fact, there exist many other proprietary parsers. These parsers can potentially also be vulnerable to BIL and XXE if their developers lack knowledge about such attacks. However, they are out of the scope of our study.

**Table 3.1:** List of 13 popular XML parsers selected for our assessment.

No	Parser	Language	Version	Short Description
1	JDOM2	Java	2.0	An XML library built specifically for Java, available at <a href="http://www.jdom.org">www.jdom.org</a>
2	NanoXML	Java	2.2	A small non-validating parser for Java, available at <a href="http://nanoxml.sourceforge.net/orig">nanoxml.sourceforge.net/orig</a>
3	NanoXML-LITE	Java	2.2	A successor NanoXML built for performance
4	Std-DOM	Java	1.7	The standard DOM parser built in Sun/Oracle Java
5	Std-SAX	Java	1.7	The standard SAX parser built in Sun/Oracle Java
6	Std-STAX	Java	1.7	The standard STAX parser built in Sun/Oracle Java
7	WOODSTOX	Java	4.2	A high-performance XML processor, available at <a href="http://woodstox.codehaus.org">woodstox.codehaus.org</a>
8	XERCES-JDOM	Java	2.11	The Apache Xerces2 DOM parser, available at <a href="http://xerces.apache.org/xerces2-j/dom.html">xerces.apache.org/xerces2-j/dom.html</a>
9	LXML-ETREE	Python	3.3.5	A Python XML parser available at <a href="http://lxml.de">lxml.de</a>
10	Std-ETREE	Python	2.7.6	The standard XML parser built in Python
11	PERL(XML::LibXML)	Perl	5.18.2	A Perl Binding for libxml2, tested in OSX 10.9
12	PHPDOM	PHP	5.5.9	The standard DOM parser built in PHP
13	MSXML (DOMDocument)	C#, Javascript,...	8.0.0	The Microsoft XML parser (MSXML) widely used in Windows

### 3.1.2 Subject Selection

We first selected the parsers that come with modern programming languages, including Java, Python, PHP, Perl, C#. Then, we expanded our selection to widely-used, open-source XML parsers. In total, we selected the 13 most commonly used parsers, e.g., the standard Java DOM, Python ETree, Microsoft XML parser (MSXML). Table 3.1 lists these parsers, their current versions and languages, and provides short descriptions.

We evaluated the adoption of the selected parsers in GitHub<sup>1</sup> and Google Code<sup>2</sup> to assess how widely they are used in practice. GitHub and Google Code are highly popular open source hosting systems. Though there exists a few more project hosting systems, such as [sourceforge.net](http://sourceforge.net), we focused on GitHub and Google Code since they do index source code very well, thus making it easier to query for the use of XML parsers in the source code of hosted projects.

On GitHub, for the Java parsers we used its search feature to query for the XML parsing classes. For the other parsers, the queries are conjunctions of the name of the corresponding XML processing classes or libraries and the names of the methods that parse XML inputs, e.g., “`xml.etree.ElementTree`” AND “`parse`”. On Google Code, we used Google Search<sup>3</sup> with a *site* directive to narrow the search to solely *code.google.com*, and the queries were similar to those for GitHub. For both repositories, we filtered the results to the specific language that a parser supports. Table 3.2 shows the frequency with which these parsers on GitHub and Google Code were adopted. These numbers might be over-approximated since the search

<sup>1</sup><https://github.com>

<sup>2</sup><https://code.google.com>

<sup>3</sup><http://google.com>

### 3.1 Experimental Study

**Table 3.2:** The use of the parsers in open source systems, data collected from GitHub and Google Code as of August 12th, 2014.

Parser	Query	GitHub	Google Code
JDOM2	org.jdom2.input.SAXBuilder	2,861	9,380
NanoXML	net.n3.nanoxml.IXMLParser	1,410	291
NanoXML-LITE	nanoxml.XMLElement	6,057	4,380
Std-DOM	javax.xml.parsers.DocumentBuilder	112,638	58,900
Std-SAX	javax.xml.parsers.SAXParser	43,307	11,200
Std-STAX	javax.xml.stream.XMLStreamReader	84,826	4,840
WOODSTOX	org.codehaus.stax2.XMLStreamReader2	252	251
XERCES-JDOM	org.apache.xerces.parsers.DOMParser	3,444	1,440
LXML-ETREE	“lxml import etree” + parse	16,012	21,200
Std-ETREE	“xml.etree.ElementTree” + parse	27,905	43,100
PERL(XML::LibXML)	“XML::LibXML” + parse_file	1,024	990
PHPDOM	DOMDocument + loadXML	71,217	32,300
MSXML (DOMDocument)	MSXML.DOMDocument + load	24,671	565
<b>Total</b>		395,624	188,837

can return code that was commented out or unused (discussed in Section 3.1.5). The total number of adoptions of the parsers in both repositories goes above half a million. Except *WOODSTOX*, which is adopted about 500 times, the others are much more frequently used, ranging from a few thousand to a hundred thousand times. This clearly shows that the selected parsers are widely used.

#### 3.1.3 Experimental Procedure

Our experimental procedure consisted of the following steps: (i) writing code to invoke each parser and pass XML files as input, (ii) preparing representative XML input files that can apply *BIL* and *XXE* attacks, and (iii) running each parser to parse every prepared XML input files and analysing CPU time, memory used, and the outputs to determine whether the parser is vulnerable to *BIL* or *XXE*. We define a parser as *XXE*-vulnerable if it attempts to expand the content of an XML input file to include a system file, a user file, or an external resource (such as from a URL to a web page). We consider a parser as *BIL*-vulnerable if it requires exceptionally high CPU time and memory when parsing XML Bombs in comparison to parsing regular XML files (inputs that a system expects) of a similar size.

For each parser we wrote code that invokes the parser and gives a file path as input to the parser. We made sure that such code is minimal: all it does is to instantiate the parser class and invoke a parsing method, without changing any default property of the parser. We observed that this is a common usage pattern of developers when they adopt a parser, keeping every property to default. Following is an excerpt of code that we wrote to invoke Std-DOM:

```
...
try {
    df= DocumentBuilderFactory.newInstance();
    docBuilder= df.newDocumentBuilder();
    docBuilder.parse(inputFile);
} catch (Exception e){
...

```

For the *BIL* category, we wrote a simple tool that can generate XML files (XML bombs) of desired size to reveal *BIL* vulnerabilities. Each XML bomb is characterised by the number of recursive reference loops and the number of references per loop. The example in Section 2.1.1 has a size of 10x10. In our experiments, we prepared a set of 10 XML bombs of sizes 5x10, 6x10, ..., 15x10.

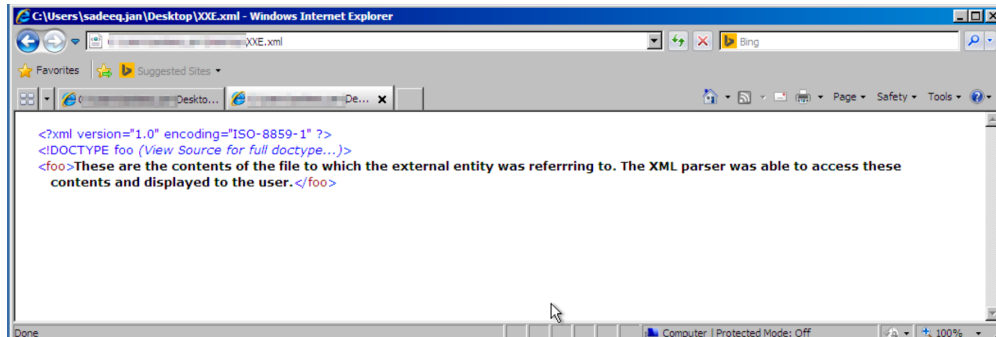
For the *XXE* category we selected three XML files, called XXE attacks, for the parsers: the first one has an entity that points to a UNIX system file (/etc/passwd), the second one points to a user file in the same directory of the XML files (user.txt), and the last one points to a Windows system file (c:\\ Windows \\win.ini). These files represent user and system files of the Linux, Mac OSX, and Windows operating systems.

It is important to note that this testing methodology can be reused to test new parsers for BIL and XXE attacks.

### 3.1.4 Results on the Parsers

#### XXE

Concerning XXE, we manually inspected the results obtained from each parser when they were fed with the three XXE attacks. We found that the vulnerable parsers attempt to expand the parsing results to include the content of the referred files, specified in the XXE attacks. If the expansion is successful, the content of the referred file (e.g., /etc/passwd) is



**Figure 3.1:** An output example of Microsoft Internet Explorer that uses a XXE-vulnerable parser. The browser expands the content of a text file referred to by the XXE attack and displays it.

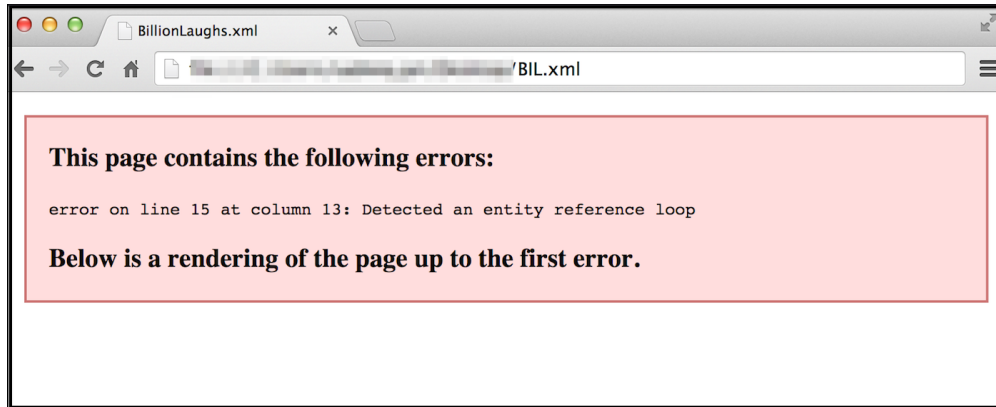
included in the parsing result. Otherwise, an exception stating an access permission error is returned. Nevertheless, both cases indicate an XXE vulnerability because the parsers try to access the content of the referred file. The other non-vulnerable parsers blocked the entity expansion and returned an error reporting the issue.

As a concrete example, Figure 3.1 depicts the parsing result of a vulnerable parser where the parser was able to acquire the content of the file referred to by the entity in the XML file. In this example, this content has been included within the tag `<foo>`.

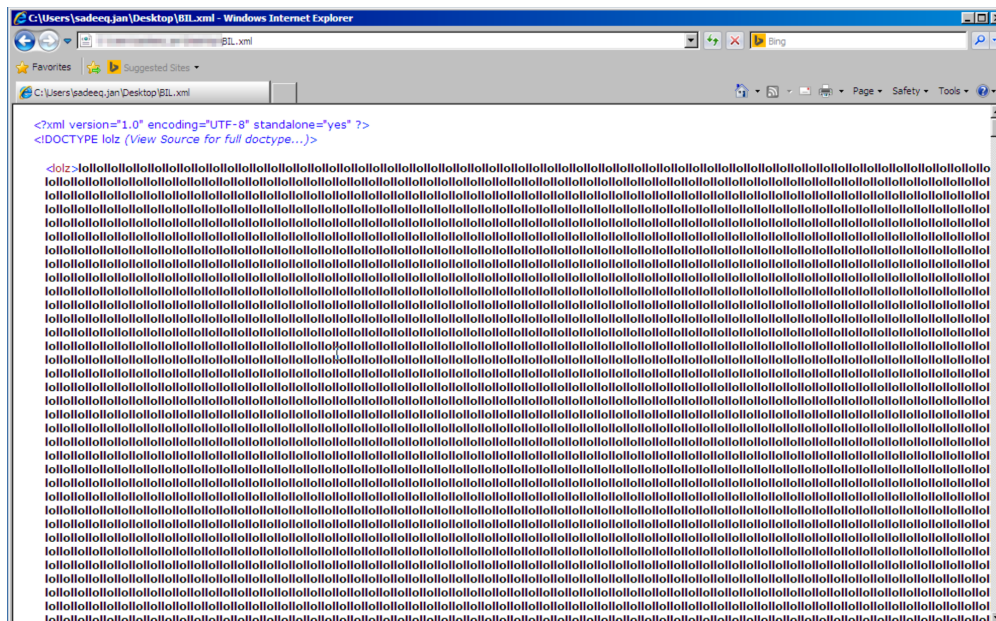
## BIL

Regarding BIL, we observed the CPU time and memory consumption of each parser when parsing each of the generated XML bombs. The size of each of these XML bombs was less than 1KB. Since regular XML files of size ranging from 1KB to 10KB require a small amount of memory (less than 1Mb in most cases) and a second of CPU time, the generated BIL attacks with similar size should result in similar behaviour if a parser is not vulnerable. Moreover, a non-vulnerable parser should be able to detect malicious entity reference loops in input XML files and raise an exception. Otherwise, when a parser exhibits significant deviation in terms of memory consumption and CPU time, it is regarded as BIL vulnerable.

As an example, Figure 3.2 represents the parsing output of an XML bomb file where the parser detected an entity reference loop. This is considered to be the desirable behaviour of the parser. However, some parsers could not detect these entity reference loops in our experiment, thus, making them vulnerable to BIL attacks. As an example, Figure 3.3 shows



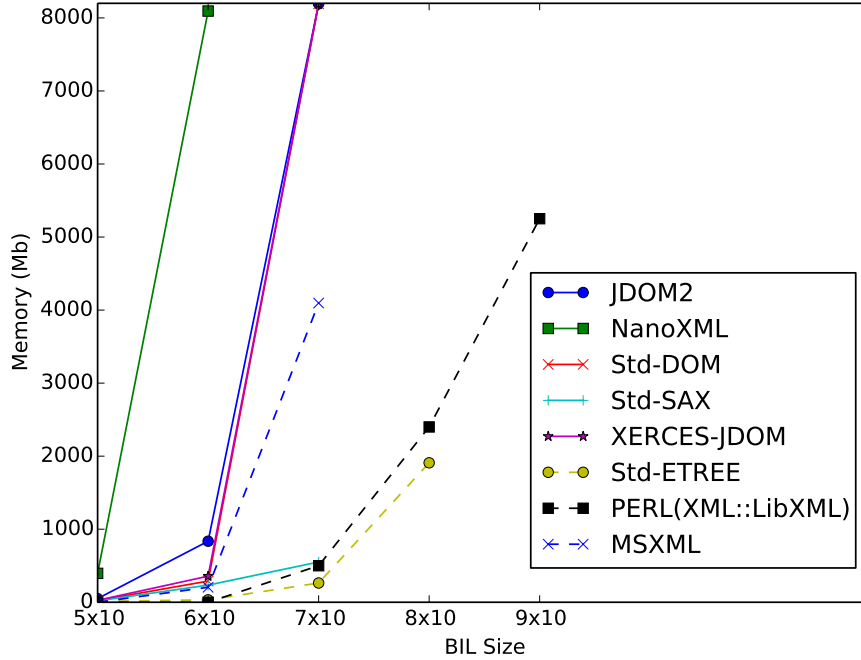
**Figure 3.2:** An output example of Google Chrome that recognises an input XML Bomb and raises an exception.



**Figure 3.3:** An output example of Microsoft Internet Explorer that uses a BIL-vulnerable parser. The browser expands recursively the content of an XML Bomb, occupies the system’s CPU and memory, and renders the system unresponsive.

the parsing result of the same XML bomb file by a vulnerable parser where it could not detect the entity reference loops and kept expanding the entity “lol”.

Figures 3.4 and 3.5 depict the memory and CPU time required for the eight *BIL*-vulnerable parsers to parse XML bombs of different sizes. We observe that the amount

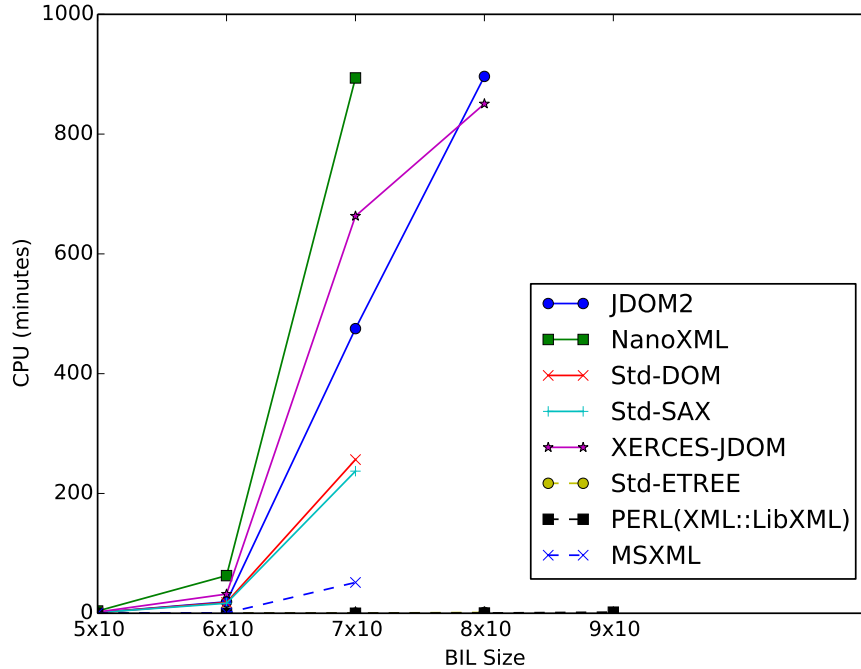


**Figure 3.4:** Memory consumption of the parsers when parsing XML Bomb files of different sizes (specified in  $M_xN$ ,  $M$  is the number of recursive reference loops;  $N$  is the number of references per loop).

of memory required to parse XML bombs increases exponentially, for all the vulnerable parsers. For the XML bomb of size  $6x10$ , the parsers require at least 33Mb RAM to parse the input. When the size is equal or greater than  $7x10$ , the amount of memory consumed increases significantly, from 200Mb up to 8Gb, which is the limit we set for each run.

Comparing to the memory consumption, the results with respect to CPU time differ slightly. Std-ETREE and Perl (XML:LibXML) require less than 2 minutes to parse BIL files up to size  $9x10$  thanks to their fast underpinning XML processing library<sup>1</sup>. The others demand much more CPU time, from 51 minutes with BIL  $7x10$ , up to a few hundred of minutes. The CPU time and memory consumption of the Java parsers were measured on a Linux node (2.4GHz, 1024 Gb RAM) of the UL HPC platform [21]. For the others, we used a Mac (3.5GHz, 16Gb RAM). This clearly shows the overwhelming cost in terms of CPU time and memory consumption when a vulnerable parser undergoes *BIL* attacks.

<sup>1</sup>libxml2, available at <http://xmlsoft.org>



**Figure 3.5:** CPU time required for the parsers to parse XML Bomb files of different sizes (specified in  $M_xN$ ,  $M$  is the number of recursive reference loops;  $N$  is the number of references per loop).

Table 3.3 summarises the results that we obtained. Out of 13 parsers of different languages, 8 (61.53%) are vulnerable to BIL and 7 (53.85%) are vulnerable to *XXE*. It is surprising that more than half of the parsers (most of them are extensively used, see Table 3.2) neglect these vulnerabilities. If the adopters of the parsers remain unaware of their presence, it is highly likely that their systems will be at risk due to *BIL* or *XXE* attacks.

Since Std-DOM and Std-SAX are very popular, we have investigated further and found out that they are vulnerable from version 1.5 backward<sup>1</sup>. From version 1.5 to 1.7, these parsers are vulnerable only when the original implementation of XML processing library, Apache Xerces<sup>2</sup>, is present in Java Classpath. Thanks to its dynamic binding feature, Java uses Apache Xerces for parsing XML instead of its internal implementation, making Std-DOM and Std-SAX vulnerable.

<sup>1</sup>[http://docs.oracle.com/javase/1.5.0/docs/guide/xml/jaxp/JAXP-Compatibility\\_150.html](http://docs.oracle.com/javase/1.5.0/docs/guide/xml/jaxp/JAXP-Compatibility_150.html)

<sup>2</sup><http://xerces.apache.org>



**Table 3.3:** Summary of *BIL* and *XXE* vulnerabilities in the parsers. We report which parsers are vulnerable to *BIL* and *XXE*.

Parser	Vul. to <i>BIL</i>	Vul. to <i>XXE</i>
JDOM2	Yes	Yes
NanoXML	Yes	Yes
NanoXML-LITE	No	No
Std-DOM	Yes	Yes
Std-SAX	Yes	Yes
Std-STAX	No	No
WOODSTOX	No	No
XERCES-JDOM	Yes	Yes
LXML-ETREE	No	No
Std-ETREE	Yes	No
PERL(XML::LibXML)	Yes	Yes
PHPDOM	No	No
MSXML (DOMDocument)	Yes	Yes
<b>Total</b>	8	7

To summarize, regarding research question **RQ1** we found that:

*BIL and XXE attacks are successful in many modern XML parsers. Among the ones selected for experimentation, more than half are vulnerable.*

### 3.1.5 Results on Open Source Systems

In the previous section, we reported that many parsers, including the most adopted one (Std-DOM), are vulnerable to *BIL* and *XXE* attacks. During our study, we also experienced many applications that crashed or hung when opening an XML bomb file, such as Microsoft Visual Studio Express 2013<sup>1</sup>. Therefore, we extended our study to investigate the research question **RQ2** to see whether software systems that use vulnerable parsers properly prevent *BIL* and *XXE* attacks.

Specifically, we picked a large number of open source systems that adopt Std-DOM and are hosted on GitHub. Std-DOM is the most used parser as Table 3.2 has shown, and GitHub

<sup>1</sup><http://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx>

### 3.1 Experimental Study

**Table 3.4:** A sample of 99 open sources projects among those selected in our study. The projects are accessible by appending these names to github.com, as of August 2014.

/godfreynolan/AndroidBestPractices	/AuScope/MDU-Portal	/v5developer/maven-framework-project
/970815940/desginModel	/kozakvoj/PPM	/0x17/Pecker
/jack2w/microcard	/jab416171/Android-Battleship-Client	/mmp/amazon-rest-api
/dippe/RooGo	/bobfreitas/http-client-tester	/rkday/jsipp
/mavenlab/jetset	/khassen/JavaCollectionFichierJaxBInitiation	/lshain-android-source/external-roboelectric
/iambus/xquery-b	/eyupdalan/edalanxmlgettersaver	/xpavlic4/skgaAndroid
/cpliakas/solr-config-validator	/bohdbantan/task_3	/mindblender/uPortal-old
/RamKancharla/2012_R3	/leandersabel/uPortal	/mltech/mlt-lib
/rlm33/TPV-RASS	/tsaikd/KDJLib	/renatoathaydes/OsgiMonitor
/brooklyncentral/brooklyn	/Gawkat/TimeWarpEngine	/ImpressiveCode/ic-depress
/Emmsii/Dungeon-Crawler	/screamconjoiner/jPath	/BBK-PiJ-2012-88/PSO
/guzziye/test	/amitkapps/pocs	/protocol7/Artemisa
/leveluplunch/levelup-java-examples	/nemrioff/NemerovCommonTest	/juphich/toolab
/sci4me/NSAClicker	/Albaniusz/java.mkkyong	/nerotech1989/ServerMonitor
/jirrick/Superfarmar	/Koziolek/kursynbp	/TechnicPack/LauncherCore
/gongchangxing/Test	/redmoo84/PPS_ITSM	/GMOD/Apollo
/zeng233/myproject	/tefreestone/myBYU	/panipilos/seerc-email-service
/smeza/srv_client	/AuScope/C3DMM	/escidoc/srw-repository
/tomtrath/homecan	/Rembau/xmlTest	/mkimberlin/sencha-touch-experiments
/wbssyy/DPminingFromCode	/antouk/rstext	/julieklein/ProteasixDatabase_PMAP
/chandrasekharab/mongoexp	/bulain/zk-demo	/icplayer/icplayer
/Bert89/luca_perilongo	/faramde/Harar-Emmanuel	/HemanandRajamani/RJUnit
/rodmidde/confluence-citation-plugin	/unja66/PK300	/bharcod/TwitterScraper
/garbray/SOA	/himadri77/SCRadioListener	/geoserver/geoserver-history
/brandom-iava/iava-dp	/minslers/by.minsler.xml	/chuntakli2/DriverApp
/stestaub/entityLoadingBug	/jlamandecap/saml-assertion-tools	/benbenedek/j2ee-homework
/gaohoward/jbm-to-hornetq	/angusws/tcx2nikeplus	/YusukeNumata/Training
/RasThomas/MedArt	/jost125/MI-W20-FLICKER	/JoelJ/JCcss
/BretonJulien/GamePlayerXML	/net900621/owl-eye	/web-builder/mdr
/Lewuathe/HD	/Rakurai/dip	/ArnaldoTrujillo/Comercial_Suit
/sklay/njztsm	/berk/banksystem_example	/gbhl/bhl-europe
/powerbush/mtk75m	/rohitkochar/FlickrSync	/tadayosi/larvae
/clem87/RSSAgregat	/EstarG/JavaWeb	/irina-andreevna-ivanova/ivanova_p01

enables us to quickly search and download source code for analysis. We used the same query as used in Table 3.2 for Std-DOM: “*javax.xml.parsers.DocumentBuilder*” on GitHub. Among the results, we selected the first 1000 Java source files that contained the query keyword. We then further filtered all files that did not actually parse XML inputs (e.g., some classes import “*javax.xml.parsers.DocumentBuilder*” but do not use it) and thus obtained 749 Java classes belonging to 628 open source projects. Table 3.4 shows a sample of 99 of them for references. All these classes together with other artefacts used in our experiments are available at <http://people.svv.lu/sadeeq/bilxxe>.

Going back to the over-approximation concern that we mentioned in Section 3.1.2 about the number of adoptions of our selected parsers by software developers, our inspection of the 1000 Java source files demonstrates that approximately 75% (749 out of 1000) of these classes actually use the Std-DOM parser, while the remaining 25 % do not. Assuming 25% is an accurate estimate for the remaining parsers in Table 3.1, the total number of source classes that use our selected parsers should still number more than 400,000. Also, note that our search results are based on only two repositories (GitHub and Google Code) among other repositories where these selected parsers could be used. This increases the number of their adoptions. Therefore, we are certain that our selected parsers are widely used by software developers.

Our assessment on whether a system that makes use of Std-DOM deals appropriately with *BIL* and *XXE* vulnerabilities is based on the application of known fixes, i.e., properly setting the attributes of the parser (through the `DocumentBuilderFactory` class) before using it to parse an XML input. For example:

```
dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/
  disallow-doctype-decl", true);
```

In our assessment, we seek for the presence of any of the following attributes and their values in the source code of the selected Java classes.

Attribute	Value
"http://apache.org/xml/features/- disallow-doctype-decl"	true
"http://xml.org/sax/features/- external-general-entities"	false
"http://xml.org/sax/features/- external-parameter-entities"	false
"JDK_ENTITY_- EXPANSION_LIMIT"	a numeric value
"FEATURE_SECURE_- PROCESSING"	true

Out of 749 selected Java source files (belonging to 628 GitHub projects) that use Std-DOM to parse XML inputs, we found only one file that properly sets one of the above properties to avoid being attacked through *BIL* and *XXE* vulnerabilities. Among the remaining files, 735 classes (98,13%) are clearly vulnerable. The other 14 classes cannot be confirmed

### 3.1 Experimental Study

**Table 3.5:** Tested applications that are vulnerable to BIL and XXE.

Application	Description
websphere-portal-plugin	A plugin for WebSphere Portal for deploying WAR, EAR, PORTLETS, EXPORT/IMPORT XMLACCESS. <a href="https://github.com/JuanyongZhang/websphere-portal-plugin">https://github.com/JuanyongZhang/websphere-portal-plugin</a>
File-Archiver-Main	An application to combine a number of files together into one archive file. <a href="https://github.com/DymaKulia/FileArchiverMain">https://github.com/DymaKulia/FileArchiverMain</a>
AppDF	A Project to facilitate easy uploading of an android application along with its supporting files to several appstores by creating a single archive AppDF file. <a href="https://github.com/onepf/AppDF">https://github.com/onepf/AppDF</a>
source2XMI	Convert the Java source code to XMI file. <a href="https://github.com/wbssyy/source2XMI">https://github.com/wbssyy/source2XMI</a>
jbm-to-hornetq	A tool to facilitate migration from JBM <sup>1</sup> to HornetQ <sup>2</sup> messaging platform. HornetQ is an open source asynchronous messaging project from JBoss. <a href="https://github.com/gaohoward/jbm-to-hornetq">https://github.com/gaohoward/jbm-to-hornetq</a>
fastcatsearch	An open source distributed search engine. <a href="https://github.com/fastcatsearch-/fastcatsearch">https://github.com/fastcatsearch-/fastcatsearch</a>
bimoku Crawler	A web crawler. <a href="https://github.com/cncduLee/bbks-crawler/tree/master/crawler/bimoku/crawler">https://github.com/cncduLee/bbks-crawler/tree/master/crawler/bimoku/crawler</a>
blog	A Java blog engine. <a href="https://github.com/IgorInger/blog">https://github.com/IgorInger/blog</a>

to be vulnerable with certainty since they use a *DocumentBuilder* object created elsewhere in their corresponding projects and security properties might be set from there. Nevertheless, these results indicate that developers (at least the owners of the selected projects) have neglected to address these vulnerabilities.

In addition, speculating that there could be workarounds to deal with vulnerabilities, we downloaded eight random systems (Table 3.5) and analysed their entire source code. We found that one of them had a vulnerable Java class which used the parser but the class was not used elsewhere in the project (i.e., orphan code), and the others seven were vulnerable: there was no mitigation along the control flow from reading XML inputs until they are parsed. Therefore, we conclude it is unlikely that developers made use of other methods to

deal with *BIL* and *XXE* vulnerabilities.

Regarding research question **RQ2** we found that:

*It is highly likely that systems that use a BIL- or XXE-vulnerable XML parser do not apply any proper mediating measure and are hence vulnerable.*

### 3.1.6 Discussion and Recommendations

Our extensive study has demonstrated that *BIL* and *XXE* attacks are in most cases neglected by the developers of XML parsers and software systems that adopt them. Since these attacks are well-known and applying them is straightforward (it is easy to create XML test files and send them to a target system), leaving them unaddressed before deployment might have severe consequences. As demonstrated from our results, a vulnerable XML parser can consume a huge amount of memory and CPU time as the result of an attack. This typically renders the system running the XML parser unavailable for legitimate users. Similarly, the confidentiality of information residing on the system running the vulnerable XML parser is at risk. Exploiting *XXE*, an attacker can get access to such information.

#### Recommendations for Software Developers

Because software systems that improperly use vulnerable parsers are also vulnerable, we recommend that developers of such systems should pay special attention to preventing such attacks if they decide to adopt a third-party XML parser, even if it is provided by a high-profile vendor, such as Oracle or Microsoft. In order to block *BIL* and *XXE* attacks, software developers should gain full understanding of the XML parser that they are considering to adopt and avoid its insecure features (e.g., using Schema instead of DTD). If external entity references are required, they should refer to trusted sources only. Known vulnerabilities of the parser and their fixes should be investigated and input sanitisation should be done before parsing XML content. Adequate security testing of the parser should also be performed.

#### Recommendations for Parser Developers

Developers of XML parsers need to be fully aware of all potential XML-based attacks and should be able to provide countermeasures wherever possible. It was observed, during our experiment, that some vulnerabilities can be exploited because of the features allowed in the default configurations of XML parsers. Parser developers should provide Secure Default

Configurations and provide alerts when any potentially insecure feature is enabled via making changes to the default configurations. Parser developers should perform security testing of their parsers. They should also provide better documentation including the potential risks of enabling any feature. This would guide software developers in using their parser in a secure way.

### Threats to Validity

Regarding the validity related to whether or not experimental results can be considered generalisable and representative, we have selected a large number of parsers from various programming languages and domains, and considered in our study their latest versions. Furthermore, we evaluated also their adoption on GitHub and Google Code, the two most popular open source repositories, to make sure that the selected parsers are used in practice. As a result, we are fairly confident that the vulnerabilities we detected both in parsers and the systems using them suggest a worrying but representative state of practice.

Moreover, although we consider only one parser in the evaluation of 628 open source projects, it is the most popular one; and given the clear results we obtained - only one of the projects properly deals with BIL and XXE attacks, it is unlikely that mitigation techniques are implemented in other projects adopting other vulnerable parsers.

## 3.2. Related Work

There is a large research body investigating the potential exploits in web services, e.g., [5, 22, 23, 24, 25, 26, 27]. Tiwari and Singh [27] described various attacks on atomic as well as composite web services, including denial of service and injection attacks (SQL/XML/X-PATH) along with their impacts and countermeasures. Similarly, Gupta and Thilagam [5] provided insights about various forms of XML-based attacks on web services and possible countermeasures. XML oversize payload (here referred to as BIL) and XXE attacks are two out of 18 attacks against web services described in the paper. Their descriptions are limited and no wide scale experiment on parsers and systems is performed to assess the impact of these vulnerabilities in practice. Orrin [28] discussed the SOA/XML Threat Model and the new XML/SOA/Web 2.0 attacks and threats in detail. XML attacks are classified into four main categories: Payload/Content Threats, XML Misuse/Abuse, XML Structure Manipulation, and Infrastructure Attacks. BIL and XXE attacks are discussed under the XML Structure Manipulation category.

