2016-01-01

A Unified Cyber-Enhanced Approach for Detecting Cross-Site Scripting Attacks on Web Applications

Bhanukiran Gurijala
University of Texas at El Paso, bhanukiran.gurijala@gmail.com

A UNIFIED CYBER-ENHANCED APPROACH FOR DETECTING CROSS-SITE SCRIPTING ATTACKS ON WEB APPLICATIONS

BHANUKIRAN GURIJALA

Doctoral Program in Computer Science

APPROVED:

Ann Q. Gates, Ph.D., Chair

Salamah I. Salamah, Ph.D., Chair

Luc Longpré, Ph.D.

Natalia Villanueva-Rosales, Ph.D.

Miguel Velez-Reyes, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

DEDICATION

To My Beloved Family,
Especially To
Mother, Lavanya Gurijala and
Father, Narasimhulu Venkata Gurijala

A UNIFIED CYBER-ENHANCED APPROACH FOR DETECTING CROSS-SITE SCRIPTING ATTACKS ON WEB APPLICATIONS

by

BHANUKIRAN GURIJALA, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

August 2016

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my advisor and mentor Dr. Ann Q. Gates. Her professionalism and meticulousness in pursuit of excellence are a great source of inspiration that I continue to learn and pass forward to others in my career. I would like to express my greatest gratitude to my co-advisor and mentor Dr. Salamah I. Salamah.

I am very grateful to my committee for their invaluable contributions, feedback, and support: Dr. Luc Longpré, for his invaluable feedback about security domain and automata; Dr. Natalia Villanueva-Rosales, for her invaluable feedback in knowledge representation; Dr. Miguel Velez-Reyes, for his consistent support and invaluable feedback with encouraging words.

Additionally, I want to thank the professors and staff of the Computer Science Department and the CyberShARE Center of Excellence at the University of Texas at El Paso for their support and enhancing my learning experience.

I would also like to thank my father, mother, and sister for their love and support. My father, Narasimhulu Venkata Gurijala, has always been supportive of my decisions and plans. My mother, Lavanya Gurijala, has always encouraged me to chase my dreams and supported me in their pursuit. My sister, Indumathi Gurijala, has been supportive and encouraging. I would like to thank my best friend Akash Agarwal and his wife Deepavali Chakravarti who are no less than my family. They have continuously supported me and encouraged me through thick and thin.

This work is partly supported by CAHSI and CyberShare through grants from National Science Foundation (NSF) CNS-1042341 and HRD-1242122. Any opinions, findings, and conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the NSF.

ABSTRACT

Cyber-security is one of our nation's most critical security priorities, and its importance continues to grow with the pervasiveness of computers and Web-based applications. In particular, cross-site scripting (XSS) is one of the most common and dangerous types of injection attacks that exploit input validation vulnerabilities. XSS has intensified due to: 1) lack of extensive security domain knowledge of software engineers who are involved in building and/or maintaining Web-applications; and 2) lack of proper software development processes focused on security, resulting in fixes to security vulnerabilities late in the software development lifecycle. Indeed, the cost benefits of removing defects, in particular security-related faults, earlier in the lifecycle is well documented. The **research goal** is to reduce successful XSS attacks through a unified approach that identifies malicious and suspicious inputs/outputs based on customized application-specific knowledge. The Intrusion Detection Approach (IDA), which is defined in this dissertation, captures XSS-related domain knowledge from national catalogs of attack patterns and uses it to generate application-specific XSS patterns for monitoring IO by integrating technologies and techniques such as ontologies, provenance, formalizations and security-related domain knowledge. The work hosts a security knowledge base that is easily maintainable whenever new attacks or new ways of launching attacks come into use. Updating the security knowledge base results in monitoring, identifying, and preventing XSS attacks without any changes to the Web application. Risk analysis combined with provenance provides a unique way of prioritizing formalized patterns based on sensitivity level of assets and trends of threats. Using the XSSMon tool, which realizes the IDA, the author conducted a case study on two versions of a commercial Web application to compare the effectiveness of XSSMon. The results showed that XSSMon had higher success rates in identifying XSS-related attacks. Specifically, the overall success rates were 4.79% and 28.01% for the original and latest version of the Web application, respectively, and 88.36% and 100% for the initial and latest version of XSSMon, respectively. The results of this case study can be extended to other Web applications that accept similar equivalence classes of IO.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 PROBLEM STATEMENT AND GOAL

Cyber-security can be defined as security measures applied to computers to provide a desired level of protection with respect to confidentiality, integrity, and availability (**CIA**) concerns. Confidentiality refers to the property that data should only be viewable by authorized parties; integrity to the principle that only authorized users are allowed to change data; and availability to the principle that data and computer resources will always be available to authorized users [1]. Cyber attacks undermine the CIA concerns or information resident on it. The importance of cyber-security is grows as the integration of computers into more and more aspects of modern life continues. This in turn has increased cyber-attacks and the need to mitigate them, in particular in dynamic Web applications that accept input from the user and performs actions based on the input. The heart of the issue is the introduction of distrusted content into a dynamic page, where neither the Web site nor the client has enough information to detect the event and take protective actions. Such distrusted content can be introduced through malicious code provided by one client for another client, or through malicious code sent by a client for itself.

Distrusted content or malicious scripts result in one of the most common application-level attacks known as Cross Site Scripting (**XSS**). XSS was the second most prevalent consequence of vulnerability exploitation for the first half of 2013 at 18% [2] and was ranked third for the entire year of 2013 [3] [4] and has consistently stayed in the top three risks for Web applications since 2002 [2-6]. With XSS, every input and output has the potential to be an attack vector, which does not occur with other vulnerability types. In addition, XSS has numerous subtleties and variants. The **goal** of the work is to reduce successful XSS attacks through a unified approach that identifies malicious and suspicious inputs/outputs based on customized application-specific knowledge. This dissertation introduces the Intrusion Detection Approach (**IDA**), which realizes the goal by using cyber-enhanced technologies, i.e., ontologies and provenance, to manage application-specific knowledge for anomaly detection and XSS knowledge from national vulnerability data bases for

misuse detection. The customized and formalized knowledge can be used monitor inputs and outputs to Web applications to determine if they are capable of launching XSS attacks.

1.2 MOTIVATION

Cyber-attacks are placed among top five risks the world is likely to face over the next decade [7] [8]. Cyber-attacks, in particular XSS, pose a severe threat to Web-application security due to: a) lack of extensive security domain knowledge in software engineers who are involved in building and maintaining Web-applications; and b) lack of proper software development processes focused on security, resulting in fixes to security vulnerabilities late in the software development life cycle (SDLC). Many approaches have been presented for detecting and preventing security breaches; however, the approaches are reactive. Section 2 describe the approaches to detect security breaches.

By capturing, maintaining, and sharing XSS-related knowledge, it is posited that XSS attacks can be prevented by supporting application-specific customization of XSS knowledge to create an intermediate layer between Web applications and its users. Furthermore, this would address the aforementioned challenges by providing customization of existing knowledge about XSS attacks for a Web application leading to enhanced security. In addition, the approach will provide the ability to restructure and customize the knowledge to address changes to the Web application over the course of its life cycle by keeping abreast with the new and emerging weaknesses and attack patterns.

1.3 CHALLENGES

The challenges for developing a comprehensive security solution for XSS attack are as follows [9]:

1. **System requirements needed for XSS attack**
   - **i.** Malicious hackers have developed many tools that can launch XSS attacks by bypassing the safeguards employed by the Web applications. The entry points of

vulnerable XSS Web applications can be found using automated tools. PHP Charset Encoder is an example tool to launch attacks.

ii. The most basic data manipulations that exploit the vulnerability and launch XSS attacks are simple to perform. No special tools are needed to launch XSS attacks. Only a Web browser is needed.

iii. New evasive mechanisms can easily be found on the hacker's discussion forums, where new hacking attempts and success stories are discussed.

2. Social factors

i. XSS vulnerabilities arise due to coding practices. Web applications are developed by programmers with varied experience and with little or no knowledge of security needs of an application. Coding vulnerabilities vary from site to site, and there is no single patch available to fix all XSS vulnerabilities.

ii. Web applications are usually maintained by a team that is generally not the same team that developed the application, which increases the chances of introducing new XSS vulnerabilities with each change request.

3. Diverse business needs

i. Businesses prioritize increasing the customer base to increase the revenue. The advent of technologies, such as AJAX, Web Services and Web 2.0, have contributed to more aggressive schedules and a practice of prioritizing features that are not inclusive of security. Usually, requirements evolve and change during implementation that forces the developers to change functionality of the application without concern for changes to security mechanisms.

ii. Web applications are developed to meet diverse business needs. Therefore, the security mechanism applicable for one Web application may not be applicable for the other.

The approach aims to address the aforementioned challenges.

1.3 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

Definitions

Definitions	Meaning
Blacklist or Block-list	A list of discrete entities that have been previously determined to be associated with malicious activity
Fuzz testing or Fuzzing	Black-box software testing technique that involves using malformed or semi-malformed data injection in an automated fashion.
Fuzzer	Program that injects automatically the fuzz inputs into a program.
Pattern	The regular and repeated way in which something happens or is done
Payload	The malicious code that performs a destructive operation
Whitelist	A list of discrete entities that are known to be benign and are approved for use

Acronyms

Acronyms	Meaning
AOP	Aspect Oriented Programming
CAPEC	Common Attack Pattern Enumeration and Classification
CSRF	Cross Site Reference Forgery
CVE	Common Vulnerabilities and Exposures
CWE	Common Weaknesses Enumeration
DFA	Deterministic Finite Automata
DOM	Document Object Model
DSI	Document Structure Integrity
FPSIV	Five Primary Security Input Validation
FSA	Finite State Automata
FSM	Finite State Machines
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
IRM	Inline Reference Monitor
IVV	Input Validation Vulnerability
MBT	Model Based Testing
MDD	Model Driven Development
MFE	Malicious File Execution
NFA	Non-deterministic Finite Automata
OWASP	Open Web Application Security Project
OWL	Web Ontology Language
PQL	Program Query Language
QAS	Quality Attribute Scenarios
RFI	Remote Malicious File Inclusion
SAW	Semantic Abstract Workflow
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
SOP	Same Origin Policy
SQL	Structured Query Language

Acronyms	Meaning
SVM	Support Vector Machines
UIV	User Input Validation
WfMC	Workflow Management Coalition
WfMS	Workflow Management Systems
XCS	Cross Channel Scripting
XML	Extensible Markup Language
XSD	XML Schema Definition
XSS	Cross Site Scripting
ZAP	Zed Attack Proxy

Abbreviations

Abbreviations	Meaning
e.g.	For example
i.e.	That is
ID	Identification

1.4 ORGANIZATION

The organization of rest of the dissertation document is as follows: Chapter 2 presents the background information to enhance understanding of the rest of the document; it also includes a comprehensive survey of XSS detection and prevention approaches. Chapter 3 presents the approach; it details the steps of the approach needed to accomplish the research goal. Chapter 4 presents the design and implementation details of the XSSMon tool developed based on the described approach. Chapter 5 presents the setup, design, and results of the case study and their analysis that support evaluation of the XSSMon tool to evaluate tool designed based on the approach. Chapter 6 compares the more closely aligned systems described in Chapter 2 to the IDA approach. Chapter 7 presents the conclusion, which includes a summary, significance, limitations of the work and future directions of the effort. Appendix A includes a detailed description of the techniques and approaches for detecting and preventing XSS attacks surveyed. Appendix B presents the test suite for performing unit testing of the designed tool. Appendix C presents the CRC cards for the tool design. Appendix D presents the details for each of the methods including pre- and post-conditions and test design strategy for unit testing.

CHAPTER 2: BACKGROUND

This section provides background information to enhance understanding of the rest of the document. This section is organized as follows: Section 2.1 provides an overview of cyber-security and cyber-attacks from a historical perspective, including an overview of Web application vulnerabilities and the main categories of XSS; Section 2.2 provides an overview of techniques, methods, and tools that are available for detecting and/or preventing XSS vulnerabilities; Section 2.3 provides an overview of cyber-enhanced techniques and technologies, including an overview of ontologies and their current usage in capturing security related domain knowledge and an overview of attack-trees and their usage in different phases of SDLC with respect to software security.

2.1 HISTORICAL OVERVIEW OF CYBER-SECURITY AND CYBERATTACKS

Early computer systems offered high security with relatively less functionality and availability as compared to the current software systems [1]. As software vendors increased functionality, moved to PCs and later to distributed computing and Web services, data availability has increased by orders of magnitude, thereby marking the increase in issues of confidentiality and integrity. The driving principle behind software development has been functionality and not security. The basic design of the Internet was built around shared access and trust with security measures being an afterthought. Many protocols in wide use mostly rely on trust and offer little, if any, security to their users. This model made sense when the Internet was first developed and information being transferred was not critical. Today, the Internet is used to transfer information between people, their banks, their brokers, businesses and government entities mostly using Web applications, which makes this information susceptible to cyber-attacks, identity thefts and phishing attacks.

