2013

Model checking techniques for vulnerability analysis of Web applications

Michelle Elaine Ruse
Iowa State University

Recommended Citation

Ruse, Michelle Elaine, "Model checking techniques for vulnerability analysis of Web applications" (2013). *Graduate Theses and Dissertations*. 13211.
https://lib.dr.iastate.edu/etd/13211

Model checking techniques for vulnerability analysis of Web applications

by

Michelle Elaine Ruse

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Samik Basu, Major Professor

David Fernández-Baca

Arka P. Ghosh

Robyn Lutz

Hridesh Rajan

Iowa State University

Ames, Iowa

2013

DEDICATION

I would like to dedicate this dissertation to my amazing daughter Morgana who was there to encourage me to move forward through every obstacle with her hugs and wisdom beyond her years.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGEMENTS viii

ABSTRACT ix

CHAPTER 1. INTRODUCTION 1

1.1 First Order SQL Injection Attacks 3

1.2 First Order Cross-Site Scripting Attacks 6

1.3 Contributions 8

1.4 Organization 8

CHAPTER 2. CLASSIFICATION OF FIRST ORDER RELATED WORKS 9

2.1 Introduction 9

2.2 Classification of First Order Vulnerability and Attack detection methods 12

2.2.1 Detection Type 12

2.2.2 Detection Method 14

2.2.3 Granularity 15

2.2.4 Location 16

2.2.5 Level of Automation 16

2.2.6 Test Case Source 17

2.3 Classifications of related works 18

2.3.1 Testing 19

2.3.2 Program analysis 23

2.3.3 Model checking 26

2.3.4 Code re-write 29

2.3.5 Structural matching 32

2.3.6 Taint analysis 36

2.3.7 Proxy 37

2.3.8 Browser-based 39

2.3.9 Penetration testing 41

2.3.10 Blackbox testing 42

2.3.11 Other techniques 43

2.4 Summary 44

2.4.1 Classifications 44

2.4.2 Techniques 46

2.4.3 Conclusions 48

CHAPTER 3. ANALYSIS & DETECTION OF SQL INJECTION VULNERABILITIES VIA AUTOMATIC TEST CASE GENERATION OF PROGRAMS 50

3.1 Introduction 51

3.2 A method for detecting SQL injection vulnerabilities 53

3.2.1 Translating SQL query conditions to C-programs 54

3.2.2 Application of CREST 59

3.2.3 Causal set detection: reductions 60

3.3 Method evaluation 64

3.4 Conclusions 65

CHAPTER 4. DETECTING CROSS-SITE SCRIPTING VULNERABILITY USING CONCOLIC TESTING 66

4.1 Introduction to Cross-Site Scripting 67

4.2 A method for detecting Cross-Site Scripting vulnerabilities and implementing attack prevention 69

4.2.1 Preprocessing 71

4.2.2 Translation 72

4.2.3 Testing for determining vulnerable outputs 75

4.2.4 Instrumentation for detecting Cross-Site Scripting attacks 77

4.3 Case Studies 77

4.4 Conclusions 79

CHAPTER 5. CONCLUSIONS AND FUTURE WORK 80

5.1 Summary and contributions 80

5.2 Extension to Second Order Injection Attacks 82

BIBLIOGRAPHY 87

LIST OF TABLES

Table 1.1 Acronym definitions 1

Table 2.1 First Order SQLID Works by Technique 19

Table 2.2 First Order XSSD Works by Technique 20

Table 2.3 Summary of classified First Order SQLID countermeasures by year . . 21

Table 4.1 GotoCode Projects Tested 78

Table 4.2 Online Bookstore Variables 79

LIST OF FIGURES

Figure 1.1 Example query with dependent sub-query 5

Figure 1.2 Facebook's XSS vulnerability of 2008 7

Figure 2.1 Classification of Injection Attack Detection methods 12

Figure 3.1 SQL query with nested sub-query 55

Figure 3.2 (a) Possible execution tree of TRANSLATE; (b) Result of translation; (c) Partial execution graph explored by CREST. 58

Figure 3.3 Requirements 61

Figure 3.4 3-valued Decision Tree 62

Figure 3.5 3-valued Decision Diagram 62

Figure 3.6 Rules for (a) redundant tree removal; (b) generalization of test values; (c) removal of duplicate test values 63

Figure 3.7 SQL query with UNION 65

Figure 4.1 Approach overview 69

Figure 4.2 Illustrative example JSP code: welcomePage.jsp 70

Figure 4.3 Mapping for JSP to Java translation 71

Figure 4.4 Grammar for adapting Java String to char arrays 73

Figure 4.5 Illustrative example code converted to Java: welcomePage.java 74

Figure 4.6 Finite state automaton representing vulnerable output 75

Figure 5.1 (a) Direct Single Injection Attack; (b) Direct Multiple Injection Attacks; (c) Indirect Injection Attacks 83

Figure 5.2 Second Order SQL Injection Vulnerability Detection 85

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation. First and foremost, I am grateful to Dr. Samik Basu for his guidance, immense patience, and support throughout the research and the writing of this dissertation. I would also like to thank my committee members for their efforts and contributions to this work: Dr. David Fernández-Baca, Dr. Arka P. Ghosh, Dr. Rajan Hridesh, and Dr. Robyn Lutz. I especially want to thank Dr. Robyn Lutz for her wisdom, guidance and encouragement that helped me realize my setbacks were not insurmountable. Special thanks to my devoted lab-mates, Tanmoy Sarkar, Zach Oster (Benevolent Overlord of LaTeX), and Youssef Hanna for their help and encouragement and my wonderful professors during my course work, Dr. Pavan Aduri, Dr. Samik Basu, Dr. Oliver Eulenstein, Dr. David Fernández-Baca, Dr. Shashi Gadia, Dr. Vasant Honavar, Dr. Yan-Bin Jia, Dr. Leslie Miller, Dr. Hridesh Rajan, Dr. Gurpur Prabhu, and Dr. Srinivas Aluru. Big thanks to Linda Dutton, the best graduate secretary even during retirement and a dear friend.

I would like to thank my Grams who has been there for me my entire life, my family, friends, Grand View University family and Luther Memorial Church family for their patience and understanding during the writing of this work. “Are you done yet?” is my least favorite phrase, but I know it was asked it out of concern and love. A special thanks to D.L., for reminding me to take care of myself and patiently helping through the final stretch. Finally, I am thankful for my Divine Creator, in whom all things are possible.

ABSTRACT

Injection Attacks exploit vulnerabilities of Web pages by inserting and executing malicious code (e.g., database query, Javascript functions) in unsuspecting users' computing environment or on a Web server. Such attacks compromise users' information and system resources, and pose a serious threat to personal and business assets. Methods have been devised to counter attacks and/or detect vulnerabilities to injection attacks in queries and/or in application source code. We define a classification for these query and application level methods and use this to classify a representative body of works that address injection attacks. We investigate and develop a framework where queries and vulnerable fragments of applications (written in query and application languages) are identified and analyzed offline (statically), and at runtime the vulnerable fragments are monitored to detect possible injection attacks. At its core, our framework leverages model checking, program analysis and concolic testing. Results show the effectiveness of our framework compared to the existing ones in three dimensions: first, our framework can detect vulnerabilities that go undetected when existing methods are used; second, our framework makes offline analysis of applications time efficient; and finally, our framework reduces the runtime monitoring overhead by focusing only on query conditions and application fragments that are vulnerable to injection attacks.

CHAPTER 1. INTRODUCTION

Table 1.1 Acronym definitions

SQLI	SQL Injection	XSS	Cross-Site Scripting
SQLIV	SQL Injection Vulnerability	XSSV	Cross-Site Scripting Vulnerability
SQLIA	SQL Injection Attack	XSSA	Cross-Site Scripting Attack
SQLID	SQLIV and SQLIA Detection	XSSD	XSSV and XSSA Detection
IA	Injection Attacks	FOID	First Order Injection Detection

Security is a prevalent concern to businesses with a Web presence and to everyday users. The Web serves as a convenient portal between end-users and company resources such as user accounts. A Web application is a computer program that allows end-users to interact with a Web server by sending and receiving data, possibly accessing a database or other system resources, via a browser. Such applications that facilitate a rich online experience (personal financing, social networking, business transactions, et cetera) are burdened with securing the privacy of sensitive data while permitting data exchange. A Web application accepting input is vulnerable to the class of attacks known as injection attacks, which include SQL Injection and Cross-Site Scripting attacks.

Injection attacks top the *OWASP Top 10 - 2010: The Top Ten Most Critical Web Application Security Risks* list published by the Open Web Application Security Project (2010 is its latest version) [1]. They are classified into two categories: reflected and stored, (also called immediate and persistent, respectively, and First Order and Second Order Attacks, respectively). Reflected attacks are immediate and maliciously disclose useful information resulting directly from the injection within a Web browsing session. Stored attacks can have a broader victim base. They can be saved by an attacker during a session, then persist in the datastore to be later retrieved by any number of unsuspecting users during their respective Web browsing sessions. In this thesis, we primarily focus on reflected, or First Order, injection attacks, and specifically SQL Injection and Cross-Site Scripting attacks.

Injection attacks occur when input passes from the browser to the server application, possibly onto the database and even back to the user’s browser; and this input contains malicious values/scripts that can alter the behavior of the Web application and cause unexpected results. A typical successful attack begins in the client-side browser where a Web application is rendered, giving an attacker opportunity to input malicious data via the browser. This data is then sent to the server, where it may reach the back-end database via a query, resulting in a SQL Injection attack, or it may be sent back to the attacker’s client-side browser for execution, resulting in a Cross-Site Scripting Attack. These attacks can return sensitive information or give unauthorized access to system resources immediately, as these are First Order attacks. Second Order attacks can victimize on an unsuspecting user when the previously stored malicious data is retrieved and becomes part of a query or the rendered Web page. One extreme measure to counter such attacks would be to disallow any user input to Web applications; such a measure is impractical for any Web application that interacts with end-users. A Web application without input has very limited functionality, it does not have the ability to query the user in order to display pertinent information. Thus, there is no access to back-end databases to retrieve personal information (e.g., a bank account balance) nor is there the capability to perform transactions (e.g., online bill pay). Therefore, the primary measure for countering injection attack involves identifying possible malicious inputs and altering them, thus rendering them benign. This is referred to as the *sanitization method*.

Sanitization methods can be fallible or lack the inclusiveness sufficient to thwart all attacks. For example, sanitization methods for SQL Injection Attacks may inspect input intended for inclusion in a SQL query or the query itself for the insertion of unexpected SQL keywords and remove them. The ubiquitous injection string “’ OR ‘1’=‘1’--” adds the SQL keyword “**OR**” to create a tautology-based attack (described later in this chapter). These sanitization methods will fail to detect attacks that do not inject a keyword. Cross-Site Scripting attacks occur when a script is injected and executed on a victim's browser (typically containing the keyword "script"). The goal of a sanitization method might be removal of keywords. Malicious users could discover and circumvent this removal. They could create a string that will contain "script" after an instance of "script" is removed. The injected string could flank "script" with "scr" and "ipt": "scrscriptipt". Similarly, persistent malicious users can bypass other sanitization tactics (e.g., encoding symbols commonly used in attacks). Thus, the use of sanitization methods alone will not ensure safety from injection attacks.

With universal Web-based access to sensitive information and resources, the prevalent threat of injection attacks must be acknowledged and addressed. Measures to detect and prevent attacks are a way to respond to the threat. Another is to detect code that is vulnerable and remove the vulnerability or add monitoring mechanisms prior to deployment. To better protect information and resources from this type of attack, we must understand the attacks and current countermeasures against them. Such knowledge can inform development of new methodologies to mitigate threats.

The rest of this chapter is organized as follows. Section 1.1 describes First Order SQL Injection and briefly introduces detection methods. Similarly, Section 1.2 discusses Cross-Site Scripting and its detection methods. Section 1.3 defines the contributions of this thesis. Finally, Section 1.4 details the remainder of the chapters.

1.1 First Order SQL Injection Attacks

A SQL Injection Attack occurs when malicious data is injected into a database query, via Web page input, to gain sensitive information from or unauthorized access to system resources (e.g., a database). SQL Injection Attack (SQLIA) refers to the situation when such injection occurs via a SQL query, i.e., when malicious data value(s) and/or code is input into a Web page and subsequently injected into a SQL query. SQL Injection Vulnerability (SQLIV) refers to weaknesses in the Web application source code (or the query itself) susceptible to such injections. SQLIAs occur when there are SQLIVs that are not adequately monitored in the source code. SQL Injection Detection (SQLID) can endeavor to determine SQLIV, SQLIA or both.

A simple SQL query example which takes which takes input `$name` and `$password` is the following: `SELECT * FROM Users WHERE name = $name AND password = $password`. Each input is contained in the **WHERE** clause and in a *condition* (e.g., *condition1* is “`name = $name`”). If the Web application source code allows user input from the browser to be assigned to `$name` and/or `$password` without adequate sanitization, this query can be injected with malicious code and an attack can be created. When user input leads to “`$name = ’ OR ‘1’=‘1’--`”, the first condition in the **WHERE** clause becomes “`name = ‘’ OR ‘1’=‘1’--’`”. Thus, the **WHERE** clause now contains a tautology “`name = ‘’ OR ‘1’=‘1’`”. Furthermore, the second condition in **WHERE** clause “`password = $password`” is ignored during the evaluation of the clause conditions, since it follows the injected comment symbols “`--`”. The end result is `Users` table can be accessed without proper authorization (matching name and password), since the **WHERE** clause is always true without matching name or password. This simple query with malicious string “`’ OR ‘1’=‘1’--`” injected via `$name` now becomes `SELECT * FROM Users WHERE name = ‘’ OR ‘1’=‘1’--’ AND password = ‘’`. The role of SQLID methods is to try to identify the vulnerability that allows the injection into the variable `$name` or the malicious data that causes the attack “`’ OR ‘1’=‘1’--’`”.

This is one example of the class of attacks known as SQL Injection. Methods have been proposed to detect SQL Injection attacks and vulnerabilities, with some methods removing the vulnerability or thwarting the attack. Generally categorized in the body of works addressing SQL Injection, methods are static [2, 3], dynamic [4, 5, 6, 7], or a combination of the two [8, 9, 10, 11, 12, 13]. These static code analyses typically detect vulnerabilities, not actual attacks. Attacks and/or vulnerabilities are detected by the dynamic and combination techniques. Chapter 2 contains a detailed classification of these works.

```
SELECT username, password FROM Users
WHERE  lastname = $lastname AND firstname = $firstname AND
          $status IN (SELECT statuses FROM STATUS
                      WHERE pid = $pid  OR  pname = $pname)
```

Figure 1.1 Example query with dependent sub-query

In some works structure change is used to detect vulnerabilities and/or attacks. In the simple query `SELECT * FROM Users WHERE name = $name AND password = $password`, the **WHERE** clause's *condition1* (`name = $name`) **AND** *condition2* (`password = $password`) is replaced by *condition1* (`name = ‘’`) **OR** *condition2* (`‘1’=‘1’`) and appended with a comment (“`--`”) which contains the **AND** and intended *condition2*. Techniques detecting an unexpected structure (e.g., parse trees [14, 4, 15]) will find this vulnerability/attack. Others find it with source code program analysis [2, 16, 17, 18, 11], code re-writes [9, 19, 20, 21], model checking [10, 22, 23, 24, 25, 26, 8], testing [5, 27, 13, 3, 28], and proxy use [29]. However, a query's intended structure need not be modified to create a tautology and launch a tautology-based attack.

The example query in Figure 1.1 contains a nested sub-query and is vulnerable to non-structure altering tautology-based SQL Injection. A domain-safe input (e.g., expected data-type and/or expected value range) for the variable `$status` could cause a tautology in the **WHERE** clause of the main query (as long as `$lastname` and `$firstname` do not create contradictions). Thus, the vulnerability lies in the nested subquery condition clause and its susceptibility to becoming a tautology.

We propose a solution in Chapter 3 that will find tautology-based vulnerabilities, even those that do not alter the SQL query structure, and does so without extensive source code analysis. Our method does not look solely at the syntax of the query, but also considers its semantics. It analyzes the query using concolic testing with input test generation to pinpoint vulnerable query conditions. Concolic (concrete plus symbolic) testing is a software verification technique that combines concrete values with symbolic execution, including a constraint solver to generate subsequent test cases [30].

1.2 First Order Cross-Site Scripting Attacks

According to *OWASP Top 10 - 2010: The Top Ten Most Critical Web Application Security Risks* list, Cross-Site Scripting (XSS) is listed as number two. "Cross-Site Scripting (XSS) attacks occur when: 1. Data enters a Web application through an untrusted source, most frequently a Web request. 2. The data is included in dynamic content that is sent to a Web user without being validated for malicious code" [1].

Cross-Site Scripting Attacks (XSSAs) are injected via Web page inputs or via the address bar in a URL and are executed in the browser. The crux of an XSSA is to launch a script (e.g., <`script`>`malicious script`</`script`>), or cause a victim to launch a script in the browser. Injected code for XSS contains tags (`script` or `javascript:`) and tag symbols "<" and ">", some subset of tags and symbols that concatenate with other strings to create script tags, or a script call in its entirety. A First Order XSS attack is immediately executed or triggered by an event (e.g., `mouseover`) and does not persist beyond the HTTP session in which it was injected.

Facebook, which has grown in popularity since its inception, has had publicly scrutinized XSS Vulnerabilities (XSSVs). In a 2008 attack [31], a job position script was vulnerable to the URL injection shown in Figure 1.2 that displays a user's cookie (Note: this is a seemingly innocuous attack, a user displaying his or her own cookie. However, the implications are the XSSV itself and the possibility that an attacker has comprised a user's system and is capturing the displayed cookie).

Figure 1.2(a) shows encoded symbols (e.g., %3C, %22) which are decoded by the browser, bypassing any security measure that does not check for this encoding. Figure 1.2(b) displays the decoded attack. This type of XSS, called self-XSS, occurs via social engineering ruses,

```
http://www.facebook.com/jobs/position.php?st=
%22%3E%3Ciframe%20src=http://xssed.com%3E
%3C/iframe%3E%3Cscript%3Ealert(document.cookie);%3C/script%3E
```

(a)

```
http://www.facebook.com/jobs/position.php?st=
"><iframe src=http://xssed.com>
</iframe><script>alert(document.cookie);</script>
```

(b)

Figure 1.2 Facebook's XSS vulnerability of 2008

where an attacker emails (or posts) a link that a victim clicks (or copies and pastes) launching the attack. Such was the case in a 2011 Facebook attack [32], wherein users copied and pasted a link containing malicious JavaScript which caused sharing of offensive content.

Cross-Site Scripting attacks can be detected and/or blocked in the browser or on the Web server at runtime. These attacks exploit the vulnerabilities that can be detected dynamically or statically offline, prior to application deployment. A few SQLID approaches, e.g., testing [5, 27] and model checking [26], are encompassing enough to be applied to XSS detection (XSSD). Methods applied to XSSD also include proxy [33, 34], browser policy [35, 36, 37, 38, 39], and parse tree [40, 41]. Some server-side approaches use static code analyses: testing [13], program analysis [42], code re-write [43], and model checking [22]. Chapter 2 includes a more detailed review of these and additional methods for XSSD.

In Chapter 4, we present a server-side solution to identify vulnerabilities via testing using the concolic test engine jCute [44]. Our concolic method uses an initial generated concrete value and subsequent concrete values based on symbolic constraint solving.

1.3 Contributions

1. ***Classification of First Order Injection Vulnerability and Attack Detection Methods.*** We have defined a comprehensive classification of works that address First Order SQLI and XSS. This classification defines and describes properties of First Order SQLID and XSSD methods which are then used to classify a representative body of existing works. Based on the classification of existing methods and their advantages and draw-backs, we propose preferable characteristics for these methods.

2. ***First Order SQL Injection Vulnerability Detection.*** We have proposed a solution that discovers First Order SQLIV by evaluating the query outside the code environment and by considering semantic dependencies in the query other methods fail to analyze. Our method will detect tautology-based vulnerabilities that do not alter the structure of the SQL query. It models the syntax and semantics of the query, including any subqueries, and it applies concolic testing to detect vulnerabilities.

3. ***First Order Cross-Site Scripting Vulnerability Detection.*** We have developed a framework that detects First Order XSSV and instruments source code for runtime XSSA monitoring. Our framework is also capable of identifying XSSV due to both conditional copy (of input to output) and concatenation of input and/or strings with the use of a concolic testing tool.

1.4 Organization

In Chapter 2, we present a comprehensive classification describing properties that can be attributed to techniques proposed to address First Order Injection Vulnerabilities and Attacks. We also outline various methods used in the detection of two types of Injection Attacks (IA) addressed in this thesis, SQLI and XSS. Chapter 3 proposes a technique to address vulnerabilities to SQLIAs. In Chapter 4, we propose a technique to address Web applications' vulnerabilities to XSSAs. Finally, in Chapter 5 we discuss future avenues of research, specifically focusing on how our proposed methods can be directly extended to detect and prevent Second Order IA.

CHAPTER 2. CLASSIFICATION OF FIRST ORDER RELATED WORKS

In this chapter, we define and describe a comprehensive classification for First Order Injection Attack and Vulnerability detection approaches. Furthermore, we categorize a representative body of works using this classification and organize the works by primary vulnerability and/or attack detection technique (e.g., testing, program analysis, model checking). We begin with the recorded history of the two types of First Order Injection Vulnerabilities which are the focus of this thesis, SQL Injection and Cross-Site Scripting. This chapter is organized as follows: Section 2.1 presents the documented history of SQL Injection and Cross-Site Scripting vulnerabilities, Section 2.2 defines and describes our classification, Section 2.3 classifies works addressing these First Order Injections, and finally Section 2.4 summarizes observations on the classification and methods addressing First Order SQLI and XSS.