Automated approaches to detecting vulnerabilities in web service frameworks (containers) have been investigated, e.g., [29, 30, 31, 32, 33]. Mainka et al. [31] described their approach to automated penetration testing of web services frameworks. They implemented a tool called WS-Attacker. It is extensible with additional plugins for web service specific attacks. Evaluation has been performed on some web services frameworks, e.g., Apache Axis2<sup>1</sup> and JBoss<sup>2</sup>, to detect two web service specific attacks: WS-Addressing spoofing<sup>3</sup> and SOAPAction<sup>4</sup>. Oliveira et al. [33] developed a similar penetration testing tool, WSFAggressor, for security testing of web services frameworks. In fact, this tool is an extension of the WS-Attacker using a small set of its functionality but providing more web service specific attacks. Chang et al. [34] proposed a fuzz testing approach to vulnerability identification and analysis in web service architecture.

Prevention and detection approaches to XML Denial of Service (XDoS) Vulnerabilities in web services have been investigated. Falkenberg et al. [8] discussed some techniques for targeting XML-based message formats using various XML properties and developed an approach to XDoS penetration testing of web services. The approach is based on measuring deviations in response time of a web service between original (unchanged) and tampered requests. Suriadi et al. [35] investigated the XDoS vulnerabilities in few web services platforms and described the corresponding effects on CPU and memory consumption of the target. Other similar work on XDoS can be found in [36, 37, 38].

In summary, existing work has discussed various XML vulnerabilities, their impact, and how to detect them in web services and their containers. However, no work has been carried out to study, in a systematic and rigorous manner, the presence of XXE and BIL vulnerabilities in modern XML parsers and open-source systems, which is the goal of this chapter.

### 3.3. Summary

In this chapter, we study the potential of two major types of XML-based attacks: XML Billion Laughs (*BIL*) and XML External Entities (*XXE*) that may undermine today's XML parsers and systems making use of those parsers. We conducted a systematic and large-scale experiment to test the most popular XML parsers for these attacks by measuring their impact on CPU time and memory consumption. Our main objective is to provide representative,

---

<sup>1</sup><http://axis.apache.org>

<sup>2</sup><http://www.jboss.org>

<sup>3</sup>[http://www.ws-attacks.org/index.php/WS-Addressing\\_spoofing](http://www.ws-attacks.org/index.php/WS-Addressing_spoofing)

<sup>4</sup>[http://ws-attacks.org/index.php/SOAPAction\\_spoofing](http://ws-attacks.org/index.php/SOAPAction_spoofing)

unbiased results of the extent of the problem in popular parsers and open source systems. We designed our experiment to achieve these objectives and reported the results in great detail.

We have studied 13 XML parsers that are widely used in open source systems hosted on GitHub and Google Code. Each was tested against BIL and XXE test cases. Executing these tests on the vulnerable parsers took exceptionally high amounts of CPU time and memory that could not have been efficiently carried out without our HPC platform [21]. The obtained results show that most of the selected parsers are vulnerable to *BIL* and *XXE* exploits. Furthermore, we extended our experiment to evaluate more than 700 classes from 628 open source systems that use a vulnerable XML parser and found that all but one of them are vulnerable as well, thus showing that parsers' vulnerabilities are not properly addressed by the systems using them.

Such alarming results call for software developers to take appropriate security measures before using these vulnerable XML parsers in their software development projects. Moreover, parser developers need to fix the problems and/or provide better documentation to help developers configure such parsers to secure their usage.



## Chapter 4

# Testing XML Gateway and Back-end Web Services for XML Injections

XML Injection (*XMLi*) attacks aim at manipulating XML messages that are processed by a target system in order to compromise it. They may be as simple as malformed XML messages to crash a target but they can also be more sophisticated and dangerous by carrying nested attacks, aiming at obtaining sensitive information or changing the underlying business logic of the target.

Existing approaches and tools for the detection of *XMLi* vulnerabilities in web services are based on fuzz testing [12, 13]. They inject XML meta-characters (e.g.,  $>$ ,  $<$ ), aiming at altering the structure of XML messages to detect if a service under test (SUT) is vulnerable. These meta-characters are often rejected by protection mechanisms (e.g., XML gateway) or the SUT itself when they parse and validate the XML messages. As a result, and as confirmed by our empirical results, these approaches are largely ineffective, especially if the goal is to detect subtle vulnerabilities in the way XML messages are processed.

In this chapter, we propose an automated testing approach and tool for *XMLi*, called SOLMI<sup>1</sup>. SOLMI covers a wide range of *XMLi* attacks. More importantly, it makes use of a constraint solver to automatically generate well-formed and valid XML messages with respect to given domain constraints, which are also carrying malicious content. By doing this, we make sure that the generated XML messages are not easily recognised, and hence, are more likely to penetrate into the SUT to exploit vulnerabilities.

---

<sup>1</sup>SOLMI is available for download upon request

---

The key contributions of this chapter include:

- The taxonomy of *XMLi* attacks, leading to the definition of XML mutation operators to be used for testing purposes.
- We propose a novel approach and tool called SOLMI, relying on the defined mutation operators to manipulate XML messages and a constraint solver to generate valid though malicious XML messages, which are test cases in our context.
- SOLMI has been evaluated on a financial system including 44 complex web services at the back-end that are protected by an XML gateway (firewall). Results are promising as SOLMI is much more effective in generating successful attacks that can bypass the gateway (78.86% of all generated tests) as compared to a state-of-the-art tool based on fuzz testing, which did not manage to generate any malicious, bypassing attack.

In our experimental setting, the 44 web services connect to the same XML gateway at the front-end, each of them has an independent interface at the gateway, and incoming input messages to each service are treated differently. Although we consider only a single gateway, such an industrial setting with many web services is rather hard to set up. Further, since there currently exist only a few industry-strength XML gateways with a sizeable market share (IBM DataPower Gateway<sup>1</sup>, Axway<sup>2</sup>), obtaining experimental results even with one of them can provide useful insights.

**Assumptions:** This chapter presents one part of the dissertation in which we aim at automated testing to uncover *XMLi* vulnerabilities involving user inputs and all related systems, including the front-end web applications (service consumers) and the back-end web services (service providers). In this chapter, we assume that the front-end web applications are vulnerable to *XMLi*, and thus, that manipulated and malicious XML messages can be created. As a result, we can focus on the back-end side and assess whether service providers are vulnerable. We investigate an approach for generating injected XML messages that mimic those that might be produced by the front-end web applications and test whether the back-end services are vulnerable to *XMLi* attacks.

The remainder of the chapter is structured as follows. Section 4.1 provides a taxonomy of *XMLi* attacks. Section 4.2 discusses the proposed mutation operators and the overall SOLMI approach. Section 4.3 describes the architecture of our developed tool. Section 4.4 shows our evaluation results and discussions, based on real-world financial services. Section 4.5 discusses related work. Finally, Section 4.6 concludes the work.

---

<sup>1</sup><http://goo.gl/L8Dzs3>

<sup>2</sup><https://www.axway.com/>

**Table 4.1:** XML Meta-characters

Character	Consequence
<	Opening a tag without closing it.
&	This is a character for escaping meta-characters, which makes an XML malformed when being used alone.
>	Closing a tag without opening it.
'	It makes the name specification syntactically incorrect when added to an attribute name.
"	Similar to the previous one.
<!--	This sequence of characters represents the beginning/end of a comment and is not allowed in attribute values.
]]>	This is a delimiter for the CDATA section and is not allowed in values of elements.

## 4.1. Taxonomy of *XMLi* Attacks

We have surveyed the state of the art regarding *XMLi* attacks and propose to classify them into four types, based on the way they change the XML structure. This classification helps us better understand the variants and intents of *XMLi* attacks. Moreover, this classification will be a basis to derive test generation strategies.

### 4.1.1 Type 1 - Deforming

Attack input values of Type 1 are XML meta-characters, such as <, >, ]]>, that are introduced to compromise the structure of generated XML messages. The target service, when processing such malformed XML messages, might crash or behave unexpectedly due to triggered exceptions.

Table 4.1 lists the most common examples of XML meta-characters. When any of these meta-characters is injected into an XML message, it will render the message malformed.

### 4.1.2 Type 2 - Random closing tags

Attack input values of Type 2 are random XML closing tags (e.g., `</test>`), aiming at deforming the generated XML messages to reveal their structure. This type of attacks tries to gain information from error messages triggered by the manipulated XML messages when they are not properly treated. Specifically, when receiving a message of this kind, the XML parsers used by the SUT often returns an error message stating that there is a mismatch between an opening and a closing tag. The error message may also reveal the names of two elements: the element that was just injected (`</test>`) and the name of the preceding element, thus revealing the latter. For example, if the *username* input receives `</test>`, it will create a SOAP message containing “... `<username></test>` ...”. This might lead to the SUT raising an error message revealing the name of the *username* element.

### 4.1.3 Type 3 - Replicating

Attack input values of Type 3 are strings of characters consisting of XML tag names and malicious content. They aim at replicating elements of an XML message for malicious purposes, such as to manipulate the application logic of the SUT to get access to protected data. Consider the following sample value for the *E-mail* input of the web portal: “`a</mail><userid>0 OR true</userid><mail>a@b.com`”. When it is used, the portal produces the SOAP message depicted in listing 4.1:

```
<soap:Envelope>
  <soap:Body>
    <user>
      <username>tony</username>
      <password>Un6R34kb!e</password>
      <userid>500</userid>
      <mail>a</mail>
      <userid>0 OR true</userid>
      <mail>a@b.com</mail>
    </user>
  </soap:Body>
</soap:Envelope>
```

**Listing 4.1:** Example of SOAP message manipulated with *XMLi* Type 3.

The *user* element has been injected with two *userid* and two *mail*. The last *userid* sub-element contains an SQL injection tautology (detailed in the next section). When parsing XML and binding its content to business code, parsers tend to only consider the last instance if they expect only one XML element of a type. As a result, if the *user* element is appended to the registration database, such a malicious content may trigger SQL injection exploitation.

### 4.1.4 Type 4 - Replacing

Attack input values of Type 4 are similar to those of Type 3 but they involve multiple input fields in order to comment out some existing XML elements and inject new ones with malicious content. The aim is to produce not only malicious XML messages but also to make them valid with respect to the domain constraints, as defined using XML Schema Definition (XSD). The purpose of attacks of Type 4 is similar to that of Type 3.

In the web portal example in Section 2.2.1, we have shown one example of *XMLi* Type 4 with the following inputs:

```
Password = "Un6R34kb!e</password> <!--"  
E-mail = "--><userid>0</userid><mail>a@b.com"
```

These inputs result in the SOAP message in Figure 2.4 in which the system-generated *userid* element has been commented out and replaced by an injected one.

This type of attacks is potentially more dangerous than Type 3 ones as the injected content will certainly be used by the target receiving the XML messages. Moreover, since it leads to XML messages that are well-formed and valid, the malicious content has a higher chance of interfering with the target's business logic and causing harmful impacts.

## 4.2. SOLMI: A FRAMEWORK FOR *XMLi* TESTING

We introduce *SOLMI* (*SOL*ver and *Mutation*-based test generation for XML Injection), a framework for testing web services against *XMLi* attacks. *SOLMI* is equipped with a set of mutation operators that can manipulate XML in order to generate all the four types of *XMLi* attacks. In particular, for Type 3 (Replicating) and Type 4 (Replacing) in which

*XMLi* attacks carry nested attacks in the form of XML content, SOLMI relies on a constraint solver and attack grammars to generate the nested attacks (also called as malicious content), making them more effective in circumventing the validation mechanisms of web services.

This section, first, discusses our proposed mutation operators for generating each type of *XMLi* attacks. Then, we describe in detail how malicious content (nested attacks) are generated for *XMLi* of types 3 and 4. Finally, we define the general test generation strategy implemented in SOLMI and the oracles for the detection of successful *XMLi* attacks.

### 4.2.1 Mutation Operators

We propose five mutation operators (MO) to manipulate XML messages and generate *XMLi* attacks. The first two are used to create *XMLi* attacks of Type 1, while each of the subsequent operators corresponds to attacks of Type 2 to 4, respectively. The names of the MOs and their description are listed in Table 4.2.

#### **Operator: MO\_der\_meta**

##### **Description:**

Adds a randomly selected XML meta-character into the content of a randomly selected element of the input XML message.

##### **Rationale:**

XML meta-characters such as < and > are special characters for defining the elements of an XML message (like keywords in a programming language). This mutation operator injects those characters into XML messages to render them malformed. When receiving a malformed XML message, the SUT might exhibit unintended behaviours.

#### **Operator: MO\_der\_att**

##### **Description:**

Removes a double quote from the value of a randomly selected attribute of a randomly selected element of the input XML message.

##### **Rationale:**

Double quotes are used as delimiters for containing attributes' values. Removing one from the value of any attribute will deform the message.

#### **Operator: MO\_clo**

**Description:**

Adds the closing tag: `</test>` into the content of a randomly selected XML element of the input XML message to mimic *XMLi* attacks of Type 2.

**Rationale:**

As explained in Section 2.2.1, adding such a closing tag to an XML message will deform the input message. However, when encountering such tag, XML parsers often raise an error stating that they expect other elements of the message and reveal their names. Hence, structural information of the message might be leaked in an unintended way to malicious users.

**Operator: MO\_replica****Description:**

Replicates an XML element to create a new one, injects malicious content generated using constraint solving, and inserts the new element right after the original one.

**Rationale:**

This operator aims at emulating *XMLi* attacks of Type 3. When an element is replicated, its last instance is often the one considered by the SUT. If that instance contains harmful content, it can lead to a security breach.

**Operator: MO\_replace****Description:**

Replace an XML element by: (1) replicating the element with new and malicious content like we do with `MO_replica`, (2) commenting out the selected element, and (3) injecting the new element at the original location of the commented-out one.

**Rationale:**

This operator aims at emulating *XMLi* attacks of Type 4. Since the selected element is commented out, the new and malicious content is certain to be considered by the SUT. As a result, it has a high chance of impacting the targeted service if it is vulnerable.

### 4.2.2 Producing Nested Attacks

The mutation operators `MO_replica` and `MO_replace` need malicious content to produce *XMLi* attacks of Types 3 and 4 that are likely to lead to security breaches. Relying on recent advances in constraint solving, we consider domain constraints and attack grammars in order

## 4.2 SOLMI: A FRAMEWORK FOR *XMLi* TESTING

---

**Table 4.2:** Summary of the proposed mutation operators for manipulating XMLs

MO name	Description
<i>Deforming</i>	
MO_der_meta	Adds a randomly selected XML meta-character into a randomly selected element of the input XML message
MO_der_att	Removes a quote from the value of a randomly selected attribute of a randomly selected element of the message
<i>Random closing tags</i>	
MO_clo	Adds <code>&lt; /test &gt;</code> into the content of a randomly selected XML element of the message
<i>Replicating</i>	
MO_replica	Replicates an XML element, injects the new content into it and puts it at the location right after the selected element
<i>Replacing</i>	
MO_replace	Replicates an XML element, obtains a new content, comments out the selected element, and injects the new one at its location

to generate valid and harmful content. In other words, this will ensure that the generated content, when being embedded in XML, satisfy domain constraints and are malicious at the same time. As a result, our final generated *XMLi* attacks are in a better position to circumvent filtering mechanisms, such as input validation, and hence, have a high chance of uncovering *XMLi* vulnerabilities.

### 4.2.2.1 Domain Constraints

The XML messages exchanged between services usually adhere to some protocols, in which the content of XML elements are restricted according to domain constraints. XSD is the most popular format used for specifying such constraints on XML content. XSD specifications can be stored into separate files or can also be included into WSDLs (Web Service Definition



Language<sup>1</sup>) to define service interfaces, data types and constraints.

Domain constraints are defined as XSD restrictions on the length, value range, and content pattern of XML elements. Their content must satisfy the constraints as otherwise they are considered to be invalid, and so are the XML messages that contain them.

The listing below shows two examples of XSD constraints specified for the *username* and *password* elements of the registration database. The first one requires that *username* must have a length between four and 32 characters, and must contain only characters of the alphabet from “a” to “Z” and numerical digits. The second requires that *password* must have a length greater than or equal to eight.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
  ...
  <simpleType name='StringUserType'>
    <restriction base='string'>
      <minLength value='4' />
      <maxLength value='32' />
      <pattern value='[a-zA-Z0-9]+' />
    </restriction>
  </simpleType>

  <simpleType name='StringPasswordType'>
    <restriction base='string'>
      <minLength value='8' />
    </restriction>
  </simpleType>
  ...
</schema>
```

**Listing 4.2:** Examples of XSD constraints.

---

<sup>1</sup>[www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl)

#### 4.2.2.2 Attack Grammar

Given that we ensure domain constraints are satisfied for nested attacks, we ensure that the content of XML elements are valid. However, we want them to be malicious as well.

As an example here, we use SQL injection (*SQLi*), which is an attack technique in which attackers inject malicious SQL code fragments into input strings. If inputs lack proper sanitisation, they might be concatenated into SQL queries, changing maliciously the intended logic of the queries. We use *SQLi* as an example in this chapter as it is one of the most exploited vulnerabilities. However, our approach is generic, BNF grammars of other types of attacks, such as XSS [39], can be used in place of *SQLi*.

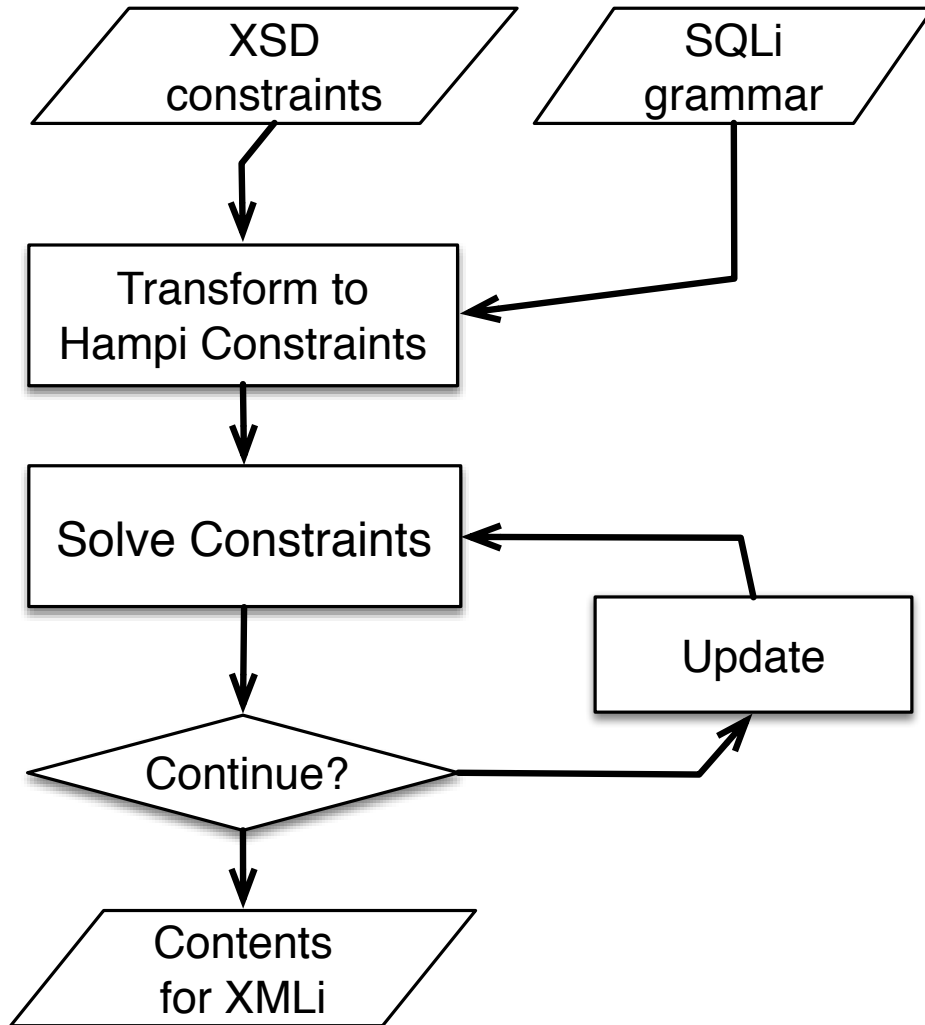
We reuse the context-free BNF grammar for *SQLi* strings proposed by Appelt et al. [40]. This grammar has been shown to generate effective *SQLi* attacks. In our context, by relying on such an attack grammar and XSD domain constraints, we can automatically generate content which is valid, yet malicious, for XML elements. The next section discusses in detail how this is done.

#### 4.2.2.3 Producing Malicious Content using Constraint Solving

Recent advances in theorem proving have given rise to a number of solvers, e.g., [41, 42, 43], that are capable of dealing with sophisticated constraints, such as string constraints or regular grammars.

We propose to use such constraint solvers to automatically produce content for XML elements such that they satisfy the associated XSD constraints. Moreover, since our security testing goal is to assess the SUT with respect to vulnerabilities, we want such content to be malicious. Therefore, we combine and transform XSD constraints and the proposed *SQLi* grammar into constraint specifications for the solver Hampi [42]. We then rely on the solver for generating concrete values for XML elements. We selected Hampi because it is the only solver that supports the targeted categories of XSD constraints and context-free grammars. In addition, it has been previously used in other contexts, e.g, concolic testing [44].

Figure 4.1 shows the procedure to generate malicious string content for XML elements. First, XML constraints and the *SQLi* grammar are transformed into Hampi constraints. Then, Hampi is invoked to solve the constraints and produce string solutions that satisfy them. Hampi is deterministic as it returns the same solution when called multiple times. However, as the SUT should be tested with different malicious content in order to maximise



**Figure 4.1:** Workflow for producing content using constraint solvers.

the chance of vulnerability detection, when needed, we update Hampi constraints to exclude the previously found solutions to enforce the search for new solutions.

### 4.2.3 Mutation-Based Test Generation

We propose two mutation-based test generation strategies for SOLMI. The first strategy is dedicated to the three mutation operators, `MO_der_meta`, `MO_der_att`, and `MO_clo`. It injects a randomly selected meta-character or a closing tag into an input XML sample

message to create a new one at each invocation. The behaviour of the first strategy resembles what existing “fuzzy” testing tools do, e.g., [12, 13].

The main strategy that we investigate combines a constraint solver and the two more advanced mutation operators (MO\_replica and MO\_replace) to generate XML messages containing *XMLi* attacks of Types 3 and Type 4. The generation procedure is provided in Algorithm 1. The inputs for the algorithm include: a sample XML message, the XSD schema that restricts XML messages, an attack grammar (such as *SQLi*), and a mutation operator. Next, we iterate through available XML elements, apply the mutation operator on each selected element, extract its constraints and transform them with the grammar into Hampi constraints, and then invoke Hampi to solve the constraints. Once a new malicious content is available, we inject the new content into the obtained mutant to create a new test case. Finally, we add the newly generated test case that emulates an *XMLi* attack into the output test suite.

The sample input messages for the strategies can be taken from existing functional test suites if available. Otherwise, they can be derived automatically from XSD using tools such as XMLMate [45] or Ws-taxi [46].

In Algorithm 1, the selection of XML elements (line 2 and line 19) is currently exhaustive, meaning that each and every element is in turn considered. Further, at line 13, we update input constraints so as to exclude previously-generated content in subsequent test cases (depicted in Figure 4.1).

### 4.2.4 Test Oracle

The objective of the test oracle is to assess whether the SUT is vulnerable to malicious content such as *XMLi*. To achieve this, once a test is submitted to the SUT, we analyse the SUT’s behaviour and its response to the test to search for symptoms of vulnerabilities, for example regarding *XMLi*:

1. If the SUT is protected by a security mechanism, like an XML Gateway or a Web Application Firewall, does the test bypass that layer or not? If the test is blocked, then the SUT is not vulnerable to that specific test. Otherwise, the test is potentially problematic as it can circumvent one security layer. However, other symptoms need to be considered.
2. If the test reaches the SUT and makes it crash, then the SUT is vulnerable.

## 4.2 SOLMI: A FRAMEWORK FOR XMLi TESTING

---

---

**Algorithm 1** SOLMI test generation algorithm:

---

**Require:** *xml*: a sample XML message.

*xsd*: the XSD Schema of the XML messages.

*attack-grammar*: an attack BNF grammar, e.g. *SQLi*

*mo*: mutation operator, MO\_replica or MO\_replace.

*max\_s*: maximum number of malicious content per element.

*budget*: targeted number of tests.

**Ensure:** *TS*: set of output test cases.

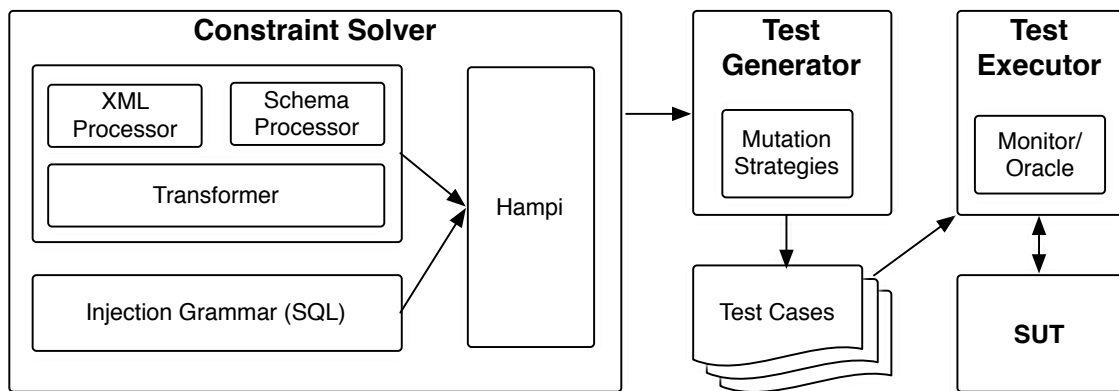
```
1:  $TS = \emptyset$ 
2:  $e = selectElement(xml)$ 
3: while  $size(TS) \leq budget$  and  $e$  is not null do
4:    $mu = apply(mo, e)$ 
5:    $c = extractConstraint(e, xsd)$ 
6:    $hampiInput = transform(c) + attack-grammar$ 
7:    $counter = 0$ 
8:   while  $counter \leq max\_s$  do
9:      $s = solve(hampiInput)$ 
10:    if  $s$  is not null then
11:       $t = inject(mu, s)$ 
12:       $TS = TS \cup t$ 
13:       $hampiInput = updateConstraints(hampiInput, s)$ 
14:       $counter = counter + 1$ 
15:    else
16:      break {exit this inner loop when HAMPI fails to solve the constraint.}
17:    end if
18:  end while
19:   $e = selectElement(xml)$ 
20: end while
21: return  $TS$ 
```

---

3. If the test reaches the SUT and triggers a response with a non-sanitised error message like parsing errors disclosing source code or data structure, then the SUT is vulnerable.
4. If the test reaches the SUT and the nested attack manifests itself (e.g., for *SQLi*, exposing or manipulating data), the SUT is vulnerable.
5. If the test reaches the SUT and the SUT successfully detects, blocks, and reports the malicious content of the test then the SUT is not vulnerable.
6. In other cases, when it is difficult to evaluate the response of the SUT as malicious or normal, executions must be logged and inspections are needed to make sure that no exploitation is possible.

### 4.3. TOOL SUPPORT

We developed a tool, called *SOLMI*, which implements all the proposed mutation operators and relies on Hampi. The inputs of the tool include a target web service, samples of XML messages, the associated XSD Schema, and an Attack Grammar (e.g., *SQLi*). The tool generates tests which are mutated versions of the original XML messages. The tool further submits the generated tests to the SUT and analyses its responses.



**Figure 4.2:** The components of SOLMI.

Figure 4.2 depicts the architecture of the tool. It contains three main components: Constraint Solver, Test Generator and Test Executor.

- **Constraint Solver.** The Constraint Solver is responsible for identifying the constraints associated with the selected element of the XML message, invoking Hampi to solve them and obtaining the new content of the selected element.
- **Test Generator.** This module generates the tests based on the proposed algorithm. These tests are the mutated versions of the input XML messages, which are valid according to the associated schema constraints but still contain malicious content, i.e., *SQLi* attack patterns.
- **Test Executor.** This module is responsible for sending the tests to the SUT and analysing the corresponding responses according to the test oracle discussed in Section 4.2.4.

## 4.4. EXPERIMENTAL EVALUATION

In this section we evaluate our proposed SOLMI framework. We compare it with ReadyAPI [12], a state-of-the-art web service testing tool, by running both of them against 44 financial web services<sup>1</sup> that are protected by an XML gateway.

ReadyAPI was selected as a baseline because it is the successor of SoapUI [47], which is widely used for testing web services. ReadyAPI combines several tools (e.g, LoadUI NG, Secure, API Monitoring in AlertSite) and was developed by SmartBear. In short, it helps software testers in performing functional, load, and security testing of their web services. The test generation of ReadyAPI is based on a dictionary of attack payloads, which are injected in XML messages. The tool provides several security scans for web services, e.g. Malformed XML, XML Bomb, Weak Authentication. We use the “Malformed XML” security scan since it is the most comparable to the features of SOLMI.

The XML gateway is the first layer of defence and aims to block malicious requests that target the web services. As our tests contain malicious content, they should ideally be blocked by this gateway. Otherwise, the web services are at risk unless they compensate by having their own sanitisation and verification procedures. In this chapter, we limit our evaluation to this first layer of defence, the XML gateway. We evaluate SOLMI and ReadyAPI in terms of their ability to generate malicious messages (tests) that can bypass the gateway to reach the protected web services.

We investigate the following research questions:

**RQ1 [Effectiveness]:** *Are the tools able to generate malicious messages (tests) bypassing the first layer of defence (the XML gateway) and thus reaching the targeted web services?*

Tests that can bypass the gateway can potentially lead to security breaches in the protected web services. The gateway should normally block the attacks generated by the tools. From a testing standpoint, a tool is deemed effective if it can generate bypassing tests to demonstrate that the gateway is vulnerable.

**RQ2 [Cost]:** *What is the cost of using the tools in terms of generation and execution time?*

Cost can be evaluated both in terms of test generation and execution time. Test generation time is important given that SOLMI relies on constraint solving to search for malicious content, an operation that is known to be time consuming in many cases. The test execution

---

<sup>1</sup>These are web services of a financial company whose name cannot be revealed due to security concerns.

time depends on the responsiveness of the SUT, the number of tests executed, and the test output evaluation. However, in our context, since the XML gateway and the web services typically respond to a service request in less than a few milliseconds, the execution time may not be as important as the test generation time, from a practical standpoint.

#### 4.4.1 Subject Application

The experimental evaluation was performed on 44 web services of a company who manages financial transactions and bank card processing. These web services are protected from unauthorised accesses or injection attacks, including XML Injection, SQL Injection, by the XML gateway.

As shown in Figure 4.3, the XML gateway is the first to consume XML messages coming from the service consumer (a Bank in this scenario). For every protected web service, the gateway exposes an equivalent service interface along with its corresponding XML Schema with constraints to validate incoming messages. The gateway is also configured with an XML threat policy to block XML-based attacks. The gateway forwards an XML message to its targeted service only if it does not violate any constraints nor the threat policy. Otherwise, the message is blocked.

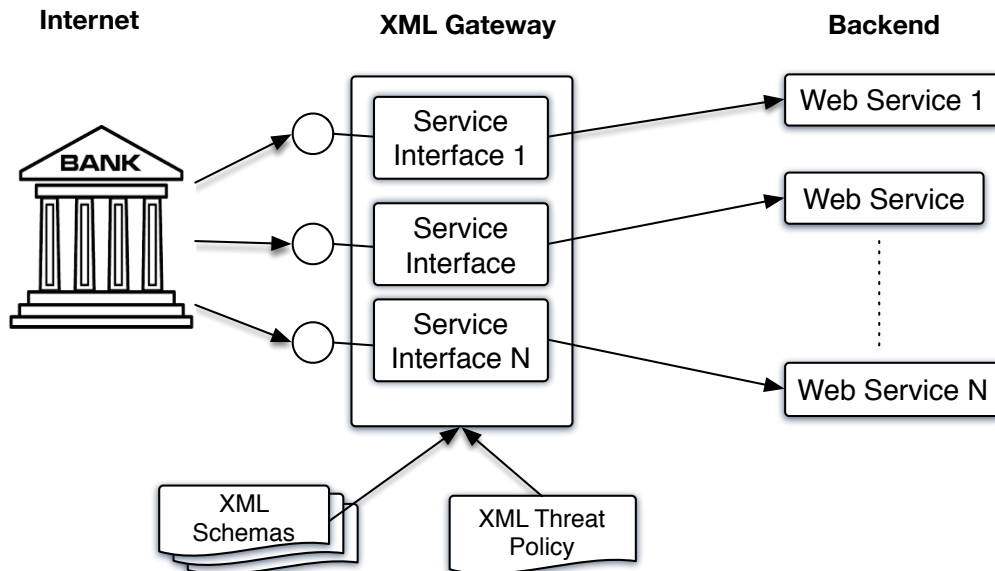


Figure 4.3: XML Gateway and Web services



The 44 web services include a total of 443 distinct XML elements of 75 different data types. Each data type is associated with a set of XSD constraints, such as string length and regular patterns.

#### 4.4.2 Variables

To investigate our first research question about effectiveness, we measure the total number of tests generated ( $T$ ) and the number of tests that successfully bypass the XML gateway ( $T_p$ ).

ReadyAPI exercises each web service parameter with a set of predefined attacks in its dictionary. Therefore, we expect that the value of  $T$  for ReadyAPI is proportional to the number of parameters of the web services under test.

SOLMI involves two strategies. Applying the mutation operators `MO_der_meta`, `MO_der_att`, and `MO_clo`, behaves similarly to ReadyAPI because each parameter is systematically exercised with all the XML meta-characters or the special closing tag (`</test>`). So the number of tests generated by these mutation operators is also proportional to the number of parameters of the web services.

The second strategy for SOLMI involves applying `MO_replica` or `MO_replace` with constraint solving. So their  $T$  value is not only dependent on the number of parameters but also on the constraints specified in the XSDs of the services. Since the constraint solver can find multiple content solutions for an XML element, the total number of tests can be very large. In the considered services, since there are several web services that have more than 40 XML elements, this would lead to a large number of generated tests. Therefore, we cap the maximum number of new tests obtained for each XML element to 20.

The second variable  $T_p$  is the number of distinct tests that successfully bypass the XML gateway. It is used to evaluate the effectiveness of the generation tools. It also captures the security level of the gateway: the greater the value of  $T_p$ , the greater the risk to the underlying web services. Since our tests contain malicious content, the XML gateway should detect and block them if it works in accordance to its intended requirements.

To answer the second research question regarding cost, we measure the time to generate tests  $G_{time}$  and execution time  $E_{time}$ . The test generation time  $G_{time}$  is important to assess the scalability of our approach, especially for the two mutation operators `MO_replica` or `MO_replace` because they involve constraint solving.  $G_{time}$  may not be significant for

ReadyAPI since it applies directly a set of attacks in its dictionary to a parameter and no significant computation overhead is involved.

### 4.4.3 Results

We applied ReadyAPI and SOLMI to all the 44 web services, protected by the XML gateway. Table 4.3 shows the obtained results in terms of the total number of tests  $T$ , the number of tests that successfully bypassed the gateway ( $T_p$ ), the percentage of bypassing tests, the test generation time  $G_{time}$  and test execution time  $E_{time}$ .

ReadyAPI resulted in 4430 tests which were submitted to the XML gateway. Only a small number (2.37%) of these tests were able to bypass the gateway. We investigated these bypassing tests and found that they bypassed because they were neither malformed nor malicious, despite what could be expected.

Regarding SOLMI, the operator `MO_der_att` was not applicable to the subject because no attributes were used in these web services. The `MO_der_meta` mutation operator, which mutated each web service parameter with the XML meta-characters, resulted in a total of 1772 tests for the 44 web services. These tests were then submitted to the XML gateway and were all blocked by the gateway. This happened because the tests (XML messages) generated by the mutation operator were malformed and hence were rejected by the XML gateway. The results for `MO_clo` are similar except that it generated a lower number of tests (443). This is due to the fact that `MO_clo` mutates each parameter once with the random closing tag.

Regarding the operators `MO_replica` and `MO_replace`, each mutation operator resulted in 3236 tests. All the tests generated with the `MO_replica` were blocked by the gateway because they violate a cardinality constraint specified in the XSDs of the web services. The most successful mutation operator was `MO_replace`, which generated 78.86% bypassing tests. The XML gateway failed to block them. Recall that `MO_replace` generates *XMLi* attacks of Type 4, which are the most difficult to detect. These tests contain malicious content that can potentially harm the web services.

This high bypassing rate is attributed to our approach based on constraint solving since the generated tests satisfy all XSD constraints by design, and therefore, most of them are able to break the first layer of defence. This concludes the first research question:

## 4.4 EXPERIMENTAL EVALUATION

**Table 4.3:** Summary of the results with ReadyAPI and SOLMI

Test Strategy	MO Operator	#Tests ( $T$ )	#Bypass. Tests ( $T_p$ )	%	Gen.	Exec
					Time $G_{time}$ (s)	Time $E_{time}$ (s)
ReadyAPI	Malformed XML	4430	107	2.37	1 min	55.62
SOLMI	MO_der_meta	1772	0	0	0.7	51.46
	MO_clo	443	0	0	0.35	9.97
	MO_der_att	NA	NA	NA	NA	NA
	MO_replica	3236	0	0	2929.99(≈49 min)	140.21
	MO_replace	3236	2552	78.86	2998.97(≈50 min)	160.99

*SOLMI and ReadyAPI were able to generate tests that bypass the gateway. Among the two, SOLMI was significantly more effective as 78.86% of its generated tests were able to reach the web services.*

For the second research question, we measured the test case generation time  $G_{time}$  and test execution time  $E_{time}$ . ReadyAPI does not distinguish the two time values. However, we noted that for each web service, the test generation took less than 1 second that amounts to less than a minute for all the 44 web services. Similarly for SOLMI with MO\_der\_meta, MO\_der\_att, and MO\_clo, all the tests were generated in less than a minute.

Test generation for SOLMI with MO\_replica and MO\_replace, which involved constraint solving, is expectedly more expensive. SOLMI was able to generate the 3236 total tests in about 50 minutes, that is 0.92 second per test. However, we also observed a variation among the test case generation time for different web services. Some XML elements have more sophisticated constraints than others, thus requiring more time for the solver. Despite such variation, the worst time to generate a test case was still less than a few seconds. Hence, our approach is scalable and applicable, in terms of test generation time, in the practical context where such testing is taking place.

As expected, the test execution time is low for all approaches ranging from 9 seconds to a few minutes for all the 44 web services. It has therefore no practical incidence. The execution time of ReadyAPI and SOLMI (MO\_der\_meta, MO\_der\_att, and MO\_clo) is small because most or all of their tests are malformed, and thus, are detected quickly and blocked by the gateway, i.e. the web services are not involved. On the contrary, the tests generated by SOLMI with MO\_replica or MO\_replace are well-formed, and many of them reach the web services. Therefore, the execution time of MO\_replica or MO\_replace is higher than that of the others, though it has no practical consequences.

This addresses the second research question:

*SOLMI, with MO\_replica or MO\_replace, takes by far the longest time to generate tests. However, the average amount of time for SOLMI to generate a test case is 0.92 second (or less than one hour for all 44 web services), which is in practice fully acceptable.*

#### 4.4.4 Discussion

The first strategy of SOLMI with MO\_der\_meta, MO\_der\_att, and MO\_clo could not generate any test that bypassed the XML gateway. This suggests that a simple application of fuzzing, a quite popular approach to such testing, may not be effective with realistic applications. Such test strategy may convey the misleading conclusion that a system is secure while it is not.

The results obtained from ReadyAPI tool are not reliable since only 2.37% of the tests were able to bypass the gateway and were found, surprisingly, not to be malicious. The tool was therefore not able to generate effective attacks. Figure 4.4 depicts a test generated by ReadyAPI, that bypassed the gateway. ReadyAPI mutated only the parameter *MerchantLocality* with the character ‘;’, while keeping the original values for the other parameters. This test is well-formed and does not contain any malicious content. It was therefore allowed by the XML gateway to reach its target web service.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:lu="...">
  <soapenv:Header/>
  <soapenv:Body>
    <lu:perform>
      <lu:resInput>
        <lu:UserName>...</lu:UserName>
        ...
        <lu:MerchantLocality>LUXEMBOURG;</lu:MerchantLocality>
        ...
        <lu:MerchantCountry>...</lu:MerchantCountry>
      </lu:resInput>
    </lu:perform>
  </soapenv:Body>
</soapenv:Envelope>