The earliest Web applications used the CGI protocol to interface with external applications. The applications shared security issues that resulted from the common text parsing approaches used. Applications were often vulnerable to input validation weaknesses, buffer overflows, and

denial of service attacks. As the Web gained popularity, Web browsers and HTTP protocol evolved. Commercial Web-server software emerged that allowed Web developers to produce custom plugins or extensions and filters that ran within the Web server process. This allowed developers to interact with a Web request at multiple stages in the serving process, which exposed Web servers to potentially vulnerable code and malicious user input, increasing the risk of Web server crashes and imported security vulnerabilities. Attacks against Web servers were generally concentrated on the core Web-server code and supporting libraries, or manipulated URL elements and injection of unexpected user data. The attacks usually fell into standard vulnerability categories such as: buffer overflows, sample code, input validation attacks, format string attacks, canonicalization attacks, encoding attacks, privilege escalation, form tampering, user created content, XSS, SQL injection, insecure direct object reference, remote malicious file inclusion (RFI), cross-site request forgery (CSRF), access control weaknesses, authentication and session management failures, data confidentiality failures, and poor error handling [10].

2.1.1 Overview of Web Vulnerabilities

Buffer-overflow attacks occur when an application tries to insert data into an available buffer without checking the size of the buffer that can result in overflow of the buffer [10]. Attackers insert their malicious code into adjacent areas of the stack or heap by inserting larger values that overflow the buffer. These malicious scripts can subsequently be executed by an attacker with the privileges of the Web-server user.

Attackers can launch input validation attacks when the user input is inadequately validated and sanitized. Format string attacks can occur when application code attempts to display unfiltered user-passed variables. Canonicalization attacks occur when equivalent forms of canonical file names are handled differently by a Web-server with unexpected results. Sample applications or test scripts that were often badly coded and were not intended for production use include early Apache test-cgi scripts (CVE1999-0070).

Encoding attacks exploit incorrect handling of various forms of character encoding by the Web server that allows attackers to bypass regular expression filters and launch attacks. Attackers can exploit application or operating system (OS) behavior to execute code with higher than expected privileges resulting in privilege escalation attack. Form-tampering attacks can be launched by client-side manipulation of hidden form fields or modification of browser cookies through local application proxies that alter Web application behavior. In a user-generated content attack, back-doors or malicious server are uploaded into vulnerable Web applications to assist in attacks.

Attackers are sometimes able to manipulate direct references to internal application objects such as file names and user IDs. Insecure direct object reference attacks can include changing an account ID in a dynamically generated Web page from an application that subsequently fails to check if the attacker's user ID is associated with changed account ID.

RFI attacks occur when an application improperly trusts user submitted files or references to external objects such as remote URLs and then evaluate and execute the malicious contents. These are common in PHP applications.

CSRF occurs when an attacker is able to trick an already authenticated user into performing malicious action, which may succeed due to design weakness in the target application and the user's Web browser automatically supplies cached credentials.

Access control weaknesses occur when developers fail to programmatically enforce strict access control allowing attackers to perform unauthorized actions or view restricted material.

Authentication and session management failures result from poorly implemented systems that allow attackers to obtain credentials or tokens to impersonate valid users. Data confidentiality failures occur when user credentials and other confidential data are not protected by correctly implemented proven cryptographic standards. Failure to correctly handle unexpected errors can crash an application revealing internal configuration information or bypass security systems leading to attacks that result from poor error handling.

SQL injection and XSS are the most popular injection attacks. A Web application is vulnerable to SQL injection attacks when malicious content can flow into SQL queries without being fully sanitized allowing the attacker to trigger malicious SQL operations by injecting SQL keywords or operators [11]. Malicious SQL statements can be introduced into a vulnerable application using different input mechanisms including user inputs, cookies, and server variables [12]. Second-order SQL injection attack is a special case, where the attacker stores malicious content into the database and triggers its execution at a later time. These attacks can lead to authentication bypass, information disclosure and other problems.

2.1.2 XSS

Dynamic Web applications accept input from users and perform actions based on the input. Lack of proper input validation results in XSS, which was first discovered in 1990s. It is one of the most common type of injection attacks that exploit input validation vulnerability [13], in which malicious scripts are injected into otherwise benign and trusted Websites [14].

Initially, only two primary types of XSS attacks were identified: Stored XSS (persistent) and Reflected XSS (non-persistent). *Stored XSS*, also referred to as *Persistent XSS*, generally occurs when user input is stored on the target server, such as in a database, message forum, visitor log, or comment field. Subsequently, a victim retrieves the stored data from the Web application without that data being made safe to render in the browser. With the advent of HTML5 and other browser technologies, the attack payload can be permanently stored in the victim's browser, such as an HTML5 database and never sent to the server at all [15]. *Reflected XSS*, also referred to as *Non-Persistent XSS*, occurs when user input is immediately returned by a Web application in an error message, search result, or any other response. The message or response may include some or all of the input provided by the user as part of the request without that data being made safe to render in the browser and without permanently storing the user provided data [15].

Amit Klein [16] identified and defined a third type of XSS, which is termed as *Document Object Model (DOM) Based XSS*. In this type of XSS, the entire tainted data flow from source to

sink takes place in the browser, i.e., the source and sink of the data is in the DOM and the data flow never leaves the browser. For instance, the source could be the URL of the page or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data.

A fourth type of XSS, which was identified and defined in 2007, is known as *Induced XSS* [17]. XSS of this type can be launched when the Webserver has HTTP Response splitting vulnerability. An attacker can change the entire HTML content by manipulating the HTTP header of the server's response to include a request parameter that has not been validated.

The four categories of XSS attacks overlap, i.e., it is possible to have both Stored and Reflected DOM-Based XSS, as well as Stored and Reflected Non-DOM Based XSS. Around mid-2012, the OWASP (Open Web Application Security Project) with involvement of the cyber-security research community proposed two new terms to organize and classify the types of XSS attacks that can occur: Server XSS and Client XSS [15].

Server XSS occurs when untrusted user supplied data is included in an HTML response generated by the server. The source of this data could be from the request, or from a stored location. This leads to two types of server XSS: *Reflected Server XSS* and *Stored Server XSS*. In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it [15].

Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. The source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server that leads to two types of client XSS: *Reflected Client XSS* and *Stored Client XSS* [15].

With these new definitions, DOM-Based XSS simply becomes a subset of Client XSS, where the source of the data is somewhere in the DOM, rather than from the Server. The induced

XSS would also fit into this classification proposed by OWASP [15]. It would be a subset of Reflected Server XSS.

To launch any type of aforementioned XSS-attacks there are three generic steps that need to occur in sequential order: locate vulnerability, inject attack script, and execute attack script [18]. With XSS, every input and output has the potential to be an attack vector, which does not occur with other vulnerability types. In addition, XSS has numerous subtleties and variants which makes its detection and/or prevention difficult.

2.2 TECHNIQUES AND METHODS FOR DETECTING AND PREVENTING XSS VULNERABILITIES

Fig. 2.2 presents a categorization and classification that is based on a thorough and systematic review of literature on techniques and methods for detecting and/or preventing XSS vulnerabilities. We classify the work into three broad categories: server-side, client-side, and generic that addresses the work that that is geared towards detecting and/or preventing server-side, client-side or in general XSS-attacks. The work that did not specify whether the approach was geared towards server-side or client-side is classified under a generic category. Each of these categories are further refined into stored, reflected and general type of XSS attacks, which are further refined into approaches for detecting, preventing, or supplementary. Each of the approaches are further refined into whether they are used for detecting or preventing vulnerabilities or attacks. Further refinement is based on whether static analysis, dynamic analysis, modeling, secure programming, or other technique is used to detecting or preventing vulnerabilities and attacks. Static analysis involves reviewing the source code or byte code of an application to find faults. Dynamic analysis entails examining the behavior of an application in runtime. Secure Programming involves using libraries that take care of security attributes or following programming guidelines or rules and practices for secure development of Web applications. Modeling involves using different modeling notations for detection or prevention of XSS related vulnerabilities and attacks. The approaches that did not fall into any of the above mentioned

categories are categorized as “Others”. In Fig. 2.2, the classification is shown only for server-side stored XSS as an example; however, a similar classification scheme can be followed for others.

The research included a survey of 131 research papers on techniques for detection and prevention of XSS-related attacks and vulnerabilities. The results of the survey are summarized in tables for each of the following categories: server-side, client-side, and generic XSS as identified in Fig. 1. For a detailed description of the summarized work, please refer to Appendix A. Tables 2.2 to 2.4 provide summaries of the techniques for prevention and detection of XSS-related to attacks and vulnerabilities on the server-side, client-side, and generic XSS, respectively. The summary of all techniques for detection and prevention of XSS-related vulnerabilities and attacks and supplementary work is presented in Table 2.5.

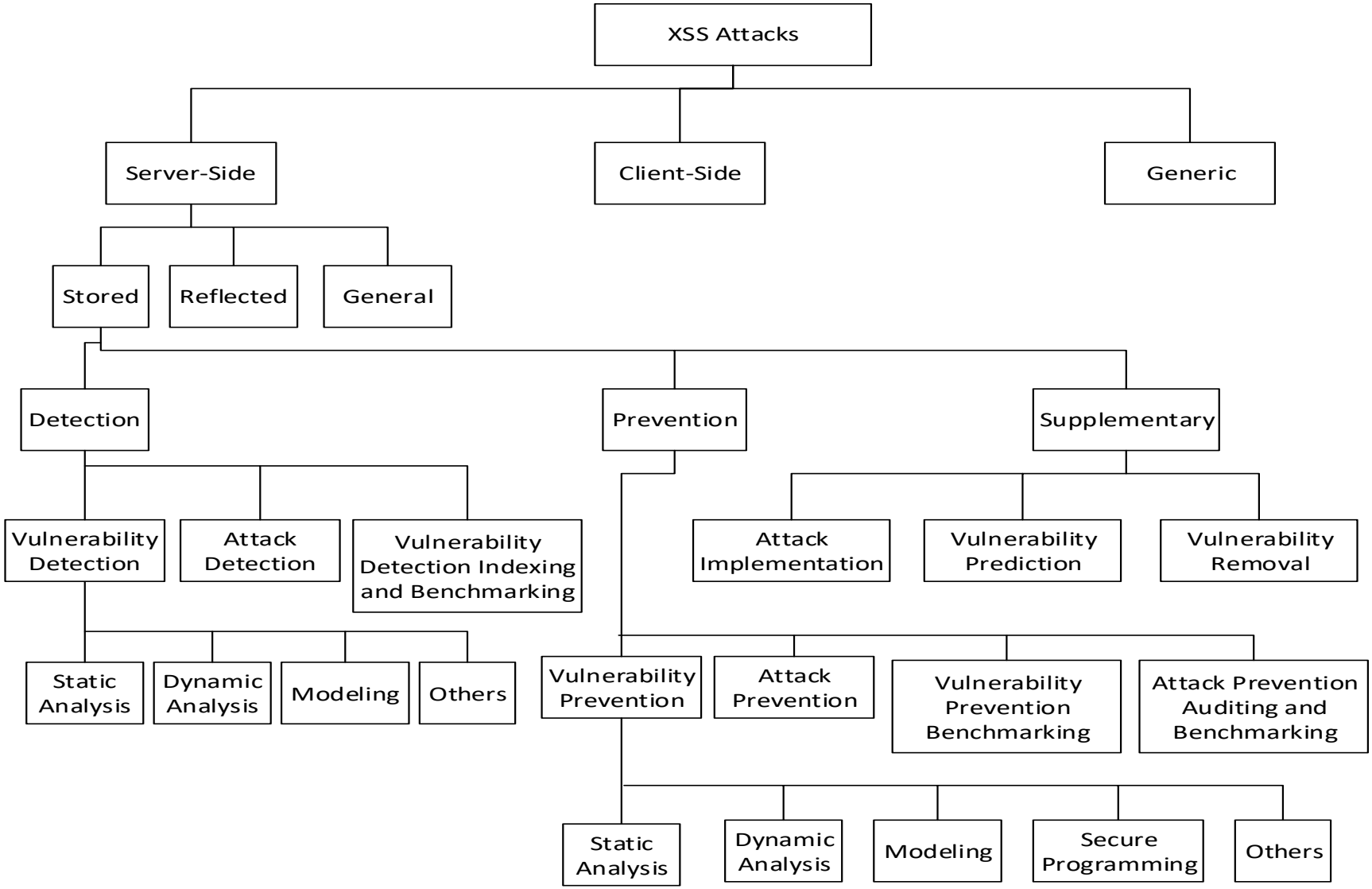

Figure 2.2: Classification categories for approaches to detect and prevent XSS-attacks.

Table 2.2: Summary of Server-Side XSS prevention and detection approaches.

General Approaches	Stored	Reflected	General
Detection			
Vulnerability Detection	**Static:** [19], [20], [21] **Dynamic:** [22]	**Static:** [19], [21] **Dynamic:** [22]	**Static:** [23] , [24], [25], [26], [27], [28] **Dynamic:** [27], [29], [30]
Attack Detection	**Dynamic:** [31], [32], [33], [34]	**Dynamic:** [32], [33], [34]	**Dynamic:** [35], [36], [37], [38], [39], [40], [41], [42] **Modeling:** [43], [44], [45], [46] **Others:** [47]
Prevention			
Vulnerability Prevention			**Static:** [48]
Attack Prevention	**Dynamic:** [31], [49], [50], [33], [51], [52], [53], [54], [34] **Others:** [55]	**Dynamic:** [56], [49], [50], [33], [51], [52], [53], [54], [34] **Others:** [55]	**Static:** [48], [28], [57], [58] **Dynamic:** [59], [60], [61], [39], [40], [41], [30], [62], [42], [63], [64], [65] **Modeling:** [66], [67], [68], [63], [65] **Secure Programming:** [69], [70]
Supplementary			
Vulnerability Removal	**Static:** [20]		**Static:** [23]

Table 2.3: Summary of Client-Side XSS prevention and detection approaches.