2.1 Introduction

In *OWASP Top 10 - 2010: The Top Ten Most Critical Web Application Security Risks* list Injection attacks are most critical. Even prominent Web sites with resources to ensure the implementation of security measures are not immune to this threat [1]. In January 2012, Amazon's Zappos.com fell victim to a data breach exposing 24 million customers' private information after attackers exploited an application vulnerable to SQL Injection [45]. On July 12, 2012, Yahoo urged users to change their passwords immediately, after their subdomain Yahoo Voices was the target of a successful SQL Injection that revealed 453,492 unencrypted Yahoo account passwords, over 2,700 database table and field names and 298 MySQL variables [46]. Similarly, password leaks affected 6.5 million users at the professional networking site LinkedIn [47], an undisclosed percent of the 40 million users at the social music site last.fm [48], 1.5 million at the dating site eHarmony [49] and 420,000 at the social networking site formspring [50].

The first documented SQL Injection Vulnerability was from US-CERT/NIST (United States Computer Emergency Readiness Team/National Institute of Standards and Technology) and was released to the NVD (National Vulnerability Database) [51] in 2001, CVE-20011-1460 [52]. This vulnerability allowed bypassing user authentication in the user parameter in `article.php` of PostNuke versions 0.62-0.64. Soon after, several techniques were proposed addressing this security flaw. SQL injection attacks persist and new more complex attacks emerge. Countermeasures are continually implemented to combat this threat.

In 1999, the first documented Cross-Site Scripting vulnerabilities recorded by US-CERT/ NIST from the NVD, CVE-1999-1357, applied to various UNIX operating system and the Netscape browser, versions 4.04 through 4.7 [53]. In this vulnerability, the character "0x8b" is converted to the *less-than* symbol ("<"), and the character "0x9b" character is converted to the *greater-than* symbol (">"), allowing script injection in CGI programs[1] that do not filter (i.e., sanitize) these characters. The NVD reported a few documented vulnerabilities from 1999 until 2001, when numerous vulnerabilities were reported. Among the 2001 vulnerabilities in the US-CERT Vulnerability Notes Database are the following: "Apache Tomcat vulnerable to Cross-Site Scripting via passing of user input directly to default error page" [54], "Lotus Domino Server R5 vulnerable to Cross-Site Scripting via passing of user input directly to default error page" [55], and "IBM WebSphere vulnerable to Cross-Site Scripting via passing of user input directly to default error page" [56]. Significant research efforts to understand XSS on a theoretical level began a few years later.

Of course, Web site administrators took immediate measures and hacked simplistic defenses; however, the research community took on a more holistic approach to the problem. The onslaught of attacks was met with an onslaught of attack and vulnerability detection and pre-

[1] CGI programs, usually written in a scripting language, are treated today as part Web application content and not generally distinguished from such applications.

vention techniques. In general, vulnerability prevention comes in Web application development and post-deployment patching of source code; attack prevention occurs at run-time, interceding the attack.

We formulate our own techniques for SQLID and XSSD in Chapters 3 and 4, respectively, but first we wish to understand the contributions and impact of existing techniques that detect First Order Injection vulnerabilities and attacks. This understanding begins with a structured classification that provides a representative overview of SQLID and XSSD techniques to provide a consistent, comprehensive characterization of existing solutions.

The works describing or deriving SQLID and/or XSSD methods do not typically use the same characteristics for evaluation or comparison. Our aim is to provide a consistent classification and review of works to help guide research efforts in addressing First Order Injection Detection (FOID). FOID can be further classified as attack detection or as vulnerability detection. Methods that aim to detect actual attacks, address SQLIA and XSSA, while methods that aim to find vulnerable queries and/or code address SQLIV and XSSV.

Contributions With this classification and review of a representative body of FOID methods, our aim is

- ***to give a road map of existing FOID research.*** This chapter gives an overview of a representative body of First Order SQLID and XSSD works.
- ***to provide an evaluation and comparison guide for reviewing existing and proposed FOID methods.*** We use our road map of works to compare known techniques for SQLID and XSSD. This provides insight into characteristic combinations that are possible and preferable for future SQLID and XSSD approaches.
- ***to provide a foundation for coordinating future research on FOID.*** Other researchers have contributed comparisons and surveys for SQLID [57, 58, 59, 60], and surveys for XSSD [57, 61]; however, to the best of our knowledge, this is the first com-

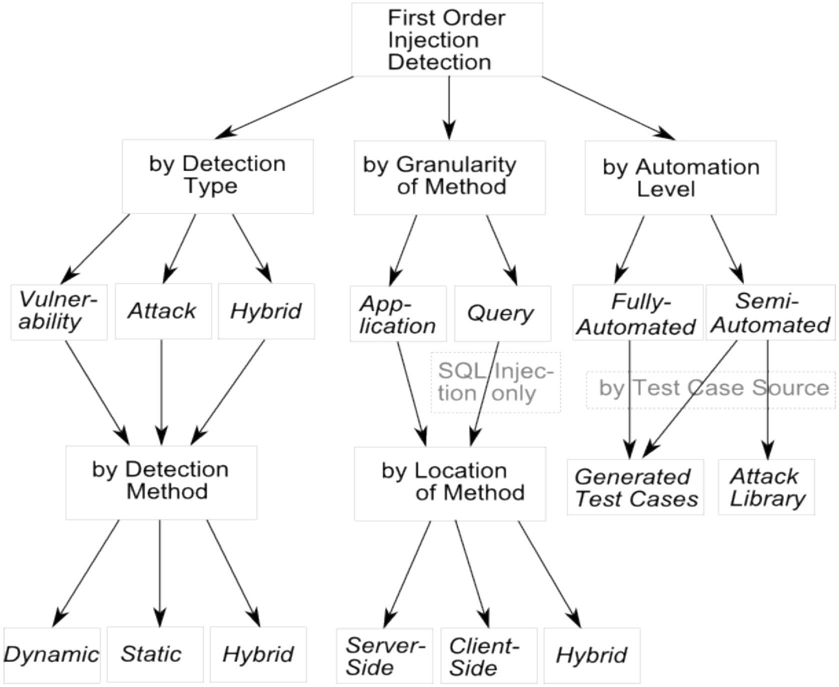


Figure 2.1 Classification of Injection Attack Detection methods

prehensive classification of methods detecting First Order SQLIA, SQLIV, XSSA and XSSV.

2.2 Classification of First Order Vulnerability and Attack detection methods

In this section, we detail characterizations of techniques that address SQLI and XSS vulnerabilities and attacks. We define *Detection Type*, *Detection Method*, *Granularity*, *Location of Method*, *Automation Level* and *Test Case Source*. Figure 2.1 outlines our classification of categories and some category inter-dependencies relevant to current and future research in First Order Injection Vulnerability and Attack Detection. The remainder of this section defines each of these categories.

2.2.1 Detection Type

The *Detection Type* indicates the method's primary goal: vulnerability detection, attack detection, or a hybrid of the two. With vulnerability detection, the overall objective is to inform the developer and/or site administrator of vulnerabilities, to modify the source code by removing vulnerable sections and/or by adding monitors, or both. Similarly, attack detection may inform of an attack, prevent the attack completely by blocking it, prevent the attack by modifying the attack-containing code, or some combination thereof. Hybrid method objectives can include combination of subsets of vulnerability and attack objectives. Although a basic classification, detection type gives insight into different technologies, which have been applied to date and which should persist in future research, based on desired objectives and given limitations.

- ***Injection Vulnerability Detection***: An injection attack vulnerability is the query or code segment(s) susceptible to injection attacks. *Vulnerability* detection methods determine the presence of susceptibilities in an application or in a particular query. Some methods specifically pinpoint potentially offensive hotspots (application code locales) [8, 28]. Most often vulnerability analysis occurs off-line; however, in [5, 62, 63], the authors have presented runtime vulnerability analysis methods. Vulnerability detection finds code weaknesses as early as the design and development phase of an application. Once the vulnerability is discovered, the method's final step may be to modify the source code to remove the vulnerability, to add checks at the hotspots, or to inform about the vulnerability in a log, report or other output.

- ***Injection Attack Detection***: SQLIA results from a malicious query and XSSA results from a malicious script to allow unauthorized access to resources (stored information or system resources). Attacks are the result of malicious user input(s) used in a query or injected into code to be rendered on the browser. To help secure Web sites, developers can include user input sanitization, but malicious query and script formation may still occur (e.g., user input follows the application's validation tests but is unsafe, user input bypasses sanitizing measures, sanitizing measures are inadequate, singularly benign inputs and/or strings are concatenated to create a malicious value). SQLIA and XSSA must be discovered at runtime. Most methods aim to prevent the attack, either by stopping it or by replacing the attack with benign code. *Attack* detection methods can inform by notifying the administrator via server-side log files, or other means, that an attack has occurred.

- ***Hybrid Vulnerability/Attack Detection***: *Hybrid* vulnerability and attack detection combines the discovery of both vulnerabilities and attacks. These combination techniques often contain various phases, including vulnerability detection proceeded by attack detection, both defined above. The overall goal of a hybrid method can be two-fold, one goal based on the vulnerability detection phase and another based on the attack detection phase. During the vulnerability phase, the aim is to find the code weaknesses, followed by either code patching or code modification at the weaknesses. During the attack phase, the aim is to detect the actual attack, followed by blocking the attack, making the attack benign, and/or reporting the attack.

2.2.2 Detection Method

The method describes the analysis performed on the Web application for both SQLID and XSSD or on the query alone for SQLID. Analysis can be static, dynamic or a combination of the two.

- ***Static Methods***: Static methods generally perform vulnerability detection, since vulnerabilities, unlike attacks, can be discovered off-line, prior to deployment. Overhead that would be incurred by a static method is often prohibitive for runtime deployment. As a result static methods are used in the pre-deployment testing phase. Such testing gives developers and/or administrators the opportunity to address vulnerabilities before they can be exploited. A disadvantage to static techniques is the need for access to the source code, which may not be available if the application being tested for vulnerability is developed by a third-party.

- ***Dynamic Methods***: Dynamic methods are typically applied to attack and hybrid vulnerability/attack detection types to find runtime attacks; however, some vulnerability detection methods use dynamic methods [5, 64, 62]. These method types can require source code access, execution of the source code, execution of a created test code, or simulated runs of the application. Furthermore, dynamic runtime techniques can add overhead and may impede the user experience.

- ***Hybrid Static/Dynamic Methods***: Some methods combine static and dynamic analysis, thus placing them in this hybrid classification. In this hybrid methodology, static and dynamic analysis will most often occur in different phases. Generally, the static analysis phase serves to discover vulnerabilities, and the dynamic phase attacks that exploit those vulnerabilities. This need not be the case, as static analysis is not exclusive to vulnerability detection and dynamic analysis is not exclusive to attack detection.

2.2.3 Granularity

Granularity of method refers to the portion of the Web application required for detection. Some SQLID methods require the application (in part or in its entirety) while others require only the query to perform the vulnerability or attack detection. XSSD techniques are application level, as are many SQLID techniques which follows from the fact that many SQLID methods use off-line program analysis (both static and dynamic), testing, and static analyses methods.

- ***Application level***: SQLID and XSSD methods that employ application level analysis tools (e.g., scanners, program analysis, model checking) have *application level* granularity. Another common theme among application level granularity methods includes static source code preprocessing to discover hotspots (code locales wherein vulnerabilities might occur, e.g. input and/or output variables), thus we see application level static vulnerability detection. Similarly, dynamic and hybrid methods can require application execution and thus will have application level granularity.

- ***Query level***: SQLID methods having *query level* granularity require only the query for SQLI detection phase. Query level methods can incorporate a proxy or other middleware between the application and the database management system (DBMS) and can reside on the application server, proxy server, or database server. Although a query level method's detection phase may not require the entire source code, it may require preprocessing part of the source code to extract the query or query structure, such as parse tree methods [15, 65]. Other query level SQLID methods may intercept a query between the application and database for testing (on the application server, on the database server, or on an intermediary proxy server) [29, 64, 28, 66].

2.2.4 Location

Location refers to where the detection tool of the implemented method must reside: *Server-side*, *Client-side*, or *Hybrid*.

- ***Server-side***: Server-side methods are implemented and executed on the application server, on the database server, or on a proxy server. This means the user is not burdened with software installation or browser plug-in and subsequent updates. Server-side implementations typically perform vulnerability detection, using static or hybrid methods prior to deployment. Dynamic server-side attack detection methods may add transparent overhead to an application, thus they can be hidden from the user's browsing experience, as long as overhead is minimal.

- ***Client-side***: These methods look for vulnerabilities in the browser environment or via some tool running on the client that intercepts HTTP requests and/responses. Browser-side tools are not used to find SQLIA, as First Order attacks are immediate and the user would be the attacker. Browser side methods could include injectors, crawlers, some testing methods; however, testing is generally performed server side.

- ***Hybrid Server- and Client-side***: This describes a combination of server-side and client-side tools working in tandem. Deployment of hybrid methods requires additions to both server and client environments, and often communication and coordination between them.

2.2.5 Level of Automation

Level of automation describes how much or how little user interaction is required for detection technique implementation. Some methods require users to supply test cases, define rules or interact with the detection tool, while others require no interaction.

- ***Fully-automated***: *Fully-automated* techniques require only the source code (for application level and certain query level) or the query itself (for query level) as input. They do not require user intervention for any reason. If the user is required to, or has the option to, define rules or attacks patterns describing SQLI and XSS vulnerabilities and/or attacks, we do not classify the method as fully-automated. Fully-automated techniques that require test cases rely on automatic test case generation, not on a library or user-defined set of rules. Any detection type (vulnerability, attack, or hybrid) may be fully-automated. Server-side static methods are most commonly fully-automated, generating test cases automatically where applicable. Dynamic and hybrid methods can be fully-automated as well, as long as any necessary test case generation is automatic.

- ***Semi-automated***: *Semi-automated* techniques require some user intervention for execution. The user may be required to supply test cases, define rules, supply attack patterns or libraries, or somehow interact with the tool to discover SQLI and XSS vulnerabilities and/or attacks. A semi-automated technique may have automated test case generation but may still rely on user interaction, such as user-defined attack patterns or rules [17].

2.2.6 Test Case Source

Some methods generate test cases to try to create attack vectors, to test for vulnerable code or to discover malicious attack patterns. All method types (static, dynamic and hybrid) could utilize some form of test case generation, but not all do. Test case generation, if present, is classified as one of the following:

- ***Automated test case generation***: *Automated test case generation* does not require user intervention, the method implementation itself generates test cases. Some methods mutate the tests based on intermediate results to generate subsequent test cases (i.e., concolic testing).

- ***Attack library***: An *Attack library* acting as a blacklist contains the specific attacks, attack patterns or other specifications detailing what should be disallowed. Libraries acting as whitelists contain safe values or patterns. Libraries can be defined by the tool authors and/or the tool users (e.g., server administrators or end-users). Since, at some point, libraries require definition, methods with an attack library that implement (an) automated process(es) can only be semi-automated.

We can now classify works related to SQLI and XSS using the categories outlined above. In the next section (2.3), we present a chronological survey that classifies methods accordingly and describes their technologies.

2.3 Classifications of related works

Among early FOID works, researchers applied software testing, program analysis and model checking techniques to applications. As these techniques revealed their limitations, researchers explored other methodologies. Using our classification, we outline the progression of FOID methods and identify their characteristics. Table 2.1 and Table 2.2 summarize SQLID and XSSD works by Detection type, respectively. In Table 2.3, SQLID works are categorized by Detection type, Detection method, Granularity, and Automation level.

In Table 2.1, works addressing SQLID are classified by approach or underlying technology of the approach. This summary also illustrates the progression of techniques applied to SQLID in order from left to right: program analysis, model checking, code re-write, structural matching, taint analysis, proxies, and various testing (concolic, penetration, blackbox). In Table 2.2, works addressing XSSD are also categorized by underlying technology of the approach, including Web crawler program analysis, browser-based, proxy, model checking, concolic testing, detection system, code re-write and structural matching.

In the remainder of this section, we describe vulnerability and attack detection and/or prevention techniques and identify their characteristics according to our classification. We aim to discover favorable characteristics among the works to inform continued research in FOID.

Table 2.1 First Order SQLID Works by Technique

	Testing	Program Analysis	Model Checking	Code Re-write	Structural Matching	Taint Analysis	Pen Testing	Black-Box Testing
Vulnerability Detection	[5]^{1} [27] [13]^{19} [3]^{16} [28]	[2] [16]^{3} [17]^{5} [18]	[10] [22]^{18} [23] [24]^{8} [25]^{21}	[9]^{12} [19] [20]	[14]^{13}		[62]^{22} [63]^{23}	
Attack Detection				[21]^{4}	[15]^{9} [65]^{10} [4]^{14} [29]^{17} [64]^{20}	[7]^{11} [12]^{15}		[66]^{24}
Hybrid Vulnerability & Attack		[11]^{6}	[26]^{2} [8]^{7}					

^{1}WAVES ^{2}BMC ^{3}SQLRand ^{4}WebSSARI ^{5}bddbddb ^{6}PQL ^{7}AMNESIA ^{8}SQLUnitGen ^{9}SQLGuard ^{10}SQLCHECK ^{11}WASP ^{12}StringBorg ^{13}Sania ^{14}CANDID ^{15}SMask ^{16}SAFELI ^{17}SQL-IDS ^{18}QED ^{19}ARDILLA ^{20}SQLProb ^{21}Apollo ^{22}MySQL1Injector ^{23}v1p3r ^{24}SENTINEL

2.3.1 Testing

Testing is used in many phases of the software development cycle to find errors in a program. Not all errors can be detected with testing techniques, nor can the absence of errors be verified with testing techniques. Software testing tools may have the functionality to generate test cases; thus they provide a natural extension to Web application testing for generating test inputs and finding errors that could make applications vulnerable to Web-based attacks. Besides traditional testing, concolic testing has also been applied to Web applications for security testing. "Concolic testing automates test input generation by combining the concrete and symbolic (concolic) execution of the code under test" [30].

Among one of the first works, in 2003, Huang et al. [5] have designed *WAVES* (*W*eb *A*pplication *V*ulnerability and *E*rror *S*canner) to address both SQLIV and XSSV. Following software testing procedures, the authors have analyzed application source code to determine poor coding practices by applying fault injection, behavior monitoring, dynamic analysis and

Table 2.2 First Order XSSD Works by Technique

	Web Crawler	Program Analysis	Browser-based	Proxy/ Firewall	Model Checking	Testing	Code Re-write	Structural Matching
Vulnerability Detection	[5][1]	[42][5] [67]		[33]	[22][11]	[27] [13][7]	[20]	
Attack Detection			[37] [35] [36][2] [38] [39][9]	[34][4] [68][13]			[69] [43][6]	[70][12] [41][3] [40] [71] [72][8]
Hybrid Vulnerability & Attack					[26][10]			

[1] WAVES [2] BEEP [3] BLUEPRINT [4] Noxes [5] Pixy [6] Noncespaces [7] ARDILLA [8] XSSDS [9] E-GUARD [10] BMC [11] QED [12] XSS-Guard [13] SWAP

blackbox testing. The resulting WAVES architecture includes crawlers to determine all pages of the Web site with HTML forms, providing a blackbox, dynamic application analysis. These pages are parsed to determine input and other relevant fields. Next, injectors are used to inject attacks from a library of attack patterns as input for runs of the application. Then the behavior of the page is monitored and algorithms are used to determine if an attack was successful after the application responds to the submitted request. Any error messages that are normally sent to the user (dialog boxes, pop-ups, etc.) are suppressed and logged. WAVES provides a Web application interface, thus only source code execution is necessary. WAVES is an application-level, dynamic, server-side vulnerability and error scanner that relies on a library of injection patterns to discern vulnerabilities making it semi-automated.

Five years later, in 2008, researchers have explored more testing solutions. Fu et al. [3] have applied testing to SQLID with symbolic values in *SAFELI* (*S*tatic *A*nalysis *F*ramework for discov*E*ring sq*L* *I*njection vulnerabilities). SAFELI is a server-side, static analysis framework that performs symbolic execution on an application as follows. The application source code is instrumented for symbolic execution, at each SQL submission "hotspot" a constraint string is constructed by consulting a pre-set stored library of attack patterns (in this case regular

Table 2.3 Summary of classified First Order SQLID countermeasures by year

	Year	Type	Method	Granularity	Automation Level
WAVES[5]	2003	vuln.	dynamic	application	semi-auto
BMC[26]	2004	hybrid	hybird	application	fully-auto
WebSSARI[16]	2004	vuln.	static	application	fully-auto
Gould, et al.[2]	2004	vuln.	static	application	fully-auto
SQLrand[21]	2004	attack	hybrid	application	semi-auto
bddbddb[17]	2005	vuln.	hybrid	application	semi-auto
PQL[11]	2005	hybrid	hybrid	application	semi-auto
AMNESIA[8]	2005	hybrid	hybrid	application	fully-auto
SQLUnitGen[24]	2006	vuln.	hybrid	application	semi-auto
SQLGuard[15]	2005	attack	hybrid	query	semi-auto
SQLCHECK[65]	2006	attack	hybrid	query	semi-auto
WASP[7]	2006	attack	dynamic	application	semi-auto
StringBorg[9]	2007	vuln.	hybrid	application	semi-auto
Sania[14]	2007	vuln.	hybrid	application	semi-auto
CANDID[4]	2007	attack	dynamic	application	semi-auto
SMask[12]	2007	attack	hybrid	application	semi-auto
SAFELI[3]	2008	vuln.	static	application	semi-auto
Wassermann, et al.[27]	2008	vuln.	dynamic	application	semi-auto
SQL-IDS[29]	2008	attack	hybrid	query	semi-auto
Lam, et al.[10]	2008	vuln.	hybrid	application	semi-auto
QED[22]	2008	vuln.	hybrid	application	semi-auto
Thomas, et al.[19]	2009	vuln.	hybrid	application	semi-auto
ARDILLA[13]	2009	vuln.	hybrid	application	semi-auto
Yu, et al.[23]	2009	vuln.	static	application	semi-auto
SQLProb[64]	2009	attack	dynamic	query	semi-auto
Apollo[25]	2010	vuln.	hybrid	application	fully-auto
MySQL1- Injector[62]	2010	vuln.	dynamic	application	semi-auto
v1p3r[63]	2010	vuln.	hybrid	application	semi-auto
Ruse, et al.[28]	2010	vuln.	hybrid	query	fully-auto
Johns, et al.[20]	2010	vuln.	static	application	semi-auto
Yu, et al.[18]	2011	vuln.	static	application	semi-auto
SENTINEL[66]	2012	attack	dynamic	query	fully-auto

expressions), and finally the string constraint solver uses the constructed constraint string to generate vulnerabilities. This semi-automated approach serves to inform developers of code vulnerabilities. Earlier that same year, Wasserman et al. [27] have utilized concolic testing (concrete plus symbolic testing) in Web applications to find insecurities, both SQLIV and XSSV. They have proposed an algorithm with an automated input test generation and runtime values for dynamic code analysis and constraint solving. Their method uses SQL injection test oracles, making it semi-automated. An advantage to concolic testing is the detection of attacks resulting from the concatenation of strings that alone do not form a threat, but together do. This hybrid static and dynamic, application-level, server-side approach detects vulnerabilities.