```

Injected Character  
by the ReadyAPI

**Figure 4.4:** An example of a test case generated by ReadyAPI, sensitive information was pixelated.

The effectiveness of SOLMI with MO\_replace is attributed to the constraint solving and mutation based-test generation technique. The higher bypassing rate of 78.86% was made

possible by the test strategy of generating well-formed, valid, and yet malicious XML messages. All these tests indeed include malicious content. Consider the test case shown in Figure 4.5, it was generated by SOLMI for a specific web service in our SUT. It contains 4 parameters *UserName*, *IssuerBankCode*, *RequestId* and *CardNumber*. The value of the parameter *RequestId* includes malicious content (“0 || 1=1” - an SQLi attack), which was generated using the Hampi constraint solver. The application `MO_replace` has commented out the previous element and inserted the new *RequestId* element along with the malicious content. The values of the other three XML elements were kept from the original XML message. If the web service directly concatenates the received parameter values into a SQL Select query, the resulting query becomes:

```
select * from Cards where RequestID = 0 || 1=1
```

Upon execution, this SQL query can result in disclosure of sensitive information from the company’s database. Similarly, other tests generated by SOLMI that were not blocked by the gateway could also result in security breaches.

In practice, web services rely on the XML gateway for security. Our results indicate that the gateway in our case study, which is operated in a professional and critical context, is not protecting web services against sophisticated attacks generated by SOLMI and is, therefore, vulnerable to *XMLi*. This is in contrast with the misleading results of simple attacks based on fuzzing. As malicious tests can reach the target web services, it is highly probable that web services can be compromised when a proper sanitisation of inputs is missing, as it is often the case in practice. We recommend that web services should adopt the principle of *defence in depth* for security i.e. providing security at multiple layers. They should not only rely on the XML gateway for security but also provide their own security mechanism to protect against known vulnerabilities such as XML Injection.

SOLMI is not limited to only SQL injections. It is a generalizable approach and can be easily adopted to test web services for other types of nested attacks. For instance, to test web services for Cross Site Scripting attacks using SOLMI, it suffices to replace the SQLi grammar with a Cross Site Scripting (XSS) grammar to generate valid tests that contain XSS attack payloads. Such grammars are often available already (e.g. [48]) and can be utilized.

```

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:lu="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <lu:perform>
      <lu:resInput>
        <lu:UserName>[REDACTED]</lu:UserName>
        <lu:IssuerBankCode>[REDACTED]</lu:IssuerBankCode>
        <!--
          </lu:IssuerBankCode>
          <lu:RequestId>1424777321381UserNam</lu:RequestId>
          <lu:CardNumber>[REDACTED]</lu:CardNumber>
        -->
        <lu:RequestId>0 || 1=1</lu:RequestId>
        <lu:CardNumber>[REDACTED]</lu:CardNumber>
      </lu:resInput>
    </lu:perform>
  </soapenv:Body>
</soapenv:Envelope>

```

**Figure 4.5:** An example of a test case generated by SOLMI, sensitive information was pixelated.

## 4.5. RELATED WORK

### 4.5.1 Testing for Vulnerabilities in Web Services

Several studies have been carried out on the detection of vulnerabilities in web services [5, 22, 23, 24, 26, 27, 49, 50]. An experimental study performed by Vieira et al. [49] that involved 300 publicly available web services, revealed a large number of vulnerabilities. Most of these vulnerabilities were related to SQL Injections. There exists a large research literature on the detection of SQL Injection vulnerabilities in web services e.g. [40, 51, 52, 53, 54]. However, a very limited research effort has been devoted to testing approaches for XML-based vulnerabilities, especially *XMLi*.

In the previous chapter, we have studied two XML-based vulnerabilities called *Billion Laughs (BIL)* and *XML External Entity (XXE)* in modern XML parsers and found most of them to be vulnerable. Some other work investigated penetration testing for XML-based Denial of Service (XDoS) vulnerabilities in web services [8, 35, 38].

Aydin et al. [55] have used automata-based static analysis to learn vulnerability signatures (i.e, automata) from deliberately vulnerable web applications and to further utilise these automata to generate test inputs for other web applications. In contrast, our work does not

require any deliberately vulnerable applications. Furthermore, they targeted *SQLi* and XSS while we focus on *XMLi*.

To the best of our knowledge, only Rosa et al. [56] targeted the detection of *XMLi* attacks in their research work. They proposed a strategy based on ontology to build a knowledge database and use it for detecting *XMLi* attacks. In the approach, known attack elements are modelled as classes, relationships, and instances in an ontology. It is then used to detect attacks when they occur by matching them with the modelled instances or by checking if they are new attacks on the basis of classes and axioms of the ontology. Their evaluation has shown that the approach performs better compared to the traditional signature-based detection approaches. This work focuses on intrusion detection, not on security testing. In contrast, our work tackles the test generation problem by generating effective *XMLi* attacks using constraint solving.

The OWASP [20] testing guideline defines a testing methodology for XML Injection. It starts with a discovery phase where an XML meta-character is inserted in the SUT. This may reveal information about the structure of the XML. Having this information, the tester can then insert XML data and tags to manipulate the structure or business logic of the web service. OWASP has also provided a tool called WSFUZZER [13] that facilitates SOAP penetration testing with fuzzing features. However, when running the tool with our subject web services, we found that it could not deal with the complex structure of their WSDLs (nested XML elements). This resulted in an execution error, which prevented any comparison with *SOLMI*.

Chunlei et al. [32] also proposed a fuzz testing approach for web services. Fuzzing approaches do not consider the associated domain constraints, hence, their effectiveness is often limited, as shown by our empirical results.

Our proposed approach differs from existing work by accounting for the wide variations in XML Injection attacks as described in Section 4.1. This increases our chances of finding vulnerabilities. Moreover, the integration of constraint solving and input mutation for test generation makes our approach more effective as it produces attacks that are more difficult to detect as they are both well-formed and malicious.

### 4.5.2 XML Test Data Generation

The following work targets functional, not vulnerability testing. But they share one of our starting premises, which is the use of XML Schemas, and also produce XML documents.

There exists a large research body investigating the generation of test data for XML-based web services [45, 46, 57, 58]. Havrikov et al. [45] proposed a search-based test generation technique for XML-based systems. A tool named *XMLMate* has been implemented that uses program structure and existing XML schemas and inputs, to generate new and valid XML test inputs. The tool uses genetic algorithms to evolve a random or sample initial population to achieve higher branch coverage. Experimental evaluation of *XMLMate* yielded good results in terms of finding unique failures in test subjects.

Xu et al. [58] proposed a testing methodology for XML-based communication in accordance with an XML Schema. Web services and applications are tested with respect to how well they validate input XML messages according to their XML Schemas. The method generates test cases from existing XML messages and schemas via schema perturbation operators. These test cases are then used to transmit invalid data to the web service or application. An evaluation on two web services revealed only 33 % of the known faults. A similar approach was presented by Offutt et al. [57] where data perturbation is used for testing web services. Test cases are generated by modifying data values and their interaction based on the types defined in the XML schema.

The approaches discussed here for XML test data generation target the functional testing of web services. They do not focus on vulnerability testing. However, the output of their approaches might be used in SOLMI to start our generation procedure.

## 4.6. Summary

We discussed a taxonomy of all known *XMLi* types and proposed an effective testing approach for *XMLi* in web services based on constraint solving and input mutation. The effectiveness of a testing approach is highly influenced by the quality of test data generation. We have focused on a test generation strategy that generates attack payloads (malicious content) which satisfy the associated domain constraints defined using XML Schema Definition (or WSDL) and attack grammars (e.g., SQL injections in our experiments). The malicious content generation is automated using a constraint solver (Hampi). Test cases (i.e., malicious XML messages) are, then, generated by mutating existing XML messages and combining them with malicious content to generate nested attacks. As a result, generated tests are valid according to XSD constraints, yet malicious at the same time.

We have carried out an experimental evaluation to compare our proposed approach with a state-of-the-art tool based on fuzz testing and known attack patterns. Our subject is a real-



world financial system with an XML gateway at the front-end that is protecting the 44 back-end web services, including a total of 443 input parameters. Our approach (SOLMI) using constraint solving and input mutation delivers promising results. Approximately 78% of the tests generated by SOLMI, which were all attacks with malicious content, were able to bypass the XML gateway and reached the target web services. Only 2.37% of the tests produced by ReadyAPI, a state-of-the-art commercial tool, could bypass the gateway. Furthermore, against expectations, these tests turned out to be non-malicious. In other words, our tool was able to find vulnerabilities in a professionally configured gateway whereas fuzz testing was misleading in suggesting it was secure. Despite using a constraint solver, the computing cost of using SOLMI is affordable in practice as it takes 0.92 seconds on average to generate each test case.

## Chapter 5

# Testing Front-end Web Applications for XML Injections

This chapter focuses on the automated testing for XML injections (*XMLi*), a prominent family of attacks that aim at manipulating XML documents or messages to compromise XML-based applications. More specifically, we target the front-end web applications of SOA systems, i.e., front-end web applications are the systems under test (SUTs) in our context. Among other functionalities, they receive user inputs, produce XML messages, and send them to services for processing (e.g., as part of communications with SOAP and RESTful web services [10, 11]). Such user inputs must be properly validated to prevent *XMLi* attacks. However, in the context of large web applications with hundreds of distinct input forms, some input fields are usually not properly validated [59]. Moreover, full data validation (i.e., rejection/removal of all potentially dangerous characters) is not possible in some cases, as meta-characters like “<” could be valid, and ad-hoc, potentially faulty solutions need to be implemented. For example, if a form is used to input the message of a user, emoticons such as “<3” representing a “heart” can be quite common. As a consequence, front-end web applications can produce malicious XML messages when targeted by *XMLi* attacks, thus compromising services that consume these messages.

As described in Section 1.2, fuzz testing approaches (e.g., ReadyAPI [12], WSFuzzer [13]) are not capable of detecting *XMLi* vulnerabilities in web applications. They can generate only simple test cases using XML *meta-characters* (e.g., <), which are blocked by the applications. Furthermore, some attacks could be based on the combination of more than one input field, where each field in isolation could pass the validation filter unaltered.

---

In this chapter, we propose an automated and scalable approach to search for test cases (attacks) that are effective at detecting *XMLi* vulnerabilities. Given that the SUT is a web application that communicates with web services through XML messages, we first identify a set of possible malicious XML messages (called *test objectives*, or TOs for brevity, in this dissertation) that the SUT can produce and send to those services. These TOs are identified using fully automated tool SOLMI (described in the previous chapter), that creates malicious XML messages based on known XML attacks and the XML schemas of the web services under test. Then, we use a specifically-tailored genetic algorithm to search the input space of the SUT (e.g., text data in HTML input forms) in an attempt to generate XML messages matching the generated TOs. Search is guided by an objective function that measures the difference between the actual SUT outputs (i.e., the XML messages toward the web services) and the targeted TOs. Our approach does not require access to the source code of the SUT and can, therefore, be applied in a black-box fashion on many different systems. The current chapter focuses on the generation of test inputs and is complementary to the automated solution for generating TOs that we presented in the previous chapter.

Note that proper input validation in the front-end can prevent many of the possible security attacks. However, in the context of large web applications with hundreds of distinct input forms, some input fields are typically not properly validated as a result of time pressures, changes, and lack of security expertise.

Furthermore, some attacks could be based on the combination of more than one input field, where each field in isolation could pass the validation filter unaltered. In some cases, full data validation (i.e., rejection/removal of all potentially dangerous characters) is not possible, as meta-characters like `<` could be valid, and ad-hoc solutions need to be implemented (which could be faulty). For example, if a form is used to input the message of a user, emoticons like `<3` representing a “heart” can be quite common.

We have carried out an extensive evaluation of the proposed approach on two case studies. The first study consists of 20 experiments on six web applications that simulate bank interactions with an industrial bank card processing service. These web applications have different levels of complexity in terms of the number of inputs, their data types and the validation technique. The second study includes a third-party application used for training purposes and an industrial web application having millions of registered users, with hundreds of thousands of visits per day. Results are promising, as our proposed search-based testing approach is effective at detecting *XMLi* vulnerabilities in both case studies, within practical execution time. The evaluation of our approach on such diverse systems, including a large industrial web application, is a sizable and useful empirical contribution.

The remainder of the chapter is structured as follows. Section 5.1 describes the testing context for this chapter. Section 5.2 presents our proposed approach and the tool that we developed for its evaluation. Section 5.3 reports and discusses our evaluation on two case studies including research questions, results and discussions. Section 5.4 discusses related work. Finally, Section 5.5 concludes the chapter.

### 5.1. Testing Context

A SOA system typically consists of a front-end web application that generates XML messages (e.g., toward SOAP and RESTful web services) upon incoming user inputs (as depicted in Figure 5.1). The front-end system often performs various transformation techniques on the user inputs before generating the XML messages, e.g., encoding, validation or sanitisation. XML messages are consumed by various back-end systems or services, e.g., an SQL back-end, that are not directly accessible from the net. In this chapter, we focus on the front-end web application and aim to test if it is vulnerable to *XMLi* attacks. We consider the web application as a black-box. This makes our approach independent from the source code and the language in which it is written (e.g., Java, .Net, Node.js or PHP). Furthermore, this also helps broaden the applicability of our approach to systems in which source code is not easily available to the testers (e.g., external penetration testing teams). However, we assume to be able to observe the output XML messages produced by the SUT upon user inputs. To satisfy this assumption, it is enough to set up a proxy to capture network traffic leaving from the SUT, and this is relatively easy in practice.

The security of the front-end plays a vital role in the overall system's security as it directly interacts with the user. Consider, for instance, a point of sale (POS) as the front-end that creates and forwards XML messages to the bank card processors (bank-end). If the POS system is vulnerable to *XMLi* attacks, it may produce and deliver manipulated XML messages to web services of the bank card processors. Depending on how the service components process the received XML messages, their security can be compromised, leading to data breaches or services being unavailable, for example.

### 5.2. Approach

This section describes our search-based testing [60] approach to detect *XMLi* vulnerabilities. We first describe the TOs that are used to guide the search for malicious test inputs and

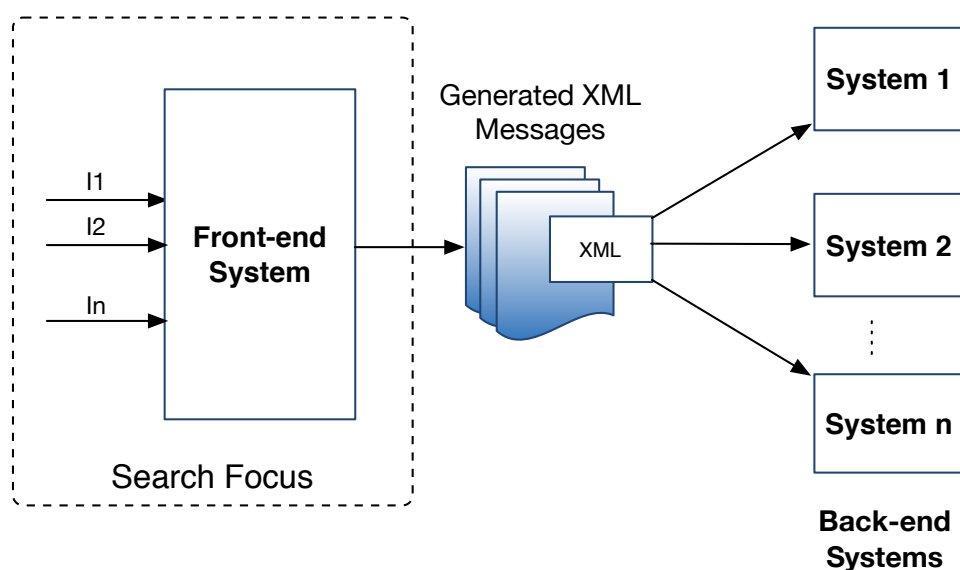


Figure 5.1: Testing Context

demonstrate the presence of vulnerabilities. A search-based technique that generates inputs to reach such TOs is then introduced. Along with the discussion of the technique, we describe in detail its building blocks, including input encoding and the fitness function.

### 5.2.1 Test Objectives (TOs)

In our approach, TOs are specific outputs of the SUT (i.e., XML messages to web services) that contain malicious content. If there exist inputs (e.g., forms in HTML pages) that can lead the SUT to generate such malicious XML outputs, then the SUT is considered to be vulnerable.

A TO is said to be covered if we can provide inputs which result into the SUT producing the TO. Our focus in this chapter is to search for such user inputs. Since the TO is malicious by design, the SUT is not expected to do so unless it is vulnerable. In other words, we search for user inputs that can lead the SUT to generate malicious messages and send them to the web services behind the corporate firewall, which cannot be contacted directly by an attacker. Sending such TOs to the backend systems/services could severely impact them depending on the malicious content that these TOs carry.

We define four types of TOs based on the types of *XMLi* attacks described in Chapter 4: *Type 1: Deforming*, *Type 2: Random closing tags*, *Type 3: Replicating* and *Type 4: Replacing*.

The intent and impact of each of these *XMLi* attack types are different. *Type 1* attacks aim to create malformed XML messages to crash the system that processes them. *Type 2* attacks aim to create malicious XML messages with an extra closing tag to reveal the hidden information about the structure of XML documents or database. Finally, *Type 3* and *Type 4* aim at changing the XML message content to embed nested attacks, e.g., SQL injection or Privilege Escalation.

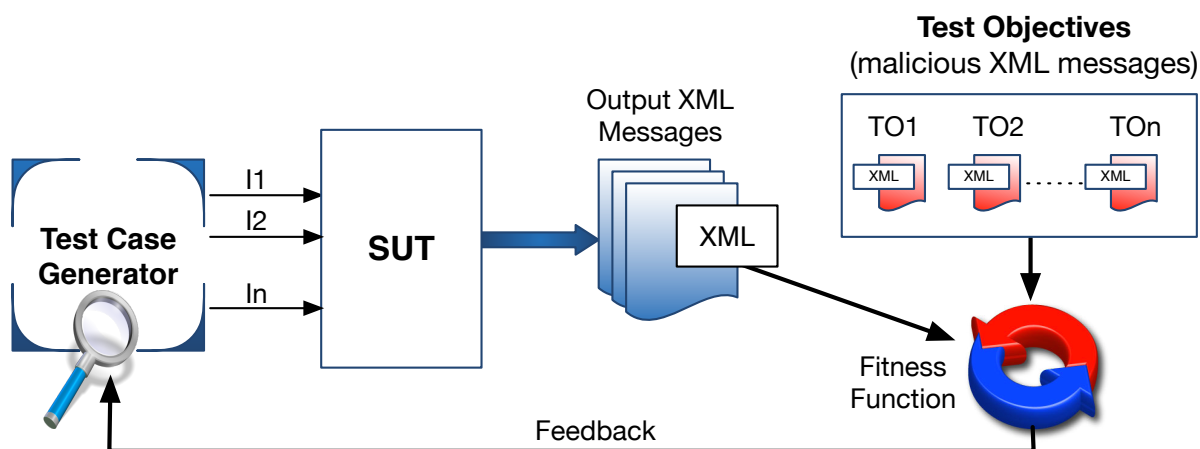
To obtain TOs for a given SUT, we first sample diverse and non-malicious output XML messages (recorded during normal execution of the SUT). Then, we automatically modify those XML messages to inject malicious intent in them. In Chapter 4, we have developed an approach and tool to automate the TO generation. Such automation requires sophisticated techniques, as we need to be able to generate XML data that is not only syntactically valid, but also conformant to their given schema (e.g., in XSD format). Such schema might have non-trivial constraints. This is important, as otherwise wrongly formatted XML messages would be trivially discarded by the parsers in the web services processing them.

Ensuring diversity in the generated TO set is important to the success of our technique. Since we consider the SUT as a black box, we do not know a priori how inputs are related to output XML messages. Having a diverse set of TOs increases our chances of figuring out such relationships and detecting *XMLi* vulnerabilities. Therefore, when generating TOs, we make sure that each XML element and attribute of the messages is modified at least once with all the types of *XMLi* attacks described in the previous chapter.

Consider the example of user registration given in Section 2.2.1: Figure 2.4 shows a possible TO where the *userid* element is manipulated, i.e., the original element *userid* has a value 500, which has been commented out and replaced with the new *userid* element having a value of 0.

### 5.2.2 Search-based Testing

Once a set of TOs for the SUT is generated, we want to test whether the SUT, with certain user inputs, can produce any of these TOs. If the SUT does not properly block malicious inputs that lead to any of the TOs, it is considered vulnerable. The testing problem is now defined as a search problem: seeking for malicious input values (i.e., *XMLi* attack strings) that, when submitted to the the SUT, lead the SUT to produce XML messages matching the TOs.



**Figure 5.2:** The overall search-based approach to generating tests for reaching the TOs.

As the number of possible input strings for the SUT can be extremely large, random testing without any guidance would most likely be ineffective. Also, since we consider the SUT as a black-box and hence we are not aware of how user inputs are transformed into output XML messages, it is infeasible to use a simple deterministic algorithm based on direct input-output matching. Therefore, we use a search algorithm, namely a genetic algorithm (GA) [61], that seeks to evolve the output of the SUT towards the TOs. Given a TO and an output XML message, the objective (fitness) function guiding search is measured by the *distance* between them.

GA is inspired by the mechanisms of natural adaptation. In a nutshell, it starts from an initial population of individuals (encoded in *chromosomes*) and iteratively evolves them by selecting elite individuals (having the best fitness) and producing offspring by applying crossover and mutation operators.

Our GA-based approach is depicted in Figure 5.2. Each test case is a GA individual, composed of a number of string values to be assigned to input parameters of the SUT. The number of input strings depend on the number of inputs of the SUT. For each TO, our approach first generates an initial population of random test cases. Then, it iteratively selects and evolves them based on the feedback from the fitness function. This process is repeated until optimal fitness is achieved (outputs match a TO) or we run out of computational budget (e.g., timeout). Note that we consider each TO separately. TOs are independently created based on different types of *XMLi* attacks. The coverage of different TOs requires different test cases and, hence, we cannot consider the coverage of multiple TOs at the same time.

In the subsequent sections we discuss in detail how test cases are encoded and the way we define and measure the fitness function.

### 5.2.2.1 GA Encoding and Reproduction

The encoding of chromosomes is a pivotal task when addressing a search problem with a GA. For a given SUT with  $N$  input parameters, we represent a test as a chromosome of  $N$  genes. Each of these genes is one string corresponding to an input parameter. We consider every input of the SUT as a string regardless of its original data type, since (i) inputs to the web application SUT will be sent through HTTP and (ii) malicious *XMLi* attacks are strings. These input strings can contain alphanumeric and special characters (e.g., %, || or &).

By default, we consider all the printable ASCII characters as the alphabet for the input strings. However, depending on the given TOs, we can reduce the alphabet to containing only characters that can possibly compose the TOs. We investigate this point further in Section 5.2.2.3.

Each input string is represented as an array of ASCII characters. The lengths of the arrays for the genes are set based on the expected maximum lengths of the corresponding input parameters. We use a special symbol to denote the “empty” character, i.e., absence of character, to fill up the array when the input string is shorter than the maximum length. Furthermore, in this way the lengths of input strings can vary during the search, as these empty spaces can be filled with new characters.

Reproduction to create offspring is done through standard crossover and mutation operators. Crossover is a genetic operator that is used to vary the programming of chromosome pairs from one generation to the next: a pair of individuals is selected, the chromosome of each is cut into two parts at the same random cut point, two new offspring are formed by concatenating the head part of one parent with the tail part of the other parent. Mutation on an offspring is done by randomly selecting a position in the chromosome and swap its corresponding character with a new one that is randomly selected from the alphabet.

To select chromosomes for crossover and mutation, we use the binary tournament selection technique [62] as recommended in the literature [63, 64].

### 5.2.2.2 Fitness Function

The fitness (objective) function that guides our search for effective test inputs is defined as the distance between the output XML message and the TO. Given a TO and an individual (i.e., a test case consisting of inputs to the SUT), its fitness is measured as the distance between the XML message that the SUT produces upon the execution of the test case and



the TO. Our search evolves such individuals, aiming at reducing the distance so that the TO can be reached.

A TO is essentially a malicious XML document that might be malformed or contain potentially dangerous values (e.g., SQL injection). The structure of the TO might, therefore, be broken. As a result, we consider TOs as plain text, i.e. just as strings. We use string edit distance (also known as Levenshtein distance) between pairs of strings for the fitness measure. In short, edit distance between two strings is the minimum number of editing operations (inserting, deleting, or substituting a character) required to transform one into the other. It is also supported by fast algorithms [65] and open source implementations.

Consider the following simple example of a TO of a SUT that has a single input parameter  $I$ . The TO is represented as the string:

```
<test>data OR 1=1</test>
```

Upon a test case  $t$  in which  $I = "OR \%"$ , the SUT generates the following XML message:

```
<test>data OR %</test>
```

The fitness value of  $t$  is measured as the edit distance between the TO and the XML message, which in this case is equal to three, as we need to modify the “%” character into “1”, and then add the two characters “=1”.