General Approaches	Stored	Reflected	General
Detection			
Vulnerability Detection	**Static:** [21] **Dynamic:** [71], [72]	**Static:** [21] **Dynamic:** [71], [72]	**Static:** [73] **Dynamic:** [74] **Modeling:** [75], [76]
Attack Detection	**Dynamic:** [77], [34]	**Dynamic:** [77], [34]	**Static:** [78]
Prevention			
Attack Prevention	**Dynamic:** [79], [77], [80], [81], [52], [53], [54], [34]	**Static:** [82] **Dynamic:** [82], [77], [80], [81], [52], [53], [54], [34]	**Static:** [83] **Dynamic:** [84], [85], [86], [87], [88], [89], [90], [91], [92], [83], [74], [63], [93], [94], [65] **Modeling:** [63], [65]

Table 2.4: Summary of Generic XSS prevention and detection approaches.

General Approaches	Stored	Reflected	General
Detection			
Vulnerability Detection	**Static:** [95], [96] **Dynamic:** [97]	**Static:** [98], [99] **Dynamic:** [100], [97] **Modeling:** [101]	**Static:** [102], [103], [104], [105], [106], [107], [108], [109], [110], [111], [112], [113] **Dynamic:** [114], [115], [116], [110], [111], [117], [113] **Modeling:** [118], [117], [112], [119]
Attack Detection	**Static:** [120] **Dynamic:** [120]	**Static:** [120] **Dynamic:** [120]	**Static:** [121], [112] **Dynamic:** [122], [123] **Modeling:** [123], [124], [125], [112] **Others:** [126]
Vulnerability Detection Indexing and Benchmarking			**Others:** [127], [128]
Prevention			
Vulnerability Prevention			**Secure Programming:** [129], [130]

General Approaches	Stored	Reflected	General
Vulnerability Prevention Benchmarking			**Others:** [131]
Attack Prevention	**Static:** [132] **Dynamic:** [133], [132]	**Static:** [132] **Dynamic:** [133], [132] **Modeling:** [134]	**Static:** [113] **Dynamic:** [135], [136], [137], [113], [138] **Modeling:** [139], [140] **Secure Programming:** [141] **Others:** [142]
Attack Prevention Auditing and Benchmarking	**Modeling:** [143]	**Modeling:** [143]	**Modeling:** [144] **Others:** [145], [146]
Supplementary			
Attack Implementation	**Dynamic:** [97] **Modeling:** [147]	**Dynamic:** [97]	**Dynamic:** [117] **Modeling:** [117]
Vulnerability Prediction			**Modeling:** [148], [149]

Table 2.5: Summary of All XSS prevention and detection approaches.

	Server-Side XSS			Client-Side XSS			Generic XSS		
General Approaches	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**
Detection									
Vulnerability Detection	**Static:** [19], [20], [21] **Dynamic:** [22]	**Static:** [19], [21] **Dynamic:** [22]	**Static:** [23] , [24], [25], [26], [27], [28] **Dynamic:** [27], [29], [30]	**Static:** [21] **Dynamic:** [71], [72]	**Static:** [21] **Dynamic:** [71], [72]	**Static:** [73] **Dynamic:** [74] **Modeling:** [75], [76]	**Static:** [95], [96] **Dynamic:** [97]	**Static:** [98], [99] **Dynamic:** [100], [97] **Modeling:** [101]	**Static:** [102], [103], [104], [105], [106], [107], [108], [109], [110], [111], [112], [113] **Dynamic:** [114], [115], [116], [110], [111], [117], [113]

	Server-Side XSS			Client-Side XSS			Generic XSS		
General Approaches	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**
									Modeling: [118], [117], [112], [119]
Attack Detection	**Dynamic:** [31], [32], [33], [34]	**Dynamic:** [32], [33], [34]	**Dynamic:** [35], [36], [37], [38], [39], [40], [41], [42] **Modeling:** [43], [44], [45], [46] **Others:** [47]	**Dynamic:** [77], [34]	**Dynamic:** [77], [34]	**Static:** [78]	**Static:** [120] **Dynamic:** [120]	**Static:** [120] **Dynamic:** [120]	**Static:** [121], [112] **Dynamic:** [122], [123] **Modeling:** [123], [124], [125], [112] **Others:** [126]
Vulnerability Detection Indexing and Benchmarking									**Others:** [127], [128]
Prevention									
Vulnerability Prevention			**Static:** [48]						**Secure Programming:** [129], [130]
Vulnerability Prevention Benchmarking									***Others*:** [131]
Attack Prevention	**Dynamic:** [31], [49], [50], [33], [51], [52], [53], [54], [34] **Others:** [55]	**Dynamic:** [56], [49], [50], [33], [51], [52], [53], [54], [34] **Others:** [55]	**Static:** [48], [28], [57], [58] **Dynamic:** [59], [60], [61], [39], [40], [41], [30], [62], [42], [63], [64], [65]	**Dynamic:** [79], [77], [80], [81], [52], [53], [54], [34]	**Static:** [82] **Dynamic:** [82], [77], [80], [81], [52], [53], [54], [34]	**Static:** [83] **Dynamic:** [84], [85], [86], [87], [88], [89], [90], [91], [92], [83], [74], [63], [93], [94], [65]	**Static:** [132] **Dynamic:** [133], [132]	**Static:** [132] **Dynamic:** [133], [132] **Modeling:** [134]	**Static:** [113] **Dynamic:** [135], [136], [137], [113], [138] **Modeling:** [139], [140] **Secure Programming:** [141] **Others:** [142]

	Server-Side XSS			Client-Side XSS			Generic XSS		
General Approaches	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**	**Stored**	**Reflected**	**General**
			Modeling: [66], [67], [68], [63], [65] **Secure Programming:** [69], [70]			**Modeling:** [63], [65]			
Attack Prevention Auditing and Benchmarking							**Modeling:** [143]	**Modeling:** [143]	**Modeling***: [144]* **Others:** [145], [146]
Supplementary									
Attack Implementation							**Dynamic:** [97] **Modeling:** [147]	**Dynamic:** [97]	**Dynamic:** [117] **Modeling:** [117]
Vulnerability Removal	**Static:** [20]		**Static:** [23]						
Vulnerability Prediction									**Modeling:** [148], [149]

2.3 CYBER-ENHANCED TECHNOLOGIES

2.3.1 Knowledge representation, Classifications, and Ontologies

The continual rise in cyber-attacks, which pose a threat to CIA concerns denting cyber-security, has created the need to formally and systematically document security domain knowledge that can be used and shared. One of the ways is to develop an ontology that captures knowledge in related fields, provides a common understanding of domain knowledge, identifies common recognition vocabularies, and gives a clear definition of the relationship between these vocabularies from different levels of formal patterns [150]. This subsection describes efforts to define ontologies that capture security domain knowledge.

Mylopoulos et al. [151] described merging knowledge base and information system management at an early level of development using a language called Telos for representing knowledge about systems. The knowledge base is divided into four sub-worlds namely, subject world, usage world, system world, and development world.

Avizienis et al. [152] provide a detailed taxonomy that contains classes of *faults*, *fault modes*, classification of *fault tolerance techniques*, and *verification* approaches. In this taxonomy, the main threats to dependability and security are defined as *failures*, *errors*, and *faults*. They classify the main means to attain security and dependability attributes into *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*.

Landwehr et al. [153] provide a taxonomy that is based on three basic questions about each observed security flaw: genesis (how did it enter the system?), time of introduction (when did it enter the system?), and location (where in the system did it manifest?).

In [154], the authors propose a data model that characterizes the domain of computer attacks and intrusions as an ontology and implement that data model with an ontology representation language. At the topmost level of the ontology, they define the class *Host* that is victim of *Attack* class. Attack class is described by the properties *Directed to*, *Effected by*, and *Resulting in* that correspond to *System Component, Input, and Consequence* class objects

respectively. The *System Component* class is comprised of the subclasses (*Network*, *System*, and *Process*). The class *Consequence* is comprised of several subclasses which include (*Denial of Service*, *User Access*, and *Probe*). Finally, the class *Input* is characterized by the predicates *Received from* and *Causing*, where *Causing* defines the relationship between the *Means of attack* and some *input*. *Received from* links *Input* and *Location*. The class *Location* is an instance of *System Component* and is restricted to instances of the *Network* and *Process* classes. *Means of attack* contains the subclasses: *Input Validation Error*, *Logic Exploits* that are further refined.

Viljanen [155] analyzed thirteen different computational trust models and derived a common vocabulary for describing facts that are considered for trust calculation in the reviewed trust models. The models can be classified as *identity-aware*, *action-aware*, *business value aware*, *capability-aware*, *competence-aware*, *confidence-aware*, *context-aware*, *history-aware* and *third-party-aware* in their input factors. The trust ontology comprises many ontological structures; *trust* is a relationship between two principals, the subject, *trustor*, and the target, *trustee*.

Geneiatakis and Lambrinoudakis [156] described an ontology for SIP-VoIP (Session Initial Protocol-VoIP) based services that can be applied either to find a countermeasure against attacks or to test the security robustness of SIP-VoIP infrastructure. *SIP_attack* and *SIP_message* are the two main concepts of the ontology. Specifically, any SIP attack employs a SIP message that is forwarded to a target node trying to cause a specific consequence. The SIP_attack is directed by a target and causes a consequence. It has two subclasses: *malformed* and *flood*.

Denker et al. [157-159] developed several ontologies for security annotations of agents and web services, using DAML (DARPA Agent Markup Language) and later OWL (Web Ontology Language). The defined ontology is composed of two sub-ontologies: *security mechanisms* that capture high-level security notations and *credential* that defines authentication methods. The goal of these ontologies is to enable high-level markup of Web resources, services, and agents while providing a layer of abstraction on top of various web service security standards. These ontologies represent well-known security concepts and enable their users to interconnect security standards.

The NRL Security Ontology presented in [160] is organized around seven separate ontologies (*Main Security Ontology*, *Credential Ontology*, *Security Algorithms Ontology*, *Security Assurance Ontology*, *Service Security Ontology*, *Agent Security Ontology*, *Information Object Ontology*). Three of them are based on existing ontologies in DAML: *Service security ontology*, describes security annotation of semantic web services; *Agent security ontology*, enables querying of security information; and *Information object ontology*, describes security of input and output parameters of web services. The four remaining ontologies: *Main security ontology*, describes security protocols, mechanisms and policies; *Credentials ontology*, specifies authentication credentials; *Security algorithms ontology*, describes various security algorithms; and *Security assurance ontology*, specifies different assurance standards.

Vorobiev and Han presented a security attack ontology for Web services [161]. The ontology brings together a set of attacks (*attacks on Web services*, *probing attacks*, *CDATA Field attacks*, *WS DoS attacks*, *WS DoS attacks*, *Application attacks*, *SOAP attacks*, *XML attacks*, *semantic WS attacks*).

Ekelhart et al. [162] described a security ontology framework based on four parts: the first part is the security and dependability taxonomy from [152], the second part presents the underlying risk analysis methodology, the third part describes concepts of the IT infrastructure domain, and the fourth part provides a simulation enabling enterprises to analyze various policy scenarios. The ontology 'knows' which threats endanger which assets and which countermeasures could lower the probability of occurrence, the potential loss or the speed of propagation for cascading failures.

Assali et al. [163] presented work that develops a knowledge base containing ontologies for the analysis of industrial risks describing concepts used for the achievement of a risk analysis.

Dobson and Sawyer [164] presented an ontology of dependability by merging two conceptualization models (IFIP model: described by the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance & UMD model: Unified Model of Dependability). Some of the IFIP attributes are themselves goals of security (Availability, Integrity, Maintainability, and

Confidentiality). The ontology covers some security aspects such as *Failure*, Dependability Threat (*Error*, *Fault*), Dependability Attributes (*Availability*, *Integrity*, *and Confidentiality*)

Tsoumas et al. [165] define a security ontology that is constructed by using Asset, Stakeholder, Vulnerability, Countermeasure, and Threat concepts in OWL and propose security framework of an arbitrary information system which provides security acquisition and knowledge management. The security ontology acts as a container for the IS security requirements.

Karyda et al. [166] use OWL to propose a security ontology for developing secure applications. The described ontology is composed of *assets* (data asset, hardware data); *countermeasures* (e.g., identification and authentication, network management, auditing services, and physical protection); *objectives*, *persons* (e.g., insider, stakeholder, and attacker); and *threats* (e.g., attacks and technical failures). They validate their ontology using nRQL queries to demonstrate its use in various contexts. They apply it to e-government scenarios: e-tax and e-voting.