In the following year (2009), Kieżun et al. [13] have presented an automatic technique to detect SQLIV and XSSV, both immediate and persistent (First and Second Order, respectively). They have implemented a tool, *Ardilla* that employs a concolic method that generates an example input, marks user input as taints to be tracked symbolically through the application (even into the database, where applicable), and mutates example input to create concrete exploits. Exploits are verified against a library of SQLIA patterns at the statically computed sensitive sinks (spots possibly susceptible to SQL Injection). However, the authors report that their constraint solver will under-approximate symbolic variable values. This hybrid dynamic and static method employs concolic testing, similar to the method proposed by Wassermann et al. in [27] using concolic technique for both SQLIV and XSSV detection, and is an application-level, server-side, semi-automated vulnerability detection method with automated test case generation.

In 2010, we [28] have presented a SQLIV query level tool that develops a model of the query capturing the dependencies of sub-queries. The model is analyzed with a concolic testing tool to automatically generate inputs and find conditions that make the query vulnerable, creating a *causal set* of the vulnerability. This causal set represents sets of condition values in the query that could lead to attacks. This hybrid, fully-automated method reports no false positives or false negatives when finding vulnerabilities to tautology-based attacks. It is described in detail in Chapter 3.

In conclusion, testing techniques are generally employed off-line (pre-deployment) to inform the developer of weaknesses and some even patch weaknesses in the code. However, even best practices cannot secure code from all vulnerabilities.

2.3.2 Program analysis

Applied to Web applications, program analysis can aid in finding behaviors that may make the application susceptible to Web-based attacks. Some methods applied to FOID include static analysis (which does not require code execution), dynamic analysis (which requires code execution), control and data flow analysis, pointer and alias analysis.

In 2004, Huang et al. [16] have presented an enhanced tool to address both SQLI and XSS. *WebSSARI* (*Web* application *S*ecurity by *S*tatic *A*nalysis and *R*untime *I*nspection) implements an algorithm that captures the semantics of information flow in an application. WebSSARI performs static analysis to discover vulnerable code sections and automatically inserts run-time guards in vulnerable sections. The ability to pinpoint weak spots limits the number of instrumented guards needed, minimizing added run-time overhead. However, this method looks at the symptoms of the error (the behavior resulting from the error) not the actual origin of the error (the line(s) of code responsible for the error). It inserts guard at calls to potentially vulnerable functions exhibiting these symptoms, not a the code section within the function wherein the error itself exists. This fully-automated, static, server-side, application-level vulnerability detection method aims to modify the code with sanitization methods only, it does not include the implementation of a runtime attack monitoring technique. Sanitization can be a good defense; however, in the case of inadequate sanitization or attacks that circumvent sanitization methods, a run time monitoring technique could serve as a secondary check. Gould et al. [2] have verified correctness of SQL query strings using a static program analysis method. For instance, a dynamically constructed SQL query string in an application may use a variable of datatype string in the query for an expected numeric field value in the database, an error a type system such as Java's will not deem incorrect. An example from [2] is a query with the following `SELECT` clause: `SELECT ‘$’ || (RETAIL/100) FROM INVENTORY`, where `||` indicates concatenation. Many database systems will not cast `(RETAIL/100)` to a string; thus this will result in a runtime error. Their tool verifies the correctness of query strings to find errors such as this and others. It uses a Finite State Automata (FSA) representation of the string and processes it with a modified context-free language (CFL) reachability algorithm to check SQL syntax. The reachability algorithm matches the grammar rules to the string; and strings with errors indicate possible vulnerabilities. This static vulnerability testing technique does not address attacks directly, but instead detects runtime errors that could be associated with vulnerabilities. It is application-level, server-side and fully-automated vulnerability detection with automated test case generation that addresses SQLI.

In 2005 Lam et al. [17] have developed a context-sensitive analysis tool, called *bddbddb*, based on deductive databases. Their tool stores information as relations that are accessed via *Datalog*, a logic query language used for deductive databases, and automatically translates each database query into a BDD (Binary Decision Diagram) program [73]. This program includes the BDD representation, BDD operations, database query optimizations and optimizations for BDD variable assignment. For simplicity, the authors have presented a subset of Datalog, *PQL* (*P*rogram *Q*uery *L*anguage), to define vulnerable patterns, applicable to both SQLI and XSS. They have implemented a semi-automated, server-side, application-level method. It utilizes static pointer alias analysis and dynamic query execution technique to solve user-defined PQL queries for vulnerability detection. The method has shown to have a low false positive rate and to produce no false negatives. PQL is described in detail in [11] by Martin, Livshits and Lam. PQL queries verify if queries match attacks. This hybrid static and dynamic, hybrid vulnerability and attack detection application-level method is semi-automated, requiring user-defined, attack-identifying PQL queries. This approach statically finds vulnerability matches using context-sensitive, flow-insensitive program analysis which minimizes necessary code instrumentation points. With sound static checkers, their method produces false positives but no false negatives. Finally, instrumented source code dynamically catches and mitigates PQL rule violations. In a subsequent work, Lam et al. [10] have extended PQL to allow users to declare information flow patterns. They have applied context-sensitive, flow-insensitive information flow tracking. This flow tracking finds vulnerabilities in a program and, if errors too numerous to analyze are found, then a model checking analysis automatically generates input attack vectors to reveal the vulnerability statically. Model checking is a more precise static analysis that uses a model to exhaustively check against specifications, thus it has the potential to find a complete set of attack vectors by simulating program execution on all inputs. This is used to instrument runtime monitoring into the application for dynamic attack detection. We classify this as a program analysis technique since model checking is not necessarily applied after the program analysis is. Although this application-level, server-side method has fully-automated test-case generation, it is only semi-automated due to the reliance on user-defined PQL queries, which can define both SQLIV and XSSV patterns.

In 2006, Jovanovich et al. [42] have used a dataflow analysis that is inter-procedural and context-sensitive to find XSSVs in a program. First they use dataflow analysis to find vulnerable hotspots. Data flow analysis tracks taints through the program to see if it can reach sensitive sinks (routines that send data to browser) unsanitized. This followed by alias and literal analysis for more precise results. Aliases refer to variables that share the same memory location, thus a tainted variable's aliases must also be labeled tainted. Literal analysis keeps track of variables' and constants' possible values at each program point to aid in taint analysis. The authors have implemented a system, *Pixy*, to perform the analyses with an average of one false positive per each vulnerable result. This server-side method uses static source code analysis to find XSSVs. In another work addressing XSSV, Jovanovich et al. [67] have performed static source code analysis for vulnerability detection for more precise analysis. The authors have integrated their tool [42] for this data flow analysis and enhanced their previous work with an a new alias analysis approach that specifically targets scripting language semantics (PHP). This alias analysis includes shadow values to compute relationships among all variables at each program point including local and global variables. The taint analysis with these aliases then reveals when sensitive sinks can be reached. This static, server-side methods discovers XSSVs.

In 2011, Yu et al. [18] have extended their earlier work addressing SQLI and XSS [23] which introduced two phases: vulnerability analysis (using attack patterns) and vulnerability signature generation (which is fully-automated), to include a third phase, sanitization generation. In this third phase, the framework automatically creates patches to *match-and-block* or *match-and-sanitize*. *Match statements* inserted will halt execution upon matching a vulnerability signature. *Replace statements* will replace the string matching the signature with the string after deletion of a set of characters from the input such that the string no longer matches the signature. The overall goal of this static, server-side, application-level method is to eliminate the vulnerabilities (including SQLIV and XSSV) in the application code.

Program analysis has continued to be a viable technique SQLIV and XSSV detection. In newer work, additional technologies are applied to improve upon previous programming analysis solutions. If not complemented by an attack detection, program analysis is limited to vulnerability detection. The overhead of program analysis inhibits feasibility of its use at runtime, when attack detection must occur. This application-level technique can offer precise vulnerability detection, and can be fully-automated as long as any test case generation is also automatic.

2.3.3 Model checking

Model checking is also among the first techniques applied to FOID. Model checking techniques create a model of the program or some portion of the program to verify if it meets some set of specifications using temporal logic and does so exhaustively. Web applications are modeled as an input language of a model checker, negation of specifications describing possible attack pattern or exploitation of vulnerabilities are expressed in temporal logic. Satisfaction of properties imply absence of an attack or vulnerability. If a counterexample is identified, then the counterexample provides information regarding how attack is deployed or vulnerability exploited.

In 2004, Huang et al. [26] have employed bounded model checking, *BMC*, to identify application code sections vulnerable to SQLI and XSS. Once found, the algorithm automatically patches the code with runtime guards. This is similar to the authors' previous work [16] which finds errors and instruments code. However, in this work, the counterexamples of model checking make code instrumentation more precise. Patching, in the from of input sanitization, occurs where errors are first found, not at other code locales where the affects of the errors may be found. This proposed solution is a fully-automated hybrid method and is applied at the application level on the server to statically find and modify vulnerable source code.

In 2005, Halfond and Orso have created a fully-automated model-based approach called *AMNESIA* [8], (*A*nalysis and *M*onitoring for *NE*ut-ralizing *S*QL-*I*njection *A*ttacks). AMENSIA is a hybrid method and identifies both vulnerabilities (hotpsots) and attacks for SQLI. This server-side, application-level static program analysis includes the creation of a non-deterministic finite state automata (NFSA) to build anticipated query models for the application. First the code is scanned for identification of hotspots (any code locale that sends queries to the database). Next, the anticipated query models (character-level NFSAs expressing all possible strings) are built, these models represent queries that could be built by the application. Runtime monitors are instrumented at these hotspots to check actual query strings against the anticipated query model, parsing the string query as a database according to the SQL grammar. If the model is not accepted, it is identified as a SQLIA. AMENSIA does so with no false positives; however, false negatives can occur when attack and benign query structures match an overly conservative model or have identical SQL structures. Attacks that do not alter the SQL anticipated query structure will be overlooked. The next year, AMNESIA was included as part of the *SQLUnitGen* testing tool [24]. This 2006 work by Shin et al. uses static and dynamic analysis to pinpoint vulnerable code locations by identifying how input is manipulated in the code. First, AMNESIA is used to build a query model which includes input flow information. Next, a modified version of an existing Java test case generation tool (JCrasher[1]) is implemented. Finally, the vulnerabilities are displayed in a call graph. False

[1] http://code.google.com/p/jcrasher/

negatives can occur due to insufficient attack pattern definition. Test cases are generated only for user input read from input methods and then passed as method arguments to be used in queries. SQLUnitGen has automated test case generation but is semi-automated due to source code modification required prior to use in the 2006 version. This application-level, server-side tool finds SQLIVs.

In 2008, Lam et al. [10] have presented a language called Programmable Query Language (PQL) for specifying patterns with the objective of addressing SQLI and XSS (PQL was previously described in this section with the same authors' program analysis-based methods). They find security vulnerabilities with a static context-sensitive, flow-insensitive information flow tracking technique that employs goal-directed model checking when there are numerous errors. This application-level, hybrid method automatically generates input vectors that will help reveal vulnerabilities in the code; however, it is semi-automated since it allows user-declared information flow patterns in PQL. Next, Martin and Lam [22] have presented *QED* (*Q*uery-based *E*vent *D*irector), a model checking technique that automatically generates XSS and SQLI Attacks and uses goal-directed model checking to discover vulnerabilities at the application level. Despite the fact that this method has fully-automated attack vector generation, it is semi-automated due to users' ability to supply taint-based vulnerability specifications in PQL [11].

In 2009, Yu et al. [23] have developed a string analysis based framework that automatically generates vulnerability signatures for SQLIV and XSSV detection, given attack pattens (regular expressions). These signatures are constructed via forward symbolic reachability analysis followed by backward symbolic reachability. Forward symbolic reachability determines possible string variable values, represented as deterministic finite automata (DFAs). These possible values are compared to the given attack pattern to determine vulnerabilities. Backward symbolic reachability analysis computes all possible inputs that exploit vulnerabilities, represented as DFAs, called vulnerability signatures. Both the forward and backward analysis provide over approximations; thus, some vulnerabilities found in the forward reachability analysis may be false positives. This application-level, server-side, semi-automated static method requires attack pattern definition.

Apollo, by Artzi et al. [25] in 2010, is a technique combining concrete and symbolic execution (concolic testing) and explicit-state model checking to detect vulnerabilities created by runtime errors and by malformed HTML, which could include SQLI and XSS. The authors have implemented Apollo, a PHP-specific tool, that dynamically discovers possible input, using concrete and symbolic execution to track the flow in the application. Apollo is a fully-automated, server-side, hybrid method that begins with a static analysis of JavaScript and collecting of static HTML documents, followed by dynamic test case generation. The dynamic test case generation is followed by monitoring the application for crashes and validating HTML output via flow-tracking.

The different approaches that employ model checking model the code, the HTML form, or the query itself (for SQLID). Various methods perfom control flow analysis, goal-directed model checking, and concolic testing. The specifications checked against also vary, predefined specifications and user defined queries in a query language specification. Like program analysis, model checking is limited to vulnerability detection unless coupled with an attack detection phase. Model checking requires the application source code, and can be fully-automated if test cases are automatically generated. Unlike typical program analysis, model checking results in counterexamples to reveal more information about the vulnerability.

2.3.4 Code re-write

The basis of some FOID techniques is alteration of the original source code or query and using that alteration (or lack thereof) to detect vulnerabilities or attacks. Code re-writing is another FOID solution.

Introduced in 2004, one proxy-based code re-write SQLID technique has been presented by Boyd and Keromytis [21], *SQLrand*. First, an application must be retrofitted or designed to have altered queries with SQL keywords appended with random numbers, generated by an Instruction Set Randomization (ISR). The runtime proxy removes and validates the numbers on modified keywords or deems bare keyboards as malicious before sending queries to the database. The authors have used only regular expressions (regex) to match SQL keywords followed by integers, not (a) specific integer key(s). Thus, as implemented, any attacker need only add a sequence of digits after injected keywords. Even with a key, the possibility of key discovery exists. This method will not discover SQLIA that do not rely on SQL keyword injections. With static analysis to retrofit code and dynamic runtime proxy, this semi-automated, server-side method relies on an attack library of regular expression for SQL keyword matching.

In 2007 BravenBoer et al. [9] have applied syntax embedding to address string manipulations, particularly concatenations, that create attacks. They have designed a hybrid tool, *StringBorg*, which acts on the server-side statically to parse application source code files and generate a language-specific application programming interface (API). This API maps the guest language to the host language, thus embedding the grammar of a guest language (e.g., SQL) into a host language (e.g., Java). For example, Java code becomes "antiquotes" and SQL fragments become "quotes". StringBorg performs a transformation (called *assimilation*) of quotes to API calls. Thus by construction the code is less vulnerable. StringBorg is semi-automated, relying on an attack library. They use SQL as an example embedded language, but note that their application is not limited to SQL as the embedded language and can be applied to other languages and thus could detect other attacks, i.e., XSS. The authors have claimed that the API guarantees no injection attacks can occur; however, only injection attack vulnerabilities in which SQL keywords are injected are thwarted, as it analyzes the syntax of queries. Furthermore, they have assumed that input will be concatenated with constants, which may not be the case. Multiple inputs concatenated together could cause an injection attack when the inputs themselves do not contain keywords or an attack, but once concatenated the resulting string contains a keyword and an attack. An input value devoid of SQL keywords can still lead to an attack, if the input results in a tautology-containing query.

In 2009, Van Gundy and Chen [43] have applied ISR techniques to the problem of XSSA in Noncespaces. The Web application is tasked with adding random prefixes of tags in the XML namespace for each document. If a document is devoid of this random prefix, or the prefix is incorrect, the client distinguishes it as untrusted, otherwise it is trusted. If a user is able to guess the random prefix, an attack will be seen as trusted. This dynamic, client- and server-side method finds and disables attacks. Also, Thomas, et al. [19] have presented a technique, Prepared Statement Replacement-Algorithm (PSR-Algorithm), based on replacing SQL queries in the source code with prepared statements. A prepared statement contains the query structure and bind variables (placeholders for variables). For each bind variable there is a setter method which assigns the variable, performs type checking and will neutralize invalid characters (i.e., single quotes). Their method begins with static code inspection to find SQLIVs. Given the source code and the line numbers of the SQLIVs, the PSR-Generator generates prepared statements as replacement code for the SQLIVs. The algorithm itself then checks for security via unit testing. This hybrid static and dynamic, server-side, application-level method removes vulnerabilities. In the authors' case studies, 6% of SQLIVs remained. The PSR-Generator is automated; however, the SQLIV line number discovery is not part of the algorithm and steps for code inspection are carried out via third party tools to gather the algorithm's input. Prepared statements are a defense against SQLIAs that change the query structure by addition of SQL keywords or other values (i.e., the insertion of an additional row into the database through one vulnerable variable in a row insertion query) and SQLIAs that result from invalid data values (i.e., an out of range value). Attacks that result from valid data values are not detected. For example, a query that contains a condition comparing a value to the result of a sub-query, where the value is of appropriate datatype yet it creates a tautology (such as in Figure 1.1).

In 2010, Athanasopoulos et al. [69] have applied ISR to separate legitimate client-side code from potential attacks using a framework that applies to the browser environment *Isolation Operators*(IO) and *Action Based Policies*. The application of an *IO*, the **XOR** function, will randomize and isolate the JavaScript source in a page. Thus, all the JavaScript code is transposed to a new domain, the **XOR** domain. This differs from ISR methods that randomize keywords or instructions in that it randomizes the entire source code. The browser must then *deisolate* the code to execute the script. The authors have reported low computation overhead, since **XOR** is a instruction-set-independent CPU instruction in today's hardware platforms. Their framework addresses various types of First Order XSSAs. In the same year, Johns et al. have proposed a technique "to outfit modern programming languages with mandatory means for explicit and secure code generation which provide strict separation between data and code" [20]. Thus they address the assumed safety of data other than input that fails to be sanitized as was done in many previous techniques. They have achieved their technique by creating embedded syntax in the code, requiring the developer to explicitly create the semantics of the code, and separating embedded data and code within the application. This static, server-side technique is used for both SQLIV and XSSV. We classify it as semi-automated since the authors advise post-parsing review of code.

Code re-write techniques must access the application source code, which is only an option for developers and typically system administrators. These techniques can find either vulnerabilities, attacks or both.

2.3.5 Structural matching

Structural matching techniques use structural representation of code or portions of code for vulnerability or attack detection. One such structure is a parse tree. A parse tree is an ordered tree representing the structure of a set of symbols (e.g., a string). The parsing is performed based on the syntax of a language (e.g., SQL). FOID methods use structural matching to represent the syntactical structure of queries or code, to check for modified or deviant strings (i.e., strings not following the syntax) which could indicate a vulnerability or attack.

In 2005, Buehrer et al. [15] have developed *SQLGuard* to identify and thwart SQLIAs. Programmers must implement this semi-automated technique via calls to the static class SQL-

Guard, which parses and builds strings to represent the query. SQLGuard creates two parse trees: one for the intended query, one for the actual query. Attack detection is achieved by comparison of these two parse trees. If they align exactly, the actual query does not contain an attack. SQLGuard does not account for attacks that do not alter query structures (e.g., have identical actual and intended parse trees), but instead inject input values of the expected datatype that result in attacks. This static (class calls) and dynamic (parse tree comparison) method adds little overhead. It measures the result of the input instead of attempting to sanitize the input before executing the query. It still suffers false negatives when an attack query's parse tree matches an expected structure. The parse trees are compared at query level in this server-side technique implementation. Also in 2005, Kruegel, et al. [71] have presented an anomaly detection system, using various techniques to detect XSSAs. Their system detects and scores anomalies found in the server log files which must conform to the Common Log Format. This approach compares HTTP requests and their parameters to program-specific profiles. Thus, it is a focused analysis when compared to general anomaly detection. Also, for implementation, the expected structure of the requests must be defined in profiles, thus not fully-automated. Anomaly scores are calculated with various models, and scores that fall above a detection threshold are reported as anomalous. This server-side dynamic method requires profile and threshold definitions and is a runtime technique comparing requests to profiles, which adds overhead.

In 2006, Su and Wassermann [65] have identified improperly sanitized input as an antecedent to command injections. *SQLCHECK*, their hybrid attack detection method checks for syntactic changes in queries by comparing dynamically queries' parse trees analogous to [15]. However, their query level method differs from [15] with the use of compiler parsing techniques and context-free grammars. Calls to SQLCHECK are added manually in this semi-automated technique. The authors have reported low runtime overhead and no false negatives and false positives; and they have claimed to be the first to formally define command injection in the context of Web applications. However, they fail, as does [15], to detect an attack in a query that adheres to appropriate datatype values, as the attack parse tree structure will match the benign tree structure.