The lower the fitness value is, the closer we are to cover a TO. We consider the TO to be covered when the corresponding fitness value is 0. It means that the generated XML message and the TO are identical.

### 5.2.2.3 Reducing the Search Space

The search space for a SUT is characterized by three factors: the number of input parameters of the SUT, the maximum string lengths for the values of those inputs, and the alphabet from which strings are created. They should be controlled in order to reduce the search space and improve the performance of the GA. The number of input parameters of the SUT is normally fixed. However, assuming that some domain knowledge is available i.e., if we know that some parameters are not involved in producing XML outputs, then they can be excluded from the search. Their values can be fixed with valid data taken from the functional test suite of the SUT.

The maximum string length values, i.e., the size of the genes, corresponding to the input parameters of the SUT, should also be adapted to the nature of the parameters. For instance, parameters for *name* and *password* are often shorter than a parameter for *description*. Again,

if we know any upper bound for input lengths, we should limit the gene size accordingly. Nevertheless, since the input strings are used or concatenated to create XML messages, their lengths are smaller than the lengths of the XML messages. As a result, we should always set the string lengths smaller than the lengths of TOs.

We should also consider to restrict the alphabet to contain only the specific set of characters that appear in the TOs. If the TOs do not contain some characters, we omit them from the alphabet. As the GA has to work with fewer characters in this case, we expect an improvement in performance. We investigate the benefit of limiting the alphabet in the experimental evaluation.

### 5.2.3 Tool Implementation

We developed a tool in Java that implements the technique presented in this chapter. The inputs of the tool include the TOs that contain malicious intents for a target SUT. The tool generates test cases, which include the values for the input parameters of the SUT (e.g., input values in HTML forms). When these tests are run, the tool compares the XML outputs of the SUT to the TOs for fitness calculation. The search is guided by this fitness function for generating new test cases. The process is repeated until the TOs are covered or the tool runs out of time.

The main components of the tool are: Test Case Generator and Test Executor. The test case generator is the core component of the tool. It is implemented on top of jMetal [66] - an object oriented, Java-based framework for optimization. The test executor provides an interface between the SUT and the test case generator. It takes the input values generated by the test case generator and submits them to the SUT (e.g, through a HTTP POST with all the needed cookies set up). The XML messages sent by the SUT toward the web services need to be intercepted and then sent back to the test case generator where the fitness is calculated. This can be easily done by setting up an HTTP proxy between the SUT and the web services.

The test case generator is generic and can be used with any application. The test executor has been modularized so that it can easily be instantiated for a specific SUT that has a different user interface (e.g., HTML web forms with different parameter names) and that generates XML files in different ways (e.g., SOAP messages or data bodies in HTTP POST messages toward RESTful web services).

## 5.3. Evaluation

We evaluate the effectiveness in vulnerability detection and the execution cost of the proposed approach through a series of experiments grouped into two studies, namely *Study 1: Controlled Experiments* and *Study 2: Third-party Applications*. The common characteristic of the subject applications in these studies is that they receive user inputs, produce XML messages, and process or send them to associated web services. In Study 1, we have two front-ends, called *SBank* and *SecureSBank*, that simulate a bank’s interactions with a real-world bank card processing service<sup>1</sup>. Since we designed these front-ends, we can control the number of inputs and validation mechanisms in them to investigate how they influence the effectiveness of our approach.

Differently from Study 1, Study 2 involves third-party independent subject applications. They enable us to evaluate how well our approach scales and to which extent our results generalize.

### 5.3.1 Research Questions

In this chapter, we investigate the following six research questions:

**RQ1** [*Effectiveness*]: *To what extent is our search-based approach effective in detecting XMLi vulnerabilities?*

Since our TOs correspond to malicious XML messages, being able to identify inputs to generate them would demonstrate that the SUTs are vulnerable. Our approach is therefore deemed effective if it can find input strings that lead to the production of TOs, i.e., cover TOs.

**RQ2** [*Comparison with Random Search*]: *How does our search-based approach perform compared to random search?* Random search [62] is typically adopted as a baseline in SBSE research [15].

**RQ3** [*Cost*]: *What is the cost, in terms of execution time, of applying the proposed approach?*

We should consider the cost for deriving TOs and the computational cost of the GA. Since the TO derivation is addressed in the previous chapter and is not the focus of the current chapter, we only discuss here the input search cost. As GAs are notorious for being

---

<sup>1</sup>The name of the company cannot be revealed due to non-disclosure agreements.

computationally expensive, we investigate whether the cost in terms of execution time affects the applicability of our approach.

**RQ4** [*Impact of Restricted Alphabets*]: *How does restricting the alphabet to the characters in the TO affect the GA performance?*

As described in Section 5.2.2.3, instead of using the complete alphabet of all possible characters, we could use a restricted alphabet by omitting the characters unused in the TOs. It is expected that the performance of the GA would improve as the overall search space would be reduced. We assess whether the magnitude of the improvement is practically significant.

**RQ5** [*Influence of Input Setting*]: *To which degree the number of input parameters and their length settings affect the GA performance?*

When there are more user inputs, the GA has to spend more time searching. Also, the length of user inputs can be variable (Var) and depend on specific inputs, or be fixed (Fix) for all inputs. The former may help reduce the search space, but it requires selecting proper lengths for inputs, which often requires domain knowledge. The latter is easier as one has to select a single maximum length for all inputs, but it might unnecessarily increase the search space. We study these trade-offs in **RQ5**.

**RQ6** [*Influence of Input Validation*]: *Does search-based testing work in presence of input validation?*

The SUT may implement protection mechanisms that do validate the inputs. When such mechanisms detect malicious inputs, they often react to malicious content by generating error messages in the HTTP responses, and often end up not generating any XML to be sent toward the target web services. This is a challenge for the GA, as no useful fitness value would be produced, generating a fitness plateau hampering effective search. It is hence important to investigate whether our proposed approach works in such circumstances, i.e., how it is affected by input validation.

### 5.3.2 Metrics

We rely on the following metrics to help answer the above research questions.

- Coverage Rate ( $C$ ): Coverage rate  $C$  is the ratio of the number of covered TOs over the total number of TOs. We denote the coverage rates for the GA as  $C_{ga}$  and for random search as  $C_{rn}$ .

- Average Execution Time ( $T$ ): Each experiment on a specific TO is repeated 10 times to account for randomness. We then measure the average execution time  $T$  (minutes) across runs and per TO for specific subject applications. We denote the execution time as  $T_{ga}$  and  $T_{rn}$  for the GA and random search, respectively.

### 5.3.3 Study 1: Controlled Experiments

Beside checking the effectiveness of the proposed approach on applications that are known to be vulnerable, this study investigates the influence of the input space (number of inputs, their lengths, and the alphabet) on the GA performance.

#### 5.3.3.1 Subject Applications

Our first study has been carried out with two web applications, *SBank* and *SecureSBank*, that simulate the web front-end of a banking system that receives user inputs, produces XML messages, and sends them to a bank card processing service. The XML messages share the same structure as those used in production. *SBank* and *SecureSBank* were, however, developed in our lab specifically for this study. We wanted to be able to configure the number of user input parameters and apply validation mechanisms. *SBank* and *SecureSBank* were deliberately designed to be vulnerable.

Both applications can have up to three input parameters, including *UserName*, *IssuerBankCode*, and *CardNumber*. The applications generate XML messages using the inputs submitted by the user. Figure 5.3 shows an example of such an output XML message. Note that the *RequestID* element in the XML message is generated by the application automatically. Users are not authorized to tamper with its value unless they do so maliciously. The generated XML messages are then forwarded to the web services of the card processing company.

*SBank* directly inserts the user inputs into the XML elements of the message without validation. This makes the application vulnerable to *XMLi* attacks. *SecureSBank* is similar to *SBank* except that one of the input parameters is validated. The application validates the input parameter *IssuerBankCode* and generates an error message if malicious content is found. The aim of evaluating *SecureSBank* is to test our approach in the presence of input validation.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:lu="http://schemas.xmlsoap.org/soap/envelope/lu/">
  <soapenv:Header/>
  <soapenv:Body>
    <lu:perform>
      <lu:resInput>
        <lu:UserName>Tom</lu:UserName>
        <lu:IssuerBankCode>0231</lu:IssuerBankCode>
        <lu:RequestId>28278111TOM</lu:RequestId>
        <lu:CardNumber>123456789</lu:CardNumber>
      </lu:resInput>
    </lu:perform>
  </soapenv:Body>
</soapenv:Envelope>

```

Figure 5.3: An example of output XML message created by *SBank*.

### 5.3.3.2 Test Objectives

As described in Section 5.2.1, we created TOs based on the four *XMLi* injection techniques proposed in the previous chapter. For *SBank* and *SecureSBank*, TOs are created by applying these four *XMLi* techniques to every element of the sample XML message (Figure 5.3). The *Type 4: Replacing* attack is a more advanced form of *XMLi* that requires at least two XML elements where the value of one must be auto-generated by the application. Therefore this attack can only be applied to the *RequestId* as it is the only element auto-generated by *SBank/SecureSBank*. This results in 10 (3 attacks x 3 elements + 1 attack x 1 element) representative TOs in total.

An example of a TO for *SBank* is shown in Figure 5.4. It contains four XML elements *UserName*, *IssuerBankCode*, *RequestId*, and *CardNumber*. The value of the parameter *RequestId* includes malicious content (“0” || 1 = 1 - an embedded SQLi attack). The application of the *XMLi* technique *Type 4: Replacing* resulted in commenting out the previous element and inserting the new *RequestId* element along with malicious content. If the web service that consumes such malicious TO executes a SQL query by directly concatenating the received values into it, then the resulting query may get executed:

```
Select * from Cards where RequestID = "0" || 1=1
```

The condition in this SQL query is a tautology and results in returning all cards’ information when executed. If the front-end system can generate this XML message from user inputs, it is considered vulnerable to *XMLi* attacks.