Firesmith [167] presents a taxonomy of safety-related requirements: "*Safety requirements*" are requirements obtained from threats analysis. "*Safety-significant requirements*" includes non-safety requirements that can cause hazards and safety incidents. "*Safety constraints*" are constraints that directly impact safety and are derived from laws, policies, standards, and industrial practices. "*Safety system requirements*" specify aspects of the primary system.

Mouratidis et al. [168] introduce the concept of *security constraint* as a separate concept to the existing concepts of Tropos. Existing concepts, such as goals, tasks, resources, are defined with and without security in mind. For example, a goal should be differentiated from a secure goal, the latter representing a goal that affects the security of the system. Widely used security-engineering concepts such as *security features*, *protection objectives*, *security mechanisms* and *threats* are introduced in the Tropos ontology to make the methodology applicable by software engineers, as well as security engineers.

Massacci et al. [169] described an extended ontology for security requirements. The top of the taxonomy is adapted from DOLCE, a foundational ontology intended to account for basic

concepts that underlie natural language and human cognition. Some of their presented concepts include: Objects (*Proposition*, *Situation*, *Entity*, *Relationship*) – Entities (*Actor*, *Action*, *Process*, *Resources*, *Assets*) – Relationships (*do-dependency*, *can-dependency*, *trust-dependency*) – Propositions (*Fact*, *Claim*, *Argument*, *Domain-Assumption*, *Quality Proposition*, *Goal*).

Herzog et al. [170] described an OWL-based ontology of information security. The ontology is built around the following top-level concepts: *assets*, *threats*, *vulnerabilities* and *countermeasures*. These general concepts together with their relations form the core ontology which presents an overview of the information security domain in a context-independent and application neutral manner. In order to be practically useful, the core ontology is populated with domain-specific and technical vocabulary that constitute the core concepts and implement the core relations. The ontology contains 88 threat classes, 79 asset classes, 133 countermeasure classes, and 34 relations between these classes.

Similar to Herzog et al. work, Fenz and Ekelhart [171] presented an ontology (500 concepts) that aims to cover a broader spectrum: their ontology models a larger part of the information security domain, including non-core concepts such as the infrastructure of organizations. In the high level concepts of the ontology, there is a *threat* that can give rise to follow-up threats, representing a potential danger to organization's *assets* and affects specific security concerns (*confidentiality*, *integrity*, *availability*) by exploiting a *vulnerability* in the form of a physical, technical, or administrative weakness causing damage to certain assets.

Laclavik et al. [172] described a semi-automatic ontology based text annotation tool – OnTeA. It works on text, in particular domain as described by domain ontology and uses regular expression patterns for semi-automatic semantic annotation.

Gandhi et al. [173] developed a semantic template for each conceptually distinct weakness type of CWE to facilitate its use in organizing and integrating vulnerability information recorded in large project repositories. Gandhi et al. [174] presented a methodology to develop precise and accurate nature language descriptions of common software weaknesses based on CWE through lightweight formal modeling using Alloy.

2.3.2 Attack Trees (AT)

Attack trees provide a formal, methodical way of describing security of the system based on varying attacks. Attack trees are represented in a tree structure with a goal at the root node and different ways of achieving that goal as leaf nodes [175].

Attack trees have been used to model diverse security systems in different settings. Camtepe and Yener [176] presented a formal methodology for modeling and detection of complex attacks in three phases: a) extending AT to capture temporal dependencies between components and expiration of an attack, b) using enhanced AT build tree automation that accepts sequence of actions from input message streams, and c) construct enhanced parallel automation that has each tree automation as a subroutine. Dewri et al. [177] use an attack tree model of the network to address system administrator's dilemma of selecting a subset of security hardening measures ensuring cost of implementation is minimal and within budget along with minimizing cost of residual damage. Morais et al. [178] presented a model-based attack injection approach for security protocol testing aiming at vulnerability detection. They use attack tree models to describe known attacks and derive injection test scenarios that are later converted into specific fault injector script to test security properties. Ning et al. [179] built a penetration attack tree model that can describe, organize, classify, manage and schedule attacks for attack resistance test by defining node of attack tree model and re-describing the relation of attack tree nodes.

CHAPTER 3: APPROACH

The **goal** of the work is to reduce successful XSS attacks through a unified approach that identifies malicious and suspicious inputs/outputs based on customized application-specific knowledge.

The goal of the work aligns with the aim of software assurance, whereby the system under development will be protected from known XSS vulnerabilities. The definition [180] of software assurance follows: *"Application of technologies and processes to achieve a required level of confidence that software systems and services function in the intended manner, are free from accidental or intentional vulnerabilities, provide security capabilities appropriate to the threat environment, and recover from intrusions and failures."*

3.1 INTRODUCTION

The first step to prevention of an intrusion or security breach is to detect it. There are two broad ways to detect intrusion: anomaly detection and misuse detection. The anomaly detection is based on detecting intrusions by looking for activity that is different from a users' or systems' normal behavior. The misuse detection is based on detecting intrusion by looking for activity that corresponds to known intrusion techniques (signatures) and system vulnerabilities. The described Intrusion Detection Approach (**IDA**) addresses both.

The data flow diagram of the framework for intrusion prevention method is depicted in Fig. 3.1.1. In Fig. 3.1.1, all processes and external interactors highlighted in green are dynamic and application-specific and form the crux for anomaly detection. All processes and external interactors highlighted in blue are not application-specific, they are information-specific and hence mostly static, i.e., done less frequently, only when there are any new versions or updates available in any of the data sources. These static processes form the crux for the approach and in particular misuse detection.

The five main steps in the IDA are:

1. Identify and capture application-specific asset information.
2. Build the knowledge base (KB).

3. Derive threat-specific blacklist patterns from knowledge.
4. Customize application-specific knowledge.
5. Process the input/output (IO).

The data for anomaly detection is based on the validated specification of acceptable input values (in the case of XSS). The data for misuse detection is derived from the knowledge inferred using data acquired and integrated from disparate data sources such as Common Attack Pattern Enumeration and Classification (CAPEC), Common Weakness Enumeration (CWE), Open Web Application Security Project (OWASP) XSS cheat-sheets, and attack payloads. All the data from disparate sources is acquired, integrated, and formalized along with capture of associated provenance and later inferred knowledge is maintained in a KB to support sharing and reuse. The captured provenance provides trust, credibility, and recency about data. The KB stores the CAPEC information as a "threat" entity, the CWE is stored as a "weakness," and the XSS cheat-sheet information is stored as a "blacklist character set."

The formalized knowledge from anomaly detection model and misuse detection model is used to scan the inputs coming from the user and outputs sent to the user. If an intrusion or security breach is detected in an application, the structured and customized prevention for the intrusion or security breach is performed based on the inputs from stakeholders and experts on how to handle intrusions and violations for that particular application.

The detailed description of each of the steps that constitute the approach is provided in the following sections of this chapter. Section 3.2 presents and describes the structure for storing the data, its associated provenance and the application-specific information required for preventing XSS-related intrusions and it is the crux for all the steps of the described approach and supports sharing and reuse. Sections 3.3 to 3.7 present and describe each step of the described approach.

3.2 STRUCTURE FOR REPRESENTING KNOWLEDGE

To manage a complex domain such as cyber-security for Web applications, a non-trivial amount of information is captured from various sources. Knowledge is then derived from this

captured information. To facilitate collection, sharing, and reuse, it should be acquired using appropriate techniques and tools, such as ontologies and provenance and expressed using standard formats.

As described in Chapter 2, an ontology provides a formal description of concepts and entities along with the relations between them. It is founded on Description Logics (DL) and expressed using languages defined by the Web Ontology Language (OWL). The current standard is OWL 2. OWL is intended to be used when the information contained in the documents needs to be processed by applications as opposed to situations where the content only needs to be presented to humans.

Provenance plays an integral role in enhancing trust and credibility of information and data especially, when information and data retrieved from various sources undergoes processing and changes by different security experts using the described method. Provenance refers to the sources of information such as entities and processes involved in producing or delivering an artifact.

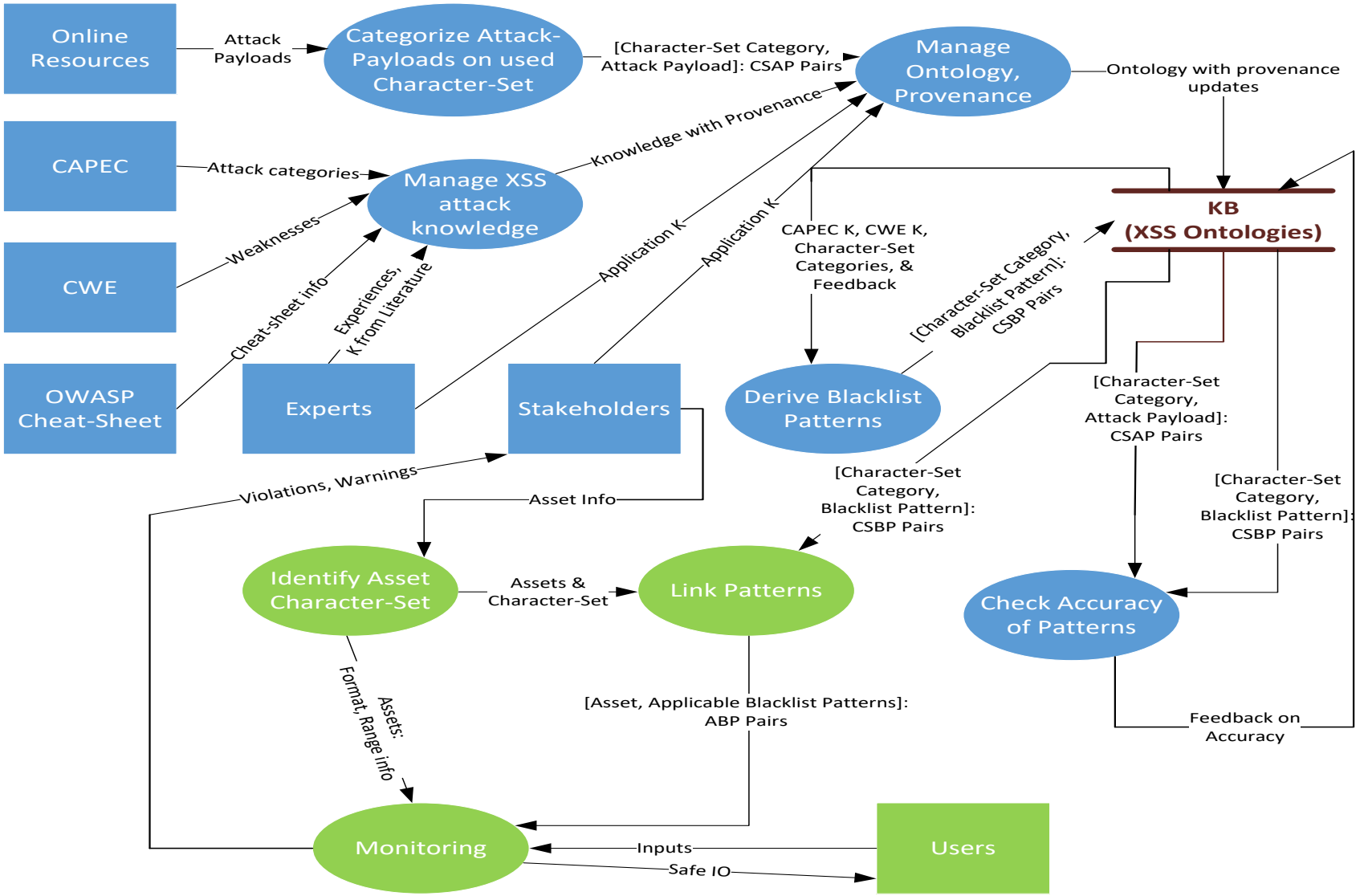

Figure 3.1.1: Dataflow Diagram for the Intrusion Detection Approach.

Prov-O [181]defines the OWL 2 Web Ontology Language encoding of the Prov Data Model. It is a lightweight ontology that can be adopted in a wide range of applications. Prov-O is used for representing the information that is captured and retrieved from various sources. Fig. 3.2.1 depicts the structure of the security ontology that is built on the Prov-O. The *Entity*, *Agent*, and *Activity* classes and their relations model the provenance that needs to be captured for credibility, trust, and decision-making purposes. All the classes capturing information about XSS are all designed as sub-classes of the *Entity* class in the Prov-O model. *Person*, *Organization*, and *Software Agent* are three sub-classes of the *Agent* class, and appropriate sub-classes are used to capture information about the agent influencing an Entity. Three sub-classes are added to the *Activity* class to capture specific information about an activity. The *Creation*, *Modification*, and *Retrieval* sub-classes of the *Activity* class capture information about creation, updates, and retrieval of *Entity*. Provenance can be captured at different levels of granularity; the decision about the level of granularity of information that needs to be captured for IDA is based on the provenance information that is captured in sources such as CAPEC and CWE.