Sania, a hybrid method for SQLIV detection, by Kosuga et al. [14] in 2007, is implemented during an application's development and debugging phases. First, the tool determines potential hotspots in an HTTP request where input is added. Next, it generates and inserts an attack string at the hotspot. Finally, comparison of parse trees of the intended query and the actual query will deem the hotspot vulnerable (differences found) or not (parse trees match). This syntactic-based comparison will fail to detect hotspots vulnerable to attacks that use valid datatype values, as do other parse tree methods, with similarly reported false positives. This application-level server-side method is semi-automated, requiring an attack code list. Another parse tree method has been presented by Bandhakavi et al. [4]. They have developed *CANDID*, *CAN*didate evaluation for *D*iscovering *I*ntent *D*ynamically, a dynamic tool that converts Web applications to safe applications (e.g. by retrofitting code). CANDID is a parse tree method similar to [15] and [14] in limitations, but differs by computing the symbolic expression of the query. CANDID mines "programmer intended queries by dynamically evaluating runs over benign candidate inputs" [4]. This server-side, application-level method aims to prevent SQLIAs and is semi-automated, employing an attack library. This symbolic evaluation is a precursor the use of concolic methods (previously described in this chapter) where the concrete candidate would be used to generate test cases for symbolic variables (along with further symbolic execution including an automated theorem prover or constraint solver).

In 2008, Johns et al. [72] have created a prototype of a passive server-side XSSA detection system called *XSSDS*. This tool compares incoming and outgoing script code in the HTTP request and response pairs. This method for first order XSSA detection is based on the observation of a direct relationship between user input and injected scripts: injected scripts are present in their entirety in both the HTTP request and the HTTP response. Thus, simple matching of incoming data and outgoing scripts will find such fully contained scripts. Non-script HTML is ignored. This dynamic, server-side method discovers XSSAs. A 2008 work by Kemalis and Tzouramanis [29] is *SQL I*njection *D*etection *S*ystem (*SQL-IDS*), a hybrid static and dynamic method that detects SQLIAs. Each query is intercepted server-side between the application and database and tested for validity according to a set of specification rules. Once verified as a non-threat by intended query and actual query syntactic structure comparison (similar to parse tree methods [15, 14, 4]), it continues to the database. If verification fails at least one specification rule, the query is marked as an attack and information is logged for the Web administrator or programmer. In this semi-automated technique, specifications rules offer chances for attack detection beyond parse tree comparison; however, they require user specification rule creation. The authors have reported no false positives nor false negatives in their preliminary experimental outcomes. The SQL-IDS detection itself is query level.

In 2009, Nadji, et al. [40] have designed an XSS defense algorithm, Document Structure Integrity (*DSI*). DSI first tracks untrusted data in the server and browser. It syntactically isolates any user data at the parser level. Then server-specified policies establish the confinement of untrusted data. Finally, the structures of the intended parse tree (without user data and benign inputs) and the actual parse tree of a Web page are compared. If these structures are dissimilar, then an attack is detected. This server- and client-side method detects attacks. That same year, Ter Louw et al. [41] have introduced *BLUEPRINT*, a tool that aims to minimize trust on browser content, applicable to minimizing XSSAs. This algorithm intercedes the normal flow of HTML through the following: HTML Lexer/Parser, Document Generator, JavaScript Lexer/Parser, JavaScript Runtime Environment, and Document Object Model (DOM) API. The goal of the approach is to eliminate dependence on the browser's parsers that may produce unreliable results. The HTML parse tree not containing dynamic code will be constructed on the application server. The client-side browser will generate a parse tree to be sent to the browser's document generator without allowing browser parsing. These two-steps ensure that the intended parse tree, when compared to the actual parse tree, will reveal any unauthorized script nodes. BLUEPRINT has a server-side component and a client-side script library contained in each Web page that is output by the program. This dynamic server- and client-side method will detect XSSAs. Another method comparing structures was presented by Bisht and Venkatakrishnan [70]. Their server-side framework aims to prevent XSSAs. *XSS-*

Guard dynamically creates the intended pages of an application, including application-intended scripts, called shadow pages. The server then uses these shadow pages to compare to actual pages during user browsing. If an unintended script is detected, it is removed from the page before it can be executed. This dynamic, server-side approach finds and removes XSSAs. Liu, et al. [64] have presented *SQLProb*, a SQL proxy-based blocker. This two-phase parse tree method first collects the query in a proxy between the application and the database and stores it in a repository. In the query validation phase, user input is extracted to employ an alignment algorithm for pairwise alignment of the actual query and the queries in the repository, then this input is validated using a parse tree. The parse tree is traversed depth-first to determine if the set of leaf nodes representing user inputs is a superset of the expect parse tree leaf node set, denoting an attack. This query-level dynamic analysis requires no source code access, and finds SQLIAs to block them. This fully-automated, server-side, blackbox technique is language independent as well. Detection relies on attacks in which the parse tree structure deviates from those of benign queries.

The various approaches compare the generated HTML structure, query structure, or some part of the HTML structure. They compare an expected structure to the actual structure when the application is executed with test cases or at runtime. Structural matching, such as parse tree comparison, are successful in finding attacks which alter the structure of the SQL query or program code; however, attacks that abide by syntax rules and do not alter query structure can go undetected. Some of these methods do report false negatives and false positives, lacking desired precision.

2.3.6 Taint analysis

Taint analysis is a dynamic technique that tracks the flow of tainted (possibly altered) variables through a program. The taint originates in variables that can be influenced by an external user (e.g., user input) and it is passed through variable manipulations in the program. Taint analysis considers information flow, while program analysis can consider this flow and other program behaviors both statically and dynamically.

Halfond et al. [7] have introduced in 2006, and refined in a second work in 2008 [6], their highly automated tool to protect existing Web applications from SQLI. In [7], the authors have presented a dynamic application-level tool, *WASP* (*W*eb *A*pplication *S*QL Injection *P*reventer), to detect and prevent SQLIAs on the server. In [6], their framework for experiments is extended to include more open source applications, generate malicious input for these applications, and adding to the set of inputs for previous applications. WASP utilizes positive taint analysis and syntax-aware evaluation. Using positive taint, WASP fetches trusted values (which are more easily established than pernicious values) from a MetaStrings library and tracks them through the application to identify trusted parts of a query. The authors of WASP have reported no false positives. This technique is not fully-automated and requires a whitelist (of allowable scripts) that we classify as an attack library.

In 2007, Johns and Beyerlein [12] have introduced *SMask* to address code injection attacks, including SQLIA and XSSA. Since generic data and executable code are not differentiable from each other, SMask approximates data/code separation using string masking and requires policy files for attack detection. SMask statically marks intended code in string values, so that strings injected during an HTTP request will remain unmarked and thus dynamically detected. This static server-side application-level code-integration approach detects and prevents attacks. Although code instrumentation is automatic, policy files are required for attack prevention, thus SMask is semi-automated.

Whether tracking potentially malicious taint or positive taint, these techniques for FOID dynamically detect attacks, SQLIA and XSSA.

2.3.7 Proxy

Not all techniques require fully accessing or executing application source code. Instead, some methods intercept the query or data being sent, and using a proxy is a way to intercept data. Once intercepted, the data is checked to see if it contains an attack using some pattern matching, thus another method is also applied.

In 2004, Ismail et al. [33] have employed a proxy as a means finding XSSVs. Their method consists of inspecting HTTP response and request, *Response Change Mode* and *Request Change Mode*, respectively. In *Response Change Mode*, the local proxy checks for the presence of special characters (e.g.,"<", ">"). If found, the request is copied to the detection/collection server and then forwarded; if none found, the request is sent without being copied. Once the server generates its response, if special characters were present, this response is compared to the collected request copy for matching special characters. If the response contains these characters, the server is marked as vulnerable to XSS and the client is sent an escape encoded response. If the response does not match, it is forwarded normally. In *Request Change Mode*, the request is checked for special characters. If found, a copy is saved to the detection/collection server and randomly seeded sequential numbers are inserted, flanking each parameter. A dummy response message is generated when this request is sent on to the server. If this dummy response proves XSS vulnerable, the original request is escape encoded and sent on to the server and the user is notified via an embedded alert HTML message in the response page. If no XSSVs are detected the original request is sent on and no message is sent to the user. This semi-automated, dynamic, client-side system serves to detect XSSVs and inform via the HTML response page and a central repository.

Kirda et al. [34], in 2006, have developed a browser-reliant method called *Noxes* to detect XSSAs. One of the hindrances to detecting XSSAs on the client-side is distinguishing mischievous JavaScript code from benign code. This Windows-based personal Web proxy uses both automated and user-defined security rules. The user can define rules manually, interactively while surfing the Internet, or in *snapshot mode* where the tool creates rules based on observing the user's browsing. Noxes fetches all HTTP requests, checks the policies, then allows or inhibits the HTTP response. Noxes is a dynamic, client-side XSSA mitigation tool, which the authors have claimed to be the first client-side solution.

In 2009, Wurzinger et al. [68] have introduced *SWAP*, *S*ecure *W*eb Application *P*roxy. *SWAP* consists of a reverse proxy that catches HTML responses and a modified Web browser that detects script content. Client-side deployment of SWAP is transparent but requires alterations in Web applications. The reverse proxy relays traffic between the server and clients. It forwards each response to a JavaScript detection component (a modified browser to inform about script content) to find embedded JavaScripts, then returns the responses to the browser. To differentiate benign and malicious scripts, the Web application scripts have been encoded into identifiers called script IDs, thus any discovered script is considered malicious. This dynamic, client- and server-side solution detects and thwarts XSSAs.

Proxy methods can be fully-automatic. Proxies typically detect runtime attacks to prevent and/or report them, as such, for SQLIA they must reside on the application server, a proxy server or a database server, and for XSSA, they can reside client-side and/or server-side.

2.3.8 Browser-based

Browser-based method typically apply to XSSD due to the fact that these attacks typically occur when a malicious script is executed in the end-user's browser.

In 2007, Garcia-Alfaro and Navarro-Arribas [35] have surveyed approaches for preventing XSSAs against Web applications. They have proposed a server-side security method using certificates for defining authorization policies and requiring enforcement of such policies on the client. The authors have implemented policies for prevention of attacks on Firefox, as an extension to the same origin policy. This dynamic, client- and server-side method does require compliant client-side browsers. Jim, et al. [36] have presented a browser methodology to allow or disallow script execution in a tool called *BEEP* (*B*rowser-*E*nforced *E*mbedded *P*olicies). BEEP is a method based on the idea that a Web page can contain embedded policies. These policies identify which scripts may run on the page. Unauthorized scripts should be devoid of embedded policy and thus, not executed. Modifications required to implement BEEP include adding policies to Web applications and support to browsers. This dynamic, client- and server-side method detects XSSAs. Vogt et al. [37] also have implemented a browser-based solution for XSS. Instead of relying on a server-side detection, this solution acts on the browser and allows the user to have a protection layer in which to judge the safety of moving on to a third-party site. Before the user is given the ability to decide, the method itself determines how the browser uses sensitive data. Sensitive data sources include various HTML objects. This sensitive data is marked and dynamically tracked through the browser, including through dependencies. Before that marked data is sent to third site, several options are available from logging it to stopping it, and prompting the user for a decision. This dynamic, client-side browser-based solution detects XSSAs.

In 2009, Athanasopoulos et al. [38] have implemented a method to find XSSAs that bypass previous browser-enforced policies (i.e., [36]). It uses HTML headers to designate execute and no-execute policies. Their framework consists of three elements: client-side code separation (during development client-side code should be separated), client-side code isolation (Web servers should apply *isolation operators* to client-side code), and action-based policy enforcement (execute or no-execute). The authors have deployed an implementation of their browser-dependent method in the Firefox browser and were investigating Safari and Chromium. Theirs is a dynamic, server- and client-side proposal to detect XSSAs.

In 2011, Stephen et al. [39] have developed an Enhanced XSS Guard Algorithm, *E-Guard*. This passive detection system positioned between the browser and Web server applies to XSS. The goal of the algorithm is to list a Website on a blacklist, whitelist or greylist for sites. For scripts themselves, there is a blacklist (untrusted scripts), whitelist (trusted scripts) and grey-based list (undetermined). To evaluate a Website, first, each list is set to empty. Then the blacklist and whitelist are initially populated manually with known scripts of each type. Next, the number of whitelist and blacklist scripts are calculated for a site, the majority type determines the site list type and a tie places the site in the grey list (not yet judged and to be re-visited after more sites have been categorized). Since E-Guard is a rule-based heuristic, false negatives may be present, but no false positives. This dynamic client- and server-side, semi-automated method detects XSSAs.

Browser-based methods require specific browser use, browser compliance or continual updates in accordance with new browser patches and version releases. Browser-side methodologies have the disadvantage of relying upon Web developers' and Web browsers' compliance in the inclusion of policy information, such as optional tags (e.g., HTTP header field "`Referer`" that identifies the requesting site's address) or overhead added to user Internet browsing by demanding decisions.

2.3.9 Penetration testing

Used in industry, penetration testing attempts to exploit vulnerabilities in an application to reveal its susceptibility to attacks. It is easily applied to the specific security risks of SQLI and XSS.

MySQL1Injector Web scanner by Bashah Mat Ali et al. [62] in 2010 is an automated penetration testing tool that dynamically detects SQLIVs in applications. Although not the first scanner approach, it is among the first non-commercial tools comparing itself to research methods included in this chapter. The scanner injects attacks in PHP-based Websites, from a list of attacks, to predict the number of infected fields in the database. We classify MySQL1Injector as semi-automated because it relies on an attack library to detect SQLIVs. *V1p3R* ("viper") [63] is another penetration testing tool. It differs from other penetration testing that randomly generates queries by using a knowledge base of heuristics to inform query generation. First, viper dynamically gathers information about the structure of the application, including pages, form actions and links, acting as a Web crawler navigating the pages, following hyperlinks. Next, the tool traverses the structure to identify HTML form input parameters. Then viper generates SQL injection attacks from the knowledge base of heuristics. Finally, the results of the attack are stored in a log file and the information is used to generate new test data (for the next iteration of attack generation) and to report vulnerabilities. This semi-automated server-side application level dynamic method tests for SQLIVs.

Penetration testing is typically dynamic, only requiring execution of the source code. A drawback to penetration testing is the reliance on known attacks and/or attack patterns for testing (unless patterns are comprehensive). MySQL1Injector and viper offer specific vulnerability testing, where general penetration testing relies on a knowledgeable tester. Only vulnerabilities are detected by penetration testing methods.

2.3.10 Blackbox testing

Blackbox testing, similar to proxy-based and penetration testing, does not examine the application code. It tests the functionality of the application.

In 2012, a blackbox testing tool *SENTINEL* [66] emerged. It is a tool to find logical flaws in applications using an Extended Finite State Machine (EFSM) to infer the Web application logic from observance of query behavior between the application and the database. This dynamic, server-side blackbox method does not access source code and is independent of application language and DBMS. SENTINEL collects the SQL queries and session variables, thus, it must either run on the server where it can access session variable storage. It creates SQL signatures by observing runtime benign queries of each source file (training), deriving their "skeleton structure" and discovering dependencies with other queries. Thus the signature contains data constraints (found in **WHERE** clause), the query parameters which give state and context variable relationships, and data constraints from other queries. Each signature has an associated set of invariants that is transformed to a function for runtime query evaluation. If a runtime query's signature exists and satisfies all invariants, it is deemed safe and allowed to pass through to the database. Once implemented, the dynamic runtime training and runtime detection are fully-automatic. In the author's implementation for PHP, modifications to the *php-mysql* module are required for query and data collection. SENTINEL's goal is to identify and block malicious queries to prevent SQLIAs. This method does consider sub-queries; however, it relies on the query structure (SQL signature) and will not detect SQLIAs resulting from tautologies formed by acceptable input data.

Blackbox testing is advantageous when application source code is not available. It is ideally fully-automatic, and language-independent.

2.3.11 Other techniques

Some notable techniques for XSSA detection use session management [74] and boundary injection and policy generation [75].

Session management In 2006, Johns has incorporated three server-side techniques in his *SessionSafe* tool to render an HTTP session immune to XSS, thus thwarting any would-be session hijacking [74]. A possible consequence of an XSSA is the unauthorized access to an application or server resource via user credentials which are contained in a session identifier (SID). An example of such access is known as XSS session hijacking, where the attacker steals the victim's credentials (e.g., cookies) and uses them to reconstruct the HTTP session in which the user was authenticated to impersonate the victim. Many dangerous scenarios can follow this session hijacking. SessionSafe strives for XSS-immune sessions to specifically avoid XSS session hijacking. The tool uses deferred loading, one-time URLs, and subdomain switching in combination to defend an application. They prevent transmission of the SID, stop recreation of a session, and limit vulnerabilities impact to vulnerable pages only, respectively. This method is not intended to replace input and output validation, a key in many Web application security measures. It is a server-side transparent (does not require code revision) tool that defends against XSSA.

Boundary injection and policy generation In 2011, Shahriar and Zulkernine [75] have developed an attack detection server-side technique, S^2XS^2. This is based on "boundary injection" to encapsulate dynamic content and "policy generation" to verify the data. A boundary defines expected HTML (e.g., number of tags) and JavaScript content, thus offers comparison policy information to check against the actual content. If this comparison fails, an XSSA has been injected. This method entails boundary injection and policy generation, policy storage, code instrumentation, feature comparison, an attack handler and a boundary remover. Run-time overhead lies only in the comparison and attack handling. Web applications are instrumented on the Web server, policies are stored there and the comparison component is server-based. This dynamic, server-side method detects and handles XSSAs.

In summary, SQLIV, XSSV and other vulnerabilities can result from developers' ignoring best practices, failing to sanitize user input or improperly sanitizing user input. The naive approach to simply prohibit scripting or access to back-end database with dynamically created queries (created at execution time) would result in loss of functionality of a Web application. With the prevalence of shared Web services, scripting cannot be disallowed. Even with proper sanitization techniques, the developer cannot predict the next intricate malicious attack from the relenting attacker. To that end, dynamic techniques using finite libraries of known attacks will always be incomplete. Static techniques that detect vulnerable input can fail if they under-approximate the vulnerabilities needing monitoring and cause inefficiency if they over-approximate. Multiple techniques used in tandem will better cover many instances. The next section details summaries based on our classifications.

2.4 Summary

In this section we outline observations of classification properties in Section 2.4.1 and of existing techniques in Section 2.4.2, including discussion of some advantages and disadvantages. In section 2.4.3, we offer guidance on future works addressing FOID with respect to our classifications.

2.4.1 Classifications

For each classification category, we discuss observed trends among the body of works classified in Section 2.3.

1. **Type**. The trend in the classified body of works begins more attack-related, followed by hybrid and vulnerability methods. Vulnerability detection is the best first line of defense to inform of vulnerabilities, but does not mitigate them. Hybrid methods that first detect vulnerable code hotspots, then add sanitization code or check for runtime

attacks typically do so to lower overhead; however, implementation can be expensive or complex and may limit desired non-malicious functionalities.

2. **Granularity**. Techniques typically are application-level, which allows for some form of code analysis and a knowledge of the application itself to better safeguard against vulnerabilities and attacks. Query level methods (for SQLID) have the advantage that they can be language-independent and DBMS-independent, applied to attack detection [64, 66] or vulnerability detection [28]. With ease of use and non-restrictive applicability, query level methods are preferable, especially if source code is inaccessible.

3. **Location**. Typically FOID tools reside server-side on the application server, proxy server, or database server (for SQLID). For both SQLID and XSSD, browser-side methods may insert attack vectors to detect vulnerabilities, while for XSSD alone browser-side methods can monitor the user's Web browsing experience to detect and thwart attacks or monitor responses from the Web server. XSSD, specifically XSSA detection, techniques lend to client-side and hybrid client- and server-side methods since the malicious code is typically executed in the browser.

4. **Level of Automation**. Fully-automated solutions are preferable for many reasons, primarily ease of use; however, they do not offer the most precision. For most of these methods, code must adhere to a specific configuration or set of rules, often following good programming practices and such sites are likely to begin with fewer vulnerabilities, since poor programming practices can contribute to vulnerable code. Semi-automated solutions may rely on attack libraries, rule sets, or user-defined attack information which may be limited, requiring additions or updates, or user intervention.

5. **Test Case Source**. When an attack library is used, whether it is limited or sufficient, the method can only be semi-automated. As long as the library is sufficient for all cases, a semi-automated tool may be preferable over one with fully-automated test case generation that fails to cover necessary cases. Methods employing automatically generated test cases with sufficient coverage that do not require other user information or interaction (fully-

automated) are equally preferable.

2.4.2 Techniques

This section outlines techniques used in the related works, some drawbacks and some benefits of each to inform future solutions.

1. **Testing**. Testing-based techniques generally look for vulnerabilities and typically do so prior to deployment of a Web application to inform the system administrator or developer of poor coding practices or other security holes in the application code. More automation (in terms of automatically generated test cases) is preferable for ease-of-use over user-defined attacks and predefined libraries; however, sufficient coverage is key. Techniques that first test, then instrument code for runtime attack detection need not be excluded from future research with improvements on testing and runtime monitoring techniques.

2. **Program analysis and taint analysis**. Since program analysis techniques evaluate the program's behavior, they typically analyze where vulnerabilities exist. Attack detection, which occurs at runtime, would require additional steps and the overhead would be prohibitive. Like testing, program analysis can offer an off-line analysis vulnerability detection within an algorithm that also detects attacks.

3. **Model checking**. Model checking is a formal technique that produces counterexamples based on errors when checking the model of a system against some specifications. For vulnerability detection methods, model checking allows for precise error detection based on the logic of the code, generally performed pre-deployment avoiding extra overhead; for hybrid detection it is an effective first step. The resulting counterexamples allow precision in locating error sources, not symptom propagation sites, thus code instrumentation (if present) is minimized.