```

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:lu="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <lu:perform>
      <lu:resInput>
        <lu:UserName>[REDACTED]</lu:UserName>
        <lu:IssuerBankCode>[REDACTED]</lu:IssuerBankCode>
        <!--
          </lu:IssuerBankCode>
          <lu:RequestId>1424777321381UserNam</lu:RequestId>
          <lu:CardNumber>[REDACTED]</lu:CardNumber>
        -->
        <lu:RequestId>"0" || 1=1</lu:RequestId>
        <lu:CardNumber>[REDACTED]</lu:CardNumber>
      </lu:resInput>
    </lu:perform>
  </soapenv:Body>
</soapenv:Envelope>

```

**Figure 5.4:** An example of test objective containing a malicious attack.

### 5.3.3.3 Experiment Settings

We conducted a number of experiments with different settings of the applications, as represented in Table 5.1. Each row in the table represents one experiment. The column *Exp. ID* is the id of the experiments. The values of *Exp. ID* is based on the corresponding applications (S for *SBank*, SS for *SecureSBank*), the number of inputs considered, whether input lengths are fixed (Fix) for all or are input specific (Var), and whether the alphabet is restricted (Y) or not (N). The *TOs* column lists the number of TOs created for the experiments. The *#Inp.* column lists the number of inputs accepted by the application version. The *Len.* column represents whether the length of the genes in a chromosome is fixed or variable. The gene’s lengths directly correspond to the lengths of the input parameters of the SUT and may affect the GA’s performance. The last column *Res. Alph.* indicates whether the GA uses a reduced alphabet or not.

All but four experiments have 10 TOs, as described in Section 5.3.3.2. Each of these four experiments (i.e. S.1.\* and SS.1.\*) have 9 TOs as they consist of the *SBank/SecureSBank* versions having one input parameter. The missing TO from each of these four experiments is the one generated using the *Type4: Replacing* attack on *RequestID* element as this TO requires at least two input parameters to be covered. It is therefore not coverable in the experiments with application versions having only one input parameter.

For these experiments, a TO is covered when it matches the output XML message,

**Table 5.1:** Experiment Settings for Study 1: SSBank stands for *SecureSBank*, Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), the input length setting (Len), and whether the alphabet is restricted (Res. Alph.).

App.	Exp. ID	#TOs	#Inp.	Len.	Res. Alph.
<b>SBank</b>	S.1.F.N	9	1	Fix	N
	S.1.F.Y	9	1	Fix	Y
	S.2.F.N	10	2	Fix	N
	S.2.F.Y	10	2	Fix	Y
	S.2.V.N	10	2	Var	N
	S.2.V.Y	10	2	Var	Y
	S.3.F.N	10	3	Fix	N
	S.3.F.Y	10	3	Fix	Y
	S.3.V.N	10	3	Var	N
	S.3.V.Y	10	3	Var	Y
<b>SSBank</b>	SS.1.F.N	9	1	Fix	N
	SS.1.F.Y	9	1	Fix	Y
	SS.2.F.N	10	2	Fix	N
	SS.2.F.Y	10	2	Fix	Y
	SS.2.V.N	10	2	Var	N
	SS.2.V.Y	10	2	Var	Y
	SS.3.F.N	10	3	Fix	N
	SS.3.F.Y	10	3	Fix	Y
	SS.3.V.N	10	3	Var	N
	SS.3.V.Y	10	3	Var	Y

meaning that fitness is 0. As GA is a randomized algorithm, to account for the statistical variation, we run it 10 times for each TO. To simplify the discussion and visualization of the experiment results, a TO is considered to be covered if the TO is covered in at least one of these 10 runs.

For each iteration of the TO, the program is terminated when the TO is covered or when there is no improvement in fitness after  $X$  evaluations. The value of  $X$  is determined based on some preliminary experiments. We use the same termination criteria for the search-based



**Table 5.2:** Results for *SBank* for the GA and Random Search in terms of coverage rate  $C_{ga}$ ,  $C_{rn}$  and average execution time  $T_{ga}$ ,  $T_{rn}$  in minutes.

Exp.ID	$C_{ga}$	$C_{rn}$	$T_{ga}$	$T_{rn}$
S.1.F.N	9/9	0/9	13.63	54.75
S.1.F.Y	9/9	0/9	10.53	54.92
S.2.F.N	10/10	0/10	28.30	49.60
S.2.F.Y	10/10	0/10	23.57	50.88
S.2.V.N	10/10	0/10	16.78	42.30
S.2.V.Y	10/10	0/10	12.18	45.48
S.3.F.N	10/10	0/10	31.88	40.42
S.3.F.Y	10/10	0/10	25.20	43.78
S.3.V.N	10/10	0/10	24.13	42.28
S.3.V.Y	10/10	0/10	19.13	47.93

approach and random search.

The GA algorithm described in Section 5.2.2 is generic. More specifically, we use generational GA [62], which is one of the single-objective optimization algorithms available in jmetal [66] and has worked well for our problem based on our preliminary experiments. The GA parameters have also been assigned values based on some small-scale preliminary experiments and are consistent with the “best practices” in the literature [66, 67]. The population size is set to 50, which is in the recommended range of 30-80 [68]. The crossover rate is set to 0.8, which also falls within the recommended range of 0.45 to 0.95 [68, 69]. For the mutation rate, we use the recommendations in [14, 69] i.e.  $\frac{1.75}{\lambda\sqrt{l}}$  where  $l$  is the length of the chromosome and  $\lambda$  is the population size.

#### 5.3.3.4 Results

The results obtained with Study 1 are shown in Tables 5.2 and 5.3 for *SBank* and *SecureS-Bank*, respectively. Our proposed approach achieved 100% TO coverage for the experiments with *SBank*. This was the case for all variants of *SBank* with different numbers of inputs (S.1.\*, S.2.\*, or S.3.\*) and the GA configurations ([F|V].[Y|N]). These results demonstrate that all the variants of *SBank* are found to be vulnerable to *XMLi*.

**Table 5.3:** Results for *SecureSBank* for the GA and Random Search in terms of coverage rate  $C_{ga}$ ,  $C_{rn}$  and average execution time  $T_{ga}$ ,  $T_{rn}$  in minutes.

Exp.ID	$C_{ga}$	$C_{rn}$	$T_{ga}$	$T_{rn}$
SS.1.F.N	6/9	0/9	14.32	52.07
SS.1.F.Y	6/9	0/9	11.70	44.82
SS.2.F.N	0/10	0/10	25.83	38.28
SS.2.F.Y	0/10	0/10	24.20	31.63
SS.2.V.N	6/10	0/10	24.60	40.72
SS.2.V.Y	6/10	0/10	19.62	36.05
SS.3.F.N	0/10	0/10	18.20	27.88
SS.3.F.Y	0/10	0/10	17.95	23.63
SS.3.V.N	6/10	0/10	23.48	36.98
SS.3.V.Y	6/10	0/10	21.05	38.85

Results for *SecureSBank* are provided in Table 5.3. They also show that all the variants of *SecureSBank*, with different number of user inputs (SS.1.\*, SS.2.\*, or SS.3.\*), have at least one configuration of the GA ([F|V].[Y|N]) that covers more than half the TOs, meaning that the variants are found to be vulnerable. However, the coverage rate is smaller than that achieved with *SBank*. This was expected, as with the presence of input validation procedures applied on the input parameter *BankCode*, four (three for SS.1.\*) TOs that require malicious inputs for this parameter are not feasible. For the other TOs, it is also more difficult for the GA to search since, whenever an unexpected input is submitted to *BankCode*, *SecureSBank* produces an error message. Since it differs from the TOs by a large distance, it introduces large fluctuations in the fitness function over time during the search.

The average execution time  $T_{ga}$  per TO ranges from approximately 10 to 31 minutes for *SBank* and 11 to 25 minutes for *SecureSBank*. Such durations in practice are acceptable as they are not expected to impact the pace of development, especially when testing is fully automated and performed off-line, for example on continuous integration systems.

Regarding **RQ2**, random search could not cover a single TO in any configuration as depicted in the  $C_{rn}$  columns in both Tables 5.2 and 5.3. This clearly shows that our search problem is far from trivial. Even with just 10 runs, Fisher exact tests on differences between success rates  $C_{rn}$  and  $C_{ga}$  provide very small  $p$ -values close to 0. Also the execution time  $T_{rn}$  for random search in each experiment is much higher than their counterparts in the GA

experiments.

Regarding the research questions **RQ1-RQ3**, we find that:

*The proposed approach is highly effective in searching for inputs to detect XMLi vulnerabilities and performs much better than random search. The average execution time per TO is acceptable in practice.*

All of the experiments that used a restricted alphabet (Exp. IDs ending with Y) performed better in terms of execution time. For instance, the average execution time of S.3.F.Y (which used a restricted alphabet) per TO is 25 minutes whereas its counterpart S.3.F.N (which used the full alphabet) stands around 31 minutes. For **RQ4**, we find that:

*Using a restricted alphabet for the GA helps significantly reduce the execution time.*

The effects of varying the number of user inputs on the GA performance are evident from the result tables of both subject applications. The application variants with one input parameter (S.1.\*, SS.1.\*) achieved the best results in terms of execution time. The variants with two input parameters performed better compared to their three-parameter counterparts. For example, S.2.\* achieved the same TO coverage in less time compared to S.3.\*.

The results of both applications indicate that keeping the input length fixed is not a viable option. The experiments where the lengths of the input parameter's lengths were variable (specific to parameters) achieved much lower execution times compared to their fixed length counterparts. For example, S.3.V.\* with variable length took less time than its counterparts S.3.F.\*. The results of *SecureSBank* comparing length options are even more interesting, as none of the configurations with two and three input parameters with fixed length (S.2.F.\*, S.3.F.\*) could cover any TO. In contrast, the variable length configurations performed much better. For instance, SS.2.V.\* with variable length achieved 6/10 coverage while its counterpart SS.2.F.\* with fixed length did not cover any TO.

Despite such observations, it is important to note that the differences between variable and fixed length options for the input parameters depend on the variation in their actual lengths. If there exists a significant variation in the actual lengths while the GA is configured to use the same length for all parameters, then the GA performance will be considerably affected. For instance, for *SBank* and *SecureSBank*, the length of the *CardNumber* parameter is 16 characters while the *BankCode* is only four characters. On the other hand, if the lengths of the parameters differ by only a few characters, then keeping it fixed (maximum length) for all parameters will not have much effect on the coverage or execution time. In short, in addressing **RQ5**, we find that:

*The number of user input parameters affects the execution time. Furthermore, using a maximum and identical fixed length for all user inputs does adversely affect coverage as well as execution time.*

Regarding input validation for **RQ6**, all variants of the application *SecureSBank* achieved much lower coverage (Table 5.3) compared to the non-validated *SBank* variants (Table 5.2). However our approach still covered more than half of the TOs in total (excluding the four infeasible TOs) for *SecureSBank*.

### 5.3.4 Study 2: Third-party Applications

We focus exclusively on the most effective experimental settings identified in Study 1 for these applications. The results of these experiments provide additional evidence to answer **RQ1**, **RQ3** and **RQ4**, The other RQs were not further investigated due to space and technical constraints, as further described below.

#### 5.3.4.1 Subject Applications

Two subject applications considered in this study are *XMLMao* and *M* (an arbitrary name to preserve confidentiality).

- **XMLMAO:** *XMLMao* is a deliberately vulnerable web application for testing XMLi vulnerabilities. It is part of the Magical Code Injection Rainbow (MCIR) [70] framework for building a configurable vulnerability test-bed. The application has a single user input parameter. It inserts inputs directly into one of the four locations in the output XML message, depending on *XMLMao*'s setting. *XMLMao* is written in PHP and consists of 1178 lines of code.
- **M:** *M* is an industrial web application with millions of registered users and hundreds of thousands of visits per day. The application itself is hundreds of thousands of lines long, communicating with several databases and more than 50 corporate web services (both SOAP and REST). Out of hundreds of different HTML pages served by *M*, in this chapter we focus on one page having a form with two string inputs. Due to non-disclosure agreements and security concerns, no additional details can be provided on this case study.

### 5.3.4.2 Test Objectives

To create TOs for *XMLMao*, we apply the first three *XMLi* mutation techniques (Types 1–3) to the sample XML message of the application. *Type 4: Replacing* is not applicable as it requires two user inputs. Since there are four locations of insertion in the XML message, we create 12 (3 x 4) TOs.

We only used four TOs for *M* (one per type), as the experiments had to be run on the laptop of one of the engineers working on *M*, as opposed to a cluster for other experiments. The TOs come from one of the SOAP web services invoked by the SUT when the target HTML form is submitted. As *M* is a large and complex system, when run on a laptop, instead of a high performance cluster of servers, response times are slow and this makes fitness evaluation more time consuming (most of the 32 minutes in Table 5.5). Even if response times for a single user are still in the order of tens of milliseconds, this made running a large number of experiments on *M* impossible, thus resulting in considering four TOs only.

### 5.3.4.3 Experiment Settings

Experiment settings for Study 2 are depicted in Table 5.4. Like Study 1, each row in the table represents one experiment. For *XMLMao*, there is only one input and, therefore, the Fix/Var length settings are not applicable. The only two configurations for *XMLMao* are: X.1.F.Y with restricted alphabet and X.1.F.N without restricted alphabet for the GA. For *XMLMao*, we keep the same termination criteria as Study 1.

For *M*, there are two user inputs. However, due to computational constraints, we only considered one configuration (M.2.F.Y) with restricted alphabet. Furthermore, we used a 30K cap for the maximum number of fitness evaluations.

### 5.3.4.4 Results

The results obtained with Study 2 are shown in Table 5.5. Our proposed approach achieved 100% TO coverage for *XMLMao*, i.e., all 12 TOs were covered. The cost in terms of execution time per TO is in the range 5-7 minutes.

With 31.87 minutes on average, execution time for *M* is much higher, although still within reasonable limits, but for a much lesser budget i.e. 30K evaluations. However, our

**Table 5.4:** Experiment Settings for Study 1: Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), the input length setting (Len), and whether the alphabet is restricted (Res. Alph.).

App.	Exp. ID	#TOs	#Inp.	Len.	Res. Alph.
XMLMao	X.1.F.N	12	1	Fix	N
	X.1.F.Y	12	1	Fix	Y
M	M.2.F.Y	4	2	Fix	Y

**Table 5.5:** Results for XMLMao and  $M$  in terms of coverage rate  $C_{ga}$  and average execution time  $T_{ga}$  in minutes.

Exp. ID	$C_{ga}$	$T_{ga}$
X.1.F.N	12/12	6.86
X.1.F.Y	12/12	5.68
M.2.F.Y	1/4	31.87

technique did manage to find inputs to cover one of the TOs. The other three TOs turned out to be infeasible due to the type of input validation that  $M$  applies. Note that, in contrast to  $XMLMao$  that is a vulnerable application implemented to study and evaluate penetration testing tools,  $M$  is an actual industrial system used in production (i.e., the found potential vulnerability was not artificially injected for the sake of these experiments).

Regarding the research questions **RQ1** and **RQ3**, we find that:

*The proposed approach is effective in finding inputs that detect XMLi vulnerabilities in third-party independent applications, within practical execution time.*

Regarding **RQ4**, the use of the restricted alphabet in XMLMao was also beneficial in terms of execution time, i.e., X.1.F.Y that used the restricted alphabet was faster with an average execution time of 5.68 minutes per TO compared to its counterpart X.1.F.N, with 6.86 minutes.

*Using a restricted alphabet for the GA results in better execution time for XMLMao.*

## 5.4. Related Work

In this section, we survey work related to vulnerability detection in web applications/services, with particular attention to XML vulnerabilities. We also discuss existing work that uses search-based approaches for security testing.

**Automated approaches for vulnerability testing:** There is a large research body investigating automated approaches for the detection of vulnerabilities in web applications/services, e.g., [23, 31, 33, 34, 71]. Bau et al. [72] performed a study to evaluate the effectiveness of the state-of-the-art in automated vulnerability testing of web applications. Their results demonstrate that such approaches are only good at detecting straightforward, historical vulnerabilities but fail to generate test data to reveal advanced forms of vulnerabilities. Mainka et al. [31] presented an automated penetration testing approach and evaluated it on several web service frameworks. They implemented a tool named WSAttacker and targeted two web service specific attacks: WS-Addressing spoofing<sup>1</sup> and SOAPAction<sup>2</sup>. Their work was further extended by Oliveira et al. [33] with another tool (WSFAGresser) targeting specific web service attacks. A common issue with most of these automated approaches is the large number of false positives, which makes their application in practice difficult. Besides, none of these approaches are dedicated towards the detection of XML injections, the objective of this chapter.

**Testing for XML Injections:** A recent survey [9] in security testing has investigated various approaches and tools for testing SQL injections and Cross-site scripting vulnerabilities in web applications and services. Unlike these two vulnerabilities which received much attention (e.g., [40, 73, 74]), only limited research targets XML injections. An approach for the detection of XML injection attacks is presented by Rosa et al. [56]. They proposed a strategy to first build a knowledge database from the known attack patterns and then use it for detecting XML injection attacks, when they occur. This approach is an improvement over the traditional signature-based detection approaches, however it focuses on intrusion detection, not on security testing. In contrast, our work is targeted towards test data generation to detect XML injection vulnerabilities in web applications.

A basic testing methodology for XML injections is defined by OWASP [75]. It suggests to first discover the structure of the XML by inserting meta-characters in the SUT. The revealed information, if any, combined with XML data/tags can then be used to manipulate the structure or business logic of the application or web service. OWASP also provided a

---

<sup>1</sup>[http://www.ws-attacks.org/index.php/WS-Addressing\\_spoofing](http://www.ws-attacks.org/index.php/WS-Addressing_spoofing)

<sup>2</sup>[http://ws-attacks.org/index.php/SOAPAction\\_spoofing](http://ws-attacks.org/index.php/SOAPAction_spoofing)

tool named WSFUZZER [13] for SOAP penetration testing with fuzzing features. However, as reported in [7], the tool could not be used with WSDLs having complex structure (nested XML elements) and is only useful in scenarios where the web services are directly accessible for testing.

In the previous chapter, we discussed four types of XML injection attacks and proposed a novel approach for testing web services against these attacks. Our evaluation found the approach very effective compared to state-of-the-art tools. However, it focuses on the back-end web services that consume XML messages and are directly accessible for testing. In contrast, our current work targets the front-ends (web applications) of SOA systems that produce XML messages for web services or other back-end systems.

In addition, while in the previous chapter we used constraint solving and input mutation for manipulating XML messages, in this chapter we use search-based testing techniques to generate test inputs for the front-end of the SUT that produces malicious XML messages. Such inputs can then help detect XMLi vulnerabilities in web applications that can be exploited through the front-ends.

**Search-based approaches for security testing:** Search-based testing has been widely investigated in the literature in the context of functional testing [67, 76, 77, 78, 79]. However, little attention has been devoted to non-functional properties of the SUT, such as security testing [80, 81].

Avancini and Ceccato [82] used search-based testing for cross-site scripting vulnerabilities in web applications. Their approach uses static analysis to look for potential cross-site scripting vulnerabilities in PHP code. Then, genetic algorithms and constraint solvers are used to search for input values that can trigger the vulnerabilities. This approach is white-box and targets a different type of vulnerabilities, i.e., cross-site scripting. Instead, our approach is completely black-box, i.e., it does not require the source code and it targets XML injection vulnerabilities.

Thomé et al. [83] proposed a search-based testing approach to detect SQL injection vulnerabilities in web applications. Their approach evolves inputs by assessing the effects on SQL interactions between the web server and database with the goal of exposing SQL injection vulnerabilities. Our work is also based on evolving test inputs but for XML injection instead of SQL. Moreover, Thomé et al. [83] used a fitness function based on a number of factors to measure the likelihood of the *SQLi* attacks. Instead, we use a fitness function based on the distance between the SUT's outputs and test objectives based on attack patterns.



Evolutionary algorithms have been also used to detect other types of vulnerabilities [84, 85]. Unlike our black-box approach for *XMLi* testing, these techniques are white-box and are focused on buffer overflow detection.

## 5.5. Summary

In this chapter, we have presented an effective search-based approach for the security testing of web applications, with a focus on *XMLi* vulnerabilities. Our approach is able to lead the system under test (SUT) to produce malicious XML messages from user inputs (e.g., HTML forms). Such web applications often act as front-ends to the web services of a SOA system. In such context, *XMLi* vulnerabilities are common and can lead to severe consequences, e.g., DoS or data breaches. Therefore, automated and effective testing to detect and fix *XMLi* vulnerabilities is of paramount importance.

The proposed approach is divided into two steps: (1) the automated identification of malicious XML messages (our test objectives, TOs) that, if generated by the SUT and sent to services, would suggest a vulnerability; (2) The automated generation of SUT inputs that generate messages matching such TOs. This chapter focuses on item (2), as item (1) was already addressed in the previous chapter.

We have evaluated our novel approach on several artificial systems and one large industrial web application. Our results suggest that the proposed approach is effective as it was able to uncover vulnerabilities in all case studies. We also found that the employed genetic algorithm works best when some domain knowledge about the system under test is available, e.g., the lengths of user input parameters and their alphabets, to restrict the search space.

## Chapter 6

# Improving Test Case Generation for XML Injections in Front-end Web Applications

Chapter 5 presented an automated and effective black-box testing approach for the security testing of web applications, with a focus on *XMLi* vulnerabilities. In this chapter, we improve upon the previous results by providing more efficient techniques to generate test cases. More specifically, the contributions of this chapter with respect to the previous chapter are:

- We investigate four different search algorithms, namely Standard Genetic Algorithm (*SGA*), Real-coded Genetic Algorithm (*RGA*), Hill Climbing (*HC*) and Random Search (*RS*), while in the chapter 5 we compared only *SGA* and *RS*.
- We evaluate a different fitness function, namely the Real-coded Edit Distance (*Rd*), to overcome the limitations of the traditional String Edit Distance (*Ed*) in our context.
- We provide an in-depth analysis by comparing all possible combinations of fitness functions and search algorithms to determine the combination that is most effective and efficient in detecting *XMLi* vulnerabilities.
- We extensively analyze several co-factors that are likely to affect the effectiveness and efficiency of the proposed approach.

We have carried out an extensive evaluation of the proposed search-based approach by conducting two different case studies. In the first study, we compared all combinations of fitness

functions and search algorithms with respect to the detection of *XMLi* vulnerabilities in (i) one open-source third-party application designed for secure-code training, and (ii) two web applications that interact with an industrial bank card processing system. We find that *RGA* combined with *Rd* is able to detect more *XMLi* vulnerabilities (better effectiveness) within a significantly lower amount of time (better efficiency) when compared to the other combinations, including the one used in the previous chapter, i.e., *SGA* with *Ed*.

To evaluate the applicability of our search-based approach in a realistic setting, we conducted a second case study involving two industrial systems. The first one is a web application having millions of registered users, with hundreds of thousands of visits per day. We focused on one of its pages with an HTML form. As our approach would be directly applicable to any system that receives HTTP messages, to show that this is indeed the case, our second case study involves a web service receiving JSON messages and generating XML messages for back-end SOAP services. Our results show that the proposed technique, when configured with *RGA* and *Rd*, successfully detects *XMLi* vulnerabilities in the evaluated industrial systems.

The remainder of the chapter is structured as follows. Section 6.1 describes our proposed approach. Sections 6.2 and 6.3 report and discuss our evaluation on two case studies including research questions, results and discussions. Further analyses regarding the various co-factors that may affect our results are presented in Section 6.4. Section 6.5 discusses related work. Threats to validity are discussed in Section 6.6. Finally, Section 6.7 concludes the chapter.

## 6.1. Approach

### 6.1.1 Search-Based Testing

In our context, applying search-based techniques requires to address three issues [15]: (i) choose an encoding schema to represent candidate solutions (i.e., test inputs); (ii) design a fitness function to guide the search for malicious test inputs; and (iii) choose and apply an effective search algorithm to generate inputs closer to the target TO. Our choices for the aforementioned tasks are detailed in the next sub-sections.

### 6.1.1.1 Solution Encoding

A given SUT requires to submit  $N$  input parameters to produce XML messages that will be sent through HTTP to the web services. Therefore, the search space is represented by all possible tuples of  $N$  strings that can be submitted via the web form, with one string for each single parameter. In this context, a string is a sequence of alphanumeric and special characters (e.g., %, || or &) that can be inserted by an attacker in the web form.

Therefore, we use the following encoding schema: a candidate test case for the SUT with  $N$  input parameters is a tuple of strings  $T = \langle S_1, S_2, \dots, S_N \rangle$  where a  $S_i$  denotes the string for the  $i$ -th input parameter of the SUT. A generic string in  $T$  is an array of  $k$  characters, i.e.,  $S_i = \langle c_1, c_2, \dots, c_k \rangle$ . The length  $k$  of the array is fixed based on the expected maximum length of the corresponding input parameter. To allow input strings with different length, we use a special symbol to denote the “empty” character, i.e., absence of character. In this way, the lengths of input strings can vary during the search even if the length of the array (i.e.,  $k$ ) in the encoding schema is fixed. In other words, the array  $S_i = \langle c_1, c_2, \dots, c_k \rangle$  can be filled with the “empty” character to represents shorter strings.

Theoretically, characters in the input string can come from the extended ASCII code as well as from UNICODE. However, in this chapter we consider only printable ASCII characters with code between 32 and 127 since, as noticed by Alshraideh et al. [86], the majority of software programs do not use characters outside this range (i.e., non-printable characters).

### 6.1.1.2 Fitness Function

The effectiveness of the search strongly depends on the guidance of the fitness function, which evaluates each candidate solution  $T$  according to its closeness to the target TO. In particular, when a candidate solution  $T$  is executed against the SUT, it should lead to the generation of an XML message that match the TO. Hence, the fitness function is the distance  $d(\text{TO}, \text{SUT}(T))$  between the target TO and the XML message that the SUT produces upon the execution of  $T$ , i.e.,  $\text{SUT}(T)$ . The function  $d(\cdot)$  can be any distance measure such that  $d(\text{TO}, \text{SUT}(T)) = 0$  if and only if  $\text{SUT}(T)$  and the TO are identical, otherwise  $d(\text{TO}, \text{SUT}(T)) > 0$ . In this chapter, we investigate two different measures for the fitness function: the *string edit distance* and the *real-coded edit distance*.

**String Edit Distance.** The first fitness function is the edit distance (or Levenshtein distance), which is the most common distance measure for string matching.

Its main advantage compared to other traditional distances for strings (e.g., Hamming distance) is that it can be applied to compare strings with different lengths [87]. In our context, the length of the XML messages generated by the SUT varies depending on the input strings (i.e., the candidate solution  $T$ ) and the data validation mechanisms in place to prevent possible attacks. Therefore, the edit distance is well suited for our search problem. In addition, recent studies [86] showed that this distance outperforms other distance measures (e.g., Hamming distance) in the context of test case generation for programs with string input parameters, despite its higher computational cost<sup>1</sup>.

In short, the edit distance is defined as the minimum number of editing operations (inserting, deleting, or substituting a character) required to transform one string into another. More formally, let  $A_n$  and  $B_m$  be two strings to compare, whose lengths are  $n$  and  $m$ , respectively; the edit distance is defined by the following recurrence relations:

$$d_E(A_n, B_m) = \min \begin{cases} d_E(A_{n-1}, B_m) + 1 \\ d_E(A_n, B_{m-1}) + 1 \\ d_E(A_{n-1}, B_{m-1}) + f(a_n, b_m) \end{cases} \quad (6.1)$$

where  $a_n$  is the  $n$ -th character in  $A_n$ ,  $b_m$  is the  $m$ -th character in  $B_m$ , and  $f(a_n, b_m)$  is zero if  $a_n = b_m$  and one if  $a_n \neq b_m$ . In other words, the overall distance is incremented by one for each character that has to be added, removed or changed in  $A_n$  to match the string  $B_m$ . The edit distance takes value in  $[0; \max\{n, m\}]$ , with minimum value  $d_E = 0$  when  $A_n = B_m$  and maximum value of  $d_E = \max\{n, m\}$  when  $A_n$  and  $B_m$  have no character in common.

To clarify, let us consider the following simple example of TO and a SUT with one single input parameter. Let us assume that the target TO is the string `<test>data OR 1=1</test>`; and let us suppose that upon the execution of the test  $T = \langle \text{OR } \% \rangle$ , the SUT generates the following XML message  $\text{SUT}(T) = \langle \text{test} \rangle \text{data OR } \% \langle \text{/test} \rangle$ . In this example, the edit distance  $d_E(\text{TO}, \text{SUT}(T))$  is equal to three, as we need to modify the “%” character into “1”, and then add the two characters “=1” for an exact match with the TO.

One well-known problem of the edit distance is that it may provide little guidance to search algorithms because the fitness landscape around the target string is largely flat [86]. For example, let us consider the target  $\text{TO} = \langle \text{t} \rangle$  and let us assume we want to evaluate the three candidate tests  $T_1$ ,  $T_2$  and  $T_3$  that lead to the following XML messages:  $\text{SUT}(T_1) = \langle \text{At} \rangle$ ,  $\text{SUT}(T_2) = \langle \text{^t} \rangle$ ,  $\text{SUT}(T_3) = \langle \text{t} \rangle$ . The messages  $\text{SUT}(T_1)$  and  $\text{SUT}(T_2)$  share two characters with the target TO (i.e.,  $\text{t}$  and  $\text{>}$ ) and have a correct length, i.e., three characters. Instead,

<sup>1</sup>The computational cost of the edit distance is  $O(n \times m)$ , where  $n$  and  $m$  are the lengths of the two strings being compared.

the message  $SUT(T_3)$  shares two characters with the target TO but it is one character shorter. Therefore, we may consider  $T_1$  and  $T_2$  to be closer to the target TO than  $T_3$  since they have the correct number of characters, two of which match the TO. However, using the edit distance, all the three tests will have the same distance to the TO since they require to change only one character, i.e.,  $d_E(\langle \mathbf{t} \rangle, \mathbf{At}) = d_E(\langle \mathbf{t} \rangle, \hat{\mathbf{t}}) = d_E(\langle \mathbf{t} \rangle, \langle \mathbf{t} \rangle) = 1$ . In this example, the edit distance is not able to distinguish between messages having the correct length (e.g.,  $T_1$ ) and messages that are shorter or longer than the TO (e.g.,  $T_3$ ).

In general, the fitness landscape around the target TO will be flat as depicted in Figure 6.1-(a): all strings that require to change (e.g.,  $\mathbf{At}$ ), add (e.g.,  $\langle \mathbf{t} \rangle$ ) or remove (e.g.,  $\langle \mathbf{tt} \rangle$ ) one single character will have a distance  $d_E$  equal to 1 while the distance will be 0 for only one single point. Thus, a search algorithm would have to explore this whole, very large neighborhood, without any particular guidance.

**Real-Coded Edit Distance.** To increase the guidance of the fitness function, in this chapter we modify the traditional edit distance by taking into account the relative distance between characters in the ASCII code. Our motivation is to focus the search on sub-regions of the large neighborhood of the target TO.

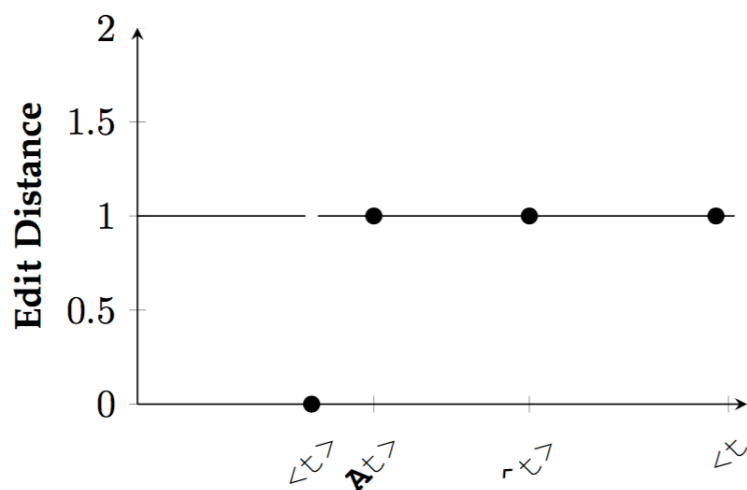
In particular, we change the recurrence relations of the traditional edit distance as follows:

$$d_R(A_n, B_m) = \min \begin{cases} d_R(A_{n-1}, B_m) + 1 \\ d_R(A_n, B_{m-1}) + 1 \\ d_R(A_{n-1}, B_{m-1}) + \frac{|a_n - b_m|}{1 + |a_n - b_m|} \end{cases} \quad (6.2)$$

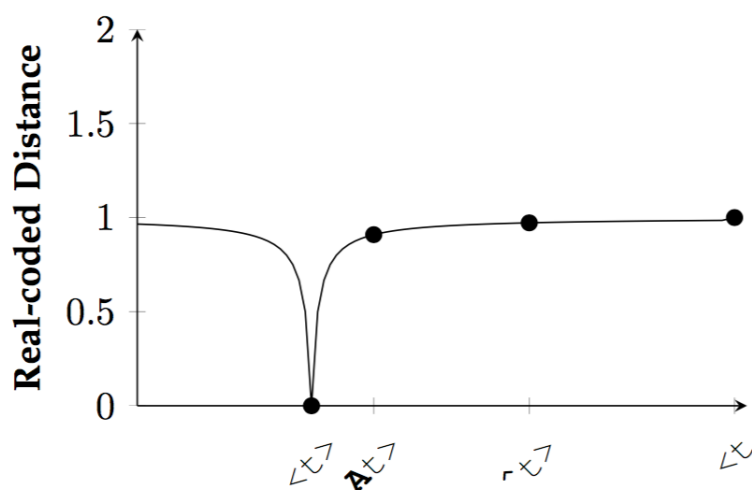
In Equation 6.2, the first two recurrence rules are identical to the traditional edit distance covering the case where  $A_n$  will match  $B_m$  by removing or adding one character, respectively. The change is applied on the third recurrence rule, which covers the case when the character  $a_n$  should be replaced by the character  $b_m$ . In the traditional edit distance, if  $a_n$  is not equal to  $b_m$  then the overall distance is always incremented by one. Instead, in Equation 6.2, if  $a_n$  is not equal to  $b_m$ , then the overall distance is incremented by the factor  $|a_n - b_m|$ , which is the absolute value of the difference between the ASCII codes for the two characters  $a_n$  and  $b_m$ .

Such an increment factor is normalized using the well-known normalization function  $\phi(x) = x/(x + 1)$  to obtain distance values within the interval  $[0; 1]$ .

To describe the benefits of this new distance, let us consider the example used previously to describe the fitness landscape of the edit distance: the target TO is the string  $\langle \mathbf{t} \rangle$ , and



(a) Edit distance



(b) Real-coded edit distance

**Figure 6.1:** Fitness landscapes for the *edit distance* and the *real-coded edit distance* for the target string  $TO = \langle t \rangle$

the tests to evaluate lead to the XML messages  $SUT(T_1) = \mathbf{A}t \rangle$ ,  $SUT(T_2) = \tilde{r}t \rangle$ ,  $SUT(T_3)$

= <t>. Using the real-coded edit distance, we obtain:

$$d_R(\langle t \rangle, \text{At}) = \frac{|60 - 65|}{|60 - 65| + 1} \approx 0.8333$$

$$d_R(\langle t \rangle, \hat{t}) = \frac{|60 - 94|}{|60 - 94| + 1} \approx 0.9714$$

$$d_R(\langle t \rangle, \langle t \rangle) = 1$$

Thus, with  $d_R$  the three tests are not equally distant to the TO anymore: the test  $T_3$  is the furthest one among the three tests. Therefore, differently from the edit distance, the new distance is able to distinguish between messages with correct length (e.g.,  $T_1$ ) and messages that are longer or shorter than the TO (e.g.,  $T_3$ ). We can also observe that the new distance returns two different values for  $T_1$  and  $T_2$  although both the two tests need to replace only one character to perfectly match the TO. This difference is given by the relative distance between the character to match in the TO (i.e., “<”) and the two characters to replace (i.e., “A” for  $T_1$  and “^” for  $T_2$ ) according to their ASCII codes. Therefore,  $d_R$  introduces a distance among characters such that small differences among strings can still lead to differences in fitness values. Though such distance may sound arbitrary, it helps the search focus on sub-regions of the neighborhood of the target TO, e.g., by preferring  $T_1$  over  $T_2$  in the example, and thus reduce the search space without decreasing our chances to reach the target.

Figure 6.1-(b) plots the fitness landscape around the target TO= $\langle t \rangle$  as well as the three candidate tests  $T_1$ ,  $T_2$ , and  $T_3$  in the previous example. In particular, the  $x$  axis orders the candidate XML messages according to their ASCII codes while the  $y$  axis reports the corresponding fitness function values produced by the real-coded edit distance. As we can observe, the plateaus are replaced by a fitness function providing more guidance by considering the arbitrary ordering of characters according to their ASCII codes.

However, as for any heuristics, our new string distance might have side effects (e.g., create new local optima) in some cases. Therefore, such heuristics needs to be empirically evaluated to check if indeed it provides the benefits we expect from the theory.

### 6.1.1.3 Solvers

Once the encoding schema and the fitness function are defined, search algorithms can be applied to find the optimal solutions, as for example malicious input strings in our case. In this chapter, we investigate four different search algorithms, which are described below.

**Random Search** is the simplest search algorithm, which uniformly samples the search space by evaluating random points. It starts with a randomly generated test  $T$  representing



the initial candidate solution to the problem. Such a solution is evaluated through the fitness function and it is stored until a new, better solution is found in the next iterations of the algorithm, or if a stop condition is reached. At each iteration, a new test  $T^*$  is randomly generated and compared with  $T$  using the fitness function. If  $T^*$  has a better fitness value than  $T$ , then it is kept as current solution for the next iterations, i.e.,  $T = T^*$ . The search ends after a fixed number of iterations. The final test  $T$  will be the best solution among all those observed across all the iterations.

In our context, a randomly generated solution  $T = \langle S_1, S_2, \dots, S_N \rangle$  is composed of  $N$  arrays of characters with a fixed length  $k$ . Each array  $S_i$  contains characters randomly taken from the set of available characters (alphabet), including the “empty” character. Therefore, a solution  $T$  is composed of arrays representing strings with variable lengths  $\leq k$ .

Since random search does not refine previously generated solutions, it has usually a low probability to reach the global optimum. However, it is often used in the software engineering literature as baseline for comparison with more advanced algorithms. Moreover, random search has been shown to outperform other search algorithms (e.g., evolutionary algorithms) when solving specific problems, such as automated software repair [88], and hyper-parameter optimization [89].

**Hill Climbing** is a local search algorithm, which iteratively exploits the *neighborhood* of the current solution to find better nearby solutions (*neighbors*). Similar to random search, hill climbing starts with a single randomly generated test  $T$ , which represents the current solution to the problem. At each iteration, a new solution  $T^*$  is taken from the neighborhood of  $T$  and evaluated against the fitness function. If  $T^*$  improves the fitness function, then it becomes the current solution for the next iteration, i.e.,  $T = T^*$ . The search ends after a fixed number of iterations or if a zero fitness value is reached, which indicates that the target TO is matched.

The key ingredient for the hill climbing algorithm is the definition of the *neighborhood*, which corresponds to the set of tests (neighbors) that can be obtained from the current solution  $T$  by applying “small mutations”. Let  $T = \langle S_1, S_2, \dots, S_N \rangle$  be the current test composed of  $N$  arrays of characters. A neighbor is obtained from  $T$  by mutating its constituent arrays using one of the following operators: *adding*, *replacing* or *deleting* characters. Each operator is performed with probability  $p = 1/3$ , i.e., the three operators are mutually exclusive (only one operator is applied at a time) and equiprobable. Given an array of characters  $S_i = \langle c_1, c_2, \dots, c_k \rangle$  of length  $k$ , the three operators are implemented as follows:

- *deleting*: each character  $c_j$  in  $S_i$  is deleted with probability  $p_d = 1/k$ . The deletion is performed by replacing the character  $c_j \in S_i$  with the “empty” character.
- *replacing*: each character  $c_j$  in  $S_i$  is replaced with a new character  $c_j^*$  with probability  $p_r = 1/k$ , where  $c_j^*$  is randomly selected from the set of printable ASCII characters.
- *adding*: a new character  $c^*$  is inserted in  $S_i$  at a random position  $j \in [1, \dots, k]$ . The character is added if and only if  $S_i$  does not contain already  $k$  non “empty” characters.

Therefore, on average only one character is removed, replaced or added in the arrays  $S_i$  contained in the test  $T$ .

Despite its simplicity, the hill climbing algorithm is very effective when the fitness forms a unimodal function in the search space, i.e., functions with only one single optimal point [62]. However, for function with multiple local optima (multimodal problems), this algorithm can return sub-optimal solutions since it converges to the first local optimum encountered during the search, even if it is not the global one [62].

**Standard Genetic Algorithm (SGA)** is a metaheuristic solver inspired by the mechanisms of natural selection and adaptation. In a nutshell, it starts with a pool of solutions, called *population*, where each solution (or *chromosome*) is a randomly generated test. Then, the population is iteratively evolved by applying well-known genetic operators, namely *crossover*, *mutation* and *selection*. At each iteration (*generation*), pairs of solutions (*parents*) are selected and re-combined using the *crossover* operator, which creates new solutions (*offsprings*) to form the population for the next generation. Other than inheriting parts (*genes*) from their parents, offsprings are further modified, with a given small probability, using the *mutation* operator. Solutions are selected according to a *selection* operator, which typically gives higher selection probability to solutions in the current population with better fitness values (fittest individuals). This process is repeated until a zero fitness value is achieved (i.e., the TO is matched) or after a fixed number of generations.

The most popular genetic operators in SGA are the *binary tournament selection*, the *multi-point crossover* and the *uniform mutation* [62]. They are defined as follows:

- the *binary tournament selection* is the most common selection mechanism for GAs because of its simplicity and efficiency [63, 64]. With this operator, two individuals are randomly taken from the current population and compared against each other using the fitness function. The solution with the best fitness value wins the tournament and is selected for reproduction.

- the *multi-point crossover* generates two offsprings  $O_1$  and  $O_2$  from two parent solutions  $P_1$  and  $P_2$  by recombining their corresponding arrays of characters. More precisely, let  $P_1 = \langle S_1, S_2, \dots, S_N \rangle$  and  $P_2 = \langle R_1, R_2, \dots, R_N \rangle$  be the two selected parents, the two offsprings  $O_1$  and  $O_2$  are generated as follows:

$$O_1 = \langle \otimes(S_1, R_1, p_1), \dots, \otimes(S_N, R_N, p_N) \rangle \quad (6.3)$$

$$O_2 = \langle \otimes(R_1, S_1, p_1), \dots, \otimes(R_N, S_N, p_N) \rangle \quad (6.4)$$

where the generic element  $\otimes(S_i, R_i, p_i)$  denotes the array obtained by cutting the two arrays  $S_i$  and  $R_i$  at the same random cut point  $p_i$  and then concatenating the head part from  $S_i$  with the tail part from  $R_i$ . Similarly,  $\otimes(R_i, S_i, p_i)$  indicates the array obtained by applying the same random cut point  $p_i$  but concatenating the head part from  $R_i$  with the tail part from  $S_i$ . Therefore, the  $i$ -th array from one parent is recombined with the corresponding array at the same position  $i$  in the other parent.

- the *uniform mutation* is finally used to mutate, with a small probability, newly generated solutions in order to preserve diversity [62]. It corresponds to the mutation operator used for the hill climbing algorithm when generating neighbors: tests are mutated by deleting, replacing or adding characters in the corresponding array of characters.

GAs are global search algorithms and are thus more effective than local search solvers for multimodal problems. This is because they use multiple solutions to sample the search space instead of a single solution (e.g., for the hill climbing) which could bias the search process [62]. On the other hand, GAs can suffer from a slower convergence to the local optimum when compared to hill climbing. Therefore, they are usually less effective and efficient for unimodal problems [62].

**Real-Coded Genetic Algorithm (RGA)** is a variant of GAs designed to solve numerical problems with real or integer numbers as decision variables (genes) [90]. The main difference between SGA and RGA is captured by the genetic operators that are used to form new solutions. In SGAs, the crossover creates offsprings by exchanging characters from the parents and, as a result, the new solutions will only contain characters that appear in the parent chromosome. Further, in SGA, diversity is maintained by the mutation operator, which is responsible for replacing characters inherited from the parents with any other character in the alphabet. Instead, in RGA, the parents are recombined by applying numerical functions (e.g., the arithmetic mean) to create offsprings that will contain new numbers (i.e., genes) not appearing in the parent chromosomes. Mutation, on the other hand, alters solutions according to some numerical distribution, such as the Gaussian distribution.

In this chapter, we investigate the usage of RGAs since they have been shown to be more effective than SGAs when solving numerical and high dimensional problems [90, 91]. In particular, our problem is numerical if we consider characters as numbers in ASCII code (as in the real-coded edit distance) and it is high dimensional (the number of dimensions corresponds to the length of the chromosomes). Indeed, maintaining the same encoding schema used for SGA, each array of characters  $S_i = \langle c_1, c_2, \dots, c_k \rangle$  of a test  $T$  can be converted in an array of integers  $U_i = \langle u_1, u_2, \dots, u_k \rangle$  such that each  $u_i \in U$  is the ASCII code of the character  $c_i \in S$  when applying real-coded crossover or mutation.

Popular genetic operators for RGA are the *binary tournament selection*, the *single arithmetic crossover* [92], and *Gaussian mutation* [93]. Therefore, the selection mechanism is the same as in SGA, whereas crossover and mutation operators are different. Before applying these two numerical operators, we convert the input strings forming a test  $T$  in arrays of integers by replacing each character with the corresponding ASCII code. Once new solutions are generated using the *single arithmetic crossover* and *gaussian mutation*, the integer values are reconverted into characters.

The *single arithmetic crossover* is generally defined for numerical arrays with a fixed length. For example, let  $A = \langle a_1, a_2, \dots, a_k \rangle$  and  $B = \langle b_1, b_2, \dots, b_k \rangle$  be two arrays of integers to recombine; it creates two new arrays  $A'$  and  $B'$  as copies of the two parents and modify only one element at a given random position  $i$  using the arithmetic mean. In other words,  $A'$  and  $B'$  are created as follows [92]:

$$A' = \langle a_1, a_2, \dots, a'_i, \dots, a_k \rangle \quad (6.5)$$

$$B' = \langle b_1, b_2, \dots, b'_i, \dots, b_k \rangle \quad (6.6)$$

where the integers  $a'_i$  and  $b'_i$  are the results of the weighted arithmetic mean between  $a_i \in A$  and  $b_i \in B$ ; and  $i \leq k$  is a randomly generated point. The weighted arithmetic mean is computed using the following formulae [92]:

$$a'_i = a_i \cdot \rho + b_i \cdot (1 - \rho) \quad (6.7)$$

$$b'_i = b_i \cdot \rho + a_i \cdot (1 - \rho) \quad (6.8)$$

where  $\rho$  is a random number  $\in [0; 1]$ . Finally, the two resulting real numbers  $a'_i$  and  $b'_i$  are rounded to their nearest integers.

In our case, parent chromosomes are tuples of strings and not simple arrays of integers. Therefore, we apply the single arithmetic crossover for each pair of arrays composing the two parents, after the conversion of the characters to their ASCII codes. More formally,

let  $P_1 = \langle S_1, S_2, \dots, S_N \rangle$  and  $P_2 = \langle R_1, R_2, \dots, R_N \rangle$  be the two selected parents; the two offsprings  $O_1$  and  $O_2$  are generated as follows:

$$O_1 = \langle \mu(S_1, R_1, p_1), \dots, \mu(S_N, R_N, p_N) \rangle \quad (6.9)$$

$$O_2 = \langle \mu(R_1, S_1, p_1), \dots, \mu(R_N, S_N, p_N) \rangle \quad (6.10)$$

where  $S_i$  is the array of ASCII codes in position  $i$  from the parent  $P_1$ ;  $R_i$  is the array of ASCII codes in position  $i$  from the parent  $P_2$ ; the elements  $\mu(S_i, R_i, p_i)$  and  $\mu(R_i, S_i, p_i)$  are the two arrays created by the single arithmetic crossover on  $S_i$  and  $R_i$  with random point  $p_i$ .

The *gaussian mutation* is similar to the uniform mutation for SGA. Indeed, each test  $T$  in the new population is mutated by deleting, replacing or adding characters in the corresponding array of characters. The main difference is represented by the routine used to replace each character with another one. With the uniform mutation, a character is replaced with any other character in the alphabet. Instead, the gaussian mutation is defined for numerical values, which are replaced with other numerical values but according to a Gaussian distribution [93]. In our case, let  $S_i = \langle c_1, c_2, \dots, c_k \rangle$  be the array of ASCII codes to mutate; each ASCII code  $c_j$  in  $S_i$  is replaced with a new ASCII code  $c_j^*$  with probability  $p_r = 1/k$ . The integer  $c_j^*$  is randomly generated using the formula:

$$c_j^* = c_j + c_j \cdot \delta(\mu, \sigma) \quad (6.11)$$

where  $\delta(\mu, \sigma)$  is a normally distributed random number with mean  $\mu = 0$  and variance  $\sigma$  [93]. In other words, the new ASCII code is generated by adding a normally distributed delta to the original ASCII code  $c_j$ . The remaining issues to solve include (1) this mutation scheme generates real numbers and not integers and (2) the generated numbers can fall outside the range of printable ASCII code (i.e., outside the interval [32; 127]). Therefore, we first round  $c_j^*$  to the nearest integer number. Finally, the mutation is cancelled if the new character  $c_j^*$  is lower than 32 or greater than 127.

## 6.2. Empirical Studies

This section describes our empirical evaluation whose objective is to assess the proposed search-based approach and compare its variants in terms of different fitness functions and search algorithms, as discussed in Section 6.1.

### 6.2.1 Study Context

The evaluation is carried out on several front-end web applications grouped into two case studies. The first study is performed on small/medium open-source applications, whereas the second one involves industrial systems. These two studies are described in detail below.

**Study 1.** The first case study involves three subjects with various web-applications. The first two subjects are SBANK and SecureSBANK (SSBANK), which both contain web applications interacting with real-world bank card processing system of one of our industrial collaborators (a credit card processing company<sup>1</sup>). Each of these two subjects has three applications that differ regarding their number of user inputs, ranging from one to three user inputs. In the following, we will refer to SBANK1 (or SSBANK1), SBANK2 (or SSBANK2), and SBANK3 (or SSBANK3) for the applications with one, two, and three user inputs, respectively. These different variants of the same applications are used to analyze to what extent the number of input parameters affects the ability of solvers and fitness functions to detect XMLi vulnerabilities (see Section 6.4 for further details). The goal of this analysis is to assess the scalability of the approach as the number of inputs increases.

Each SBANK/SSBANK application receives user inputs, produces XML messages and sends them to the web services of the card processing system. An example of XML message produced by an SBANK/SSBANK application is depicted in Figure 6.2. Such a message contains four XML elements, which are *UserName*, *IssuerBankCode*, *CardNumber*, and *RequestID*. The first three elements are formed using the submitted user inputs while the *RequestID* element is generated by the application automatically. In other words, the application logic does not allow users to tamper with the value of this element unless they do so maliciously.

Applications in SBANK are vulnerable to XML injections as there is no validation or sanitization of the user inputs. The SSBANK applications are similar to SBANK except that one of the input parameters is validated, i.e., the application checks the input data for malicious content. Before producing the XML message, the latter applications validate the user input parameter *IssuerBankCode* and generate an error message if any malicious content is found. These two applications allow us to assess, in a controlled fashion, the impact of input validation procedures on the ability of solvers and fitness functions to detect XMLi vulnerabilities.

---

<sup>1</sup>The name of the company cannot be revealed due to a non-disclosure agreement

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:lu="http://schemas.xmlsoap.org/soap/envelope/lu/">
  <soapenv:Header/>
  <soapenv:Body>
    <lu:perform>
      <lu:resInput>
        <lu:UserName>Tom</lu:UserName>
        <lu:IssuerBankCode>0231</lu:IssuerBankCode>
        <lu:RequestId>28278111TOM</lu:RequestId>
        <lu:CardNumber>123456789</lu:CardNumber>
      </lu:resInput>
    </lu:perform>
  </soapenv:Body>
</soapenv:Envelope>
```

**Figure 6.2:** An example of output XML message created by *SBank*.

The third subject of our first study is *XMLMao*, an open source web application that is deliberately made vulnerable for testing XML injection attacks [70]. It is part of the *Magical Code Injection Rainbow (MCIR)* [70] framework for building a configurable vulnerability test-bed. This application accepts a single user input and creates XML messages. It has 1178 lines of code written in PHP. We chose to include such an open source application in our evaluation to have, as part of our study, a publicly accessible system that future research can use as a benchmark for comparison.

**Study 2.** The second study consists of two subjects provided by one of our industrial collaborators which, to preserve confidentiality, are referred to by arbitrary names: *M* and *R*.

- **M:** *M* is an industrial web application with millions of registered users and hundreds of thousands of visits per day. The application itself is hundreds of thousands of lines long, communicating with several databases and more than 50 corporate web services (both SOAP and REST). Out of hundreds of different HTML pages served by *M*, in this chapter we focus on one page having a form with two string inputs.
- **R:** *R* is one of the RESTful web services interacting with *M*. This web service receives requests in JSON format, and interacts with two SOAP web services and one database. *R* is significantly smaller than *M*, as it is composed of just over 40 classes. Of the different API methods provided by *R*, in this chapter we focus on a POST that takes as body a JSON object with three string fields.

**Table 6.1:** Description of Test Objectives

App. Name	#Input	#TOs per Attack				Total #TOs
		Type 1	Type 2	Type 3	Type 4	
SBANK	1	3	3	3	0	9
	2	3	3	3	1	10
	3	3	3	3	1	10
SSBANK	1	3	3	3	0	9
	2	3	3	3	1	10
	3	3	3	3	1	10
XMLMao	1	4	4	4	0	12
M	2	1	1	1	1	4
R	3	1	1	1	1	4

As the experiments on these two systems had to be run on a dedicated machine (e.g., they could not be run on a research cluster of computers) due to confidentiality constraints, we could not use all of their web pages and endpoints. We chose those two examples manually, by searching for non-trivial cases (e.g., web pages with at least two string input parameters that are not enumerations), albeit not too difficult to analyze, i.e., given the right inputs, it should interact with at least one SOAP web service. Due to non-disclosure agreements and security concerns, no additional details can be provided on *M* and *R*.

**Test Objectives (TOs).** For each application and for each case study, we created the target TOs based on the four types of XML injection attacks described in the previous chapter in Section 5.2.1. Table 6.1 reports on the number of generated TOs collected per study subject and type of XML injection attacks. For SBANK/SSBANK applications with two and three input parameters, there are ten TOs in total: three TOs (one for each attack of types *Type 1-Type 3*) for each of the three XML elements and one additional TO for the *Type 4* attacks. Note that the *Type 4: Replacing* attack is a more advanced form of XML injection that requires at least two XML elements where the value of one of them must be auto-generated by the application. Therefore, this attack can be applied only to the *RequestID* element as it is the only auto-generated element in the application. Moreover, this attack should not be applied to web applications with only one input parameter, otherwise the resulting TO will be unfeasible. As a consequence, for SBANK/SSBANK applications with only one input parameter, we have nine TOs in total, corresponding to the attacks of types *Type 1-Type 3* for each of the three XML elements.



The XMLMao application has one user input that can be inserted in four possible locations in the generated XML messages. Therefore, we create TOs by applying each type of attack on the four XML elements. As depicted in Table 6.1, we do not have TOs for the attack of *Type 4: Replacing* because XMLMao has only one input parameter. In total, we obtain 12 TOs (3 attacks  $\times$  4 locations) for this subject. Finally, for the industrial applications ( $M$  and  $R$ ), we have four TOs, i.e., one TO for each type of attack.