The source information and version numbers of the data sources (CAPEC, CWE, OWASP cheat-sheets, etc.) provide information about the recency of data and supports decision-making about credibility and trustworthiness of data. The IDA ontology is used for knowledge management of the information retrieved and captured from various sources. Specifically, the ontology stores generic knowledge about XSS and application-specific knowledge. The generic knowledge includes information from sources such as, CAPEC, CWE, online payload resources, OWASP cheat-sheets, and security experts. The application-specific knowledge source includes experts and stakeholders. The benefits of the IDA ontology are that it supports reuse and, whenever new attacks or new ways of launching attacks come into use, updating the security knowledge base can be done easily without changes to the monitored application.

There are several security ontologies described in the literature, the details can be found in Section 2.3.1. Some of the key differences between the ontologies presented in literature and the ontology designed include: a) structure to support capture and use of provenance of the data and

information stored in the ontology, b) use of the *Characters* entity to specify allowed characters for each asset of the application to support customization of generic security knowledge to detect input validation vulnerabilities, and c) capture and use of severity and likelihood information of *Threats*, *Weaknesses*, and *Effect (*of the threat) to customize and prioritize based on user needs. The designed ontology can be found at the link: tps://github.com/bgurijala/SecurityOntology and is published through http://ontology.cybershare.utep.edu/security/ida-o. The main entities of the diagram include the following:

1. *Asset*: This is subclass of *Entity* as described in Prov-O. Asset can be categorized as a tangible or intangible asset. Tangible assets are physical objects that are not applicable here. In this case, the focus is on intangible assets that include Web forms, database, and software that are used to access the Web application. Software includes virtual elements that possess processing characteristics such as browser, text editor, or operating system. *Asset* belongs to an *Organization* (subclass of *Agent* entity that comes from Prov-O) which in this case is Web application. *Vulnerable A*ssets entity "*is a*" *Asset* that is vulnerable to a XSS-attack. The assets can be ranked based on sensitivity and risk analysis, which will be useful for prioritization and test case generation. Each asset has its associated provenance captured.
2. *Asset Range*: This is a subclass of *Entity* as described in Prov-O. The range information for each of the assets of the application is captured using this entity.
3. *Asset Format*: This is a subclass of *Entity* as described in Prov-O. The format information for each of the assets of the application is captured using this entity.
4. *Server Locations*: This is a subclass of *Entity* as described in Prov-O. This entity captures all trusted and allowed server locations information for each of the assets of the application.
5. *IO Locations*: This is a subclass of *Entity* as described in Prov-O. This entity captures all allowed IO locations information for each of the assets of the application.

6. *Characters*: This is a subclass of *Entity* as described in Prov-O. This entity captures all allowed characters' information for each of the assets of the application, including the representation of the character and its malicious representations.
7. *CharacterSet*: This is a subclass of *Entity* as described in Prov-O. This entity captures the set of characters formed by incrementally combining Characters and their possible malicious representations.
8. *Weakness*: This is a subclass of *Entity* as described in Prov-O. Weakness is associated with an asset. Each weakness can be mitigated in various ways, which is captured by *Control Measures*. The weakness information is captured along with its provenance. The "*is derived from*" relation captures the weaknesses from which a particular weakness is derived, which will help in multiple levels of checking for possible and applicable weaknesses that may not be readily visible. This provides a relatively comprehensive way of checking for known possible weaknesses.
9. *Threat*: This is a subclass of *Entity* as described in Prov-O. The weaknesses of the assets are exploited by threats. In this case, threats are mostly in the form of malicious scripts. The threat related information is captured along with their provenance. The "*is derived from*" relation of the threats captures the threats from which a particular threat is derived, which will help in multiple levels of checking for possible exploits that may not be readily visible. This provides a relatively comprehensive way of checking for known possible exploits.
10. *Control Measures*: This is a subclass of *Entity* as described in Prov-O. The weaknesses of the assets can be mitigated by implementing control measures on vulnerable assets to protect them against possible threats.
11. *Effect*: This is a subclass of *Entity* as described in Prov-O. This entity captures the effect related information of a *Threat*. The effect of a security threat can be broadly categorized into its effect on confidentiality, integrity, or availability.

12. *Effect Severity*: This is a subclass of *Entity* as described in Prov-O. This entity captures the severity of the *Effect* entity. This information can be used for filtering and/or prioritizing.
13. *Pattern*: This is a subclass of *Entity* as described in Prov-O. Each threat can be formalized as patterns, which is captured by the *Pattern* class. These formalized patterns of threats will guide the process of looking for possible malicious representations of a particular threat under consideration.
14. *Severity*: This is a subclass of *Entity* as described in Prov-O. This entity captures the severity information of the *Threat* and *Weakness* entities. This information can be used for filtering and/or prioritizing.
15. *Likelihood*: This is a subclass of Entity as described in Prov-O. This entity captures the likelihood information of the *Threat* and *Weakness* entities. This information can be used for filtering and/or prioritizing.
16. *Creation*: This is a subclass of the *Activity* entity as described in Prov-O. This entity captures the information associated with the creation of an *Entity*.
17. *Modification*: This is a subclass of the *Activity* entity as described in Prov-O. This entity captures the information associated with the modification of an *Entity*.
18. *Retrieval*: This is a subclass of the *Activity* entity as described in Prov-O. This entity captures the information associated with the retrieval of an *Entity*.

The other entities shown in Fig. 3.2.1 are those entities that are described in Prov-O [181].

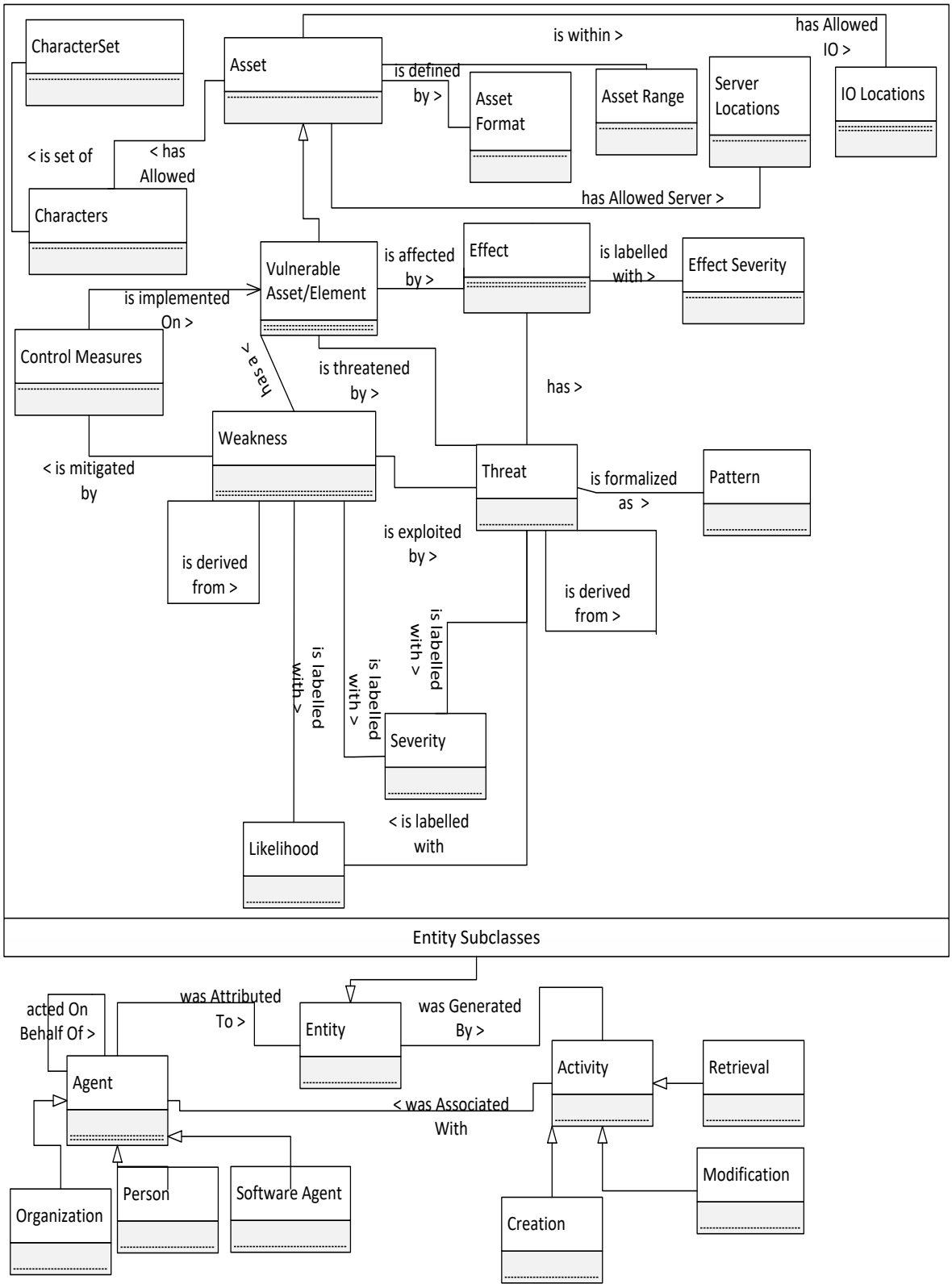

Figure 3.2.1: The IDA ontology for documenting XSS-attacks knowledge.

3.3 STEP 1: IDENTIFY AND CAPTURE APPLICATION-SPECIFIC ASSET INFORMATION

The attributes of each of the related real-world objects are considered as assets. For each attributes of the related real-world objects, the approach documents information that include the Five Primary Security Input Validation (FPSIV) attributes (i.e., type, length, character-set, format, and range) [182]. The FPSIV information can be gleaned from the Software Requirements Specification (SRS) or from the stakeholders' inputs. The FPSIV captured for each of the assets of an application form the integral part of the anomaly detection model. The Asset Character-Set is used as input for attaching and customizing blacklist patterns for misuse detection. All the asset-specific information captured is stored in the KB. Other information captured about the assets such as sensitivity level, what kind of scripts can be accepted (inline or called), trusted server names and IO locations, and script names guide the anomaly detection model. The trusted server and IO locations information captured help remediate introduction of untrusted or malicious scripts from third party Web sites or locations. The format information of an asset depicts the allowable patterns of IO, which we reference as whitelist patterns. This application-specific information is stored in the KB, which is later used to customize knowledge for monitoring. A sample template capturing all the necessary asset-specific information described above for one of the related real-world objects for an open-source project management (OPM) Web applilcation, is presented in Table 3.3.1. Provenance of the creator or modifier or retriever is captured accordingly, which will be used for credibility, trust, and decision-making purposes.

3.4 STEP 2: BUILD THE KNOWLEDGE BASE

This step involves retrieving XSS-related information such as CAPEC, CWE, OWASP XSS cheat-sheets, and attack payloads from various online sources and populating the KB. This step is done once at the start and is updated only when new versions or updates are available in the sources.

3.4.1 Populate Attack (Threat) Information into KB

Automated process retrieves the user-specified version of attack patterns' xml and xsd files maintained by MITRE Corporation from their Website (http://capec.mitre.org/data/index.html#downloads). The xml file is parsed to retrieve attack-related useful information along with its associated provenance to populate the KB. The implementation automated the xml file loading, parsing and populating the KB with attack information along with provenance. When threat-related information is retrieved from the xml file, the Retrieval sub-class of Activity captures the provenance about the retriever along with the provenance retrieved from the xml file. The threat-related information that is retrieved from CAPEC xml file includes: ID, name, summary, severity, likelihood, related weaknesses, related attack patterns, CIA impact, technical context, and content history.

Content history provides useful provenance about the creation/modification of the attack pattern. Architectural paradigm, framework, platforms, and languages information part of the technical context is retrieved to determine the applicability of the threat depending of the technical context of the application under consideration. The CIA impact captures the effect a particular threat has on the confidentiality, integrity, and availability of the assets and/or the application. This helps prioritize threats in order of what the stakeholders consider most important attribute for the application under consideration. Related attack patterns provide information about the threats that derive the threat under consideration and the threats that can be derived from threat under consideration. These drive the testing for threats that are related to the threat under consideration. Related weaknesses captures all weaknesses that the threat under consideration is known to exploit. The Severity and Likelihood of a threat captured guides the prioritization process. The summary information is one of the information that is used for construction of threat patterns, which means they help in formalizing threats as patterns that guide the detection of similar attacks that can be launched.

All the information retrieved from CAPEC xml file is stored in the security ontology (KB) as data properties or object properties of threat entity. Severity, Likelihood, related attack patterns,

related weaknesses, technical context, and content history are stored as object properties of *Threat*. Other retrieved information such as ID, name, and summary are stored as data properties of the *Threat Entity*.

Table 3.3.1: Information gathered from related real-world objects associated with the OPM.