4. **Code re-write**. Code re-write requires source code additions or amendments. Approaches that use APIs and/or attack libraries, augment SQL query keywords or re-write queries as prepared statements have a complex implementation and still leave suscepti-

ble code. In the case of most SQLID techniques, attacks may guess the SQL keyword augmentation or create attacks without keywords to circumvent these countermeasures.

5. **Structural matching**. Techniques using structural matching typically compare the intended code and/or query structure (derived previously and stored or constructed at runtime) with the actual code and/or query structure at runtime. These techniques are effective for detecting structural altering attacks; however, alone they are insufficient for assuring application safety, yet could still be viable a step of a more inclusive algorithm. Drawbacks include the added runtime overhead, for some SQLID methods the need for platform specific tools for extracting the query from the application and for some XSSD, extracting and comparing entire document structures.

6. **Penetration testing**. System administrators and developers use penetration testing (*pen testing*) to find vulnerabilities. Research also has used pen testing to discover vulnerabilities. Like other vulnerability detection techniques (e.g., model checking) pen testing offers error detection as part of a multi-step algorithm to detect and prevent SQLIAs.

7. **Blackbox**. Some typical advantages to blackbox testing techniques are the following: they are language-independent, they do not require source code access, and they are fully-automated. Some fully-automated approaches based on other techniques listed in this section are language-dependent. Blackbox testing can overcome this dependence often with an easily implemented tool; however, some still require access to language-specific system files and/or code modules.

8. **Other techniques**. Proxy methods are used for SQLIA and XSSA detection. Proxy methods can be code- and DBMS-independent, making proxy-based tools easy to implement. Browser-based methods are more adequately used to detect XSSV and XSSA, due to the nature of XSS. Detection systems have been applied to XSSA detection; they represent where research meets industry and could easily become commercialized tools.

2.4.3 Conclusions

We have presented a comprehensive classification of FOID techniques. Kindy and Pathan [59] have presented a survey of attacks, vulnerabilities and detection techniques. They categorize methods into two sets, vulnerability detection or countermeasures (attack detection possibly preceded by vulnerability detection). Johari and Sharma [60] have described and reviewed a sample of SQLID works, concluding that SQLIDs have general weaknesses, among which include runtime overhead and invasive user interaction. However, they have not outlined these for each work, as we have.

Since we cannot mitigate all risks or thwart all attacks, we should aim to minimize them. Even with current protective measures, a malicious user could still construct more complex attacks. Injection attacks will persist as long Web sites with database connectivity allow searching and updating and exist in their current configurations. There are no existing techniques to eliminate all vulnerabilities or attacks. Even extreme limitation of access wherein only hard-coded queries or scripts are executed may have security breaches for the savvy attacker to exploit and would seriously hinder the user's Web experience and encumber the developer.

We observe different characteristics and sets of characteristics among the works classified. Hybrid types for FOID are promising with vulnerability detection to find code and/or query weaknesses followed by attack detection (e.g., monitors placed in the code only at vulnerable spots). Dynamic methods and the dynamic phase of hybrid methods should ensure that they do not incur too much overhead. Fully-automated and semi-automated solutions are viable for future research as long as predefined attack libraries and test case generation ensure sufficient coverage. SQLID approaches should be server-side, except in cases for SQLIV where the tool's methodology allows for browser-side use (e.g., pen testing). XSSD approaches can be client-side, server-side or a hybrid. Although application level methods are prevalent, this need not be the case in future works as query level solutions for SQLID are also suitable.

We have defined a classification for FOID and used this to categorize a representative sample of related works. Finally, we have presented observations of the classified techniques. Our hope is that this classification and evaluation of existing techniques will serve as to inform for future work addressing FOID. In seeing the techniques applied, the characteristics that various solutions possess, researchers can aim to improve upon certain characteristics or to develop a solution with a specific set of characteristics. We have implemented a query level vulnerability detection method for SQLI, presented in Chapter 3 and an application level hybrid method to address XSS in Chapter 4.

CHAPTER 3. ANALYSIS & DETECTION OF SQL INJECTION VULNERABILITIES VIA AUTOMATIC TEST CASE GENERATION OF PROGRAMS

In this chapter, we present a query level technique to analyze and detect tautology-based First Oder SQL Injection Vulnerabilities. It distinguishes itself from other similar methods that utilize query syntax structure in their solutions by also accounting for the semantics of the query and query dependencies. Our novel technique identifies the possibilities of such attacks. The central theme of our technique is based on automatically developing a model for a SQL query such that the model captures the dependencies between various components, e.g. sub-queries, of the query. We, then, analyze the model using CREST[1] test case generator and identify the conditions under which the query corresponding to the model is deemed vulnerable. We further analyze the obtained condition-set to identify its subset; this subset being referred to as the *causal set* of the vulnerability. Thus, our technique considers the semantics of the query conditions, i.e., the relationship between the conditions, and as such complements the existing techniques which only rely on syntactic structure of the SQL query. In short, our technique can detect vulnerabilities in nested SQL queries, and can provide results with no false positives and false negatives when compared to the existing techniques.

The rest of this chapter is organized as follows. Section 3.1 briefly reviews SQLI, describes query dependencies and introduces our technique. Section 3.2 describes our technique; especially (a) the technique for translating SQL queries to corresponding C-program, (b) the application of CREST for obtaining conditions of injection vulnerability, and (c) the analysis technique deployed to obtain the minimal causal set. Section 3.3 presents advantages of our

[1] CREST: Automatic Test Generation Tool for C. Available at `http://code.google.com/p/crest/`.

proposed technique and results of its evaluation. Finally, Section 3.4 concludes the chapter.

3.1 Introduction

SQL Injection Attacks occur when malicious code is injected into a SQL query that is executed and allows unauthorized access to data or system resources. When a SQLIA is launched by exploiting a vulnerability in a Web application, the malicious data is inserted via Web page input and becomes part of some SQL query code in the application. The application with the newly injected code is sent to the Web server, and subsequently the query is executed on the back-end database that resides on the Web server or on a database server. The attack is successful if the input is not properly sanitized before being injected into a query or before being executed on the database.

Our approach includes a static server-side approach using testing to find First Order SQLIVs. We first model the query then evaluate for conditions in which the **WHERE** clause(s) of the query could contain a tautology. Thus, it is susceptible to an SQLIA. The first step in modeling the SQL query is breaking down its SQL keywords. A basic query form is the following: **SELECT** *fieldname-list* **FROM** *tablename* **WHERE** (*condition*$_1$ **AND/OR** *condition*$_2$). In the query's **WHERE** clause, a *condition* contains the comparison describing (a) desired value(s) of a field in a table within the database. Rows with fields that satisfy the **WHERE** clause are returned as result of the SQL query. We will consider a simple equality constraints in a condition to explain our method.

In the query **SELECT** `Last_Name, First_Name` **FROM** `User` **WHERE** (`Status_field = $status`), the condition "`Status_Field = $value`" contains `$value` that may be user supplied or dependent upon user-supplied data. If `$value` can contain a tautology due to user input, then the query is vulnerable to SQLI and the condition `Status_Field = $value` is the vulnerable condition. Sanitization methods may help secure the query from some tautology-based attacks by blocking keywords and escaping special characters. There are some queries susceptible to tautology-based attacks from user input that does not contain keywords or special characters, but legitimate values that can lead to SQLIA.

It is noted that fields to be returned that are listed after **SELECT** and tables listed after **FROM** are of no consequence in the creation of a tautology for the basic query. Only when the query is complex, containing sub-queries can the fieldname-list of the **SELECT** clause play a role in a tautology. An example is the following: **SELECT** `Last_Name, First_Name` **FROM** `User` **WHERE** (`Username = $username` **AND** `1 <` (**SELECT COUNT** (Login_Date) **FROM** `Login_Log` **WHERE** `Username = $username`)). Here the query contains a nested sub-query. The result of that sub-query, the returned aggregate function value of the **SELECT** statement becomes the value of a condition in the **WHERE** in which it is nested. Thus the **SELECT** clause is of consequence in the model. Sanitization methods on user input influenced values (here $username) may not prevent a tautology in the condition containing the sub-query.

Our technique addresses the tautology-based attack that sanitization methods can fail to catch. Existing SQLID sanitization techniques are based on syntactic differences due to the insertion of SQL keywords and may check for valid data values. These methods that check for unexpected values and/or SQL keywords will fail to detect a tautology created by appropriate data values. Thus, they are susceptible to both false positives, identifying benign queries as attacks, and false negatives, identifying attacks as benign queries. Our method evaluates the dependencies between conditions in the SQL queries, especially when the query is nested.

Another advantage of our technique is that, we can automatically identify the conditions under which vulnerabilities in the query can be exploited to realize a SQLIA. We refer to these conditions as the *causal set*. The query inputs, at runtime, can be checked against the causal set; if the check is successful, then the query input is deemed malicious, otherwise, the input is benign.

The causal set gives the following advantages: (a) the conditions in the runtime execution of the SQL query can be verified against the conditions in the causal set and (b) the execution can be identified to exploit a SQL vulnerability if the conditions in the causal set are satisfied.

The *contributions* of our approach are summarized as follows:

Contributions

1. We propose a new approach for SQLID that analyzes the semantic dependencies between SQL query conditions and does not rely solely on syntactic structure of the query.

2. Our approach is complementary to the existing techniques for SQLID and leads to an effective detection mechanism for SQLIVs. Since our technique is based on the semantic dependencies, it does not have any false positive or false negative results.

3. We provide a novel technique to reduce various cases that can lead to SQL injection and automatically combine these cases into a succinct summary. The succinctness allows for easy understanding of the query vulnerability and facilitates efficient monitoring of the user inputs that can lead to exploitation of the vulnerability. We refer to the summary as the causal set.

3.2 A method for detecting SQL injection vulnerabilities

In this section, we describe our technique. It consists of the following three main steps:

1. We compile SQL queries to a target language (in our case C) such that the dependencies between query conditions at various locations in the query are faithfully captured;

2. We apply an existing test generator (in our case CREST) to obtain the test cases that correspond to valuation of conditions at different locations leading to possible injection vulnerability exploitation;

3. We analyze the test cases to identify the cause of the vulnerability that can be effectively used during run-time monitoring of the query-executions.

3.2.1 Translating SQL query conditions to C-programs

In this section we describe the first step of our technique in which we translate queries. The primary objective of our translator is to generate a program (essentially using if-control construct) which captures the valuations of the conditions at different locations in the query (associated to the **WHERE** clause) and their inter-dependencies that can maliciously affect the query result. In describing our technique, we will consider the following types of conditions: atomic conditions of the form “`X = Value`”; belongs-to conditions of the form “`X` **IN** (some nested query result)”; and boolean combination (conjunction, disjunction, negation, etc.) of the above conditions. Other forms of conditional expressions can be translated by following appropriate rules of translation found in Algorithm 1.

In the event the **WHERE** condition is atomic, the query becomes vulnerable whenever the condition (after code injection) at that location becomes a tautology. We do not consider the valuation of the exact condition in the query; instead, we are interested in the valuation of the condition (after code injection) at the location where the original condition was present. For example, for an atomic condition “`X = $input`” in a **WHERE** clause of the query, we say that **WHERE** clause contains a location (say, `c`) that holds an atomic condition dependent on user input and may be affected by the user. The user can make this condition a tautology by providing an input such that “`$input =‘’ OR ‘1’=‘1’--’`”. Observe that the user input makes the original condition non-atomic (by adding a disjunction); however, we are not concerned with this exact change. We simply detect that the location `c` (where a user-input dependent condition is present) contains a condition that has become a tautology.

Similarly, for a conjunctive condition, we are interested in finding out whether the conditions at any one of the locations (say `c1` and `c2`) which contain the conjuncts can be made a tautology. This is because if any one of the conjuncts at a location (e.g., `c1`) becomes a tautology while the other (`c2`) is not a contradiction, then the query result is affected by the condition at `c1`. A simple and commonly used example illustrating this scenario is as follows. **SELECT** `name` **FROM** `users` **WHERE** `user = ‘$input1’` **AND** `passwd = ‘$input2’`.

```
SELECT  X1  FROM  T1, T2
WHERE   Y11 = $input11  AND Y12 = $input12
                       AND Y13 NOT IN
                           SELECT  X2  FROM  T3, T4
                           WHERE   Y21 = $input21
                                OR Y22 = $input22
```

Figure 3.1 SQL query with nested sub-query

In this query, there are two locations c1 and c2 for possible injection. If the user provides $input2 such that it is equal to “’OR ‘1’ = ‘1’”, the query becomes **SELECT** name **FROM** users **WHERE** user = ‘$input1’ **AND** passwd = ‘’ **OR** ‘1’ = ‘1’. Thus the user can access entries in the table without proper authorization. This intrusion is allowed as long as no other condition, i.e., at location c1, becomes a contradiction, in which case the result of the query is an empty set. Similar arguments can be provided for the dual operation: disjunction. Replacing the **AND** with **OR** in will result in a disjunction and will result in a tautology in the **WHERE** clause.

Figure 3.1 presents an example SQL query with a nested sub-query which we use to describe our technique. Notice the query has four user-input “locations” where the code injection can occur. We denote these locations as c11, c12, c21 and c22 corresponding to the user-input dependent conditions “Y11 = $input11”, “Y12 = $input12”, “Y21 = $input21” and “Y22 = $input22”, respectively. There is also one condition that relies on the results of the sub-query, c13 corresponding to “Y13 NOT IN *(results from sub-query)*”. The variables containing the input are $input11, $input12, $input21 and $input22 and the query is exploited via a tautology-based SQL injection attack if one of the following holds with actual user inputs:

1. The condition at location c11 becomes a tautology, condition at location c12 does not become a contradiction, and disjunction of the conditions at locations c21 and c22 does not become a tautology;

2. The condition at location c12 becomes a tautology, condition at location c11 does not

become a contradiction, and disjunction of the conditions at locations `c21` and `c22` does not become a tautology;

3. The conditions at locations `c11` and `c12` do not become contradictions, and disjunction of the conditions at locations `c21` and `c22` becomes a contradiction.

Proceeding further, for conditions that depend on nested sub-queries (belongs-to), we say that if the sub-query is affected by some code-injection then the belong-to condition is also affected. The query in Figure 3.1 will be used throughout to describe our technique. If conditions at locations `c21` and `c22` evaluate to a contradiction (due to code injection of the form `0 = 1`), the condition at location `c13` (associated with "`Y13 NOT IN ...`") becomes a tautology.

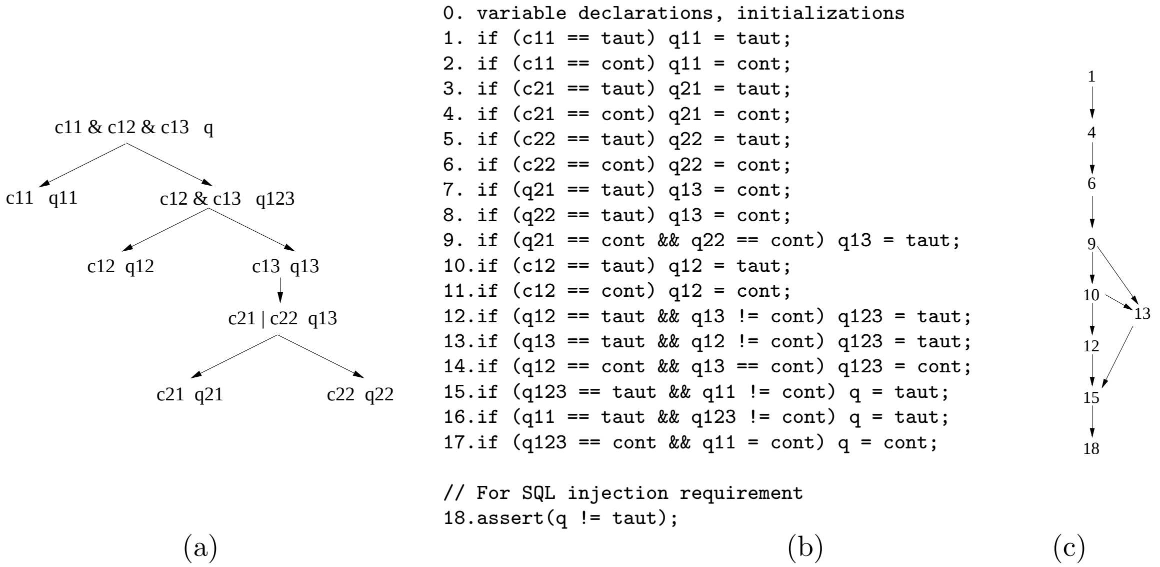
Algorithm 1 presents our translator. It takes as input a SQL query and generates program code. The first step, as noted above, is to gather the locations c of conditions associated with the **WHERE** clause of the query (Line 2). Then a subroutine `TRANSLATE`, with the condition location c and query q as parameters, is invoked (we have overloaded q to denote a query and also a variable to capture how the query is affected by the conditions in its **WHERE** clause). As outlined above, the algorithm recursively explores the query condition-locations (conjunctions, disjunctions, etc.) and, wherever necessary, analyzes the locations of subquery conditions (e.g., at Lines 23, 28). Note that if there exists a conjunctive condition at location `c`, we represent it as **AND** of the corresponding locations holding the conjuncts (see Line 10). Similarly, if there exists a disjunctive condition at location `c`, we represent it as **OR** of the corresponding locations holding the disjuncts (see Line 17).

Example 1 *Figure 3.2(a) presents the recursive exploration of query in Figure 3.1 by the translation algorithm. In the figure,* `c11` *denotes the location for the condition* `Y11=$input1`*,* `c12` *denotes the location for the condition* `Y12=$input2`*,* `c13` *denotes the location for the condition* `Y13 NOT IN ...`*,* `c21` *denotes the location for* `Y21=$input21`*, and finally* `c22` *denotes the location for* `Y22=$input22`*. Each condition location can either take the valuation (a)* `taut`*, denoting that the condition at that location has become a tautology; or (b)* `cont`*, denoting that*

Algorithm 1 Query Translator

```
procedure TRANSLATE($q$)
    Obtain condition-locations $c$ associated to WHERE clause;
    TRANSLATE($c$, $q$);
end procedure

procedure TRANSLATE($c$, $q$)
    if $c$ is atomic then
        print if ($c$ == taut) $q$ = taut;
        print if ($c$ == cont) $q$ = cont;
    end if
    if $c := c_1$ AND $c_2$ then
        TRANSLATE($c_1, q_1$); TRANSLATE($c_2, q_2$);
        print if ($q_1$ == taut && $q_2$ != cont) $q$ = taut;
        print if ($q_2$ == taut && $q_1$ != cont) $q$ = taut;
        print if ($q_1$ == cont) $q$ = cont;
        print if ($q_2$ == cont) $q$ = cont;
    end if
    if $c := c_1$ OR $c_2$ then
        TRANSLATE($c_1, q_1$); TRANSLATE($c_2, q_2$);
        print if ($q_1$ == cont && $q_2$ == cont) $q$ = cont;
        print if ($q_1$ == taut) $q$ = taut;
        print if ($q_2$ == taut) $q$ = taut;
    end if
    if $c :=$ V IN $q_k$ then
        TRANSLATE($q_k$);
        print if ($q_k$ == taut) $q$ = taut;
        print if ($q_k$ == cont) $q$ = cont;
    end if
    if $c :=$ V NOT IN $q_k$ then
        TRANSLATE($q_k$);
        print if ($q_k$ == taut) $q$ = cont;
        print if ($q_k$ == cont) $q$ = taut;
    end if
    if $c :=$ V > $q_k$ then
        TRANSLATE($q_k$);
        print if ($q_k$ == taut) $q$ = cont;
        print if ($q_k$ == cont && $V > 0$) $q$ = taut;
        print if ($V < 1$ ) $q$ = cont;
    end if
    if $c :=$ V < $q_k$ then
        TRANSLATE($q_k$);
        print if ($q_k$ == taut) $q$ = taut;
        print if ($q_k$ == cont) $q$ = cont;
        print if ($V < 1$) $q$ = taut;
    end if
end procedure
```



```
0. variable declarations, initializations
1. if (c11 == taut) q11 = taut;
2. if (c11 == cont) q11 = cont;
3. if (c21 == taut) q21 = taut;
4. if (c21 == cont) q21 = cont;
5. if (c22 == taut) q22 = taut;
6. if (c22 == cont) q22 = cont;
7. if (q21 == taut) q13 = cont;
8. if (q22 == taut) q13 = cont;
9. if (q21 == cont && q22 == cont) q13 = taut;
10.if (c12 == taut) q12 = taut;
11.if (c12 == cont) q12 = cont;
12.if (q12 == taut && q13 != cont) q123 = taut;
13.if (q13 == taut && q12 != cont) q123 = taut;
14.if (q12 == cont && q13 == cont) q123 = cont;
15.if (q123 == taut && q11 != cont) q = taut;
16.if (q11 == taut && q123 != cont) q = taut;
17.if (q123 == cont && q11 = cont) q = cont;

// For SQL injection requirement
18.assert(q != taut);
```

Figure 3.2 (a) Possible execution tree of TRANSLATE; (b) Result of translation; (c) Partial execution graph explored by CREST.

the condition at that location has become a contradiction; or (c) remain unchanged (denoting no code injection). In addition to `q` *and* `q13`*, which capture whether the top-level and the nested queries, respectively, are affected by their corresponding* ***WHERE*** *conditions, there are several other "*`q`***" variables used as intermediate data variables in the translator. Figure 3.2(b) shows the code generated as a result of the translation.* □

We say that the injection vulnerability is exploited if the valuation of conditions at locations related to user inputs are such that the program resulting from translation violates the assertion on `q`, the top-level query (Line 18 in Figure 3.2(b)).The following theorem states the correctness of the above claim.