### 6.2.2 Research Questions

Our evaluation addresses the following research questions:

- **RQ1:** *What is the best fitness function for detecting XMLi vulnerabilities?* With this first research question, we aim at comparing the two fitness functions defined for the XMLi vulnerability detection problem. In particular, we compare the performance of each solver (e.g., hill climbing), considered individually, when used to optimize the *real-coded edit distance* proposed in this chapter and the traditional *string edit distance* [16, 86]. In particular, the comparison is performed in terms of the number of detected XMLi vulnerabilities (effectiveness) and the time needed to detect them (efficiency). Therefore, in answering this research question, we consider the following two sub-questions:
  - RQ1.1** [Effectiveness]: *What is the best fitness function in terms of effectiveness?*
  - RQ1.2** [Efficiency]: *What is the best fitness function in terms of efficiency?*
- **RQ2:** *What is the best solver for detecting XMLi vulnerabilities?* In this second research question, we compare to what extent different search algorithms are able to detect XMLi vulnerabilities when optimizing the same fitness function (e.g., the string edit distance). Specifically, we compare the different algorithms discussed in Section 6.1.1.3 when optimizing the same fitness function with respect to their ability to detect as many XMLi vulnerabilities as possible (effectiveness) and the time needed to detect such vulnerabilities (efficiency). Therefore, we consider the following two sub-questions:
  - RQ2.1** [Effectiveness]: *What is the best solver in terms of effectiveness?*
  - RQ2.2** [Efficiency]: *What is the best solver in terms of efficiency?*

The goal of these two research questions is to understand which solver and fitness function combination is more effective and efficient for detecting XMLi vulnerabilities. Therefore to

answer them, we use all web applications in Study 1 to perform an extensive analysis of all possible combinations of solvers and fitness functions (see Section 6.2.3).

For the industrial applications ( $M$  and  $R$ ) in Study 2, we could not involve our industrial partners in the evaluation of all possible configurations given the high computational cost of this type of study. Indeed, such a detailed investigation involves (i) different solvers, (ii) different fitness functions, (iii) different configurations, (iv) various TOs for each application, and (v) a number of repetitions to address the randomized nature of the solvers being compared. For these reasons, Study 2 is used to evaluate the applicability of the best configuration of our search-based approach, in a realistic context, as formulated by the following research question:

- **RQ3:** *Is the proposed technique effective and efficient in detecting XMLi vulnerabilities in industrial systems?* For this research question, we focus on the two real-world applications  $M$  and  $R$  in Study 2 to understand whether the proposed search-based approach is able to detect XMLi vulnerabilities (effectiveness) in larger systems with complex input validation routines and in a reasonable amount of time (efficiency). Since the goal here is to assess the applicability of our approach in realistic conditions and because running all combinations of solvers and fitness functions is not possible on our industrial applications, we focus on assessing the best combination of solver and fitness function identified when answering **RQ1** and **RQ2**.

### 6.2.3 Variable Selection

To answer our RQs, we studied the effect of the following independent variables:

- *Fitness function:* in Section 6.1.1.2 we described two different fitness functions, i.e., *real-coded edit distance* ( $Rd$ ) and *string edit distance* ( $Ed$ ), that can be used to guide search algorithms toward the detection of XMLi vulnerabilities. The former has been widely applied in the software testing literature [86] while the latter has been introduced in this chapter. To answer **RQ1**, we compare the results achieved by each solver considered individually when optimizing the two fitness functions for each application and TO in our empirical study.
- *Solver:* given a fitness function, different optimization algorithms can be used to find optimal solutions to our problem. Therefore, this independent variable accounts for the four solvers described in Section 6.1.1.3 that could be used interchangeably for the

XMLi vulnerabilities detection problem, which are Random Search (RS), Hill Climbing (HC), Standard Genetic Algorithms (SGA) and Real-coded Genetic Algorithms (RGA). To answer **RQ2**, we compare the four solvers when optimizing the same fitness function. In other words, the comparison is performed by considering each fitness function separately.

For brevity, in the following we refer to combinations of these two independent variables (*Fitness function*  $\times$  *Solver*) as *treatments* affecting the dependent variables.

In our study, the dependent variables are the performance metrics used to compare the effectiveness and the efficiency across treatments. For effectiveness, we use the *Success Rate*, which is the ratio of the number of times a given TO is covered by a treatment  $\Omega$  to the total number of times the treatment  $\Omega$  is executed (i.e., runs). More formally, the success rate is defined as follows:

$$SR(TO, \Omega) = \frac{\# \text{ successful\_runs}}{\# \text{ runs}} \times 100 \quad (6.12)$$

where  $\# \text{ successful\_runs}$  denotes the number of times  $\Omega$  covers the TO, and  $\# \text{ runs}$  indicates the total number of runs.

For efficiency, we use the *Execution Time*, which measures the average time (in minutes) taken by a treatment  $\Omega$  to reach the termination criterion (i.e., either the TO is covered or the search timeout is reached) over the total number of runs for a given TO.

In addition to the dependent and independent variables described above, we also investigate the following co-factors that may affect the effectiveness and the efficiency across the treatments:

- *Number of input parameters*: each web application in our studies is a web-form with different input boxes where attackers can introduce malicious input strings. A higher number of input strings may increase the search time required by a given treatment to cover each TO. Therefore, we investigate the effect of this co-factor by applying each treatment on subjects with different number of input parameters. The purpose of this analysis is to measure the scalability of each treatment when the number of input parameters increases.
- *Alphabet size*: the alphabet of characters to use for generating input strings is represented by the set of printable ASCII characters. Instead of using the complete alphabet of all possible characters, we can reduce the size of the alphabet for the input parameters by omitting the characters, we know, are unused in the TOs. For example, if we observe that the target TO does not contain the character “A”, we can assume

that such a character is not useful to create malicious input strings. Therefore, we can reduce the size of the search space by removing the character “A” from the set of characters (alphabet) to use for generating malicious input. On the other hand, it may be difficult to determine what the restricted alphabet is when data validation and transformation routines are used. Therefore, we assess to what extent the usage of a restricted alphabet (positively/negatively) helps scalability.

- *Initial population*: all solvers start with an initial set of randomly generated solutions, which are tuples of randomly generated strings. Since the length of the input string that matches the target TO (upon the generation of the corresponding XML message) is unknown a priori, the length of the input strings in the initial population may affect the performance of our treatments. Indeed, if the randomly generated input strings are too long or too short (compared to the final solution) we would expect that each treatment will require more time (more edit operations) to find the malicious input string. To analyze the impact of the initial population on the performance of our treatments, we consider two different settings: (i) we generate random strings with a fixed (F) maximum lengths of characters each, or (ii) we generate strings of variable length (V) by using the “empty” character (see Section 6.1.1.1).

To perform a detailed evaluation of the effect of these three co-factors on our main treatments, we conducted a number of experiments with different settings, as summarized in Table 6.2. Each row in the table represents one experiment. The first column contains the name of the applications used in our case studies. The second column (*ExpId*) assigns a unique id to each experiment based on the application and its configuration. The third column *#TOs* lists the number of TOs in the experiment. The fourth column (*#Inp*) lists the number of input parameters. The fifth column (*PopLen.*) reports whether the length of the input strings in the initial population is fixed (Fix) or not (Var). The last column (*Res. Alph.*) indicates whether the search use a full alphabet set (Y) or restricted alphabet set (N). These configuration details are encoded in the *ExpId* values reported in the second column of Table 6.2. For example, the *ExpId* “S.2.F.Y” encodes the following settings for the SBANK (“S”) web application: it has two input parameters (“2”), input strings in the initial population have a fixed length (“F”), and a restricted alphabet set (“Y”) is used.

Therefore, we have (3 input parameters  $\times$  2 alphabets  $\times$  2 input string’s lengths = ) 12 different configurations for both SBANK and SSBANK. Instead, for XMLMao, we have only four possible configurations since this application has only one input parameter. For the industrial applications (*M* and *R*) in Study 2, we could not involve our industrial partners in the evaluation of all possible configurations given the high computational cost of this type

of study (see Section 6.2.2). For these reasons, the  $M$  and  $R$  applications are evaluated with only one configuration, M.2.V.Y and R.3.V.Y respectively. It is worth noticing that the number of input parameters for  $M$  and  $R$  is fixed since no alternative versions with a different number of input parameters are available. For the remaining setting, we opted for the configurations we empirically found to be statistically superior in Study 1. Therefore, for Study 2 we used the restricted alphabet (“Y”) and the initial population composed by input strings with variable length (“V”).

### 6.2.4 Experimental protocol

For each TO and each configuration, we executed each treatment  $\Omega$  and recorded if the TO is covered or not as well as the execution time. Each execution (i.e., run) is repeated 10 times (but only three times for the industrial systems) to account for the randomized nature of the optimization algorithms. The coverage data is binary since a given TO is either covered or not by a specific run of the treatment, whereas the execution time is recorded in minutes. This data is further used to calculate the selected performance metrics, i.e., *Success Rate* and average *Execution Time* for each TO.

For answering **RQ1.1**, we analyzed whether the success rates achieved by the solvers statistically differ when using two different fitness functions, i.e., real-coded edit distance ( $Rd$ ) and the string edit distance ( $Ed$ ). To this aim, we use the Fisher’s exact test [94] with a level of significance  $\alpha = 0.05$ . The Fisher exact test is a parametric test for statistical significance and is well-suited to test differences between ratios, such as the percentage of times a TO is covered. When the p-value is equal or lower than  $\alpha$ , the null hypothesis can be rejected in favor of the alternative one, i.e., a solver (e.g., HC) with one fitness function (e.g.,  $Rd$ ) covers the TO more frequently than the same solver but with another fitness function (e.g.,  $Ed$ ). We also use the Odds Ratio ( $OR$ ) [95] as measure of the effect size, i.e., the magnitude of the difference between the success rates achieved by  $Rd$  and  $Ed$ . The higher the  $OR$ , the higher is the magnitude of the differences. When the Odds Ratio is equal to 1, the two treatments being compared have the same success rate. Alternatively,  $OR > 1$  indicates that the first treatment achieves a higher success rate than the second one and  $OR < 1$  the opposite case.

For answering **RQ1.2**, we analyzed whether the execution time achieved by the solvers statistically differs when using  $Rd$  or  $Ed$ . To compare execution times, we use the non-parametric Wilcoxon test [96] with a level of significance  $\alpha = 0.05$ . When obtaining p-values  $\leq \alpha$ , we can reject the null hypothesis, i.e., a given treatment takes less time to cover the TO

**Table 6.2:** Experiment Settings: Experiment ID (Exp. ID) is named based on the corresponding application (App.), the number of inputs (#Inp.), length of input strings in the initial population (PopLen), and whether the alphabet is restricted (Res. Alph.).

App.	Exp. ID	#TOs	#Inp.	PopLen.	Res. Alph.
<b>SBank</b>	S.1.F.N	9	1	Fix	N
	S.2.F.N	10	2	Fix	N
	S.3.F.N	10	3	Fix	N
	S.1.F.Y	9	1	Fix	Y
	S.2.F.Y	10	2	Fix	Y
	S.3.F.Y	10	3	Fix	Y
	S.1.V.N	9	1	Var	N
	S.2.V.N	10	2	Var	N
	S.3.V.N	10	3	Var	N
	S.1.V.Y	9	1	Var	Y
	S.2.V.Y	10	2	Var	Y
	S.3.V.Y	10	3	Var	Y
<b>SSBank</b>	SS.1.F.N	9	1	Fix	N
	SS.2.F.N	10	2	Fix	N
	SS.3.F.N	10	3	Fix	N
	SS.1.F.Y	9	1	Fix	Y
	SS.2.F.Y	10	2	Fix	Y
	SS.3.F.Y	10	3	Fix	Y
	SS.1.V.N	9	1	Var	N
	SS.2.V.N	10	2	Var	N
	SS.3.V.N	10	3	Var	N
	SS.1.V.Y	9	1	Var	Y
	SS.2.V.Y	10	2	Var	Y
	SS.3.V.Y	10	3	Var	Y
<b>XMLMao</b>	X.1.F.N	12	1	Fix	N
	X.1.F.Y	12	1	Fix	Y
	X.1.V.N	12	1	Var	N
	X.1.V.Y	12	1	Var	Y
<b>M</b>	M.2.V.Y	4	2	Var	Y
<b>R</b>	R.3.V.Y	4	3	Var	Y

under analysis than another treatment. We also use the Vargha-Delaney ( $\hat{A}_{12}$ ) statistic [97] to measure the magnitude of the difference in the execution time. A value of 0.5 for the  $\hat{A}_{12}$  statistics indicates that the first treatment is equivalent, in terms of execution time to the second one. When the first treatment is better (lower execution time) than the second one,  $\hat{A}_{12} < 0.5$ . Naturally,  $\hat{A}_{12} > 0.5$  otherwise.

For **RQ2.1** and **RQ2.2**, we use the Friedman’s test [98] to verify whether multiple treatments are statistically different or not. It is a non-parametric equivalent to the ANOVA test [99] and thus does not make any assumption about the data distributions to be compared. More specifically, for **RQ2.1**, we compare the average success rates achieved by the different treatments in 10 independent runs across all web applications and configurations. Instead, for **RQ2.2** the comparison is performed considering the average execution time achieved in the 10 runs across all web applications and configurations. For both **RQ2.1** and **RQ2.2**, we use a level of significance  $\alpha = 0.05$ . When the p-values obtained from the Friedman’s test are significant (i.e.,  $\leq 0.05$ ), we apply the post-hoc Conover’s procedure [100] for pairwise multiple comparison. The p-values produced by the post-hoc Conover’s procedure are further adjusted using the Holm-Bonferroni procedure [101] to correct the significance level in case of multiple comparisons. Note that the purpose of **RQ2.1** and **RQ2.2** is to compare different solvers in terms of both effectiveness and efficiency; thus, we separately compare the four solvers described in Section 6.1.1.3 for the two fitness functions (e.g., *Rd* and *Ed*).

### 6.2.5 Parameter settings

Running randomized algorithms, and GAs in particular, requires to set various parameters to achieve acceptable results. In this study, we set the parameter values by following the recommendations in the related literature, as detailed below:

- **Mutation rate.** De Jong’s [102] recommended value of  $p_m=0.001$  for mutation rate has been used by many implementations of Genetic Algorithms. Another popular mutation rate has been defined by Grefenstette’s [103] as  $p_m=0.01$ . Further studies [14, 69, 104, 105] have demonstrated that  $p_m$  values based on the population size and chromosome’s length achieves better performance. Hence, for RGA and SGA we use  $p_m = (1.75)/(\lambda\sqrt{l})$  as mutation rate, where  $l$  is the length of the chromosome and  $\lambda$  is the population size. We also conducted some preliminary experiments with these different recommended mutation rates and we found that better results are indeed achieved when  $p_m$  is based on the population size and chromosome’s length. For HC,

we set the mutation rate to  $1/l$  (where  $l$  is the length of the chromosome) since there is no population for this solver. This parameter is not applicable for RS.

- **Crossover rate.** The crossover rate is another important factor for the performance of GAs. The recommended range for the crossover rate is  $0.45 \leq p_c \leq 0.95$  [68, 69]. In our experiments, we chose  $p_c = 0.70$  for RGA/SGA, which falls within the range of the recommended values. Notice that this parameter is not applicable to HC and RS.
- **Population size.** Selecting a suitable population size for GAs is also a challenging task since it can affect their performance. The recommended values used in the literature are within the range 30-80 [68]. From our preliminary experiments, we observed that the population size of 50 works best for RGA/SGA in our context. Such a value is also consistent with the parameters settings used in recent studies in search-based software testing [106, 107, 108]. This parameter is applicable only to population-based algorithms, i.e., RGA and SGA in our case.
- **Termination Criteria.** The search terminates when one of the following two stopping criteria is satisfied: a zero-fitness value is obtained (i.e., the target TO is covered) or the maximum number of fitness evaluations is reached. For SBANK, XMLMao and the two industrial systems, we set the maximum number of fitness evaluations to 300K. Instead, for SSBANK, we used a larger search budget of 500K fitness evaluations because it uses input validation routines, which make the TOs more difficult to cover. We also empirically found that a larger search budget is indeed needed for SSBANK compared to SBANK and XMLMao.

## 6.3. Results

This section discusses the results of our case studies, addressing in turn each of the research questions formulated in Section 6.2. Reporting the individual results along with the statistical tests for each TO, for each configuration, and for each treatment is not feasible due to the large number of resulting combinations, i.e., 2,016 in total. Therefore, we report the mean and standard deviation of the success rate and of the execution time obtained for all TOs of the same web application and with the same configuration (i.e., for each experiment/row in Table 6.2). For the statistical tests, we report the number of times the differences between pairs of treatments are statistically significant together with the average effect size measures.



**Table 6.3:** Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SBANK

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
S.1.F.N	100.00	0.00	22.22	23.86	71.11	38.87	67.78	29.49	92.22	13.02	90.00	7.07	0.00	0.00
S.2.F.N	96.00	6.99	8.00	11.35	50.00	43.97	39.00	29.61	88.00	22.01	79.00	23.31	0.00	0.00
S.3.F.N	93.00	10.59	3.00	6.75	40.00	34.96	28.00	27.81	60.00	51.64	45.00	41.16	0.00	0.00
S.1.F.Y	100.00	0.00	35.56	33.21	85.56	10.14	73.33	17.32	98.89	3.33	90.00	11.18	0.00	0.00
S.2.F.Y	92.00	7.89	21.00	20.25	45.00	28.38	44.00	36.88	89.00	14.49	90.00	10.54	0.00	0.00
S.3.F.Y	87.00	19.47	7.00	9.49	41.00	33.48	26.00	21.71	60.00	51.64	47.00	42.96	0.00	0.00
S.1.V.Y	100.00	0.00	40.00	36.40	100.00	0.00	78.89	13.64	97.78	4.41	86.67	14.14	0.00	0.00
S.2.V.Y	99.00	3.16	26.00	27.16	75.00	35.36	56.00	25.03	71.00	41.75	58.00	26.58	0.00	0.00
S.3.V.Y	93.00	13.37	6.00	8.43	69.00	41.75	46.00	26.75	70.00	48.30	37.00	34.66	0.00	0.00
S.1.V.N	100.00	0.00	21.11	27.13	77.78	33.46	67.78	27.74	96.67	7.07	86.67	11.18	0.00	0.00
S.2.V.N	100.00	0.00	3.00	6.75	61.00	50.43	41.00	35.10	76.00	35.02	63.00	32.34	0.00	0.00
S.3.V.N	90.00	15.63	0.00	0.00	55.00	47.90	24.00	23.19	60.00	51.64	29.00	29.98	0.00	0.00
Average	95.83	6.43	16.07	17.57	64.20	33.23	49.31	26.19	79.96	28.69	66.78	23.76	0.00	0.00

### 6.3.1 RQ1: What is the best fitness function for detecting XMLi vulnerabilities?

Table 6.3 summarizes the results of all treatments on the first subject SBANK in Study 1, listing the average success rate ( $SR$ ) along with the standard deviation ( $SD$ ) for each configuration. Each row in the table represents one configuration (experiment) identified by the unique id listed in the first column  $ExpId$ . The last row in the table lists the mean values for  $SR$  and  $SD$  across all configurations. As depicted in that table, for all solvers the real-coded edit distance ( $Rd$ ) achieved higher success rates compared to the string edit distance ( $Ed$ ). RGA achieved an  $SR$  of 95.83% with  $Rd$ , which is much higher than that of  $Ed$  with 16.07%.

These observations are confirmed by the Fisher’s exact test, as reported in Table 6.4. For each solver, this table lists the average Odds Ratios ( $OR$ ) of the success rates for each configuration, as well as the number of times where  $Rd$  achieved significantly higher ( $\#Rd > Ed$ ) or lower ( $\#Rd < Ed$ ) success rates compared to  $Ed$ , according to the Fisher’s exact test. The last row in the table lists (i) the average  $OR$  for all configurations (i.e., average of the columns  $Avg. OR$ ), and (ii) the total number of statistically significant cases (i.e., the sum of the  $\#Rd > Ed / \#Rd < Ed$  columns). We can observe that, for all solvers

**Table 6.4:** Average Odds Ratios (OR) of the Success Rate for SBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better ( $\# \text{Rd} > \text{Ed}$ ) or worse ( $\# \text{Rd} < \text{Ed}$ ) than the edit distance.

ExpId	RGA			SGA			HC		
	Avg. OR	#Rd>Ed	#Rd<Ed	Avg. OR	#Rd>Ed	#Rd<Ed	Avg. OR	#Rd>Ed	#Rd<Ed
S.1.F.N	216.25	8	0	2.88	0	0	2.30	0	0
S.2.F.N	223.93	10	0	2.55	0	0	2.36	0	0
S.3.F.N	235.08	10	0	2.10	0	0	5.98	1	0
S.1.F.Y	164.75	6	0	3.14	0	0	3.57	0	0
S.2.F.Y	57.59	10	0	2.13	0	0	2.39	0	0
S.3.F.Y	118.90	9	0	2.44	0	0	5.28	1	0
S.1.V.Y	162.06	5	0	7.16	0	0	3.81	0	0
S.2.V.Y	167.45	8	0	6.84	0	0	7.12	1	0
S.3.V.Y	196.65	10	0	8.79	1	0	54.14	3	0
S.1.V.N	224.83	8	0	3.59	0	0	3.22	0	0
S.2.V.N	373.24	10	0	8.08	1	0	5.25	1	0
S.3.V.N	269.64	10	0	10.36	3	0	22.89	4	0
Avg./Total	200.87	104	0	5.00	5	0	9.86	11	0

**Table 6.5:** Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for SSBANK

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
S.1.F.N	66.67	50.00	17.78	24.38	63.33	47.70	52.22	41.77	62.22	47.11	56.67	43.01	0.00	0.00
S.2.F.N	3.00	4.83	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.3.F.N	3.00	6.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.1.F.Y	66.67	50.00	23.33	28.28	56.67	43.59	50.00	39.05	63.33	48.48	57.78	44.10	0.00	0.00
S.2.F.Y	6.00	5.16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.3.F.Y	7.00	6.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
S.1.V.Y	66.67	50.00	32.22	39.62	64.44	48.51	54.44	42.75	64.44	48.51	47.78	37.34	0.00	0.00
S.2.V.Y	18.00	15.49	2.00	4.22	12.00	16.19	5.00	7.07	0.00	0.00	0.00	0.00	0.00	0.00
S.3.V.Y	21.00	19.12	2.00	4.22	6.00	12.65	3.00	4.83	0.00	0.00	0.00	0.00	0.00	0.00
S.1.V.N	66.67	50.00	13.33	20.62	62.22	47.64	46.67	37.42	65.56	49.27	44.44	37.45	0.00	0.00
S.2.V.N	4.00	6.99	0.00	0.00	2.00	4.22	2.00	4.22	0.00	0.00	0.00	0.00	0.00	0.00
S.3.V.N	5.00	9.72	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average	27.81	22.90	7.56	10.11	22.22	18.37	17.78	14.76	21.30	16.11	17.22	13.49	0.00	0.00

and for all configurations, the  $OR$  is always larger than one. The largest  $OR$  values are obtained for RGA, for which we observe that  $Rd$  is significantly better than  $Ed$  in most of the configurations ( $\approx 90\%$ ), with an average  $OR$  value ranging between 57.59 to 373.24. For

**Table 6.6:** Average Odds Ratios (OR) of the Success Rate for SSBANK application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better ( $\# Rd > Ed$ ) or worse ( $\# Rd < Ed$ ) than the edit distance.

ExpId	RGA			SGA			HC		
	Avg. OR	#Rd> Ed	#Rd<Ed	Avg. OR	#Rd>Ed	#Rd<Ed	Avg. OR	#Rd>Ed	#Rd<Ed
S.1.F.N	114.42	5	0	3.04	0	0	2.02	0	0
S.2.F.N	1.69	0	0	1.00	0	0	1.00	0	0
S.3.F.N	1.75	0	0	1.00	0	0	1.00	0	0
S.1.F.Y	107.27	4	0	2.12	0	0	2.71	0	0
S.2.F.Y	2.39	0	0	1.00	0	0	1.00	0	0
S.3.F.Y	2.68	0	0	1.00	0	0	1.00	0	0
S.1.V.Y	102.81	3	0	4.31	1	0	5.47	1	0
S.2.V.Y	4.91	0	0	2.29	0	0	1.00	0	0
S.3.V.Y	5.78	0	0	1.34	0	0	1.00	0	0
S.1.V.N	158.04	6	0	6.57	1	0	7.55	1	0
S.2.V.N	1.98	0	0	1.16	0	0	1.00	0	0
S.3.V.N	2.34	0	0	1.00	0	0	1.00	0	0
Avg./Total	42.17	18	0	2.15	2	0	2.15	2	0

the other solvers,  $OR$  is still larger than one but its magnitude is smaller when compared to that of RGA. In addition, according to the Fisher’s exact test, SGA and HC performed significantly better with  $Rd$  in only 4% and 9% of the configurations, respectively. These results indicate that the solver that most benefits from the usage of  $Rd$  is RGA.

For SSBANK in Study 1, success rate results are listed in Table 6.5. Despite the input validations in SSBANK, the solvers were able to obtain positive success rates in many configurations with both  $Rd$  and  $Ed$ . This means that some  $XMLi$  attacks can be still generated by inserting malicious inputs. Thus, the input validation procedures in SSBANK is sub-optimal, either because it is not adequately implemented, or because it is not possible to avoid all possible attacks using only input validation. We further observe that  $Rd$  achieved higher average success rates compared to  $Ed$  in all configurations. Indeed, the average improvement of the success rate when using  $Rd$  is 20% for RGA, 4% for SGA and 3% for HC. The corresponding results of the Fisher’s exact test and  $OR$  values are provided in Table 6.6. Similar to the results achieved for SBANK,  $OR$  is larger than one in most of the configurations, although the larger differences are observed for RGA. In particular, for this solver,  $Rd$  leads to an average  $OR$  ranging between 1.69 and 107.27. Instead, for the other two solvers, there is no statistically significant difference according to Fisher’s exact test for most of the cases, as confirmed by the  $OR$  values which are often around one.

For XMLMao in Study 1, the results for average success rates are provided in Table 6.7.

**Table 6.7:** Average Success Rates (SR) and Standard Deviation (SD) out of 10 runs per TO for XMLMao

ExpId	RGA				SGA				HC				RS	
	Rd		Ed		Rd		Ed		Rd		Ed		Ed	
	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD	SR	SD
X.1.F.N	100.00	0.00	35.00	38.26	78.33	24.80	70.83	29.37	98.33	3.89	91.67	8.35	0.00	0.00
X.1.F.Y	100.00	0.00	44.17	44.41	70.83	19.75	70.83	23.92	96.67	6.51	91.67	12.67	0.00	0.00
X.1.V.Y	100.00	0.00	51.67	37.86	95.83	11.65	82.50	16.03	95.00	6.74	90.00	11.28	0.00	0.00
X.1.V.N	100.00	0.00	30.00	32.19	85.83	22.75	81.67	21.25	95.00	6.74	88.33	10.30	0.00	0.00
Average	100.00	0.00	40.21	38.18	82.71	19.74	76.46	22.64	96.25	5.97	90.42	10.65	0.00	0.00

**Table 6.8:** Average Odds Ratios (OR) of the Success Rate for XMLMao application. For each solver, we also report the number of times the Success Rate obtained by the real-coded distance is statistically better ( $\# Rd > Ed$ ) or worse ( $\# Rd < Ed$ ) than the edit distance.

ExpId	RGA				SGA			HC		
	Avg. OR	$\#Rd > Ed$	$\#Rd < Ed$		Avg. OR	$\#Rd > Ed$	$\#Rd < Ed$	Avg. OR	$\#Rd > Ed$	$\#Rd < Ed$
X.1.F.N	175.32	8	0		2.50	0	0	2.81	0	0
X.1.F.Y	167.90	7	0		2.06	0	0	2.89	0	0
X.1.V.Y	94.09	6	0		4.94	0	0	2.06	0	0
X.1.V.N	156.95	9	0		2.51	0	0	2.32	0	0
Avg./Total	148.56	30	0		3.00	0	0	2.52	0	0

When applying RGA with  $Rd$ , the success rate is 100% for all configurations, which is much higher than that of  $Ed$  with 40%. This large difference is also confirmed by the Fisher’s exact test and very large  $OR$  values. Indeed, RGA with  $Rd$  is significantly better than RGA with  $Ed$  in 30 cases out of 48 (62%). The corresponding average  $OR$  values are very large, ranging between 94.09 and 175.32. In contrast, for the other two solvers, the differences are never significant when comparing the two fitness functions.

In general, for all three subjects in Study 1, we observe that Random Search (RS) always results in zero success rate, i.e., it was unable to cover any TO. This confirms the need for more advanced search algorithms to detect XMLi vulnerabilities. Furthermore, none of the solvers reached significantly higher success rates when using  $Ed$  instead of  $Rd$ . Therefore, for **RQ1.1**, we conclude that:

*The real-coded edit distance is very effective compared to the string edit distance, especially for RGA which, as shown next, happens to be the best solver as well.*

Regarding efficiency (**RQ1.2**), Tables 6.9, 6.10 and 6.11 report the average execution

**Table 6.9:** Average execution time (in minutes) results for SBANK

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
S.1.F.N	2.26	8.82	9.27	5.21	4.37	3.65	8.52
S.2.F.N	5.21	10.06	13.55	7.75	8.49	6.48	8.82
S.3.F.N	6.51	8.74	14.26	8.24	8.91	8.53	8.40
S.1.F.Y	1.52	7.19	7.13	4.73	2.20	2.33	9.11
S.2.F.Y	4.38	8.30	11.82	6.93	8.24	3.95	10.32
S.3.F.Y	5.73	8.03	12.28	7.83	9.52	7.42	9.05
S.1.V.Y	1.51	6.83	5.08	4.14	2.08	2.46	7.87
S.2.V.Y	2.95	8.14	8.92	6.14	12.10	5.65	7.90
S.3.V.Y	4.98	8.38	10.42	6.88	7.58	7.28	7.47
S.1.V.N	2.17	7.89	8.98	5.71	3.53	3.16	10.93
S.2.V.N	4.21	9.21	11.84	7.71	7.95	5.72	12.03
S.3.V.N	6.08	7.83	12.36	7.77	8.40	7.59	10.33
Average	3.96	8.28	10.49	6.59	6.95	5.35	9.23

time for the three subjects SBANK, SSBANK and XMLMao in Study 1, respectively. The results of the Wilcoxon's test, along with the  $\hat{A}_{12}$  statistics, are reported in Tables 6.12, 6.13 and 6.14. For each solver and each configuration, these tables list the effect size as well as the number of cases where the execution time for  $Rd$  is significantly lower or higher than  $Ed$ , based on the Wilcoxon's test and  $\hat{A}_{12}$  statistics.

Unlike the results of the success rates where  $Rd$  always performed better, we obtained mixed results for different solvers and applications when looking at efficiency.

For SBANK, RGA with  $Rd$  exhibited better efficiency with an average execution time of 3.96 minutes compared to 8.28 minutes for  $Ed$ . This is also confirmed by the reported low  $\hat{A}_{12}$  values (e.g., 0.14) and a significantly more efficient  $Rd$  in 80% of the cases. For SGA and HC,  $Ed$  obtained lower execution times and is significantly more efficient than  $Rd$  in 50% and 39% of the cases, respectively.

Efficiency results for XMLMao are similar to SBANK except for HC, for which the average execution time obtained with  $Rd$  is lower and is found to be significantly better than  $Ed$  in seven cases.

**Table 6.10:** Average execution time (in minutes) results for SSBANK

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
S.1.F.N	5.12	15.21	11.45	8.20	7.09	8.18	12.95
S.2.F.N	10.50	6.87	10.11	8.57	11.92	13.46	5.36
S.3.F.N	5.33	3.87	6.06	4.95	6.76	5.47	1.70
S.1.F.Y	5.49	13.88	8.77	7.96	6.67	6.11	13.86
S.2.F.Y	11.62	6.99	9.73	9.23	12.81	10.72	5.77
S.3.F.Y	6.24	4.15	5.94	4.29	7.26	4.78	1.71
S.1.V.Y	5.69	14.20	8.75	9.03	6.00	8.63	11.49
S.2.V.Y	11.40	8.89	11.42	10.41	12.48	11.60	4.85
S.3.V.Y	6.41	3.98	8.32	5.61	6.29	4.84	1.68
S.1.V.N	5.57	14.49	10.65	11.62	7.14	8.87	13.08
S.2.V.N	11.52	5.80	9.44	8.81	12.07	9.92	5.49
S.3.V.N	5.49	4.24	6.20	4.14	6.16	5.35	1.66
Average	7.53	8.55	8.90	7.74	8.56	8.16	6.63

Regarding SSBANK, the differences in average execution time obtained with  $Rd$  and  $Ed$  are not very large (i.e.,  $\approx 1$  minute), although statistically significant in favor of  $Ed$  in many cases, i.e., 25-53% for  $\#Rd > \#Ed$ .