Related Real-World Object	Fields	Type	Length	Character-set	Format	Range	Sensitivity Level	Script	Called/ Inline	Server	Script Name	IO Locs
Projekte	ID	Int*	10	Numeric	[0-9]{1,10}	0 to 4,294,967,295	2	No	N/A	N/A	N/A	N/A
	Name	Varchar	25	Alphanumeric, Apostrophe, period, hyphen	[a-zA-Z]{1,25}['.-][a-zA-Z]{1,25}		4	No	N/A	N/A	N/A	N/A
	Desc	Text		Alphanumeric, all special characters, images, links			4	Yes	Both	X, Y, Z	convImg	A, B, C
	Start	Varchar	10	Numeric, /, ., and -			4	No	N/A	N/A	N/A	N/A
	End	Varchar	10	Numeric, /, ., and -			4	No	N/A	N/A	N/A	N/A
	Status	Tinyint	1				2	No	N/A	N/A	N/A	N/A
	Budget	Float	12	Numeric, period, comma		0 to +3.4E+38	8	No	N/A	N/A	N/A	N/A

3.4.2 Populate Weakness Information into KB

Automated process retrieves the user-specified version of weakness enumerations' xml and xsd files maintained by MITRE Corporation from their Website (https://cwe.mitre.org/data/index.html#downloads). The xml file is parsed to retrieve weakness-related information along with its provenance to populate the KB. The current implementation automates the xml file loading, parsing, and populating the KB with weakness information along with provenance. When weakness-related information is retrieved from the xml file, the *Retrieval* sub-class of *Activity* captures the provenance about the retriever along with the provenance retrieved from the xml file. The weakness-related information that is retrieved from CWE xml file includes: ID, name, description summary, applicable platforms, likelihood, relationships, related attack patterns, and content history.

The content history provides useful provenance about the creation/modification of the weakness. Applicable platforms contain languages, operating systems, hardware architectures, architectural paradigms, environment, technology or common platforms on which the weakness may exist. Related attack patterns contain all threats that can exploit the weakness under consideration. This information drives the testing for threats that are related to the weakness under consideration. Relationships captures all weaknesses that derive the weakness under consideration and all weaknesses that can be derived from weakness under consideration. The likelihood of a weakness captured guides the prioritization process.

All the information retrieved from CWE xml file is stored in the security ontology (KB) as data properties or object properties of *Weakness* entity. Likelihood, related attack patterns, applicable platforms, relationships, content history are stored as object properties of *Weakness*. Other retrieved information such as ID, name, and description summary are stored as data properties of the *Weakness* entity.

3.4.3 Populate XSS-related Cheat-Sheets Information into KB

The latest version of XSS-related cheat-sheets is retrieved from OWASP Website. XSS filter evasion cheat-sheet includes various representations that can be used to bypass trivial checks for malicious strings. This information is categorized based on the character-set. Each of the possible malicious representations of a particular character using a character-set is captured in the KB and organized based on categories of the character-set. The possible malicious representations are added as data properties to the KB. The character-sets are categorized based on the minimal set of characters needed to construct a malicious string. Characters are incrementally added to the set, yielding a hierarchy of character-sets. Provenance of the retrieval is captured and added to aid in enhancing credibility, trust, and decision-making process.

3.4.4 Categorize and Populate Identified XSS Attack Payloads Information into KB

The XSS attack payloads retrieved from online sources are identified and categorized based on the character-set that is used in them. Each of the identified attack payload categories are associated/attached to their respective character-set categories created in Step 3.3 based on the characters used in the payload. The pairs of character-set categories and associated attack payloads, referred to as CSAP, are stored in the KB. The categorized attack payloads are added as object properties to the *Characters* entity. The attack payloads (specific examples/instances of attack strings used to launch XSS-attacks) are primarily obtained from online sources. Other sources of attack payloads include the output from fuzzers used in tools like Burp or Zed Attack Proxy (ZAP). Fuzz testing or fuzzing is a black-box software testing technique that involves using malformed or semi-malformed data injection in an automated fashion to find implementation bugs. A fuzzer is a program that injects automatically the fuzz inputs into a program to detect bugs. In case of security testing, the attack payloads are used to generate fuzz inputs. Provenance of the retrieval is captured and added to provide the capability to make decisions based on credibility and trust. Table 3.4.4 presents example categories and associated attack payloads.

Table 3.4.4: Character-Set and Associated Attack Payloads

Category Name	Character-Set Category	Attack-Payload Set (APS)	CSAP
C1	Alphanumeric U {&, ;, =, /, #, \, “, `}	{ P1}, where P1 = <SCRIPT a=`>` SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT>	(C1,APS^{C1})
C2	Alphanumeric U {&, ;, =, /, #, \, “, ‘}	{Q1, Q2, Q3}, where Q1= <SCRIPT a=\">\" " SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> Q2 = <SCRIPT \"a='>'\" SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> Q3 = <SCRIPT a=\">'>\" SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT>	(C2, APS^{C2})
X1	Alphanumeric U {&, ;, =, /, #, \, “}	{R1, R2, R3, R4, R5, R6}, where R1 = <SCRIPT/SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> R2 = <SCRIPT SRC=\"http://ha.ckers.org/xss.jpg\"></SCRIPT> R3 = <SCRIPT a=\">\" SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> R4 = <SCRIPT =\">\" SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> R5 = <SCRIPT/XSS SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT> R6 = <SCRIPT>document.write(\"<SCRI\");</SCRIPT>PT SRC=\"http://ha.ckers.org/xss.js\"></SCRIPT>	(X1, APS^{X1})
X2	Alphanumeric U {&, ;, =, /, #}	{S1, S2}, where S1 = <SCRIPT SRC=//ha.ckers.org/.js> S2 = <SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>	(X2, APS^{X2})

3.5 STEP 3: DERIVE THREAT-SPECIFIC BLACKLIST PATTERNS FROM KNOWLEDGE

Attack payloads are often directly used in penetration testing; however, using attack payloads directly in an intrusion-detection system is not practical or scalable because of the variations that can occur. Defining a pattern addresses this issue by generalizing similar attack-payloads aiding in detection of similar intrusions/attacks.

The blacklist patterns are derived for each of the threats using threat-related knowledge from CAPEC, CWE knowledge attached to threats, and populated knowledge from OWASP cheat-sheets stored in the KB. The generic blacklist patterns derived for each of the threats are represented as regular expressions. Currently, the derivation of blacklist patterns for each of the threats is performed manually using the example attack strings, summary, and known vulnerabilities of the threat. The derived patterns are checked/verified for accuracy using the CSAP knowledge to ensure patterns' coverage and feedback is provided to make the blacklist patterns precise. The verified/checked blacklist patterns representing formalized attack payloads are stored in the KB to facilitate detection of intrusions through pattern matching.

A threat can be formalized as multiple blacklist patterns using the knowledge base, e.g., attack pattern CAPEC 199, which documents that XSS exploits case insensitivity. If filters look for the word "script," then using "Script" or "ScRiPT" could bypass the filter. Similarly, having <scriscriptpt> can also bypass and turn out to be potentially malicious if filter replaces script by empty string. An example of one such threat-specific pattern derived for CAPEC 199 is shown in Fig. 3.5.1:

```
(leftAngularParenthesis{1,2})(?i)s((?i)(script)*)(?i)c((?i)(script)*)(?i)r((?i)(script)*)(?i)i((?i)(scri
pt)*)(?i)p((?i)(script)*)(?i)t((?i)(script)*)(rightAngularParenthesis{1,2})
```

Figure 3.5.1 Blacklist pattern sample

3.6 STEP 4: APPLICATION-SPECIFIC CUSTOMIZATION OF KNOWLEDGE

3.6.1 Infer Knowledge

Once all the information is retrieved from data sources and application-specific information is captured from stakeholders and experts, we infer knowledge from the ontology. Pellet reasoner (Version 2.4.0) is used to infer knowledge from the ontology. The inferences are performed and axioms are generated for subclasses, property assertions, class assertions, data property and object property characteristic, equivalence class, and inverse object properties. All the generated inferences in the form of axioms are added to the existing knowledge in the ontology. The inferences are saved and used in all the subsequent steps of the process. For instance, if there is an object property "i*sExploitedBy*" between the *Weakness* and *Threat* entities, then inverse object property axiom adds an object property "*exploits*" between the *Threat* and *Weakness* entities.

3.6.2 Retrieving Possible Malicious String Representations Based On Character-Set

Based on the character-set allowed by an asset (for instance, consider the allowed character-set to be: alphanumeric and &, ;, =, /, #, \, ", `) and acceptable provenance (e.g., date, or creator), possible ways of representing <script> tag and/or JavaScript element is retrieved from the KB. The results of the retrieval also include possible malicious representations of all characters that are proper subset of the allowed character-set. As an example, consider the different ways of expressing < and > based on the allowed character-set as shown in Table 3.6.2.

Table 3.6.2 Possible representations of <and > based on the allowed character-set

Possible Representations of <			**Possible Representations of >**		
&lt	&LT	<	&gt	&GT	>
<	&#60	&#060	>	&#62	&#062
&#0000060	&#x3c	&#x03c	&#0000062	&#x3e	&#x03e
&#x000003c	&#X3c	&#X03c	&#x000003e	&#X3e	&#X03e
&#X000003c	&#x3C	&#x03C	&#X000003e	&#x3E	&#x03E
&#x003C	&#X3C	&#X03C	&#x003E	&#X3E	&#X03E
\x3c	\x3C	\u003c	\x3e	\x3E	\u003e
\u003C			\u003E		

3.6.3 Construct Character-Set Specific Blacklist Patterns Using Threat-Specific Blacklist Patterns

The possible malicious string representations retrieved based on the allowed character-set in Step 3.5.2 is used to construct character-set-specific blacklist patterns (represented as **regular expressions**) using each of the derived threat-specific blacklist patterns from Step 3.4. The character-set-specific blacklist pattern is constructed for all XSS threats. For simplicity, using the same example attack pattern CAPEC 199 from Step 3.4, which exploits case insensitivity, we customize the threat-specific pattern using the knowledge retrieved for leftAngularParenthesis and rightAngularParenthesis in Step 3.5.3 to build a customized pattern as shown in Fig. 3.6.3. The regular expressions are used to perform pattern matching after transforming them into deterministic finite automata (DFA).

3.6.4 Formalize Character-Set Specific Blacklist Patterns for Pattern Matching

The customized character-set-specific blacklist patterns described in Section 3.5.3 are automatically converted into NFA (non-deterministic finite automata) and then to DFA (deterministic finite automata). The corresponding payloads (from CSAP) are then matched against the generated DFAs to check the accuracy of generated patterns in identifying malicious inputs and document the feedback i.e., the number of payloads, if any that were undetected by the pattern. The feedback is stored as log file for experts to review the accuracy of generated patterns and fine-tune it, if needed. The generated and verified DFAs for each of the blacklist patterns are stored in the KB for future use in monitoring in the IDA. The DFAs are used to perform pattern matching on the IO and flag them as described in Section 3.7.

```
leftAngularParenthesis = ((&lt | &LT | &lt; | &LT; | &#60 | &#060 | &#0060 | &#00060 | &#000060 | &#0000060 | &#60; | &#060; | &#0060; | &#00060; | &#000060; | &#0000060; | &#x3c | &#x03c | &#x003c | &#x0003c | &#x00003c | &#x000003c | &#x3c; | &#x03c; | &#x003c; | &#x0003c; | &#x00003c; | &#x000003c; | &#X3c | &#X03c | &#X003c | &#X0003c | &#X00003c | &#X000003c | &#X3c; | &#X03c; | &#X003c; | &#X0003c; | &#X00003c; | &#X000003c; | &#x3C | &#x03C | &#x003C | &#x0003C | &#x00003C | &#x000003C | &#x3C; | &#x03C; | &#x003C; | &#x0003C; | &#x00003C; | &#x000003C; | &#X3C | &#X03C | &#X003C | &#X0003C | &#X00003C | &#X000003C | &#X3C; | &#X03C; | &#X003C; | &#X0003C; | &#X00003C; | &#X000003C; | \x3c | \x3C | \u003c | \u003C))


rightAngularParenthesis = ((&gt | &GT | &gt; | &GT; | &#62 | &#062 | &#0062 | &#00062 | &#000062 | &#0000062 | &#62; | &#062; | &#0062; | &#00062; | &#000062; | &#0000062; | &#x3e | &#x03e | &#x003e | &#x0003e | &#x00003e | &#x000003e | &#x3e; | &#x03e; | &#x003e; | &#x0003e; | &#x00003e; | &#x000003e; | &#X3e | &#X03e | &#X003e | &#X0003e | &#X00003e | &#X000003e | &#X3e; | &#X03e; | &#X003e; | &#X0003e; | &#X00003e; | &#X000003e; | &#x3E | &#x03E | &#x003E | &#x0003E | &#x00003E | &#x000003E | &#x3E; | &#x03E; | &#x003E; | &#x0003E; | &#x00003E; | &#x000003E; | &#X3E | &#X03E | &#X003E | &#X0003E | &#X00003E | &#X000003E | &#X3E; | &#X03E; | &#X003E; | &#X0003E; | &#X00003E; | &#X000003E; | \x3e | \x3E | \u003e | \u003E))


(leftAngularParenthesis{1,2})(?i)s((?i)(script)*)(?i)c((?i)(script)*)(?i)r((?i)(script)*)(?i)i((?i)(script)*)(?i)p((?i)(script)*)(?i)t((?i)(script)*)(rightAngularParenthesis{1,2})
```

Figure 3.6.3 Customized blacklist pattern sample

3.6.5 Store Formalized Character-Set Specific Blacklist Patterns for Monitoring

Character-set-specific blacklist patterns formalized in Step 3.6.4 are stored in the KB for their use in monitoring IO. The formalized patterns are stored for reuse in monitoring IO of the Web application. Formalizations are performed again only if there is a change to the KB or customization requirements. This storage and reuse enhances the performance.