Theorem 1 (Sound and Complete Translation) *Given a program P generated by Algorithm 1 from a query Q, there exists some execution path in P where* `q` *evaluates to* `taut` *at the program's exit point if and only if there exists some combination of valuations of query conditions at different locations that maliciously affects the result of Q.*

Proof The proof follows directly from the semantics of the conditions and their effect on the queries. If the query condition is conjunctive, then the query is affected only when the condition in at least one of the locations becomes a tautology while conditions at other locations are not contradictions. This is carefully captured by the translation algorithm and appropriately used to generate the corresponding code. As a result, the program P will have an execution path which makes `q` (the program variable used to capture SQL injection attack at the top-level query Q) to be equal to a tautology. The similar argument holds for disjunction. The query condition containing the set condition *belongs-to* (`V IN` $\mathtt{q}_k$) will be a tautology if the result of the subquery, $\mathtt{q}_k$ is a tautology. In this case, the program P will have an execution path making `q` tautology. The negation of belongs-to follows the same argument. For the query conditions containing a range condition, a value is checked against a subquery (e.g., value `V` greater-than ($>$) q_k). For this case, if the subquery returns a contradiction and the value is greater than zero, then this condition returns a tautology. The program P will have an execution path that makes `q` a tautology. The other range condition argument is similar. Thus for the various query conditions, we will have an execution path leading to a tautology. □

The above theorem ensures that there exist no false positives or false negatives in our analysis.

In the next section we describe the test step of our method.

3.2.2 Application of CREST

We use CREST, an automatic test generation engine for C programs, to analyze the program. We consider the assertion that `q` does not evaluate to `taut` at the exit of the program (Line 18 in Figure 3.2(b)). The program keeps all variables uninitialized. More specifically, uninitialized variables are declared as CREST variables, which allows CREST to choose different valuations of these variables to generate test cases that violate the assertion. At its core, CREST relies on concrete and symbolic (concolic) execution of programs to maximize exploration of branches in a program and identify assertion violations (if they exist). Concolic testing utilizes a combination of concrete and symbolic execution to generate test cases and to effectively guide exploration of the new program paths, respectively. In concolic testing, a random test case is generated and the test program is executed with that concrete value. When the program encounters a conditional statement (e.g., instrumented assertions) these become the symbolic constraints. The concrete and symbol constraints are solved simultaneously to create a new test value and to continue creating new branches to test, until no new branches can be created. As long as the branches are finite in number and finite in length, i.e., not recursively infinite, we are assured extensive branch coverage with this testing technique. In the event the program does not contain any loops (as is the case of the result of our translations), CREST can potentially explore all possible branches and therefore can generate all possible test cases that lead to assertion violation.

Each test case assigns some values to the CREST variables and these values denote the conditions under which `q` evaluates to `taut`, i.e., a vulnerability is exploited.

Example 2 *Figure 3.2(c) shows some of the execution traces of the program in Figure 3.2(b) explored by CREST to generate test cases (each node in the trace denotes a line number of the program). The execution traces 1-4-6-9-10-12-15-18 and 1-4-6-9-10-13-15-18 correspond to the test case where* `c11`*,* `c12` *are tautologies and* `c21`*,* `c22` *are contradictions. Note that CREST may not assign* `taut` *or* `cont` *to all variables while generating a test case. For instance, the path 1-4-6-9-13-15-18 corresponds to the test case where* `c11` *is a tautology and* `c21`*,* `c22` *are contradictions. The variable* `c12` *remains uninitialized; we will refer to such values as* `unin`*.* □

In the next section we discuss the causal set.

3.2.3 Causal set detection: reductions

In the above sections, we have presented how the CREST test case generator can be used effectively to identify injection-causing requirements (i.e., the valuation of conditions at various locations). At runtime, when the user inputs are provided, they are monitored to check whether any of these requirements are satisfied. Any user input that satisfies at least one requirement will be deemed intrusive and the query will not be allowed to execute with the input, thus stopping a SQLIA. While CREST generates all possible requirements in terms of condition valuations at each location, the number of such requirements may be large, and therefore it may be ineffective to verify user inputs against each of the requirements one at a time. For instance, CREST identifies eight different cases, corresponding to the case where the condition at location `c12` is tautology. Similar cases are obtained when conditions at locations `c11`, `c21`, or `c22` become either tautologies or contradictions. In the table of Figure 3.3), we list the values of the conditions under which injection-vulnerability can be exploited in the query. The summary of these cases is that *after the user provides some input, the condition at location* `c12` *becomes a tautology, the condition at location* `c11` *does not become a contradiction, and disjunction of the conditions at locations* `c21` *and* `c22` *does not become a tautology.*

In this section, we present a reduction mechanism which results in a summarization of all cases obtained from CREST. The proposed succinctness achieves two advantages. First, the succinctness permits efficient monitoring of user inputs at runtime. Second, it removes all redundancies in the conditions, thus allowing the developer to understand the root cause of the SQL injection vulnerability in the query and to take appropriate corrective measures.

Decision tree representation of vulnerability requirements. Recall that a vulnerability requirement is given in terms of valuation of conditions at different locations of the query under consideration. The domain of valuation $\mathcal{D}$ is $\{$`taut, cont, unin`$\}$. Each requirement can be viewed as a conjunctive formula where each conjunct corresponds to a valuation of a condition at a particular location. For instance, one of the requirements is

$$\texttt{c11} = \texttt{taut} \wedge \texttt{c21} = \texttt{unin} \wedge \texttt{c22} = \texttt{cont} \wedge \texttt{c12} = \texttt{taut}$$

That is, the conditions at locations `c11` and `c12` are tautologies, the condition at location `c21` is uninitialized (i.e., not adversely affected by user input) and the condition at location `c22` is a contradiction.

The set of all requirements is therefore a disjunction of conjunctive formulas representing individual requirements. Such formulas can be represented using a (3-valued) decision tree where each node in the tree corresponds to one of the location variables and directed edges from a node represent its valuation. The edges are labeled with items $\in \mathcal{D}$. The ordering in which variables appear in the tree is pre-specified and the leaf node is termed `T` (true) node. A path from the root to the leaf in the tree corresponds to a conjunctive formula, which in turn corresponds to one possible valuation of the location variables as described by some requirement. Figure 3.4 presents a 3-valued decision tree representing the injection-requirements shown in the table (Figure 3.3).

c11	c21	c22	c12
taut	cont	cont	taut
taut	cont	unin	taut
taut	unin	unin	taut
taut	unin	cont	taut
unin	cont	cont	taut
unin	cont	unin	taut
unin	unin	unin	taut
unin	unin	cont	taut

Figure 3.3 Requirements

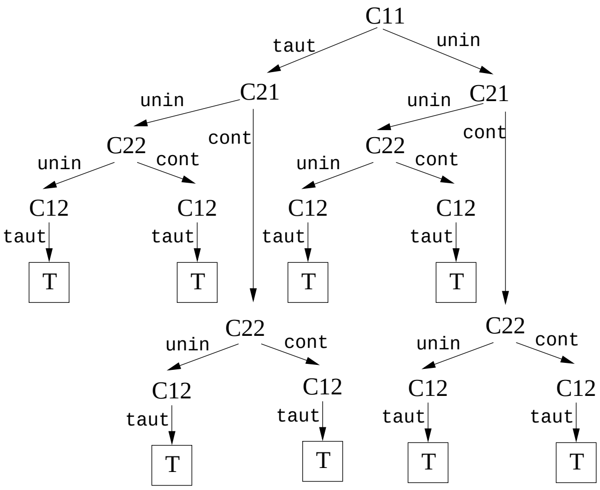


Figure 3.4 3-valued Decision Tree

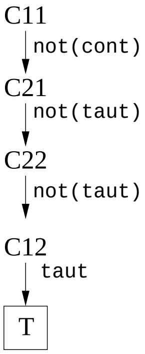


Figure 3.5 3-valued Decision Diagram

Decision trees to Decision diagrams. Decision trees can be reduced to decision diagrams which removes all duplications and redundancies from the decision tree taking into consideration the semantics of boolean operations (conjunction and disjunction) over the domain of the decision tree node-values ($\mathcal{D}$ in our case). [76]. We present rules for reducing our 3-valued decision tree to a 3-valued decision diagram in Figures 3.6(a), 3.6(b) and 3.6(c).

The first rule (Figure 3.6(a)) states that if there is a node c1 such that all its three branches go to the same node c2, then the valuation of the node c1 is not relevant, i.e., there exists some specific valuations for all variables other than c1 such that for all possible valuations of c1, there exists an injection-causing requirement. In this case, node c1 can be removed and all incoming edges to c1 are

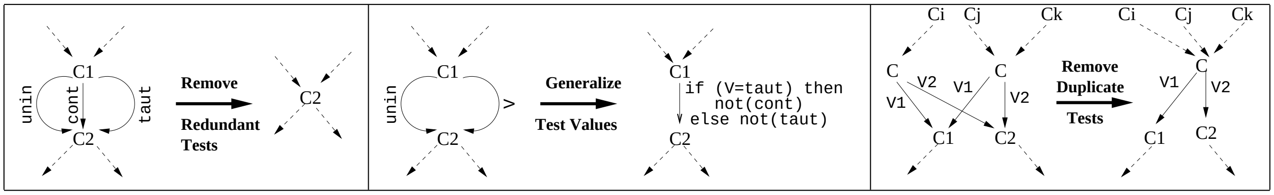


Figure 3.6 Rules for (a) redundant tree removal; (b) generalization of test values; (c) removal of duplicate test values

redirected to its child-node (c2). This rule is commonly referred to as *redundant test removal*.

The second rule (Figure 3.6(b)) corresponds to the case when there exists a node c1 in a path where one of its branches is labeled with **unin** and the other labeled with V (which is equal to either **taut** or **cont**), and both branches lead to the same node c2. In that case, the two branches from c1 are merged to reflect that the valuation of c1 is *not* equal to the *negation of* V. This merging follows from the fact that if there are at least two paths in the decision tree, one where c1 is equal to **taut** (or **cont**) and the other where c1 is equal to **unin**, and all other node values remain the same, then the valuation of c1 in these paths is equal to **not**(**cont**) (or **not**(**taut**)). We refer to this rule as *generalization*. The generalization rule depends on the domain and semantics of the valuations in a multi-valued decision tree/diagram.

Finally, the third rule (Figure 3.6(c)) corresponds to at least two identical subtrees/graphs that are rooted at two different nodes. In that case, one of the nodes is removed and all incoming edges to the removed node are redirected to the one that is not removed. This rule is referred to as *duplicate test removal*.

The application of the above rules converts a 3-valued decision tree to a 3-valued decision diagram (a DAG). Figure 3.5 presents the 3-valued decision diagram obtained from the decision tree in Figure 3.4. The steps that lead to the decision diagram are summarized as follows. Using the rule to remove duplicate tests where the test node does not have any children, only one node T is allowed in the decision diagram. All but one c12 nodes are removed from the decision tree (duplicate test removal). Similarly, there are four duplicate subtrees rooted at c22 and as such three of them are removed. The node c22 has two branches, each going to the same node c12, and as such the branches are merged (generalization) to **not**(**taut**). Similarly, duplicate test removal and generalization are applied to nodes c21 and c11 to obtain the decision diagram.

The decision diagram states that SQL injection vulnerability can be exploited by user inputs which make (a) the condition at location c12 a tautology, (b) the condition at location c11 not a tautology, and (c) the conditions at locations c21 and c22 not contradictions. This is concise and precise representation of the injection requirements shown in the table of Figure 3.3. In essence, the decision diagram captures the *causal set of requirements*. Note that for ease of explanation, in Figures 3.3, 3.4 and 3.5, we have shown one small set of requirements in the table and the corresponding decision tree and diagram. The size of the table is much larger for our example; the reduction due to summarization to a causal set obtained by generating the corresponding decision diagram, therefore, is significant.

The reduction algorithm for obtaining a decision diagram from a decision tree is well-studied [76]. It is based on recursive backward exploration of the decision tree and has a complexity of $O(N \log(N))$, where N is the total number of nodes in the decision tree. One of the challenging aspects of decision diagram is the order of the nodes (e.g., we considered the ordering c11 followed by c21, followed by c22, followed by c12) that will result in the smallest possible decision diagram corresponding to a decision tree. It is computationally expensive (NP-Complete). However, we can leverage different heuristics [73] that have been proposed to efficiently produce a “good” ordering of variables.

3.3 Method evaluation

As proved in Section 3.2 (Theorem 1), our technique does not have any false positives or false negatives (for the SQL queries syntax considered for translation). Additionally, we have claimed that our technique is likely to capture in a succinct fashion the core conditions (causal set) which, when satisfied by the user-inputs, will cause a SQL injection attack. In the following, we will use some sample examples to show that our claim holds true in practice.

The SQL query in Figure 3.1 contains four locations where user-inputs can affect the conditions. As each of the locations can take up one of three values (taut, cont and unin), there are 3^4 different test inputs. CREST can identify around 28 different injection-causing test cases (see Figure 3.3 for test cases corresponding to c12 = taut). However, our technique of reduction obtains only 4 different elements in the causal set. In short, our technique results in 85% reduction. Next, consider the SQL query in Figure 3.7

Similar to the previous example, this query also has four locations where user inputs affect the conditions; however, the dependencies between these locations are different from those in the previous

```
  SELECT deductible
  FROM policy as p
  WHERE inputPolicy = $input11 OR id = $input12
UNION
  SELECT d.insuredname
  FROM dependents as d
  WHERE inputPolicy = $input21 OR id = $input22
```

Figure 3.7 SQL query with UNION

example. CREST obtains thirteen different injection causing test cases, while our technique correctly identifies the causal set to contain cases where at least one of the locations result in a `taut` condition, and reduces that number to four (about 69% reduction).

In summary, our proposed technique has two main advantages. It does not produce any false positive or false negatives. It produces results that capture exactly the cause of SQL injection with respect to user inputs. The causal set is, therefore, precise and succinct, making it easier to monitor for injection-causing user inputs and also to take appropriate corrective measure in the event of an injection.

3.4 Conclusions

We have shown that our technique is at the same time more general and more precise than the existing techniques, as it relies on semantic dependencies between the conditions that are affected by user inputs. This method specifically focuses on SQL query vulnerabilities that are exploited by injections that lead to tautologies in query conditions.

CHAPTER 4. DETECTING CROSS-SITE SCRIPTING VULNERABILITY USING CONCOLIC TESTING

In the previous chapter we have presented a query level technique to analyze SQL queries and monitor for vulnerabilities. SQL queries are often embedded in a source language in a Web application. In this chapter, we present our application level testing tool approach that serves as the first line of defense in detecting vulnerabilities and informing selective instrumentation for runtime monitoring. Our two-phase techniques includes a translation phase followed by an instrumentation phase. The translation phase is compromised of translation and testing-based analysis. First, we statically identify input and output variables in the application. Next, the Web application is translated to the input language of a tester to determine the outputs that are likely to depend on the user inputs. Finally, in the instrumentation phase, monitors are inserted in the Web applications to check whether any of the outputs identified in the previous phase are indeed exploited by user inputs.

Our method is as efficient and effective as the available XSSD techniques. For testing we employ a concolic testing tool (jCute [44]) instead of complex code implementation of other conconlic testing based techniques [27, 37, 77, 18]. In addition to being both efficient and effective as the best available techniques, our framework is also capable of identifying XSS vulnerability conditions that occur due to the conditional copy (of inputs to outputs) and the concatenation of singularly benign input strings that form malicious strings. We present a prototype of the framework and demonstrate its effectiveness using a non-trivial JSP Web application.

The rest of this chapter is organized as follows. Section 4.1 briefly reviews the definition of XSS and introduces our technique. Section 4.2 describes our technique; especially (a) the identification of input and outputs for testing, (b) the translation of JSP to Java, and (c) concolic unit testing method. Section 4.3 presents the results of the evaluation of our method. Finally, Section 4.4 concludes the chapter.

4.1 Introduction to Cross-Site Scripting

Cross-Site Scripting attacks occur when a script is injected into an application and executed typically by the browser, granting a malicious user unauthorized access to system resources or sensitive infromation. First Order XSS attacks are successful only when certain vulnerabilities exist in the application; more precisely when such vulnerabilities remain unresolved. These vulnerabilities primarily involve allowing executable inputs from users to be directly or indirectly assigned the outputs of Web application without proper sanitization. As outputs of Web applications are executed by the browser, inputs that influence the outputs can inject unwanted potentially malicious codes that are executed. This classification of XSSAs aligns with the Code Injection Attack definition provided by Ray and Ligatti [78].

Research on XSS aims to find vulnerabilities and/or prevent attacks. One challenge in attack prevention is discerning which inputs will result in attacks. Some basic approaches have been deployed to detect and stop attacks by checking for script tags and same origin verification via HTML "`Referer`" header field[1] [34]. However, these techniques may not be effective as sophisticated attacks can easily circumvent tag-based detection, and HTML "`Referer`" field use is optional and possibly unavailable. In the recent past, more complex detection techniques [70, 40, 33, 37, 13, 43, 22] have been proposed and developed that are either based on static analysis or runtime monitoring.

Static analysis uses traditional program analysis techniques to identify vulnerable code segments via taint analysis and instruments these segments to avoid their exploitation at runtime. Typically, they are not suited to find exploitation of vulnerabilities resulting from conditional copy. Conditional copy is a technique used in Code-interference based injection attacks, wherein a variable value is transferred into another (e.g., character-by-character in a conditional code segment) with direct dependency of the copies or other data operations.

Runtime monitoring, on the other hand, relies on a library of attack patterns or specification of non-attack (allowable) patterns to detect potential attacks. As a result, runtime monitoring can incur prohibitively large overhead if non-vulnerable variables are unnecessarily monitored against attack patterns.

[1] The `Referer` HTML header field identifies the Web page address (URL) and is purposefully misspelled.

Our two-phase method employs both static analysis and runtime monitoring. The static analysis phase includes application translation and conolic unit testing technique. The runtime monitoring phase uses the facts learned from static analysis to monitor the relevant variables in the application. Any existing runtime monitoring method can be used in our runtime monitoring phase as long as it is coupled with the information generated in our static analysis phase. Therefore, we emphasize the static analysis phase.

We have developed a framework that implements our technique. Our framework takes as input Web applications written in JavaServer Pages (JSP), a prevalent application language that can contain HTML elements and embedded Java, in addition to language-specific objects and statements. Input in JSP includes HTML form input

(`<input type=[...] name="paramName">`)

which are accessible as output via JSP request objects

(`<%= request.getParameter("paramName") %>`)

and via JSP text boxes (`<%=paramName %>`). Other output can be found within embedded Java print statements (`<% out.print(ln) %>`). Our objective is to determine whether the inputs can become assigned to some outputs, identify these outputs (static analysis) and then deploy runtime monitoring to check the values of these outputs for attack patterns.

Our method is outlined as follows:

1. ***Automatic translation***. We convert the JSP application to a Java program by considering only the elements of JSP application that are relevant for determining XSS vulnerabilities. First, the JSP input and output variables are identified. Then, the JSP page is translated to a Java program, with additions of automatic input test case generation statements and assertions to be used to check for equality between inputs and outputs, and possible assignment of simple attack patterns (e.g., containing script tags: $< \ldots >$) to outputs.

2. ***Testing-based analysis***. We use the concolic testing tool jCUTE[2] to generate inputs and to test assertions. These assertions verify which inputs can contain script tag symbols and pass unsanitized (or improperly sanitized) to an output in the Java program. This implies the JSP page has a vulnerability based on the input/output pair and the affected outputs can be exploited at runtime.

[2] CUTE : A Concolic Unit Testing Engine for C and Java Automatic Test. [44]

3. ***Source code instrumentation***. Finally, we instrument the original JSP application and include calls to a Java class that monitors the values of affected outputs and whenever a potential XSS attack pattern is recognized at the outputs, the pattern is replaced with a benign string to avoid exploitation.

Contributions The contributions of our approach can be summarized as follows:

1. We propose a concolic testing based technique for detecting possible vulnerable outputs (those which are directly and indirectly assigned to from the user inputs and can contain scripting tags) followed by selective instrumentation for runtime XSS attack detection.
2. Being based on efficient concolic testing technique, our vulnerability detection method does not require expensive program analysis techniques.
3. Our technique detects the vulnerabilities caused by *conditional copy* that are typically not detected by existing techniques. Our solution can also detect attacks caused by concatenation of singularly benign inputs resulting in malicious output.
4. We have developed a prototype implementation of our technique for JSP applications and evaluated the effectiveness of our technique using real-life GotoCode applications.

4.2 A method for detecting Cross-Site Scripting vulnerabilities and implementing attack prevention

This section describes our method that is depicted in Figure 4.1.