Overall, in terms of efficiency, the real-coded edit distance is significantly better than the string edit distance for RGA, while the reverse is true for SGA and HC. One possible explanation for this difference is the better ability of the genetic operators in RGA to exploit the neighborhood of candidate solutions when using  $Rd$ . As explained in Section 6.1,  $Rd$  helps focus on sub-regions of the search space but it is necessary that the solvers are able to exploit this information to produce some benefits. To better explain this aspect, let us consider the TO=A (ASCII code 65) and let assume that the current input string is C (ASCII code 67), whose real-coded edit distance to the TO is  $|65 - 67|/(|65 - 67| + 1) = 0.67$ . When using the mutation operators of HC and SGA, the character C can be replaced by any other character with ASCII code from 32 to 127 even if only few characters in this set would lead to better  $Rd$  values, i.e., those with ASCII codes  $\in \{64, 65, 66\}$ . Therefore, the probability of replacing the character C with a better character is very low, i.e.,  $p = 3/95 \approx 0.03$ . Instead, in RGA the gaussian mutation gives higher probability to characters with ASCII codes that are closer to 67, which is the code of C. Indeed, the probability of replacing C with characters

**Table 6.11:** Average execution time (in minutes) results for XMLMao

ExpId	RGA		SGA		HC		RS
	Rd	Ed	Rd	Ed	Rd	Ed	
X.1.F.N	1.04	8.02	7.18	5.98	1.86	3.21	12.19
X.1.F.Y	1.02	6.73	7.45	6.17	1.35	2.09	8.84
X.1.V.Y	0.81	6.44	4.76	5.41	1.38	2.29	12.11
X.1.V.N	0.90	8.33	6.86	5.92	2.34	3.77	8.64
Average	0.94	7.38	6.56	5.87	1.73	2.84	10.44

with ASCII codes  $\in \{64, 65, 66\}$  is much higher in RGA when compared to HC and SGA. On the other hand,  $Rd$  is more expensive to compute than  $Ed$  since it is based on real-numbers and entails additional computations (as shown in Equation 6.2 in Section 6.1). Therefore,  $Rd$  will lead to better efficiency if and only if its additional overhead is compensated by a large saving in the number of fitness evaluations.

After manual investigation, we discovered that this is the case only for RGA. Indeed,  $Rd$  remained efficient for RGA in most of the cases due to a higher success rate than  $Ed$ , which resulted in a lower number of fitness evaluations during search. Instead, for SGA and HC the reduction in the number of fitness evaluations is small and thus it does not compensate for the additional overhead of  $Rd$  with respect to  $Ed$ . The only exception to this general rule is SSBANK, for which RGA with  $Rd$  is both more effective and less efficient than  $Ed$ . This results from the input validations performed in SSBANK, which produces an error message instead of a complete XML response whenever invalid inputs are submitted. When using  $Ed$ , computing the distance between such a small error message and the TO is much faster than doing so with a complete XML output generated upon the insertion of valid inputs.

In other words, our investigation reveals that the real-coded edit distance is more efficient, in terms of execution time, in the specific case where it achieves a much higher success rate than the string edit distance. Otherwise, if the success rates of the two fitness functions do not differ significantly, the string edit distance is more efficient.

Regarding **RQ1.2**, we conclude that,

*Unless a significantly higher success rate is achieved by the real-coded edit distance, the string edit distance leads to a more efficient search.*

To answer **RQ1**, we consider both the results of **RQ1.1** and **RQ1.2**.  $Rd$  fares better in terms of effectiveness whereas it is worse regarding efficiency. However, even when  $Rd$  leads

**Table 6.12:** Average  $A_{12}$  statistics of the execution time for SBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better ( $\# Rd > Ed$ ) or worse ( $\# Rd < Ed$ ) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
S.1.F.N	0.05	8	0	0.81	0	5	0.62	0	3
S.2.F.N	0.11	8	0	0.79	0	4	0.77	0	5
S.3.F.N	0.38	6	4	0.88	0	6	0.60	0	1
S.1.F.Y	0.11	7	0	0.75	0	4	0.61	1	2
S.2.F.Y	0.12	10	0	0.82	0	6	0.62	0	3
S.3.F.Y	0.27	6	0	0.76	0	4	0.70	1	5
S.1.V.Y	0.06	8	0	0.68	1	4	0.61	0	0
S.2.V.Y	0.03	10	0	0.73	0	4	0.60	1	3
S.3.V.Y	0.20	6	0	0.72	0	5	0.65	0	3
S.1.V.N	0.01	9	0	0.81	0	5	0.65	0	3
S.2.V.N	0.04	9	0	0.80	0	5	0.79	0	7
S.3.V.N	0.34	6	2	0.83	0	7	0.59	1	4
Avg./Total	0.14	93	6	0.78	1	59	0.65	4	39

to higher execution times, the difference with  $Ed$  ranges between 0.69 to 3.9 minutes on average, which is of limited practical consequences. Further, this relatively small difference is largely compensated with a much higher ability to detect XMLi vulnerabilities, up to an improvement of 80% in detection rate.

### 6.3.2 RQ2: What is the best solver for detecting XMLi vulnerabilities?

To answer **RQ2.1**, we compare the success rates of the four solvers (i.e., RS, HC, SGA, and RGA) for each fitness function (e.g.,  $Rd$  and  $Ed$ ). As reported in Table 6.3, the highest success rate (95.83%) for SBANK is achieved by RGA with  $Rd$ . Similarly, for SSBANK and XMLMao, RGA with  $Rd$  achieved the highest success rates of 27.81% and 100%, respectively. In contrast, the results are mixed when using  $Ed$  as fitness function: for SBANK and XMLMao, the highest success rate scores are obtained by HC (66.78% and 90.47% respectively), while for SSBANK the best success rate of 17.78% is obtained by SGA. Finally, RS fares the worst with a success rate of zero in all experiments and subjects, as it could not cover a single TO.

To establish the statistical significance of these results, we use the Friedman’s test [98]



**Table 6.13:** Average  $A_{12}$  statistics of the execution time for SSBANK application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better ( $\# \text{Rd} > \text{Ed}$ ) or worse ( $\# \text{Rd} < \text{Ed}$ ) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
S.1.F.N	0.12	6	0	0.81	0	6	0.47	1	0
S.2.F.N	0.80	0	8	0.70	0	2	0.49	1	1
S.3.F.N	0.77	0	5	0.72	0	3	0.53	0	2
S.1.F.Y	0.33	6	2	0.59	0	1	0.69	0	2
S.2.F.Y	0.87	0	9	0.55	0	0	0.65	0	3
S.3.F.Y	0.83	0	9	0.80	0	6	0.68	1	6
S.1.V.Y	0.21	6	1	0.54	1	1	0.48	1	1
S.2.V.Y	0.65	0	5	0.53	0	1	0.65	1	2
S.3.V.Y	0.68	0	3	0.69	0	5	0.75	0	4
S.1.V.N	0.28	6	1	0.49	3	2	0.56	0	0
S.2.V.N	0.91	0	9	0.52	0	0	0.67	0	4
S.3.V.N	0.93	0	10	0.75	0	7	0.63	1	5
Avg./Total	0.62	24	62	0.64	4	34	0.60	6	30

to compare the average success rates (over ten runs) achieved by the different solvers for all web applications, configuration settings, and TOs. When using  $Rd$  as fitness functions, the Friedman’s test reveals that the solvers significantly differ from each other in terms of effectiveness (p-value =  $2.58 \times 10^{-15}$ ). For completeness, Table 6.15 provides the ranking obtained by the Friedman’s test as well as the results of the post-hoc Conover’s procedure [100] for multiple pairwise comparisons. As we can observe, the best rank is obtained by RGA, which turns out to be significantly better than all the other solvers according to the post-hoc Conover’s procedure. The four solvers are also significantly different when using  $Ed$  as indicated by the Friedman’s test, yielding a p-value of  $8.2 \times 10^{-12}$ . However, as visible in Table 6.16, RGA is not the best solver with this fitness function, being ranked third above RS. The two other solvers, i.e., HC and SGA, are statistically equivalent according to the Conover’s tests, though HC obtained a slightly better rank based on Friedman’s test.

Given the mixed results obtained for the two fitness functions, we compare the best solver with  $Rd$  against the best solver with  $Ed$  to find the best treatment (i.e., combination of solvers and fitness functions). To this aim, we performed the Friedman’s test comparing the average success rates of RGA with  $Rd$  against HC and SGA with  $Ed$ . Results show these three treatments are statistically different in terms of effectiveness (p-value =  $9.17 \times 10^{-11}$ ). The post-hoc Conover procedure confirms the superiority of RGA with  $Rd$  over the other two treatments. Therefore, for **RQ2.1** we conclude that:

**Table 6.14:** Average  $A_{12}$  statistics of the execution time for XMLMao application. For each solver, we also report the number of times the efficiency of the real-coded distance is statistically better ( $\# Rd > Ed$ ) or worse ( $\# Rd < Ed$ ) than the edit distance.

ExpId	RGA			SGA			HC		
	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed	Eff.Siz	#Rd>Ed	#Rd<Ed
X.1.F.N	0.23	8	1	0.66	0	3	0.41	4	0
X.1.F.Y	0.26	8	1	0.61	0	1	0.45	2	0
X.1.V.Y	0.26	8	2	0.54	0	2	0.48	0	1
X.1.V.N	0.16	8	0	0.64	0	2	0.46	1	0
Avg./Total	0.23	32	4	0.61	0	8	0.45	7	1

**Table 6.15:** Ranking produced by the Friedman’s (smaller values of Rank indicate more effectiveness) when using  $Rd$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	RGA	1.02	(2), (3), (4)
2	HC	2.50	(3), (4)
3	SGA	2.70	(4)
4	RS	3.77	-

**Table 6.16:** Ranking produced by the Friedman’s (smaller values of Rank indicate more effectiveness) when using  $Ed$ . For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	HC	1.62	(3), (4)
2	SGA	1.87	(3), (4)
3	RGA	2.86	(4)
4	RS	3.64	-

*RGA combined with the real-coded edit distance as fitness function is the best solver in terms of effectiveness.*

Regarding **RQ2.2**, we analyze the execution time of the solvers for each fitness function (i.e.,  $Rd$  and  $Ed$ ) separately. The results of this analysis for the three subjects in Study 1 are

**Table 6.17:** Ranking produced by the Friedman’s (larger values of Rank indicate more efficiency) when using *Rd*. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	RGA	3.53	(2), (3), (4)
2	HC	2.36	(4)
3	RS	2.28	-
4	SGA	1.82	-

reported in Tables 6.9, 6.10 and 6.11. For all three subjects, the most efficient solver with *Rd* is always RGA, whose average running time ranges between 0.94 (for XMLMao) and 7.53 (for SSBANK) minutes. Further, the average execution time for HC ranges between 1.73 (for XMLMao) and 8.56 (for SSBANK) minutes, whereas it ranges between 6.56 and 10.49 minutes for SGA. The differences in execution times are also confirmed by Friedman’s test, which returned a p-value of  $6.26 \times 10^{-6}$ . To better understand for which pairs of solvers such a significance holds, Table 6.17 shows the complete ranking produced by Friedman’s test as well as the results of the post-hoc Conover’s procedure. The best rank is achieved by RGA, which significantly outperforms all the other solvers when using *Rd*. The second best ranked solver is HC, which is statistically more efficient than SGA only.

When using *Ed* as fitness function, there is no clear winner among the four solvers in terms of efficiency for the three subjects in Study 1. Indeed, HC is the most efficient solver for XMLMao and SBANK, while SGA is for SSBANK. From the statistical comparison performed with the Friedman’s test, we can definitely conclude that the four solvers are significantly different in terms of execution time. However, the post-hoc Conover’s procedure revealed that statistical significance holds only when comparing one pair of solvers (see Table 6.18): HC and RS.

To find out which treatment among all possible combinations of solvers and fitness functions is the most efficient (as measured by the average execution time), we performed Friedman’s test to compare RGA with *Rd* and HC with *Ed*, which are the best treatments for the two fitness functions (see Tables 6.17 and Table 6.18). The resulting p-value of 0.002 and the corresponding Friedman’s ranking indicate that RGA with *Rd* is significantly more efficient than HC with *Ed*. Thus, addressing **RQ2.2**, we conclude that:

**Table 6.18:** Ranking produced by the Friedman’s (larger values of Rank indicate more efficiency) when using *Ed*. For each solver, we also report whether it is significantly better than the other solvers according to the post-hoc procedure.

ID	Solver	Rank	Significantly better than
1	HC	3.07	(3)
2	SGA	2.78	-
3	RS	2.21	-
4	RGA	1.93	-

**Table 6.19:** Results on the industrial systems

Config.	TO	Successes	Avg. Iterations
M.2.V.Y	Close	0	300k
	Meta	0	300k
	Replicate	0	300k
	Replace	3	23k
R.3.V.Y	Close	0	300k
	Meta	0	300k
	Replicate	0	300k
	Replace	2	147k

*RGA combined with the real-coded edit distance as fitness function is the most efficient solver in terms of execution time.*

### 6.3.3 RQ3: How does the proposed technique scale to industrial systems?

To address **RQ3**, we carried out experiments on two industrial systems (recall Section 6.2), provided by one of our industrial partners. As the experiments had to be run on a dedicated machine, only 4 TOs, one solver (the best from the previous experiments) and 3 repetitions were carried out. Table 6.19 shows the results of these experiments.

In both cases, it was possible to solve at least one TO. The others are unfeasible, due

to the type of input sanitization carried out by those systems. Note that whether a TO is feasible or not depends on the actual implementation of the SUT. We used the actual systems without modifications, i.e., we did not inject any artificial security vulnerability to check if our technique could spot them.

In the case of  $R$ , there was no direct mapping from the JSON fields and the fields in the XML of the TO (e.g., two of the JSON fields are concatenated in one single field in the output XML of the TO), making the search more difficult compared to  $M$ . Regarding  $M$ , interestingly, one of the fields that leads to the XML injection does get sanitized. Given the TO target field:

0 or 1=1

one of the valid inputs to solve that TO was

0 or 1=1<v2

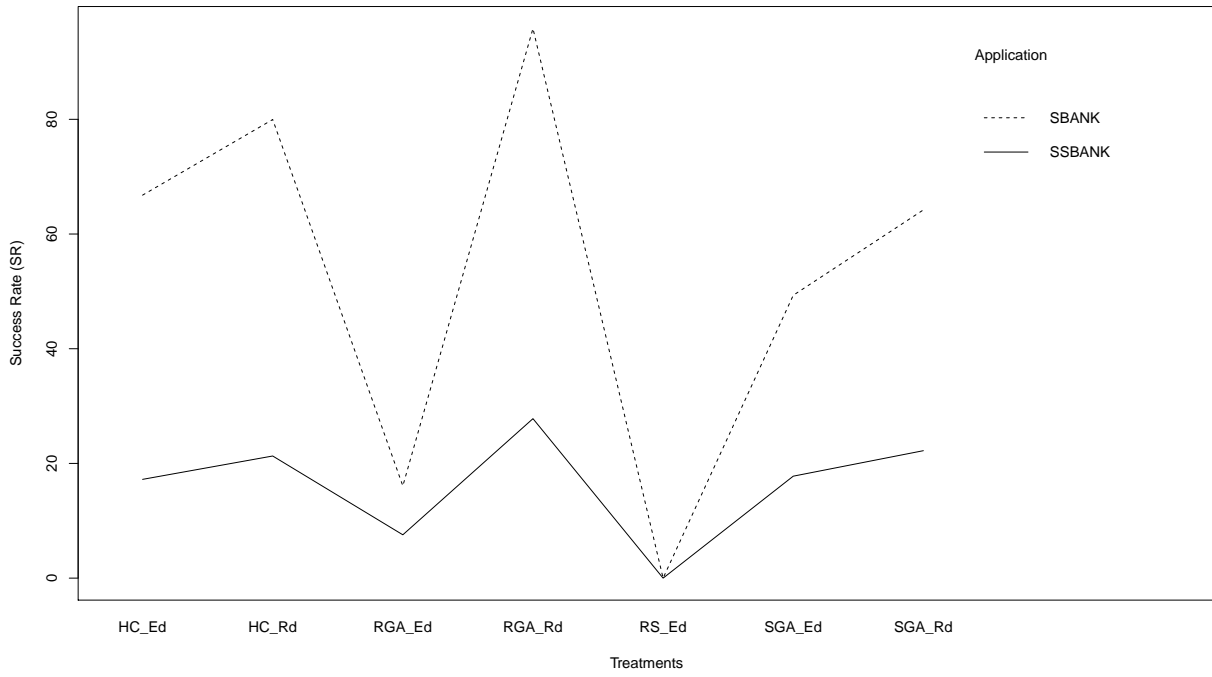
as any character including and after the first  $<$  is removed as part of the input sanitization.

*Our proposed technique was able to produce inputs that can detect XMLi vulnerabilities in the evaluated industrial systems.*

## 6.4. Additional Analysis

In this section, we investigate the various co-factors that may affect the effectiveness of the solvers. For this purpose, we use the two-way permutation test [109], which is a non-parametric test to verify whether such co-factors statistically affect or not the search effectiveness. This test is equivalent to the two-way Analysis of Variance (ANOVA) test [99].

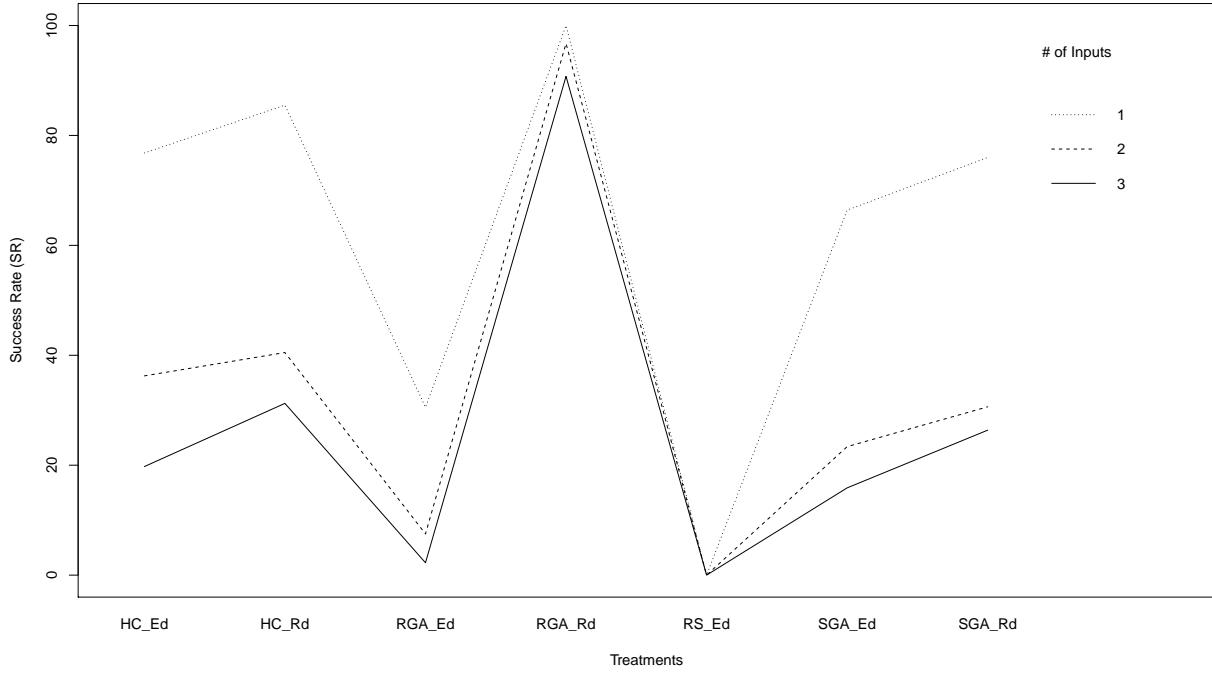
**Input validation:** To investigate the effect of input validation, we compare the average  $SR$  for SBANK and SSBANK, which are two different front-ends for the same real-world bank card processing system. The difference is that one front-end uses input validation (i.e., SSBANK) while the other not (i.e., SBANK). This analysis can be performed by comparing the results reported in Tables 6.3 and 6.5. For each treatment, the average  $SR$  for SBANK is always higher than the  $SR$  scores achieved for SSBANK. The p-value  $<0.05$  obtained from the two-way permutation test shows that the co-factor *input validation* significantly affects the performance of the solvers. This can also be observed from the interaction plot depicted in Figure 6.3: for all the solvers, the average  $SR$  of the SBANK is always higher



**Figure 6.3:** Comparison of the average success rates for SBANK (without input validation) and SSBANK (with input validation).

compared to that of SSBANK. We note that the best treatment in our study, which is RGA with *Rd*, could reach a success rate greater than 20% in the presence of input validation. Though there exist input validation routines in SSBANK, they are applied only on one input parameter instead of all three. Thus, if a front-end web application uses incomplete input validation, our proposed search-based technique is able to detect XMLi vulnerabilities in a reasonable amount of time (i.e., less than 10 minutes on average).

**Number of input parameters:** As described in Section 6.2.5, we have three different versions of SBANK and SSBANK with varying numbers of input parameters. This allows us to analyze how the success rate is impacted when increasing the number of input parameters. The results of this analysis are reported in Table 6.20, with the average *SR* for all the treatments with the same number of input parameters and for each application. We can clearly see that for most of the treatments the larger the number of input parameters, the smaller the success rates achieved by the different treatments. For example, for HC with *Ed* the success rate is 88.33% with one input parameter and it dramatically decreases to 72.50%



**Figure 6.4:** Comparison of the average success rates for SBANK and SSBANK with 1, 2 and 3 input parameters.

and 39.50% with two and three input parameters, respectively. This overall pattern is also observable from the interaction plot in Figure 6.4. The only exception to this general rule is the combination of RGA with *Rd* for which we can observe limited variation in the average success rate as depicted in Figure 6.4. Therefore, our best configuration (i.e., RGA with *Rd*) is not only more effective and more efficient, but is also more scalable as it is little affected by increasing the number of parameters, as opposed to the other treatments.

The permutation test for the number of input parameters also reveals a significant interaction between this co-factor and the *SR* ( $p$ -value $<0.05$ ). Hence, we conclude that increasing the number of inputs adversely affects the average *SR* of the solvers, i.e., the higher the number of input parameters, the more difficult is to detect XMLi vulnerabilities.

**Initial population:** As described in Section 6.2.3, the initial set of random tests can be composed by input strings with Fixed (F) or Variable (V) length. To investigate the effect of this co-factor, we compare the average *SR* obtained by each solver, for each application, when using Fixed and Variable length. The result of this analysis is reported in Table 6.21,

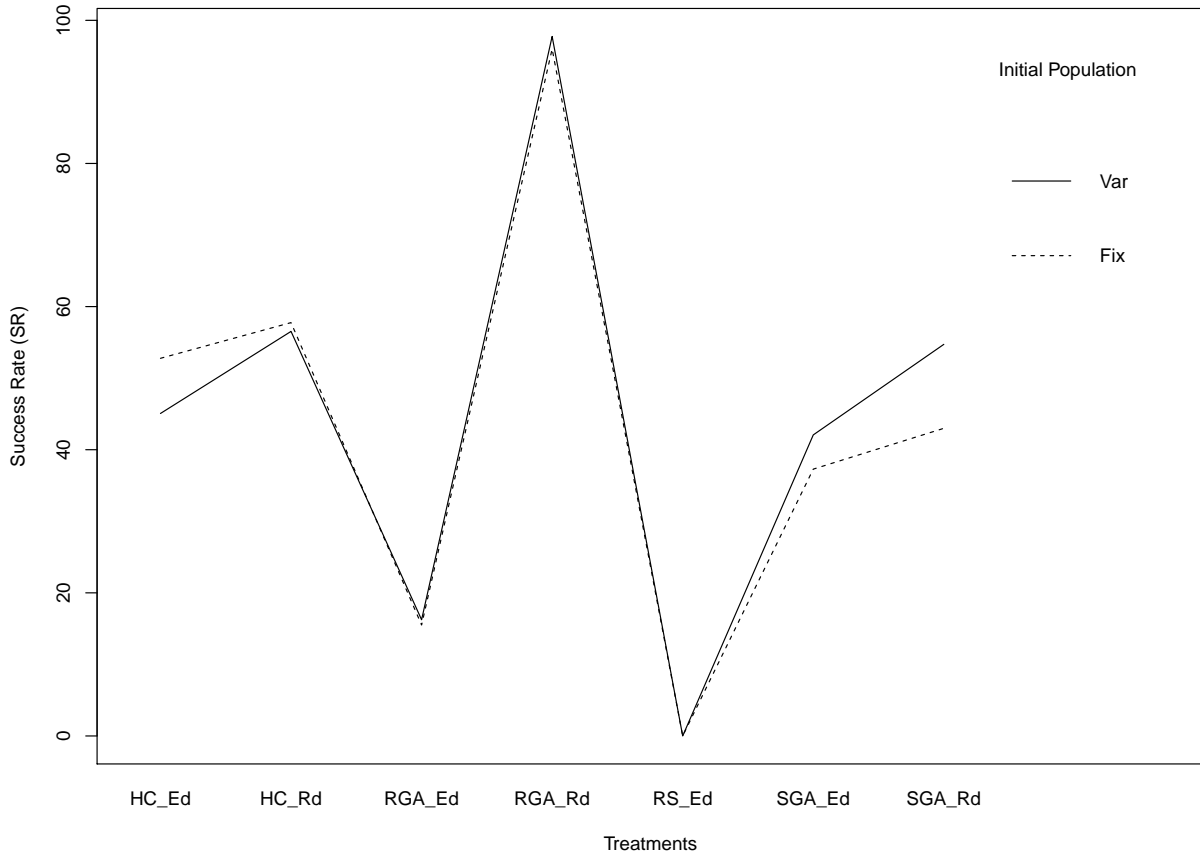
**Table 6.20:** Comparison of the average Success Rates (SR) of the experiments involving apps. with 1, 2 and 3 Inputs

App./Solver	SBANK			SSBANK		
	1 input	2 inputs	3 inputs	1 input	2 inputs	3 inputs
RGA with Rd	100.00	96.75	90.75	66.67	7.75	9.00
RGA with Ed	29.72	14.50	4.00	21.67	0.50	0.50
SGA with Rd	83.61	57.75	51.25	61.67	3.50	1.50
SGA with Ed	71.94	45.00	31.00	50.83	1.75	0.75
HC with Rd	96.39	81.00	62.50	63.89	0.00	0.00
HC with Ed	88.33	72.50	39.50	51.67	0.00	0.00
Avg/app	78.33	61.25	46.50	52.73	2.25	1.96

**Table 6.21:** Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length

Solver	SBANK		SSBANK		XMLMAO	
	Fix	Var	Fix	Var	Fix	Var
RGA with Rd	94.67	97.00	25.39	30.22	100.00	100.00
RGA with Ed	16.13	16.02	6.85	8.26	39.58	40.83
SGA with Rd	55.44	72.96	20.00	24.44	74.58	90.83
SGA with Ed	46.35	52.28	17.04	18.52	70.83	82.08
HC with Rd	81.35	78.57	20.93	21.67	97.50	95.00
HC with Ed	73.50	60.06	19.07	15.37	91.67	89.17
Avg/app	61.24	62.82	18.21	19.75	79.03	82.99

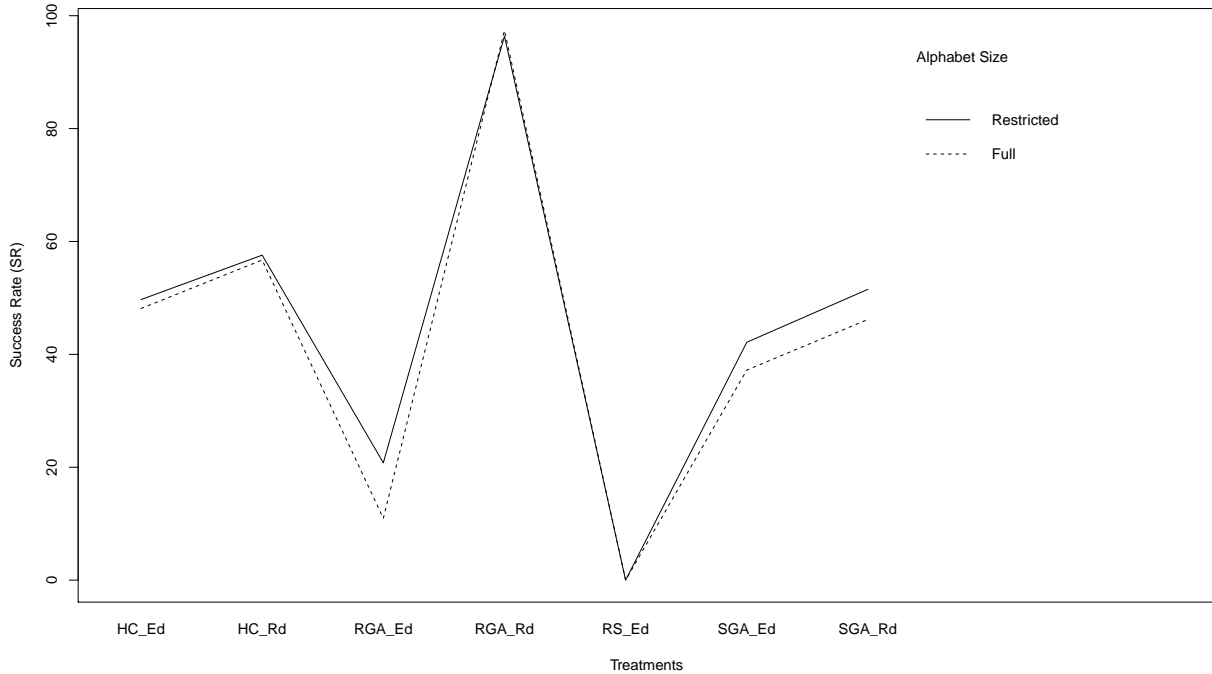




**Figure 6.5:** Comparison of the average success rates (SR) when using an initial population composed by input strings with Fixed (Fix) or Variable (Var) length.

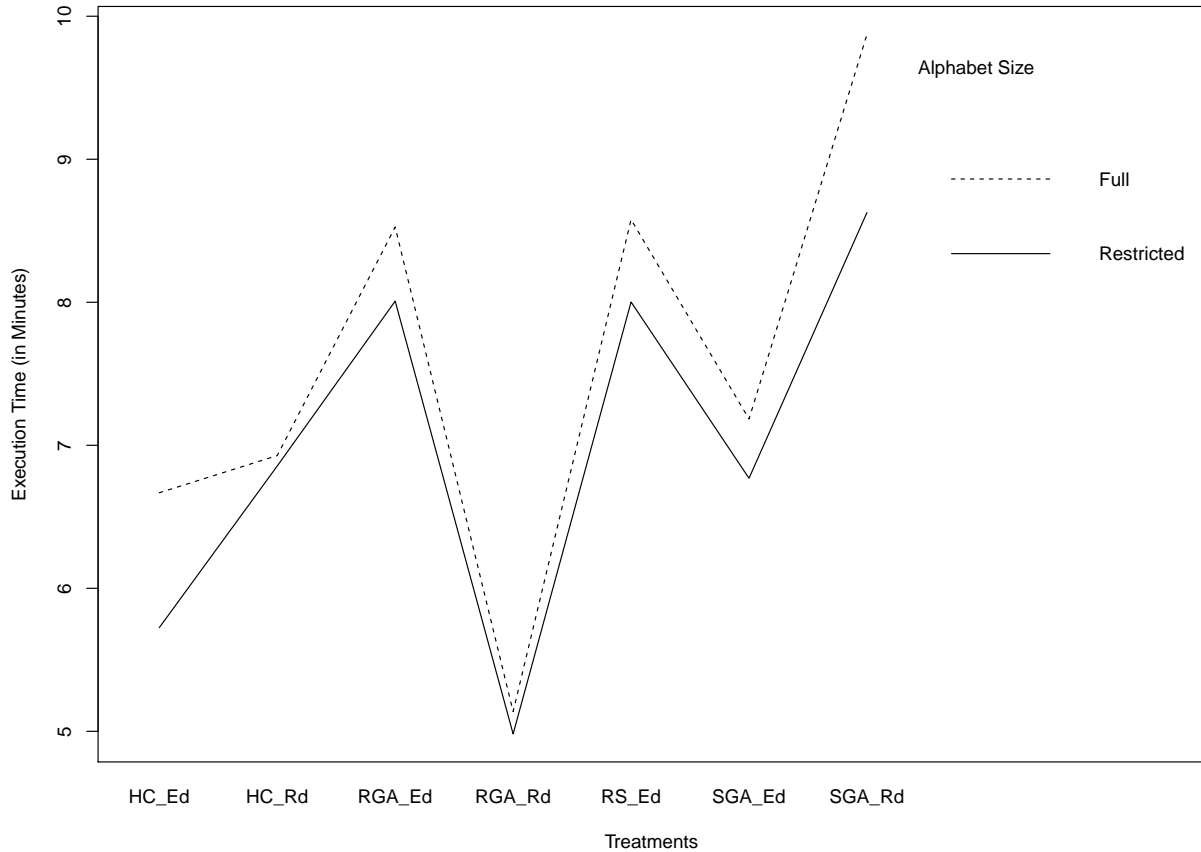
which shows the average  $SR$  achieved for each solver and application. We can see that the difference between the two types of setting is limited, i.e., it is on average 1.58% for SBANK, 1.54% for SSBANK, and 3.96% for XMLMao. These small differences can be visualized through the interaction plot in Figure 6.5 and a permutation test further shows they are not significant ( $p$ -value=0.80). Therefore, we conclude that the length of the input strings in the initial population (or the initial solution for HC) does not significantly affect the performance of the solvers.