3.6.6 Attach Applicable Blacklist Patterns to Assets

All applicable blacklist patterns are retrieved from the KB and customized based on the allowed asset character-set for a given application. The whitelist pattern associated with an asset is also retrieved for monitoring. For each of the assets in an application, the combination of whitelist and blacklist patterns of assets involved in those use-cases are used for monitoring IO.

3.7 STEP 5: PROCESS IO

The processing of IO is performed in two sub-steps. The first step involves processing the IO and flagging those that need further processing. The second step involves processing the flagged IO.

The processing of IO involves the use of both the anomaly detection model and misuse detection model. The anomaly detection model encompasses the whitelist (derived from input constraints specified including FPSIV attributes and other information captured for assets) matching, length check, range check, and comparing with allowed locations, if applicable. The misuse detection model encompasses the customized blacklist (derived from CAPEC, CWE, and other XSS-related information) matching. The inputs and outputs of an application are compared with formalized whitelists and blacklists using pattern matching. The pattern matching is performed using DFAs representing whitelists and customized blacklists. Based on the result of pattern matching, IO is flagged. Each flagged IO is then processed using the inputs of experts/stakeholders regarding how to proceed. The details of the steps are outlined in Steps 3.6.1 and 3.6.2.

3.7.1 Flag IO

The IO is compared with the formalized whitelist and blacklist DFAs using pattern matching. This pattern matching identifies compliant, potentially malicious, malicious, and suspicious IO and flags them appropriately. The flag value for all IO is by default “Unmarked”. Identifying and flagging is performed as follows:

1. All applicable patterns' potential sub-string (that can lead to attack) will be searched for in the IO of the field. If a sub-string match is found, then the input string will be marked as "Malicious."
2. If the input string matches the described format and range for the field and, if it is not already marked as “Malicious”, then it is marked as “Compliant”.
3. If the IO matches the described format of the field and is already marked as “Malicious,” then the flag value is changed to “Potentially Malicious.”
4. If the input string neither matches sub-string of blacklist pattern nor allowable format, then it will be marked as "Suspicious."

3.7.2 Process Flagged IO

Inputs on how to process the IO for each of the flag values is captured from the stakeholders/experts. While monitoring IO and after the flagging step, the IO is processed based on the flag value. If the flag value is “Compliant,” then the normal processing of the application occurs. If the flag values are “malicious”, “potentially malicious”, or “suspicious” then the IO is processed according to the inputs captured on how to process in such cases.

CHAPTER 4: IMPLEMENTATION

This chapter presents the design, implementation, and verification of the implementation of a tool, XSSMon, that is designed to monitor the IO traffic of the Web application using the approach described in Chapter 3. Section 4.1 presents the design; Section 4.2 presents the detailed design and test strategy.

4.1 DESIGN

The design of XSSMon is performed in such a way that there is high cohesion and low coupling. High-level collaboration diagram of the implementation is as depicted in Fig. 4.1.1. The design of the tool can be broadly categorized into the following packages and classes:

1. **UserInput** class is responsible for accepting and storing user inputs for futher processing.
2. **MonitorIO** class is responsible for accepting, processing, and flagging IO.
3. **reToDFA:** This package contains class(es) essential for defining/creating regular expressions, transforming regular expressions to non-deterministic finite automata (NFA), and then to deterministic finite automata (DFA). It is also responsible for building the customized application-specific regular expressions and transforming them to DFAs using knowledge from ontology.
4. **secKMgmt:** This package contains class(es) essential for reading data from various data sources, parsing the data retrieved from all data sources, and managing (loading, updating, inferring, and querying) the ontology.

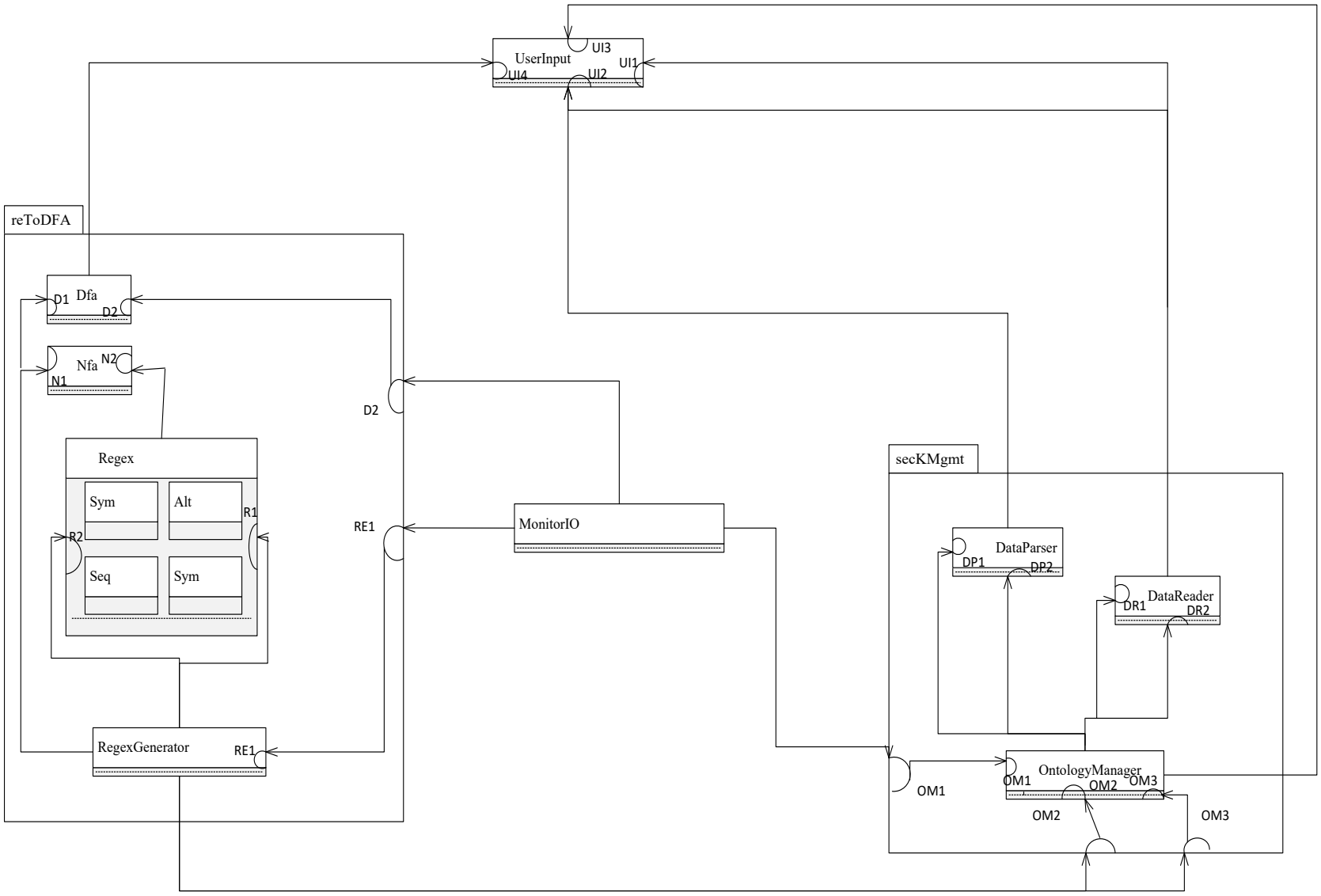

Figure 4.1.1: Collaboration diagram.

The Class-Responsibility-Collaboration (CRC) cards for the classes of XSSMon as depicted in Fig. 4.1.1 can be found in Appendix C. Summary of the contracts provided by each of the classes is presented below:

1. UserInput class provides four contracts. A brief description of each of the contracts is as follows:
    a. Knows Data Source Locations (UI1): The user-input URL locations for loading source (CAPEC and CWE) xml and xsd files.
    b. Knows Source Files Saved Location (UI2): The user-input locations for saving the loaded source (CAPEC and CWE) xml and xsd files.
    c. Knows Ontology Location (UI3): The user-input location for loading the security ontology and saving the updated and inferred ontology.
    d. Knows DFA Saved Location (UI4): The user-input location for saving the formalized patterns (customized DFAs) for reuse.
2. DataReader class provides two contracts. A brief description of each of the contracts is as follows:
    a. Load attack pattern files Contract (xml and xsd files) (DR1): The CAPEC xml and xsd files are read from the user-specified URL locations and saved into user-specified locations.
    b. Load weakness files (xml and xsd files) (DR2): The CWE xml and xsd files are read from the user-specified URL locations and saved into user-specified locations.
3. DataParser class provides two contracts. A brief description of each of the contracts is as follows:

a. Parse attack pattern file (xml) Contract (DP1): The saved CAPEC xml file is parsed and relevant information is retrieved for each of the attack patterns.

b. Parse weakness file (xml) Contract (DP2): The saved CWE xml file is parsed and relevant information is retrieved for each of the weaknesses.

4. OntologyManager class provides three contracts. A brief description of each of the contracts is as follows:

    a. Initialize Contract (OM1): The ontology is loaded from the user-specified location to update with the retrieved attack pattern and weakness information. The ontology is inferred and the inferred ontology is saved in user-specified location.

    b. Retrieve applicable malicious representations Contract (OM2): The asset-specific and applicable malicious representations are retrieved from the inferred ontology.

    c. Retrieve applicable attack patterns Contract (OM3): The applicable attack patterns are retrieved.

5. Dfa class provides two contracts. A brief description of each of the contracts is as follows:

    a. Save DFA Contract (D1): The customized blacklist and whitelist DFAs generated are saved in the user-specified location for reuse.

    b. Match Patterns Contract (D2): The formalized whitelist and blacklist patterns are matched against IO.

6. Nfa class provides two contracts. A brief description of each of the contracts is as follows:

a. Convert NFA to DFA Contract (N1): The NFA constructed from regular expression is converted to DFA.

b. Knows NFA Contract (start state, accept states, and transitions) (N2): The start, accept, and transitions of NFA for processing.

7. The Regex class provides two contracts. A brief description of each of the contracts is as follows:

    a. Make NFA Contract (R1): NFA is constructed for the regular expression.

    b. Set regular expressions (R2): The regular expressions are set to be used in construction of NFA.

8. The RegexGenerator class provides one contracts. A brief description of the contract is as follows:

    a. Customize Contract (RE1): The attack patterns are customized and formalized using asset-specific knowledge and building of regular expressions using formats of assets.

4.2 Detailed Design and Test Strategy

The verification of the implementation is performed at unit-level. In this case, each of the methods of the classes are considered as unit. To perform unit-level verification pre- and post-conditions for each of the methods are defined. Black-box testing is performed using the pre- and post-conditions. The test suite is designed using test strategy that ensures coverage of all scenarios captured by the pre- and post-conditions. The details of the pre- and post-conditions for each of the methods along with the test strategy for designing test suite can be found in Appendix D.

4.3 System Testing

Big bang integration strategy was used to integrate all the unit tested modules. System testing was performed on the system resulting from all the integrated unit tested modules. The

purpose of system testing is to check that the system as a whole works, i.e., given a set of inputs, the system yields the expected outputs.

The system level testing was performed using inputs generated from two different sources: a) string inputs generated using the xeger tool, and b) string inputs generated using ZAP's XSS file fuzzer. The asset information is used to retrieve information for performing system testing that involves anomaly detection and misuse detection by customizing blacklist patterns. The malicious inputs are derived from ZAP's XSS file fuzzer and the non-malicious inputs are derived from xeger tool. The combination of these inputs were used for system testing.

A total of 317 inputs were used to perform system testing out of which 150 input strings were derived using the xeger tool and 167 input strings were derived from ZAP's XSS file fuzzer. The number of test cases for malicious inputs were determined by the fuzz inputs generated using ZAP's XSS file fuzzer that is known to have the most comprehensive list of payloads. The ZAP's XSS file fuzzer performs fuzzing on these payloads using the processor selected to yield fuzz inputs that are used for system testing. These fuzzed inputs derived from fuzzing most comprehensive list of payloads known avoids subjective judgement as well as covers known types of malicious strings. Therefore, we believe that the number of fuzz inputs generated by the ZAP's XSS file fuzzer are good and valid to perform system-level testing and draw conclusions.

The JavaScript Unescape processor was used to generate the fuzz inputs. All 317 input strings were processed and flagged by the system. The asset on which the system level testing was performed has the same asset information as the "project description" asset of the OPM Web application presented in Chapter 5. This is because the asset information allows alphanumeric character-set including all possible special characters. This means such information would help test all possible malicious scripts generated by the fuzzer to check/measure the effectiveness of the tool in detecting XSS-related malicious inputs. The results of performing system testing are given in Table 4.3.1. All the 150 input strings generated by the xeger tool were flagged correctly as expected. This means that there were no false negatives. A total of 14 strings out of 167 input

strings were flagged incorrectly (false positives), i.e., 10.77% of malicious string generated by the fuzzer were flagged incorrectly as compliant.

Investigations were performed to determine the reasons for incorrect flagging. It was identified that the reason 14 strings were incorrectly flagged was because of incomplete knowledge in the ontology. For instance, some of the possible representations of HTML coded character-set for alphabets was not part of the ontology, which caused some of the inputs to be flagged incorrectly. Incorporating that knowledge led to correct flagging of input strings. This oversight highlights an important point: the more comprehensive the knowledge included in the ontology is, then the more precise the system is in detecting XSS-related attacks. This means that with more knowledge, the system becomes more robust against XSS-related attacks.