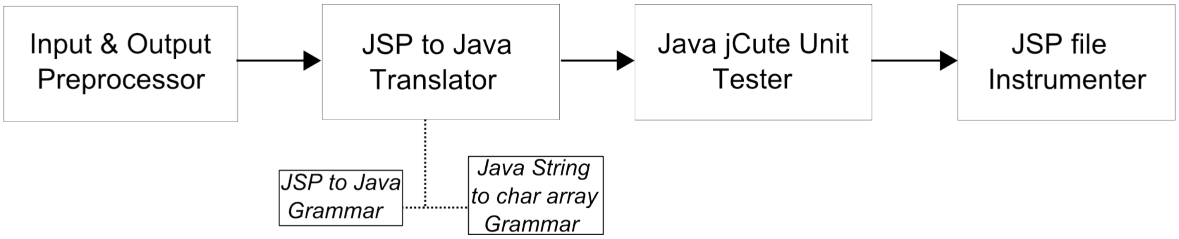


Figure 4.1 Approach overview

We use a an example to discuss the salient aspects of our technique, and to compare our technique with respect to the existing ones. Figure 4.2 shows a JSP application that displays a welcome page that

```
 1.  <%@ page import="java.util.*" %>
 2.  <%
 3.    String usrname = request.getParameter("username");
 4.    String uname = new String();
 5.    for (int i = 0; i < username.length(); i++){
 6.      if (username[i]==’a’) {uname[i]=’a’;}
 7.      else if (username[i]==’b’) {uname[i]=’b’;}
         ...
 8.    }
 9.    session.setAttribute("uName",uname);
10.  %>
11.  <HTML>
12.  <BODY>
13.  <BR>
14.  <HR>
15.  <div id ="WelcomeMessage"> WELCOME </div>
16.  <div id="Welcome<%=session.getAttribute("uName")%>">
17.  Welcome, <%=session.getAttribute("uName")%>
18.  </div>
19.  </BODY>
20.  </HTML>
```

Figure 4.2 Illustrative example JSP code: welcomePage.jsp

is displayed after a user login. We note the inputs and assignments contained in the application. In Line 3, the code assigns `usrname` user input from the HTTP `request` parameter `username`. It then copies `usrname` to `uname` character-by-character (Lines 5–8). Finally, the value of `uname` is set to a session attribute, `uName`, which is retrieved at Line 16. Thus, in this application, the input variable is `usrname` and the output is `uname`. The application can be exploited by assigning malicious input `usrname`, which gets directly assigned to `uname` via the conditional copy code segment. This is an example of conditional copy vulnerability. This illustrative example is the conditional copy in its simplest form. Web applications will not typically include a character-to-character copy; however, conditional copies may arise in more sophisticated versions. For example, a translation of an input string to some intermediate string may perform a conditional copy. Attackers can exploit such translations. Also, insufficient or inadequate sanitization methods give a false perception of security; they are also examples of conditional copy.

Our technique can efficiently detect such vulnerabilities.

We now describe our technique in detail.

4.2.1 Preprocessing

We statically process the JSP Web application one file at a time to identify the inputs and outputs of each page generated from the files. We define a grammar based on the application language to capture all variables, request and response parameters, session variables, tags, et cetera. We have enumerated constructs of the Web application langague JSP and their mapping to Java for our translation, in Figure 4.3.

```
JSP implicit object = char[] object name
JSP implicit variable  = char[] variable name
<%= expression %> = Java expression
<% scriptlet %> = Java only
<%! declarations %> = Java declarations
<%= expression %> = evaluate expression in Java context
<%= input %> = Java input variable and/or variable assignment
session.getAttribute("paramName"); = char[] paramNameInput = new char[SIZE];
session.setAttribute("paramName",name); = char[] paramName = name;
session.getParameter("paramName"); = char[] paramNameInput = new char[SIZE];
response.sendRedirect(string); = char [] output = string;
request.setParameter("paramName",name); = char[] paramName = name;
request.getParameter("paramName"); = char[] paramNameInput = new char[SIZE];
request.getQueryString(); = char[] querystringInput = new char[SIZE];
request.getQueryURI(); = char[] queryuriInput = new char[SIZE];
<%= output %> = Java output variable and System.out.println();
<%-- input %> = Java input variable and/or variable assignment
<%-- output %> = Java output variable and System.out.println();
<%! comment %> = Java comment  // or /* ... */
<!--  HTML comment >  = Java comment  // or /* ... */
<%@ include > = map include file as dependent file for inclusion
<HTML identifier> = System.out.println("HTML identifier>");
```

Figure 4.3 Mapping for JSP to Java translation

Any information that the application expects from other pages within the application (e.g., session variables that are passed between pages in a session) is also identified as input since we treat the JSP file as an atomic unit. This aligns with Wu and Offutt [79]; the authors consider "HTML file or section of a server program that prints HTML" as an atomic section in modeling and testing Web-based applications.

In our example in Figure 4.2, the input from the user is the value associated to `username` at Line 3 (`3.  String usrname = request.getParameter("username");`), which is assigned to the application input `usrname`. The output in our example is `uName`, used inside tags at Lines 16 and 17 shown below.

```
16.  <div id="Welcome<%=session.getAttribute("uName")%>">
17.  Welcome, <%=session.getAttribute("uName")%>
```

4.2.2 Translation

Once the input and output variables in an application file are detected during preprocessing, the translation of application to Java is performed. Note that, the result of translation, the Java program, maintains the following aspects of the original JSP application: the relationships between program variables, inputs and outputs, and control flow of the application. All other elements of JSP application, that relates to generating HTML page are discarded, they are either commented or placed in print statements in the result of translation. Additionally, our method requires a Java String to be represented as a character array, as concolic testing will not work with String. Figure 4.4 shows the String to character array mapping for variables and Java String methods.

We show the full translation of our example application from Figure 4.2 as the resulting Java code in Figure 4.5. The input, output and the JSP application variables are also present in the Java program. Additionally, the string variables are converted to character arrays as concolic testing cannot handle strings; however, this does not result in any over-approximation or under-approximation in the context of vulnerability detection. Lines 21–31 encode a constructor which mimics external input (in this case `username` parameter in JSP application). Lines 22–26 represent the translation of Line 3 of JSP application in Figure 4.2. Lines 36–40 is the translation of the conditional copy of the JSP application and Line 41 corresponds to assignment of `uname` to session variable `uName` in the JSP application. Identified output variables are also instrumented during translation. Lines 16–17 of Figure 4.2 correspond to Lines 46–50. The output is printed in Lines 46–47 and tested in Lines 48-50. Thus we have a test file with test case generation and testing code for the next step, testing.

```
str = { varStr | "[^null]" | " " | "null" | obj.toString() }
obj = { Object | char[] | strBuffer | boolean | char | double | float | int | long }
char[] = { char_char[] | char }
str.charAt( int )==>  {  (charArrVar.toString()).charAt(int) | " ".charAt() }
str.compareTo( obj )==> { compareCharArr(char[], *.toCharArray())}
str.compareTo( str )==> { str.compareTo(str) }
str.compareToIgnoreCase( str )==> { compareCharArrIgnoreCase(char[],char[]) }
str.concat( str )==> { char[] = concatCharArrs(char[],char[]).toCharArray(); }
str.contentEquals( strBuffer )==> { ContentEquals (char[], strBuffer.toString().toCharArray()) }
str.copyValueOf( char[] )==> { char[] = CopyValueOf(char[]) }
str.copyValueOf( char[], int, int )==> { char[] = CopyValueOf(char[],int1,int2) }
str.endsWith( str )==> { endsWithCharArr(char[], char[]) }
str.equals( obj )==> { Arrays.equals(char[], char[]) }
str.equalsIgnoreCase( str )==> { equalsIgnoreCaseCharArrs(char[], char[]) }
str.getBytes( _ )==> { getBytesCharArr(char[]) }
str.getBytes( int, int, byte[], int )==> { getBytesCharArr(int, int, byte[], int, char[]) }
str.getBytes( str )==> { getBytesCharArr(char[]) }
str.getChars( int, int, char[], int)==> { getCharsCharArrs(int,int,char1[],int, char2[]) }
str.hashCode( _ )==> { (char[].toString()).hashCode( _ ) }
str.indexOf( int )==> { (char[].toString()).indexOf( int ) }
str.indexOf( int, int )==> { (char[].toString()).indexOf( int, int ) }
str.indexOf( str )==> { (char[].toString()).indexOf( char[].toString() ) }
str.indexOf( str, int)==> { (char[].toString()).indexOf( char[].toString(), int} ) }
str.intern( _ )==> { (char[].toString()).intern( _ ) }
str.lastIndexOf( int )==> { (char[].toString()).lastIndexOf( int ) }
str.lastIndexOf( int, int )==> { (char[].toString()).lastIndexOf( int, int ) }
str.lastIndexOf( str )==> { (char[].toString()).lastIndexOf( char[].toString() ) }
str.lastIndexOf( str, int )==> { (char[].toString()).lastIndexOf( char[].toString(), int ) }
str.length( _ )==> { CHAR_LENTGH | (char[].toString()).length() }
str.matches( str )==> { (char[].toString()).matches( str) |
           (char[].toString()).matches(char[].toString()) }
str.replace( char, char | CharSequence target )==> { Replace(char, char, char[]) }
str.replace( CharSequence target, CharSequence replacement )==>
           { Replace(CharSequence, CharSequence, char[]); }
str.startsWith( str )==> { startsWithCharArr(char[], char[]) }
str.startsWith( str, int )==> { startsWithCharArr(char[], char[], int) }
str.subSequence( int, int )==> { subSequenceCharArr(int1, int2, char[]) }
str.substring( int | int, int )==> { substringCharArr(int, char[]) }
str.substring( int, int )==> { substringCharArr(int1, int2, char[]) }
str.toCharArray( _ )==> { char[] | str.toCharArray() }
str.toLowerCase( _ )==> { toLowerCaseCharArr(char[]) }
str.toString( _ )==> { char[].toString() | str.toString() }
str.toUpperCase( _ )==> { toUpperCaseCharArr(char[]) }
str.trim( _ )==> { (charArrVar.toString()).trim() }
str.valueOf( boolean )==> { (char[] = (String.valueOf( boolean )).toCharArray() }
str.valueOf( char )==> { (char[] = (String.valueOf( char )).toCharArray() }
str.valueOf( char[] )==> { char[] = (String.valueOf(char[])).toCharArray() |  char[] }
str.valueOf( char[], int, int )==> { char[] = (String.valueOf(char[],int,int)).toCharArray() }
str.valueOf( double )==> { (char[] = (String.valueOf( double )).toCharArray() }
str.valueOf( float )==> { (char[] = (String.valueOf( float )).toCharArray() }
str.valueOf( int )==> { char[] = (String.valueOf( int )).toCharArray()
str.valueOf( long )==> { char[] = (String.valueOf( long )).toCharArray() }
str.valueOf( Object )==> { char[] = (String.valueOf( Object )).toCharArray() }
```

Figure 4.4 Grammar for adapting Java String to char arrays

```
welcomePage.java:
// begin headers
import java.io.*;
    . . .
import cute.Cute;
//compiler directives from translated file
// end headers
// begin mainClass
public class welcomePage {
   // begin members
   public static javax.servlet.http.HttpServletRequest request;
   public static javax.servlet.http.HttpServletResponse response;
   public static javax.servlet.http.HttpSession session;
   public static javax.servlet.jsp.JspWriter out;
   public static int STR_SIZE= 5;
   public static char[] usrname ;
   public static char[] uname ;
   public static char[] username ;
   public static char[] uName;
   // end members
   // begin constructor
   public welcomePage() {
      username = new char[STR_SIZE];
      for (int k=0; k < STR_SIZE; k++) {
         username[k] = cute.Cute.input.Character();
         if (username[k] == ’\0’) username[k] = ’ ’;
      }
      usrname = new char[STR_SIZE];
      for (int i=0; i< STR_SIZE; i++){ usrname[i]=username[i]; }
      uname= new char[STR_SIZE];
      uName = new char[STR_SIZE];
   }
   ...
   public static void main(String[] args) {
      // call default constructor
      welcomePage mywelcomePage = new welcomePage();
      for (int i=0; i < STR_SIZE; i++) {
                 if (usrname[i] == ’a’) { uname[i]= ’a’; }
                 else if (usrname[i] == ’b’) { uname[i]= ’b’; }
                 ...
      }
      for (int i=0; i< STR_SIZE; i++){ uName[i]=uname[i]; }
      System.out.println(" <HTML>\n");
      System.out.println(" <BODY>\n");
      System.out.println(" <BR>\n");
      System.out.println(" <div id =WelcomeMessage> WELCOME </div>\n");
      System.out.println("<div id=\"Welcome"+uName.toString()+"\">");
      System.out.println("Welcome, "+uName.toString()+"");
      if (uName[0] ==‘<’) {
              System.out.print("check uName: ");
              Cute.Assert(uName[3] !=‘>’); }
      System.out.println(" </div>\n");
      System.out.println(" </BODY>\n");
      System.out.println(" </HTML>\n");
   }// end main
}//end of Mainclass
```

Figure 4.5 Illustrative example code converted to Java: welcomePage.java

4.2.3 Testing for determining vulnerable outputs

For testing, we use the concolic testing tool jCute [44] to automatically generate input test cases that determine vulnerable outputs. These outputs are directly or indirectly assigned from inputs and can contain scripting tags. Recall the description of concolic testing from Chapter 3 as follows. Concolic testing utilizes a combination of concrete and symbolic execution to generate test cases and to effectively guide exploration of the new program paths, respectively. In concolic testing, a random test case is generated and the test program is executed with that concrete value. When the program encounters a conditional statement (e.g., instrumented assertions) these become the symbolic constraints. The concrete and symbol constraints are solved simultaneously to create a new test value and to continue creating new branches to test, until no new branches can be created. As long as the branches are finite in number and finite in length, i.e., not recursively infinite, we are assured extensive branch coverage with this testing technique. There will be inserted assertions when there are output variables and there may be conditionals from the code that serve as symbolic constraints. Our method relies on careful placement of assertion statements and identifies test cases that can violate the assertions.

We check whether the output of the Java program can evaluate to a sequence of characters representing a script. In other words, we need to check whether the sequence contains a "<" proceeded by a ">". The finite state automaton (FSA) in Figure 4.6 represents such a sequence; zero or more characters followed by "<", followed by a finite sequence of characters, followed by ">", and finally, followed by zero or more characters.

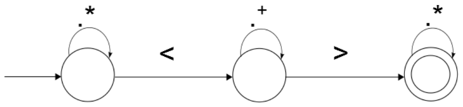


Figure 4.6 Finite state automaton representing vulnerable output

The concolic testing tool jCute can reason about characters and relationships between characters (equality, inequality, ordering); that is, it cannot check for the inclusion of a sequence of characters in a regular language (expressed as the FSA in Figure 4.6). However, one can check the presence of "<" at the beginning and ">" at the end of a character sequence where the sequence is at least three characters long. Such checking is sufficient to detect whether some variable in Java application being tested by the jCute engine can be assigned to a string that is accepted by the FSA in Figure 4.6.

Proof. The proof relies on two facts

1. The valuation of output variables in the Java program comes from an input variable or from concatenation of two or more input variables (we discard the role of program variables that are statically assigned in the program and replace them with empty strings).

2. The smallest string accepted by the FSA in Figure 4.6 is of length three, and starts and ends with "<" and ">", respectively.

Therefore, if jCute fails to identify any test case which leads to an output variable assigned to a sequence of three characters accepted by the FSA in Figure 4.6, then it will also fail to identify any test case which will result in the same output variable to be assigned a sequence of four or more characters accepted by the FSA in Figure 4.6. Conversely, if an output can be assigned to a sequence of four or more characters accepted by FSA in Figure 4.6, then jCute can identify a test case that will lead to the same output variable assigned to a sequence of three characters accepted by the FSA. □

In our translated test program in Figure 4.5, Lines 48–50 include the assertion which is violated only when the output variable `uName` (which was converted from a JSP session attribute) in the JSP application (Figure 4.2) holds a sequence of characters of length three, and starts and ends with "<" and ">", respectively. This violation indicates a vulnerability that can lead to XSS, i.e. an input containing script symbols can be assigned to the output variable, directly or indirectly. After testing, with these vulnerable output variables identified, we are now ready to selectively instrument our original JSP Web application with runtime monitors.

4.2.4 Instrumentation for detecting Cross-Site Scripting attacks

Applying the jCute testing engine, we automatically obtain test cases for which some output variable can be assigned to a three-character sequence starting and ending with "<" and ">", respectively. These are the outputs that can be exploited by injection attacks, and therefore, their valuation must be monitored at runtime. We refer to these outputs as vulnerable outputs. Note that, there can be many outputs in the application and a small number of them may be vulnerable. By utilizing jCute to find these vulnerable outputs, we minimize the number of outputs that need to be monitored in the Web application.

We instrument the original JSP application by inserting calls to a monitor module, which checks the valuation of the vulnerable outputs at program points right before they are being used in the application. Any attack detection engine could be deployed as a monitor; we have created a simple monitor. Our monitor checks for scripting elements involving `HTTP`, <`script`>, `javascript:`, `document.cookie`, `document.location`, <`img src`, and <`iframe`. If an output matches a scripting pattern, a benign error-message string replaces the malicious output, thereby neutralizing the attack.

In our example, `uName` is the vulnerable output. We add a call to the monitor method before Line 9 of Figure 4.2.

4.3 Case Studies

We use open-source JSP applications from GotoCode[3] to evaluate the feasibility and effectiveness of our technique. JSP applications consist of multiple JSP files. We manually identify the dependencies between the JSP files; dependency can be easily determined based on the exchange of information between JSP files (via session variables, request parameter

[3] http://www.gotocode.com

Table 4.1 GotoCode Projects Tested

Project Name	Number of Files Tested	Number of Files w/ Vulnerable Outputs
Online BookStore	24	9
Bug Tracking System	12	1
Employee Directory	6	5
Events	9	3
Forum	4	1
Ledger	2	0
Online Portal	23	8
Yellow Pages	8	5

forwarding, etc.). Each file is then analyzed by our technique individually and vulnerable outputs are determined.

We have selected eight projects from GotoCode to evaluate our technique. Table 4.1 presents the project names and the number of files that are analyzed in each project. The table also shows the number of files for each project in which vulnerable outputs are detected.

Table 4.2 presents a detailed result for one of the projects. The second column presents the known vulnerable variables as noted in [57]. The third column presents the vulnerable variables that are identified by our technique. Note that, our technique identifies variables as vulnerable that are previously undetected (as per [57]). Two types of timing results (in seconds) are reported. `C-Time` corresponds to the time for translating JSP application page to instrumented Java program with jCute test variables, and compiling the Java program using jCute test engine. `E-Time` corresponds to the execution of compiled Java program in jCute and identifying all the vulnerable output variable (i.e. identifying the test case/input that leads to failure of assertion).

Lines of code are included in the final column for jsp and converted java file, respectively. They demonstrate that the translation includes the string to character array conversion methods that needed since concolic testing cannot handle strings, but can handle characters. The number

Table 4.2 Online Bookstore Variables

Filename (.jsp)	Vulnerable Variable(s)	Identified Variable(s)	In	Out	C-Time	E-Time	LOC
BookDetail	itemid	itemid	17	10	13.84	84.47	806/1647
CategoriesRecord	category_id	category_id	5	7	7.79	23.38	295/1024
Login	ret_page, querystring	ret_page, querystring	6	8	6.41	8.45	174/917
OrdersRecord	order_id	order_id, member_id, item_id	9	6	9.37	159.01	360/1122
ShoppingCartRecord	order_id	order_id	6	13	8.63	10.33	322/1061

of input and output variables add to the translated line count, as the translated Java file must contain test generation and assertion code, respestively.

Timing is also included in Table 4.2. We show the average of five runs on each file. E-Time shows the time to find all identified variables in a file. We note that E-Time for Login and OrdersRecord is for multiple variables, thus we consider the average. Conditional statements represent constraints to solve in concolic testing, and thus the number of conditionals influence E-Time. In order from most to fewest conditionals is BookDetail, OrdersRecord, Categories Record, ShoppingCartRecord, and Login which is also the same E-Time ordering to find one vulnerable variable.

4.4 Conclusions

We have presented a method for identifying XSS vulnerability and detecting their exploitation. At its core, our technique relies on smart translation of Web applications and deploying a unit testing engine to determine vulnerabilities, and finally applying code instrumentation to monitor attacks that exploit those vulnerabilities. We have shown that our technique is efficient and can easily identify conditional copy vulnerabilities.

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

5.1 Summary and contributions

First, we have provided a comprehensive classification and review of a representative body of FOID methods that provides a road map of existing research and a foundation for future works. Next, we have proposed techniques for the detection of First Order SQLI and XSS and implemented a language-specific framework to evaluate these techniques. Our SQLI method is at the query level and our XSS method is at the application level, respectively. Our algorithms employ concolic testing with test case generation to find vulnerabilities.

Our query-based SQLID technique introduced a new approach that analyzed the semantics dependencies between conditions of a query, thus the technique did not rely solely on the query's syntactic structure. We employed a concolic testing tool to generate query condition values and test which combinations of condition values could lead to an attack, creating various attack cases. We also provided a novel technique to reduce the various cases and summarize them automatically into a causal set. Our work specifically addresses vulnerabilities in a query that can lead to tautology-based attacks. Since our technique is based on semantic dependencies, it does not produce false negative or false positive results. Also, with its reliance on semantic dependencies between the conditions that can be affected by user input, is both more general and more precise than the existing techniques.

The future plan is to develop a complete framework and empirically evaluate the strengths of our technique using real-life SQL queries. This will also include the investigation of various decision diagram construction and minimization heuristics, and identification of the ones that best suit our purpose. Especially, it will be interesting to take into consideration Multi-valued

Decision Diagram Library [80] that allows for representation of logical formulas over variables with any size (finite) domain and has several optimized reduction algorithms.

In the case of XSS, we have implemented a concolic testing based technique for detecting possible vulnerable outputs. These outputs guide selective instrumentation in the Web application code for runtime XSS attack detection. Since we use efficient concolic testing, our technique does not require expensive program analysis techniques. Our method detects the conditional copy vulnerability (character-by-character copy in a conditional code segment). Furthermore, our method detects attacks caused by concatenation of singularly benign inputs that result in malicious output. The prototype implementation of our technique translates JSP applications to the test language (Java). We have evaluated its effectiveness on real-life applications.

Since our XSS method was implemented for the Web application programming language JSP, additional translators for other domain-specific languages (DSLs) are natural extensions to the framework. Some Web-based DSLs include the following: PHP, Perl, Python, Ruby, Java, JavaScript, AJAX, and ASP.NET(C#, VB.NET). The future plan is to extend this technique to detect and prevent different types of injection attacks (e.g., Second Order SQLI) in Web applications written in other languages. The overall objective is to develop a generic framework that allows grammar-based automatic translation of Web applications written in any language into intermediate test-language, and automatic instrumentation of Web applications based on the results of testing the test-language. Another possible avenue for future work is the implementation of our algorithm with a new test language and new concolic testing tool or technique.