**Alphabet size:** Instead of using the complete alphabet (i.e., all possible ASCII characters), we can restrict its size by considering only the characters we determine to be used in the TOs. However, as discussed in Section 6.2.3, this strategy may be detrimental when



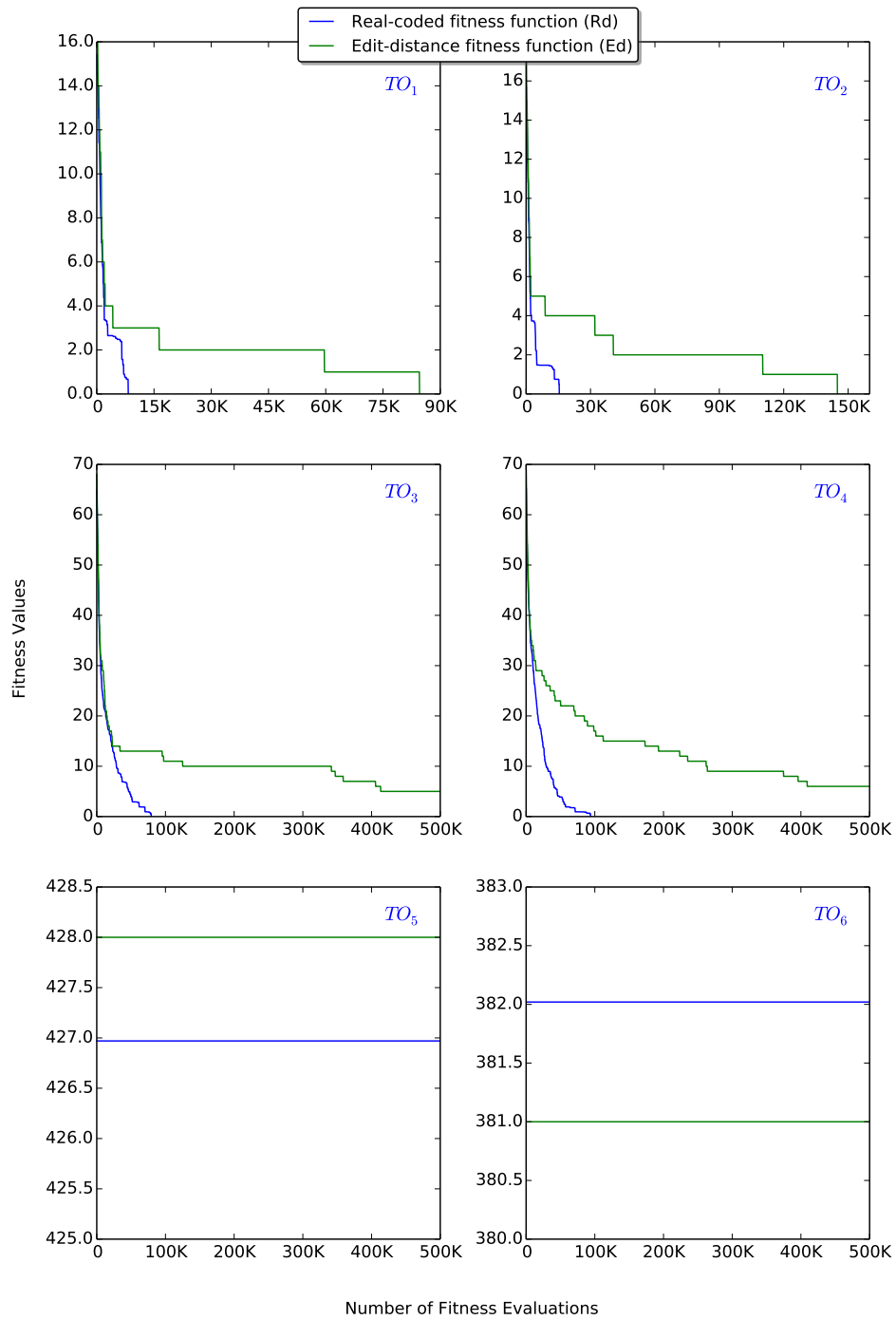
**Figure 6.6:** Comparison of the average success rates (SR) with complete (Full) and restricted (Restricted) alphabet size.

there is no straightforward relationship match between input strings and the generated XML messages, due to transformations and validation. Therefore, we want to analyze the impact of this strategy on the performance of the various treatments. The interaction plot in Figure 6.6 indicates that the effect of this co-factor on the average success rate  $SR$  is very small. Only for RGA with  $Ed$  and SGA (either with  $Ed$  or  $Rd$ ) we can observe slightly higher success rates when using the restricted alphabet size. The permutation test also reveals no significant interaction (p-value=0.55) between the success rate (i.e., effectiveness) and the size of the alphabet. With respect to efficiency, Figure 6.7 depicts the effect of the size of the alphabet on average execution time. As we can observe, the efficiency of RGA with  $Rd$  is not affected by this co-factor while all the other solvers achieved a lower execution rate with a restricted alphabet. However, the effect is still not significant according to permutation tests, i.e., p-value=0.20. Therefore, reducing the size of the alphabet is not recommended given its low impact on both effectiveness and efficiency of the various treatments (and RGA with  $Rd$  in particular) combined with the high risk of unintentionally excluding characters that may lead to  $XMLi$  attacks.



**Figure 6.7:** Comparison of the average execution time with complete (Full) and restricted (Restricted) alphabet size.

**Fitness convergence:** To investigate the convergence of the two fitness functions over time, we recorded the fitness function of the best individual in the population throughout the GA generations. For this analysis, we selected the best solver (*RGA*) and one representative case (*SS.1.F.N*) to compare the values of *Rd* and *Ed* throughout the generations for six TOs. As depicted in Figure 6.8, for TO1 and TO2, RGA with *Rd* converged much faster ( $< 30K$  fitness evaluations) to reach zero fitness (i.e., to cover the TO), while *Ed* used a large number of fitness evaluations, i.e., 90K for TO1 and 150K for TO2. RGA with *Rd* also covered the other two TOs, i.e., TO3 and TO4 in less than 100K fitness evaluations while *Ed* could not cover them in 500K fitness evaluations. The last two TOs (TO5 and TO6) are infeasible to cover since the application SSBANK always returns an error message for



**Figure 6.8:** Convergence rate for RGA with  $Rd$  and  $Ed$  for SS BANK with  $SS.1.F.N$  configuration.

invalid inputs. Therefore, the fitness function remains flat for both *Rd* and *Ed*, meaning that the search cannot converge towards zero fitness. Hence, for applications with input validation (like SSBANK), it should be expected that some TOs are simply infeasible to cover. However, our results show that it is still possible to detect XMLi vulnerabilities which are not adequately addressed by input validation routines.

## 6.5. Related Work

Related work for *XMLi* testing and search-based approaches for security testing, has already been discussed in Section 5.4. This section describes, how we improve our approach presented in the previous chapter, in terms of effectiveness and efficiency.

In the previous chapter, we presented a search-based approach for generating test inputs exploiting XML injection vulnerabilities in front-end web applications. We used the standard Genetic Algorithm along with the string-edit distance to find malicious test inputs. We evaluated our approach on several web applications including a large industrial application and we also compared it with random search. We found our proposed search-based testing approach to be very effective, as it was able to cover vulnerabilities in all case studies while the random search could not, in any single case.

This chapter extends our search-based testing approach, presented in the previous chapter, in several ways. First, we introduced a different fitness function, i.e., the Real-coded Edit Distance (*Rd*), which further improves the traditional string edit distance (*Ed*). Second, we investigated two further optimization algorithms, namely Real-coded Genetic Algorithm (*RGA*) and Hill Climbing (*HC*), in addition to the standard Genetic Algorithm (*SGA*) and random search (*RS*). Third, we enlarged our empirical evaluation using an additional industrial application. Last, we conducted an extensive evaluation by comparing all possible combinations of solvers (i.e., *SGA*, *RGA*, *HC* and *RS*) and fitness functions (i.e., *Rd* and *Ed*). Our new results show that *RGA* with *Rd* is significantly superior the previous approach [16] in terms of both effectiveness and efficiency.

## 6.6. Threats to Validity

In this section, we discuss the threats that could potentially affect the validity of our findings.

**Internal validity:** In our context, there are two main threats related to internal validity: (i) the use of randomized algorithms and (ii) the choice of parameter settings for the solvers. To mitigate the first threat, we repeated each experiment several times, i.e., 10 times for each subject of Study 1 and three times for the industrial systems in Study 2, and reported the aggregated results. The use of rigorous statistical analysis also adds support to our findings. To mitigate the threats arising from the parameter settings, especially for Genetic Algorithms, we used the parameter values that are recommended in the literature and also carried out some preliminary experiments before using them for the complete experiments (as described in Section 6.2.5). Besides, we used the same parameter settings for all solvers. However, it is worth noting that the empirical study carried out in this chapter is based on a software tool we developed. As for any software, although it has been carefully tested, we cannot guarantee that such tool is bug-free.

**External validity:** Threats to external validity concern the generalization of our findings. The empirical study is based on a small set of applications. This was due to two main reasons:

- We conducted a large empirical study with different solvers and fitness functions, which required a cluster of computers running for days. Using more subjects would have not been feasible.
- Enterprise systems are usually not accessible on open-source repositories, so we were limited by what was provided by our industrial partners. Furthermore, due to technical constraints, such systems had to be run on a dedicated machine, and not a cluster of computers.

Although this presents a threat to the generalization of our results, we have made sure to evaluate our approach with different types of applications, i.e., front-end web applications interacting with the bank-card processing system, an open source application and real-world industrial application with millions of registered users. Further, we have also evaluated applications with different levels of complexity, i.e., three versions of SBANK and SSBANK with varying number of parameters and the presence of input validation routines. Also, using real industrial systems in the case study does prove that our technique can scale to actual systems used in practice.

**Conclusion validity:** Regarding the threats to conclusion validity, we have carried out the appropriate and well-known statistical tests along with multiple repetitions of the experiments. In particular, we have used the parametric Fisher's exact test, the non-parametric

Wilcoxon test, Friedman's test and the two-way permutation test to find whether the outcomes (success rate for effectiveness and average execution time for efficiency) of the treatments differ significantly. Besides, we have also used the Odds Ratio (OR) and Vargha-Delaney ( $\hat{A}_{12}$ ) statistics to measure the effect size, i.e., the magnitude of the observed difference. Our conclusions are based on the results of these tests and statistics.

It should also be noted that being able to carry out a successful XML injection attack does not necessarily mean that the receiver of such messages (e.g., a SOAP web service) will be compromised. This depends on how the receiver is implemented (e.g., Does it have adequate level of input validation/sanitization routines?). However, in practice, internal web services (not directly accessible on the internet) might not be subject to rigorous penetration testing as the user front-end, and so might be less secure.

## 6.7. Summary

In this chapter, we have presented an effective approach to automatically generate test cases for the security testing of web applications based on metaheuristic search, with a focus on *XMLi* vulnerabilities. This work is built on the approach presented in the previous chapter, i.e., the automated generation of SUT inputs that generate messages matching the TOs (our test objectives, malicious XML messages). In this chapter, we investigated different strategies to improve the effectiveness and efficiency of test generation part.

We evaluated four different search algorithms, with two different fitness functions. We have evaluated and compared them on artificial and open source systems and two industrial systems (one being a very large web application). Our results are promising as the proposed approach was able to effectively and efficiently uncover vulnerabilities in all these case studies. In particular, a Genetic Algorithm, using a fitness function minimizing a real-coded edit distance between TOs and generated XML messages, clearly showed to be the best algorithm and appeared to be sufficiently effective and efficient, to be used in practice.

# Chapter 7

## Conclusions and Future Work

This chapter summarizes the research contributions of this dissertation and discusses potential areas for future work.

### 7.1. Summary

This dissertation investigated the limitations of existing approaches for security testing of XML-based vulnerabilities, in an industrial context. Real-world XML-based systems typically have a multi-tiered infrastructure composed of several tiers including: front-end web applications, XML gateway/firewall, and back-end web services. Effecting and efficient detection of XML-based vulnerabilities requires security testing strategies that are tailored toward each specific tier (e.g., XML gateway/firewall). However, existing security testing approaches for XML-based vulnerabilities rely only on fuzz testing strategy, which are unlikely to generate the complex tests required for detecting these vulnerabilities. Indeed, fuzzing approaches tend to generate simple tests, which are either detected at the first tier, i.e., front-end web application, or blocked at the XML gateway/firewall tier.

To increase the efficiency and the effectiveness of security testing for XML-based systems, we have proposed several security testing strategies aimed at detecting XML-based vulnerabilities for various components of multi-tiered business applications. These strategies have been developed in close collaboration with a leading financial service provider in Luxembourg/Switzerland, namely SIX Payment Services. All our proposed strategies are



automated, effective and scalable. They have been empirically evaluated on several industrial as well as on open-source systems.

In short, this dissertation made the following contributions:

Chapter 3 covered security testing for the two vulnerabilities related to XML parsers, namely *Billion Laugh (BIL)* and *XML External Entities (XXE)*. We conducted a systematic and large-scale experimental assessment of 13 most popular XML parsers. Furthermore, we also investigated several open-source systems adopting one of the vulnerable XML parsers. We proposed a testing approach that uses various performance measurements (e.g., CPU time, memory consumption), as test oracles for detecting *BIL* and *XXE* vulnerabilities. Our results show that most of the XML parsers are vulnerable and so are the systems using one of the vulnerable parsers. These results can be used to motivate software developers to take appropriate security measures to detect and overcome the vulnerabilities related to XML parsers.

Chapter 4 addressed security testing for XML Injection (*XMLi*) vulnerabilities in XML gateway/firewall and back-end web services. We developed *SOLMI*, an automated testing approach and tool, based on constraint-solving and input-mutation techniques that covers a wide range of *XMLi* attacks. Our test generation strategy first uses a constraint solver and an attack grammar to generate attack payloads that satisfy the associated domain constraints (e.g., security policies at XML gateway/firewall). Then, the generated (malicious) payloads are used to mutate existing (benign) XML messages. The resulting mutated XML messages have higher chances to bypass the gateway. We evaluated and compared our approach with a state-of-the-art tool, namely *ReadyAPI*, when testing the financial system of our industrial collaborator. The target system consists of 44 complex web services at the back-end that are protected by an XML gateway (firewall). Our results demonstrated that a large proportion (78.86%) of the attacks, generated with *SOLMI*, could bypass the XML gateway/firewall, while for *ReadyAPI*, the percentage of bypassing attack was 2.37% only.

Chapter 5 presented a novel testing approach targeting the front-end web applications. We developed an automated, black-box testing approach using Search-Based Testing (SBT). The aim is to search for such test inputs that, when submitted to the web application, can generate malicious XML messages (previously created with our tool *SOLMI*). If such inputs exist, the web application is considered vulnerable to *XMLi*, meaning that the input sanitization and transformation procedures are not able to prevent *XMLi* attacks. We used the standard *Genetic Algorithm* with the traditional string edit distance as the fitness function

to guide the search towards malicious inputs. We evaluated our approach on several artificial systems and one large industrial web application. Our results demonstrated that the approach was able to discover *XMLi* vulnerabilities in all case studies.

Chapter 6 focused on boosting the efficiency and the effectiveness of the search-based test case generation approach presented in Chapter 5. We investigated four different search algorithms, namely *Standard Genetic Algorithm (SGA)*, *Real-coded Genetic Algorithm (RGA)*, *Hill Climbing (HC)* and *Random Search (RS)*. Furthermore, to overcome the limitations of the traditional string edit distance, we evaluated a different fitness function, namely the *Real-coded Edit Distance (Rd)*. We provided a detailed analysis of all possible combinations of the search algorithms and fitness functions, to determine the best combination in terms of effectiveness and efficiency. Our results demonstrated a significant improvement over the previous search-based approach. In particular, the *Real-coded Genetic Algorithm (RGA)*, using *Real-coded Edit Distance (Rd)*, appeared to be sufficiently effective, efficient and scalable, to be used in practice for detecting *XMLi* vulnerabilities in web applications.

## 7.2. Future Work

In this dissertation, we focused on security testing strategies for detecting XML-based vulnerabilities in web services and applications. There are other injection vulnerabilities, e.g., *XPATH*, *LDAP*, or *JSON* injections, that also warrant attention from academia and practitioners. Furthermore, the continuously evolving nature of web services and applications, and the use of alternative technologies (e.g., *REST* web services), may result in new injection vulnerabilities. While the security testing strategies presented in this dissertation focuses on XML-based vulnerabilities, they can also be generalized to other types of vulnerabilities. For instance, our search-based testing approach can be applied to *Cross-site scripting* or *XPATH* injection vulnerabilities, by modifying the target malicious XML messages to the corresponding types of attacks for these vulnerabilities. Similarly, the constraint-solving and input mutation testing approach (*SOLMI*) presented in Chapter 4, can also be extended to other types of vulnerabilities by defining and adopting their specific attack grammars.

Our proposed search-based testing strategy presented in Chapters 5 and 6, can further be improved by integrating a multitarget approach. Our current approach uses a standard genetic algorithm to evolve test cases for one single target at one time, i.e., Test Objective (TO) in our context. Therefore, genetic algorithms (or in general search algorithm) are re-launched multiple times, once for each TO. However, larger web applications have many

input forms that could be exploited by attackers. As a consequence, the number of potential TOs to assess can be very large. Executing the genetic algorithm for each TO separately may affect the overall efficiency. Since the TOs are based on common attack patterns for *XMLi*, the test cases evolved during the search for one TO are likely good (close to cover) for other TOs as well. Therefore, instead of starting the search from completely random inputs for each TO, an approach that can evolve test cases for multiple TOs at the same time, is expected to be efficient. To this aim, a multi-objective optimization algorithm, e.g., NSGAI, can be tailored to optimize in parallel multiple TOs.

Finally, our search-based testing approach can be improved by considering alternative fitness functions instead of the string edit distance (or its real-coded variant). Indeed, the string edit distance is very expensive to compute since its time complexity is  $\mathcal{O}(n*m)$  (where  $n$  and  $m$  are the lengths of the two strings). Since most of the execution time is spent in the fitness calculations, a single-target search-based approach may face challenges in covering the target TO, especially when working under strict time constraints. Therefore, alternative, less precise fitness functions with linear time complexity  $\mathcal{O}(n)$  (where  $n$  is the length of the longest string), can be investigated.

# Bibliography

- [1] Web application vulnerability statistics of 2012. <http://www.ivizsecurity.com/Web-Application-Vulnerability-Statistics-of-2012.html>. Accessed: 2014-08-06. 1
- [2] T. Berry and N. Niv. Web Application Attack Report. [https://www.imperva.com/docs/HII\\_Web\\_Application\\_Attack\\_Report\\_Ed1.pdf](https://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed1.pdf). Accessed: 2014-07-05. 1
- [3] Whitehat website security statistics report 12. <https://www.whitehatsec.com/resource/stats.html>. Accessed: 2015-04-03. 1
- [4] XML Vulnerabilities Introduction. <http://resources.infosecinstitute.com/xml-vulnerabilities/>. Accessed: 2014-06-22. 3, 16
- [5] Abhinav Nath Gupta and Dr. P. Santhi Thilagam. Attacks on web services need to secure xml on web. *Computer Science and Engineering, An International Journal (CSEIJ)*, 3(5), 2013. 3, 16, 30, 54
- [6] S. Jan, C.D. Nguyen, and L. Briand. Known xml vulnerabilities are still a threat to popular parsers and open source systems. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 233–241, Aug 2015. 3, 6
- [7] Sadeeq Jan, Cu D. Nguyen, and Lionel Briand. Automated and Effective Testing of Web Services for XML Injection Attacks. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA)*, Jul 2016. 3, 6, 80
- [8] A Falkenberg, C. Mainka, J. Somorovsky, and J. Schwenk. A new approach towards dos penetration testing on web services. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 491–498, June 2013. 3, 16, 31, 54

- [9] Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. Chapter one-security testing: a survey. *Advances in Computers*, 101:1–51, 2016. 3, 79
- [10] Paul Adamczyk, Patrick H. Smith, Ralph E. Johnson, and Munawar Hafiz. *REST: From Research to Practice*, chapter REST and Web Services: In Theory and in Practice, pages 35–57. Springer New York, New York, NY, 2011. 4, 58
- [11] Simple Object Access Protocol (SOAP). <https://www.w3.org/TR/soap/>. Accessed: 2016-04-26. 4, 58
- [12] SmartBear ReadyAPI. <http://smartbear.com/product/ready-api/overview/>. Accessed: 2015-11-18. 4, 5, 33, 44, 47, 58
- [13] WS FUZZER Tool. [https://www.owasp.org/index.php/Category:OWASP\\_WSFuzzer\\_Project](https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project). Accessed: 2015-11-16. 4, 5, 33, 44, 55, 58, 80
- [14] Randy L Haupt and Sue Ellen Haupt. *Practical genetic algorithms*. John Wiley & Sons, 2004. 5, 73, 103
- [15] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Survey*, 45(1):11:1–11:61, December 2012. 6, 67, 83
- [16] Sadeeq Jan, Cu D. Nguyen, Andrea Arcuri, and Lionel Briand. A Search-based Testing Approach for XML Injection Vulnerabilities in Web Applications. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*, March 2017. 6, 97, 125
- [17] W3C XML Standard. <http://www.w3.org/TR/REC-xml/>. Accessed: 2014-07-01. 9, 10
- [18] XML Terminology. <http://en.wikipedia.org/wiki/XML/>. Accessed: 2014-02-20. 9
- [19] XML External Entity Injection. <http://securityhorror.blogspot.com/2012/03/what-is-xxe-attacks.html>. Accessed: 2014-06-27. 11
- [20] OWASP. <https://www.owasp.org/index.php>. Accessed: 2016-11-1. 12, 55
- [21] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE. 23, 32

- [22] Elisa Bertino, Lorenzo Martino, Federica Paci, and Anna Squicciarini. *Security for Web Services and Service Oriented Architectures*. Springer, 2010. 30, 54
- [23] Jinfu Chen, Qing Li, Chengying Mao, Dave Towey, Yongzhao Zhan, and Huanhuan Wang. A web services vulnerability testing approach based on combinatorial mutation and soap message mutation. *Service Oriented Computing and Applications*, 8:1–13, 2014. 30, 54, 79
- [24] Yuri Demchenko, Leon Gommans, Cees de Laat, and Bas Oudenaarde. Web services and grid security vulnerabilities and threats analysis and model. In *The 6th IEEE/ACM International Workshop on Grid Computing*. IEEE, 2005. 30, 54
- [25] Meiko Jensen, Nils Gruschka, and Ralph Herkenhner. A survey of attacks on web services. *Computer Science - Research and Development*, 24(4):185–197, 2009. 30
- [26] Vipul Patel, Radhesh Mohandas, and Alwyn R. Pais. Attacks on web services and mitigation schemes. In *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, pages 1–6, July 2010. 30, 54
- [27] S. Tiwari and P. Singh. Survey of potential attacks on web services and web service compositions. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, volume 2, pages 47–51, April 2011. 30, 54
- [28] Steve Orrin. The soa/xml threat model and new xml/so/web 2.0 attacks and threats. In *DEFCON 15*, 2007. 30
- [29] Michael R. Brenner and Musa R. Unmehopa. Service-oriented architecture and web services penetration in next-generation networks. *Bell Labs Technical Journal*, 12(2):147–159, 2007. 31
- [30] N. Antunes and M. Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, pages 301–306, Nov 2009. 31
- [31] C. Mainka, J. Somorovsky, and J. Schwenk. Penetration testing tool for web services security. In *Services (SERVICES), 2012 IEEE Eighth World Congress on*, pages 163–170, June 2012. 31, 79

- [32] Wang Chunlei, Liu Li, and Liu Qiang. Automatic fuzz testing of web service vulnerability. In *Information and Communications Technologies (ICT 2014), 2014 International Conference on*, pages 1–6, May 2014. 31, 55
- [33] R.A Oliveira, Nuno Laranjeiro, and Marco Vieira. Wsfaggessor: An extensible web service framework attacking tool. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, MIDDLEWARE '12, pages 2:1–2:6. ACM, 2012. 31, 79
- [34] Wang Chunlei, Liu Li, and Liu Qiang. Automatic fuzz testing of web service vulnerability. In *Information and Communications Technologies (ICT 2014), 2014 International Conference on*, pages 1–6, May 2014. 31, 79
- [35] S. Suriadi, A Clark, and D. Schmidt. Validating denial of service vulnerabilities in web services. In *Network and System Security (NSS), 2010 4th International Conference on*, pages 175–182, Sept 2010. 31, 54
- [36] R. Chang, Guofei Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, pages 186–199, July 2009. 31
- [37] S. Padmanabhuni, V. Singh, K.M. Senthil Kumar, and A Chatterjee. Preventing service oriented denial of service (presodos): A proposed approach. In *Web Services, 2006. ICWS '06. International Conference on*, pages 577–584, Sept 2006. 31
- [38] Xinfeng Ye. Countering ddos and xdos attacks against web services. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, volume 1, pages 346–352, Dec 2008. 31, 54
- [39] Josip Bozic, Dimitris E Simos, and Franz Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test*, pages 1–7. ACM, 2014. 42
- [40] D. Appelt, C.D. Nguyen, and L. Briand. Behind an application firewall, are we safe from sql injection attacks? In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015. 42, 54, 79
- [41] Clark Barrett, ChristopherL. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011. 42

- [42] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM. 42
- [43] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM. 42
- [44] Adam Kiezun. *Effective software testing with a string-constraint solver*. PhD thesis, Massachusetts Institute of Technology, June 2009. 42
- [45] Nikolas Havrikov, Matthias Hörschele, Juan Pablo Galeotti, and Andreas Zeller. Xml-mate: Evolutionary xml test generation. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 719–722, New York, NY, USA, 2014. ACM. 44, 56
- [46] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Ws-taxi: A wsdl-based testing tool for web services. In *ICST*, pages 326–335, 2009. 44, 56
- [47] SoapUI. <http://www.soapui.org/>. Accessed: 2015-11-18. 47
- [48] A. Avancini and M. Ceccato. Circe: A grammar-based oracle for testing cross-site scripting in web applications. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 262–271, Oct 2013. 53
- [49] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566–571, June 2009. 54
- [50] Hong Zhu and Yufeng Zhang. Collaborative testing of web services. *Services Computing, IEEE Transactions on*, 5(1):116–130, Jan 2012. 54
- [51] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 43–49, New York, NY, USA, 2010. ACM. 54



- [52] Xiang Fu and Kai Qian. Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '08, pages 34–39, New York, NY, USA, 2008. ACM. 54
- [53] A. Kieyzun, P.J. Guo, K. Jayaraman, and M.D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199–209, May 2009. 54
- [54] Ben Smith, Laurie Williams, and Andrew Austin. Idea: Using system level testing for revealing sql injection-related error message information leaks. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 192–200. Springer Berlin Heidelberg, 2010. 54
- [55] Abdulbaki Aydin, Muath Alkhalaf, and Tevfik Bultan. Automated test generation from vulnerability signatures. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 193–202, Washington, DC, USA, 2014. IEEE Computer Society. 54
- [56] T.M. Rosa, A.O. Santin, and A. Malucelli. Mitigating xml injection 0-day attacks through strategy-based detection systems. *Security Privacy, IEEE*, 11(4):46–53, July 2013. 55, 79
- [57] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, September 2004. 56
- [58] Wuzhi Xu, J. Offutt, and J. Luo. Testing web services by xml perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp.–266, Nov 2005. 56
- [59] James Ransome and Anmol Misra. *Core Software Security: Security at the Source*. CRC Press, 2013. 58
- [60] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder K Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010. 60
- [61] David E Goldberg et al. *Genetic algorithms in search optimization and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989. 63

- [62] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at <https://cs.gmu.edu/~sean/book/metaheuristics/>. 64, 67, 73, 90, 91
- [63] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991. 64, 90
- [64] Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu. Comparison of performance between different selection strategies on simple genetic algorithms. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 2, pages 1115–1121. IEEE, 2005. 64, 90
- [65] William J Masek and Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980. 65
- [66] Juan J. Durillo and Antonio J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011. 66, 73
- [67] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14(2):105–156, June 2004. 73, 80
- [68] Helen G. Cobb and John J. Grefenstette. Genetic algorithms for tracking changing environments. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 523–530, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. 73, 104
- [69] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170, 2006. 73, 103, 104
- [70] Magical code injection rainbow (mcir). <https://github.com/SpiderLabs/MCIR/>. Accessed: 2016-04-26. 76, 95
- [71] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*, pages 199–209, May 2009. 79
- [72] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, May 2010. 79

- [73] T.P. Gallagher. Automated detection of cross site scripting vulnerabilities, March 2008. US Patent 7,343,626. 79
- [74] M. Junjin. An approach for sql injection vulnerability detection. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1411–1414, April 2009. 79
- [75] OWASP XML Injection Testing. [https://www.owasp.org/index.php/Testing\\_for\\_XML\\_Injection\\_\(OWASP-DV-008\)](https://www.owasp.org/index.php/Testing_for_XML_Injection_(OWASP-DV-008)). Accessed: 2014-06-25. 79
- [76] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014. 80
- [77] Mark Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering, FOSE '07*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society. 80
- [78] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, March 2010. 80
- [79] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, pages 73–83, New York, NY, USA, 2007. ACM. 80
- [80] Wasif Afzal, Richard Torkar, and Robert Feldt. A Systematic Review of Search-based Testing for Non-functional System Properties. *Information and Software Technology*, 51(6):957–976, June 2009. 80
- [81] Sven Törpe. Search-Based Application Security Testing: Towards a Structured Search Space. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 198–201, March 2011. 80
- [82] Andrea Avancini and M. Ceccato. Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 85–94, Sept 2011. 80

- [83] Julian Thomé, Alessandra Gorla, and Andreas Zeller. Search-based Security Testing of Web Applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST 2014, pages 5–14, New York, NY, USA, 2014. ACM. 80
- [84] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting Buffer Overflow via Automatic Test Input Data Generation. *Computers & Operations Research*, 35(10):3125–3143, October 2008. 81
- [85] S. Rawat and L. Mounier. Offset-Aware Mutation Based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 531–533, March 2011. 81
- [86] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. *Softw. Test. Verif. Reliab.*, 16(3):175–203, September 2006. 84, 85, 97, 98
- [87] Leigh Metcalf and William Casey. *Cybersecurity and Applied Mathematics*. Syngress, 2016. 85
- [88] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM. 89
- [89] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. 89
- [90] Francisco Herrera, Manuel Lozano, and Jose L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial intelligence review*, 12(4):265–319, 1998. 91, 92
- [91] Ashish Ghosh and Shigeyoshi Tsutsui. *Advances in evolutionary computing: theory and applications*. Springer Science & Business Media, 2012. 92
- [92] Stjepan Picek, Domagoj Jakobovic, and Marin Golub. On the recombination operator in the real-coded genetic algorithms. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 3103–3110. IEEE, 2013. 92

- [93] Kalyanmoy Deb and Debayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1):1–28, 2014. 92, 93
- [94] Peter Sprent. *Fisher Exact Test*, pages 524–525. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 101
- [95] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005. 101
- [96] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, third edition, "1998". 101
- [97] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. 103
- [98] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617, 2008. 103, 112
- [99] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936. 103, 117
- [100] W. J. Conover and Ronald L. Iman. Rank transformations as a bridge between parametric and nonparametric statistics. *The American Statistician*, 35(3):124–129, 1981. 103, 113
- [101] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979. 103
- [102] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Ann Arbor, MI, USA, 1975. AAI7609381. 103
- [103] J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 16(1):122–128, January 1986. 103
- [104] J David Schaffer, Richard A Caruana, Larry J Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 51–60. Morgan Kaufmann Publishers Inc., 1989. 103

- [105] Jim E Smith and Terence C Fogarty. Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm. In *Parallel Problem Solving from Nature PPSN IV*, pages 441–450. Springer, 1996. 103
- [106] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013. 104
- [107] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 2017. To appear. 104
- [108] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015. 104
- [109] Rose D Baker. Modern permutation test software. *Randomization Tests, chapter Appendix*. Marcel Decker, 1995. 117