Our framework could be deployed on a Web server to test and refactor deployed or deployment-ready Web application code. A possible pre-deployment application could be an integrated development environment (IDE) plug-in for vulnerability testing. The framework in its current configuration could be extended to also detect Second Order SQLI and XSS. We describe this Second Order Injection vulnerability detection extension in the following section.

5.2 Extension to Second Order Injection Attacks

The framework detailed in this thesis detects First Order Injection vulnerabilities, specifically SQLI using a query level technique and XSS using an application level technique. Both attacks have Second Order versions which have been researched less frequently than their First Order counterparts due to their complexity. Existing works are lacking extensive definitions and descriptions of Second Order Injection Attacks; however, white papers [81, 82, 83] explain the a priori storage of malicious data, the subsequent retrieval of that data and the ensuing attack on unsuspecting victims with some illustrative examples. These papers serve as the primary source for papers directly addressing Second Order attacks specifically [13] and all SQLI [4]. We note that while this description is one possible scenario of a Second Order attack, other scenarios exist. Our aim is to provide an encompassing Second Order Injection attack definition along with detailed scenario explanations and propose a concolic testing-based technique for the detection of Second Order Injection Attack vulnerabilities.

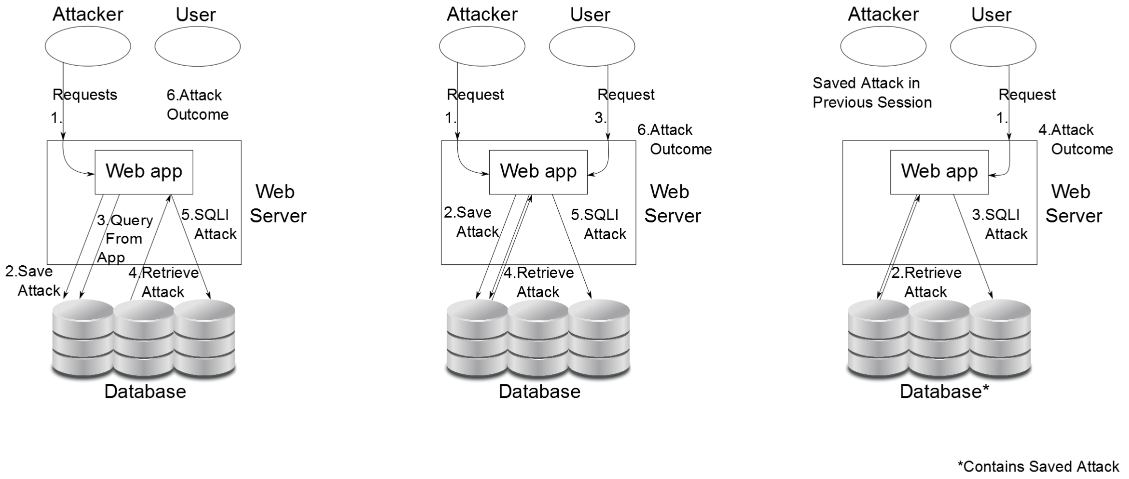
We define a Second-Order Injection Attack as an injection of malicious data that is first stored, then retrieved, and finally propagated to compromise user and/or system resources. We first address Seond Order SQL Injection Attacks (SQLIA2), wherein the event propagated and responsible for the security breach is a query injected with the malicious data. We further classify SQLIA2 that use the database for persistent storage in the following scenarios:

1. Direct Single Injection Attack (DSIA): In this type of attack, an attacker uses a single query to inject into the database malicious data that is used in a subsequent query called by the web application that is not deployed by the attacker to launch a successful attack. See Figure 5.1 (a).

2. Direct Multiple Injection Attack (DMIA): In this type of attack, an attacker uses one query to inject into the database malicious data that is used in a subsequent query called by the attacker to launch a successful attack. See Figure 5.1 (b).

3. Indirect Injection Attack (IIA): In this type of attack, an attacker uses a single query to inject into the database malicious data that is used in a subsequent query called by an

innocent victim to launch a successful attack. See Figure 5.1 (c).



(a) Direct Single Injection (b) Direct Multiple Injection (c) Indirect Injection Attacks

Figure 5.1 (a) Direct Single Injection Attack; (b) Direct Multiple Injection Attacks; (c) Indirect Injection Attacks

Typically two vulnerabilities are exploited in a second order attack, one during storage of the malicious data and one during the propagation of the attack. A few works that address First Order SQLIA described in Section 2.1 can also address SQLIA2 or some scenarios of SQLIA2 [8, 13, 4, 15]

For correctly detecting SQLIA2, a method would need to track the malicious data into the database and also track it from the database into the application at runtime. This is difficult when SQLIA2 spans more than one session, requiring expensive tracking and possibly demarcation or extraneous storage of the malicious data. Methods which detect First Order SQLIA attacks may still find SQLIA2 propagated attacks [8, 15]. As long as the attack itself is detected and thwarted, its classification as First or Second Order is inconsequential; however, First Order vulnerability detection may not discover Second Order vulnerabilities.

For Second Order SQLIV detection, expensive and extensive analysis similar to that used in Second Order SQLIA detection is required. However, we propose a method that detects the two vulnerabilities when exploited together can lead to a propagated SQLIA2. Based on the First Order XSSV detection method in Chapter 4, this method will provide a static analysis of a Web application to detect vulnerable spots which will inform the instrumentation of runtime attack monitoring. Thus, this approach will statically detect Second Order SQLIV (SQLIV2).

Our proposed method is outlined as follows:

1. ***Preprocessing***: The application is statically analyzed to identify two types of input variables, one into the application (`input`) and one into the database (`input to db`), and two types of output variables, one from the database for use in the application (`output from db`) and one from the application for rendering in the browser (`output`).

2. ***Translation***: The application is translated from its original language to the testing language, with previous identified `input` variables coded for automatic test case generation, and `input to db`, `output from db` and `output` variables coded for testing for equality to other variables.

3. ***Testing***: The translated test file is executed to determine the following equalities: `input` = `input to db`, `input to db` = `output from db`, and `output from db` = `output`. These equalities imply that sanitization methods were inadequate or were not implemented in the application, allowing potentially malicious data to pass unaltered from one input or output variable to the next. These variable pair equalities indicate vulnerability for storage of an attack (`input` = `input to db`) and some attack scenarios (combinations of variable pair equalities).

4. ***Instrumentation***: Finally, the original Web application will be instrumented to allow for runtime monitoring of vulnerable variables.

Figure 5.2 illustrates the first three steps of the proposed method. In the top box, as described in the preprocessing step, input and output variables are identified: `input`, `input to db`, `output from db` and `output`. Next is translation from the Web application language to the testing language as depicted in the second box. Following translation is testing. In

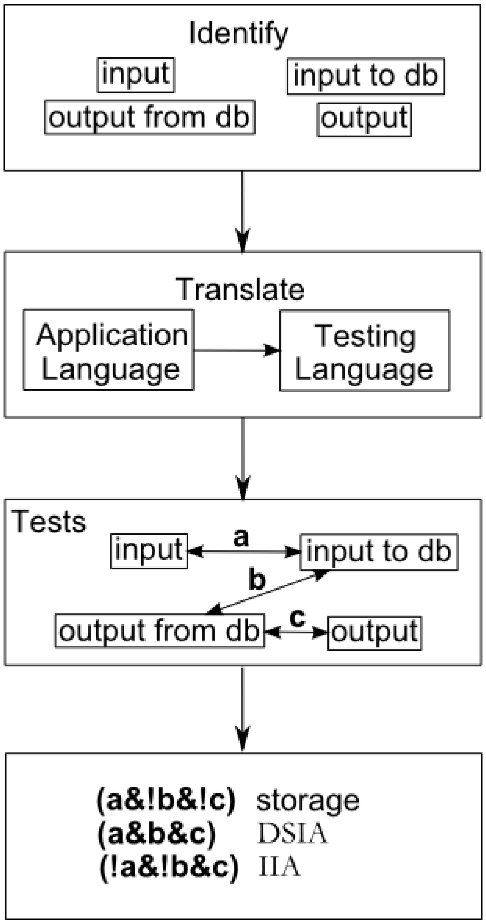


Figure 5.2 Second Order SQL Injection Vulnerability Detection

the third box, arrows indicate equality tests that occur between the variables. The final box lists expression based on the tests that, when satisfied, indicate SQLIV2. The equalities are labeled as follows: `a` : (`input` = `input to db`), `b` : (`input to db` = `output from db`), and `c` : (`output from db` = `output`). Storage occurs when data passes unaltered from an `input` variable to a `input to db` variable, represented by the expression (`a` & !`b` & !`c`). When a DSIA vulnerability is found, all the variables are equal (`a` & `b` & `c`), indicating data passing from `input`, being stored in the database, being retrieved from the database, and finally to `output`. A vulnerability to previous stored malicious data occurs when output of the database can reach the browser (!`a` & !`b` & `c`).

Using the basic components of SQLIA2 (1. Storage, 2. Retrieval and 3. Propagation), we have classified three basic types of Second-Order SQL Injection Attacks: Direct Single Injection, Direct Multiple Injection and Indirect Injection Attacks and proposed a method for Second Order SQLI vulnerability detection.

BIBLIOGRAPHY

[1] "OWASP 2010 top ten," 2010. [Online]. Available: http://www.owasp.org

[2] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 645–654. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999468

[3] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting sql injection vulnerabilities," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, july 2007, pp. 87 –96.

[4] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: preventing sql injection attacks using dynamic candidate evaluations," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 12–24. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315249

[5] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*, ser. WWW '03. New York, NY, USA: ACM, 2003, pp. 148–159. [Online]. Available: http://doi.acm.org/10.1145/775152.775174

[6] W. Halfond, A. Orso, and P. Manolios, "Wasp: Protecting web applications using positive tainting and syntax-aware evaluation," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 65 –81, jan.-feb. 2008.

[7] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 175–185. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181797

[8] W. G. J. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101935

[9] M. Bravenboer, E. Dolstra, and E. Visser, "Preventing injection attacks with syntax embeddings," in *Proceedings of the 6th international conference on Generative programming and component engineering*, ser. GPCE '07. New York, NY, USA: ACM, 2007, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1289971.1289975

[10] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '08. New York, NY, USA: ACM, 2008, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1328408.1328410

[11] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 365–383. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094840

[12] M. Johns and C. Beyerlein, "Smask: preventing injection attacks in web applications by approximating automatic data/code separation," in *Proceedings of the 2007 ACM symposium on Applied computing*, ser. SAC '07. New York, NY, USA: ACM, 2007, pp. 284–291. [Online]. Available: http://doi.acm.org/10.1145/1244002.1244071

[13] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” in *ICSE*. IEEE, 2009, pp. 199–209.

[14] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, and Y. Takahama, “Sania: Syntactic and semantic analysis for automated testing against sql injection,” in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, dec. 2007, pp. 107 –117.

[15] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, “Using parse tree validation to prevent sql injection attacks,” in *Proceedings of the 5th international workshop on Software engineering and middleware*, ser. SEM ’05. New York, NY, USA: ACM, 2005, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/1108473.1108496

[16] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in *Proceedings of the 13th international conference on World Wide Web*, ser. WWW ’04. New York, NY, USA: ACM, 2004, pp. 40–52. [Online]. Available: http://doi.acm.org/10.1145/988672.988679

[17] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, “Context-sensitive program analysis as database queries,” in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS ’05. New York, NY, USA: ACM, 2005, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1065167.1065169

[18] F. Yu, M. Alkhalaf, and T. Bultan, “Patching vulnerabilities with sanitization synthesis,” in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 251–260.

[19] S. Thomas, L. Williams, and T. Xie, “On automated prepared statement generation to remove sql injection vulnerabilities,” *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 589–598, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.08.002

[20] M. Johns, C. Beyerlein, R. Giesecke, and J. Posegga, “Secure code generation for web applications,” *Engineering Secure Software and Systems*, pp. 96–113, 2010.

[21] S. W. Boyd and A. D. Keromytis, “Sqlrand: Preventing sql injection attacks,” in *ACNS*, ser. Lecture Notes in Computer Science, M. Jakobsson, M. Yung, and J. Zhou, Eds., vol. 3089. Springer, 2004, pp. 292–302.

[22] M. Martin and M. S. Lam, “Automatic generation of xss and sql injection attacks with goal-directed model checking,” in *Proceedings of the 17th conference on Security symposium*, ser. SS’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 31–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496714

[23] F. Yu, M. Alkhalaf, and T. Bultan, “Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses,” in *Automated Software Engineering, 2009. ASE ’09. 24th IEEE/ACM International Conference on*, nov. 2009, pp. 605 –609.

[24] Y. Shin, L. Williams, and T. Xie, “Sqlunitgen: Test case generation for sql injection detection,” *North Carolina State University, Raleigh Technical report, NCSU CSC TR*, vol. 21, p. 2006, 2006.

[25] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in web applications using dynamic test generation and explicit-state model checking,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 474–494, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.31

[26] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, “Verifying web applications using bounded model checking,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 199–. [Online]. Available: http://dl.acm.org/citation.cfm?id=1009382.1009735

[27] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, “Dynamic test input generation for web applications,” in *ISSTA*, 2008, pp. 249–260.

[28] M. Ruse, T. Sarkar, and S. Basu, “Analysis & detection of sql injection vulnerabilities via automatic test case generation of programs,” in *Proceedings of the 2010 10th*

IEEE/IPSJ International Symposium on Applications and the Internet, ser. SAINT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 31–37. [Online]. Available: http://dx.doi.org/10.1109/SAINT.2010.60

[29] K. Kemalis and T. Tzouramanis, "Sql-ids: a specification-based approach for sql-injection detection," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 2153–2158. [Online]. Available: http://doi.acm.org/10.1145/1363686.1364201

[30] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 571–572. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321746

[31] "Facebook vulnerable to xss. over 70 million users are at risk," http://www.xssed.com/news/69/Facebook_vulnerable_to_XSS._Over_70_million_users_are_at_risk.

[32] "Facebook blames porn attack on browser." [Online]. Available: http://www.informationweek.com/news/security/attacks/231903115

[33] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," in *AINA (1)*. IEEE Computer Society, 2004, pp. 145–151.

[34] E. Kirda, C. Krügel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *SAC*, H. Haddad, Ed. ACM, 2006, pp. 330–337.

[35] J. Garcia-Alfaro and G. Navarro-Arribas, "Prevention of cross-site scripting attacks on current web applications," in *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II*, ser. OTM'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1770–1784. [Online]. Available: http://dl.acm.org/citation.cfm?id=1784707.1784768

[36] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *WWW*, C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, Eds. ACM, 2007, pp. 601–610.

[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *NDSS*. The Internet Society, 2007.

[38] E. Athanasopoulos, V. Pappas, and E. Markatos, "Code-injection attacks in browsers supporting policies," in *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, 2009.

[39] M. J. Stephen, P. P. Reddy, C. D. Naidu, and C. Rajesh, "Prevention of cross site scripting with E-Guard algorithm," *International Journal of Computer Applications*, vol. 22, no. 5, pp. 30–34, May 2011, published by Foundation of Computer Science.

[40] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense," in *NDSS*, 2009.

[41] M. Ter Louw and V. Venkatakrishnan, "BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers," in *Security and Privacy, 2009 30th IEEE Symposium on*, May 2009, pp. 331 –346.

[42] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 258–263. [Online]. Available: http://dx.doi.org/10.1109/SP.2006.29

[43] M. V. Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks," in *NDSS*, 2009.

[44] K. Sen and G. Agha, "CUTE and jCUTE: concolic unit testing and explicit path model-checking tools," in *Proceedings of the 18th international conference on Computer*

Aided Verification, ser. CAV'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 419–423. [Online]. Available: http://dx.doi.org/10.1007/11817963_38

[45] "2012 has delivered her frist giant data breach." [Online]. Available: http://www.teamshatter.com/topics/general/team-shatter-exclusive/2012-has-delivered-her-first-giant-data-breach

[46] "Yahoo hack leaks 453,000 voice passwords." [Online]. Available: http://www.informationweek.com/news/security/attacks/240003587

[47] "Linkedin investigating user account password breach." [Online]. Available: http://searchsecurity.techtarget.com/news/2240151334/LinkedIn-investigating-user-account-password-breach

[48] "Music site last.fm joins the password-leak parade." [Online]. Available: http://www.pcworld.com/article/257178/music_site_lastfm_joins_the_passwordleak_parade.html

[49] "Some eharmony user information stolen: An ancillary advice site that uses eharmony user names and passwords was hacked using an sql injection vulnerability." [Online]. Available: http://news.cnet.com/8301-1009_3-20031460-83.html

[50] "Formspring disables user passwords in security breach." [Online]. Available: http://news.cnet.com/8301-1009_3-57469944-83/formspring-disables-user-passwords-in-security-breach/

[51] "National vulnerability database." [Online]. Available: http://nvd.nist.gov/home.cfm

[52] "National vulnerability database, cve-2001-1460: Sql injection vulnerability in article.php in postnuke 0.62 through 0.64 allows remote attackers to bypass authentication via the user parameter." accesssed: 2012. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2001-1460

[53] "National vulnerability database, cve-1999-1167: Cross-site scripting vulnerability in third voice web annotation utility allows remote users to read sensitive data and generate fake

web pages for other third voice users by injecting malicious javascript into an annotation." [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-1999-1167

[54] U. S. C. E. R. Team, "Vulnerability note database: Vulnernability note vu#672683." [Online]. Available: http://www.kb.cert.org/vuls/id/672683

[55] ——, "Vulnerability note database: Vulnernability note vu#642339." [Online]. Available: http://www.kb.cert.org/vuls/id/642239

[56] ——, "Vulnerability note database: Vulnernability note vu#560659." [Online]. Available: http://www.kb.cert.org/vuls/id/560659

[57] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks," in *PRDC*. IEEE Computer Society, 2007, pp. 365–372.

[58] N. Antunes and M. Vieira, "Benchmarking vulnerability detection tools for web services," in *Web Services (ICWS), 2010 IEEE International Conference on*. IEEE, 2010, pp. 203–210.

[59] D. Kindy and A. Pathan, "A survey on sql injection: Vulnerabilities, attacks, and prevention techniques," in *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, June 2011, pp. 468 –471.

[60] R. Johari and P. Sharma, "A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection," in *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*, may 2012, pp. 453 –458.

[61] D. Jayamsakthi Shanmugam, "Cross site scripting-latest developments and solutions: A survey," *Int. J. Open Problems Compt. Math*, vol. 1, no. 2, 2008.

[62] A. B. M. Ali, A. I. Shakhatreh, M. S. Abdullah, and J. Alostad, "Sql-injection vulnerability scanning tool for automatic creation of sql-injection attacks," *Procedia Computer Science*, vol. 3, no. 0, pp. 453 – 458, 2011, ¡ce:title¿World Conference on

Information Technology¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910004515

[63] A. Ciampa, C. A. Visaggio, and M. 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*, ser. SESS ’10. New York, NY, USA: ACM, 2010, pp. 43–49. [Online]. Available: http://doi.acm.org/10.1145/1809100.1809107

[64] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, “Sqlprob: a proxy-based architecture towards preventing sql injection attacks,” in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC ’09. New York, NY, USA: ACM, 2009, pp. 2054–2061. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529737

[65] Z. Su and G. Wassermann, “The essence of command injection attacks in web applications,” in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’06. New York, NY, USA: ACM, 2006, pp. 372–382. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111070

[66] X. Li, W. Yan, and Y. Xue, “Sentinel: securing database from logic flaws in web applications,” in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ser. CODASPY ’12. New York, NY, USA: ACM, 2012, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133605

[67] N. Jovanovic, C. Kruegel, and E. Kirda, “Precise alias analysis for static detection of web application vulnerabilities,” New York, NY, USA, pp. 27–36, 2006. [Online]. Available: http://doi.acm.org/10.1145/1134744.1134751

[68] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, “Swap: Mitigating xss attacks using a reverse proxy,” in *Software Engineering for Secure Systems, 2009. SESS ’09. ICSE Workshop on*, may 2009, pp. 33 –39.

[69] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis, “xjs: practical xss prevention for web application development,” in

Proceedings of the 2010 USENIX conference on Web application development, ser. WebApps'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863166.1863179

[70] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 23–43. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70542-0_2

[71] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Comput. Netw.*, vol. 48, no. 5, pp. 717–738, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.01.009

[72] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, ser. ACSAC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 335–344. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2008.36

[73] M. Rice and S. Kulhari, "A survey of static variable ordering heuristics for efficient bdd/mdd construction." Technical report, UC Riverside, Tech. Rep., 2008.

[74] M. Johns, "Sessionsafe: implementing xss immune session handling," in *Proceedings of the 11th European conference on Research in Computer Security*, ser. ESORICS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 444–460. [Online]. Available: http://dx.doi.org/10.1007/11863908_27

[75] H. Shahriar and M. Zulkernine, "S2xs2: A server side approach to automatically detect xss attacks," in *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 7–14. [Online]. Available: http://dx.doi.org/10.1109/DASC.2011.26

[76] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press Cambridge,, UK, 2004, vol. 2.

[77] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *ISSTA*, 2008, pp. 261–272.

[78] D. Ray and J. Ligatti, "Defining code-injection attacks," *SIGPLAN Not.*, vol. 47, no. 1, pp. 179–190, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103678

[79] Y. Wu and J. Offutt, "Modeling and testing web-based applications," George Mason University, Tech. Rep., 2002.

[80] A. S Miner, "Implicit gspn reachability set generation using decision diagrams," *Performance Evaluation*, vol. 56, no. 1, pp. 145–165, 2004.

[81] C. Anley, "Advanced sql injection in sql server applications," *White paper, Next Generation Security Software Ltd*, 2002.

[82] ——, "(more) advanced sql injection in sql server applications," *White paper, Next Generation Security Software Ltd*, 2002.

[83] G. Ollmann, "Second-order code injection attacks," *NGS Insight Security Research*, 2004.