As described in Chapter 3, the IO is assigned a flag value of: a) suspicious if it neither matches whitelist patterns nor blacklist patterns, b) malicious if it matches only blacklist patterns and does not match whitelist patterns, c) compliant if it matches only whitelist patterns and does not match blacklist patterns, and d) potentially malicious if it matches both whitelist and blacklist patterns.

Table 4.3.1: XSSMon version 1 system test results for compliant and malicious inputs.

Input #	**Compliant**	**Percent Success**	**Potentially Malicious**	**Percent Success**	**Malicious**	**Percent Success**	**Suspicious**	**Percent Success**
150	150	100	0	100	0	100	0	100
167	14 (false positives)	91.62	116	100	20	100	17	100
Total 317	**Total** 303	95.58	**Total** 112	100	**Total** 20	100	**Total** 17	100

To affirm the results of investigations for inputs, which were incorrectly flagged as compliant, were indeed incomplete knowledge in the ontology, the HTML coded character-set for alphabets and other information were added to the ontology. The second run of system testing was performed on XSSMon version 2 and the results were recorded. The results of the system-level testing are given in Table 4.3.2. The results of the second run validate the results of the investigations for incorrect flagging of inputs was incomplete knowledge in the ontology.

Table 4.3.2: XSSMon version 2 system test results for compliant and malicious inputs.

Input #	Compliant	Percent Success	Potentially Malicious	Percent Success	Malicious	Percent Success	Suspicious	Percent Success
150	150	100	0	100	0	100	0	100
167	0	100	125	100	23	100	19	100
Total 317	**Total** 150	100	**Total** 125	100	**Total** 23	100	**Total** 19	100

Another set of fuzz inputs was generated using JavaScript Escape processor on the ZAP's XSS file fuzzer and system-level testing was performed. A total of 221 fuzzed inputs were used to perform system testing. The results of system-level testing are as depicted in Table 4.3.3. None of the fuzz inputs were flagged as *Compliant* implying that there were no false positives. The ability of the system to correctly identify and flag two different sets of fuzz inputs generated using two different processors on ZAP's XSS file fuzzer further strengthened the correctness of the system.

Table 4.3.3: Additional results of XSSMon version 2 testing results for malicious inputs.

Input #	Compliant	Percent Success	Potentially Malicious	Percent Success	Malicious	Percent Success	Suspicious	Percent Success
221	0	100	158	100	34	100	29	100

4.4 CURRENT IMPLEMENTATION OF XSSMON TOOL

The current implementation include the following features: a) loading user-specified version of CAPEC and CWE xml and xsd files, b) parsing of CAPEC and CWE xml files to retrieve relevant information, c) populating CAPEC and CWE information into ontology, d) inferring knowledge after populating and updating information, e) retrieving knowledge based on user-requested provenance, f) customizing and formalizing blacklist patterns based on allowed character-set of assets for misuse detection, g) prioritizing attack patterns based on likelihood and severity, h) retrieving asset-specific information for anomaly detection, i) formalizing formats as whitelist, and j) processing and flagging IO based on information for anomaly detection, whitelists, and blacklists.

The current implementation of XSSMon tool has the following features missing that are described in Chapter 3: a) saving the categorized attack payloads based on character-set (CSAP) used in the ontology, b) automated checking of the accuracy of generated blacklist patterns on CSAP and providing feedback, and c) saving CSBP in the ontology for reuse.

4.5 LIMITATIONS OF CURRENT IMPLEMENTATION

One of the limitations of the current implementation of XSSMon tool is the way DFAs are derived from regular expressions. The DFAs are constructed after the knowledge is retrieved and stored. Depending on the asset under consideration, the applicable DFAs are used for pattern matching. This may pose a problem if the regular expression for which DFA is being constructed is of the form: (a | b) a ((a | b) {n}). The time complexity of the approach used in the current implementation to construct DFA would take $O(2^n)$. Based on our current knowledge, this situation is not common and, therefore, the approach used in the current implementation is expected to work in all the practical situations under which the system is used. However, if the above mentioned scenario should ever occur, then a possible solution is to generate the DFA states as needed on the fly. That is to say, when such a regular expression is encountered, the DFA states that are needed are generated while processing the IO string based on the character of the IO read. This approach would impact performance and slow down processing of that particular scenario, but would help overcome the time complexity issue.

The DFAs generated from regular expressions in the current implementation are not minimal. A candidate for future work to improve the XSSMon tool is to generate minimal DFAs from regular expressions.

CHAPTER 5: CASE STUDY

This chapter presents the results of an exploratory case study that was conducted to evaluate the effectiveness of the XSSMon tool in detecting XSS attacks on an open-source project management (OPM) Web application. A *case study* is defined as "a technique for detailed exploratory investigations, both prospectively and retrospectively, that attempts to understand and explain phenomenon or test theories, using primarily qualitative analysis" [183]. Case studies can support testing complex theories in settings where there is little control over the variables, provide insights into cause and effect relationships, and answer how and why questions. The design and implementation of the case study follows the guidelines given by Easterbrook and Aranda [183].

5.1 CASE-STUDY RESEARCH DESIGN

The OPM Web application was chosen for the case study because one of its versions has been well documented to be vulnerable to XSS attacks on National Vulnerability Database (NVD) and Common Vulnerability Enumeration (CVE) Websites. For the sake of anonymity and privacy, we refer to it as the OPM original version. The tool, which is written in PHP 5, is intended for small to medium-sized businesses and freelancers. It provides an open source alternative to proprietary tools like Basecamp. The OPM was launched in November 2007, and it has been downloaded more than 500,000 times since its first release. A later version of the tool was released in 2015 to address the known XSS vulnerabilities. This version of OPM, referred to in the case study as the "OPM latest version" is not documented to be XSS vulnerable in NVD and CVE. HTMLPurifier was used in the latest version of OPM to identify, filter, and/or sanitize XSS-related malicious inputs.

The case study design is as depicted in Fig. 5.1.1. Apache server serves as the Web server, and the OPM Web application uses Mozilla Firefox browser. All the user inputs to the Web application are provided in this environment. The "add project" functionality of the application that is known to have XSS-related vulnerability is chosen. The required inputs for the fields come from the are randomly generated strings by xeger using the format information of the assets or

ZAP fuzzer generated strings. These inputs are given to the application and the requests and responses are captured. A total of 317 projects were added to the application using "add project" functionality. In these projects about 150 projects used random information generated by xeger for all the fields and the remaining 167 projects used random information generated by xeger for all fields except "desc" field for which the information generated by ZAP's file fuzzer that are known to be malicious strings. The types of input to the OPM Web application are described in Section 5.1.3.

The requests from the application and the responses from the Web-server to the application are monitored using the Network Monitor of the Firefox browser. All the requests and responses are saved as HTTP Archive format (HAR) file (JSON-formatted) from the Network Monitor of the browser to the "traffic-log file". Two traffic-log files were created: one with 150 projects and the other with 167 projects information. The traffic-log files with 150 projects can be found at the link:

https://www.dropbox.com/sh/ik41p11bqli9ytn/AACP4Uvl3NDsmG74Igag4US1a?dl=0.

The traffic-log files with 167 projects can be found at the link:

https://www.dropbox.com/sh/nxhacjiakmao602/AADF1b_t_GK6DkKN_FOj9vjda?dl=0.

In addition, the user input is saved to the "input-log file" prior to being processed by the application. The user input files corresponding to the 150 projects can be found at the link:

https://github.com/bgurijala/InputFiles/tree/master/WhitelistFiles.

The user input files corresponding to the 167 projects can be found at the link:

https://github.com/bgurijala/InputFiles/tree/master/BlacklistFiles.

The traffic-log file is parsed to create a file for each of the requests and responses. Every such created file stores the following information: time stamp, input data (as name-value pair) i.e., processed data applicable for request files, response data (name-value pair) applicable for response files, and a flag indicator for each of the asset information (name-value pairs). The flag indicator can have the following values: 00=Suspicious; 01=Malicious; 10=Compliant; and 11=Potentially Malicious. The input–log file will have an additional field indicating whether the input was

generated by the Fuzzer, indicating that it comes from a black-list pattern. The implementation for processing network traffic saved as HAR file and creating request and response files can be found at the link: https://github.com/bgurijala/NetworkIOProcess.

Another comparison was done with the OPM latest version by creating 166 projects with malicious inputs. The traffic-log files with 166 projects can be found at the link:

https://www.dropbox.com/sh/li8ooaefe24w34t/AABXFMX5JnpalNr94beneWTOa?dl=0.

All the IO will be flagged by the tool. The analysis will determine how effective was the OPM Web application was in detecting and/or dealing with malicious inputs. The results will be analyzed to determine the effectiveness of the XSSMon tool in identifying malicious and/or invalid inputs. A comparison of the effectiveness of Web application and XSSMon tool in identifying XSS-related malicious inputs is then conducted.

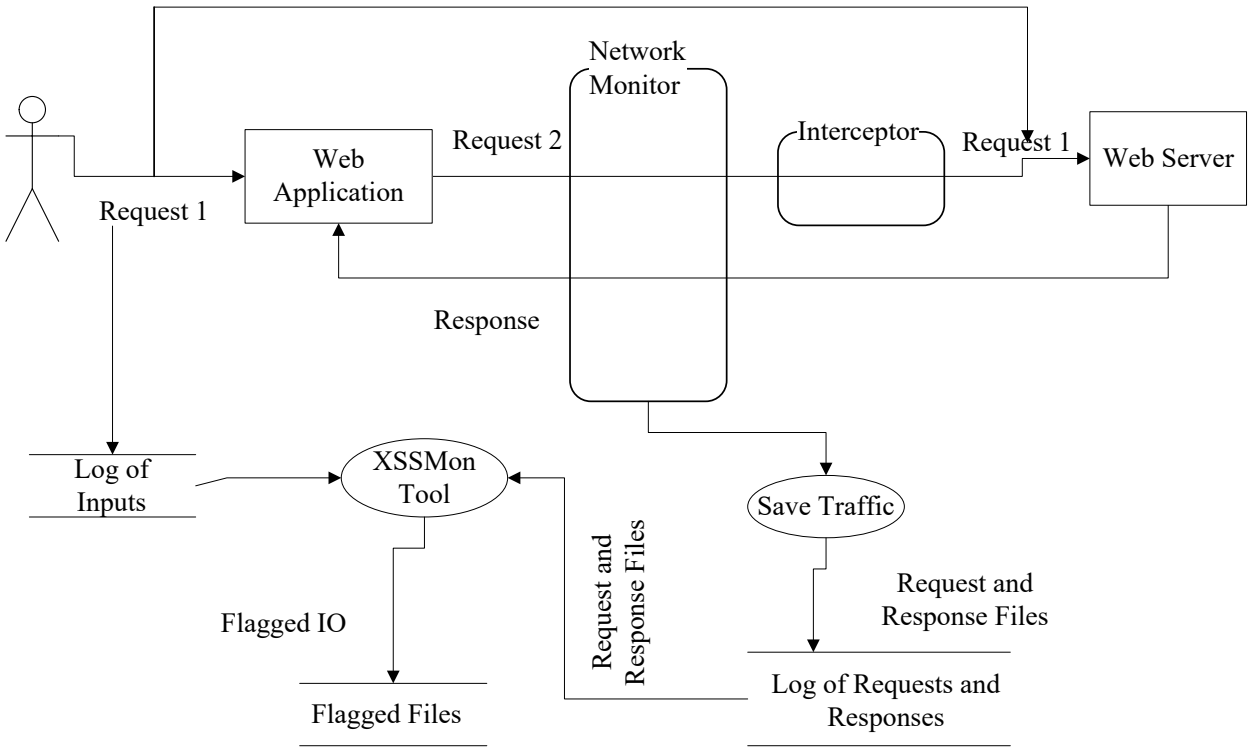

Figure 5.1.1: Dataflow diagram of case-study design.

5.1.1 Research Questions

The following are the research questions that are being posed for the case study:

1. How effective is the Web application in identifying, detecting and/or sanitizing XSS-related threats (malicious inputs) in terms of percentage of identified threats?
2. How effective is the XSSMon tool in identifying XSS-related malicious inputs in terms of percentage of identified threats?
3. How many false-positives or false-negatives does the XSSMon tool produce and why?

5.1.2 Propositions

The propositions of the case study are as follows:

1. The case study supports the finding that the OPM Web application is not effective in identifying, detecting, and/or sanitizing XSS-related threats.
2. The XSSMon tool is effective in identifying XSS-related malicious inputs that may pass due to lack or improper sanitization performed by the application.
3. The number of false positives and false negatives produced by the tool are less than 5%.

5.1.3 Units of Analysis

The possible types of inputs to the application include: a) compliant and safe strings, b) compliant and potentially malicious strings, c) non-compliant and malicious strings, and d) non-compliant and non-malicious strings. The malicious strings for the input are derived from XSS file fuzzer of Zed Attack Proxy (ZAP). The randomized XSS fuzz inputs from ZAP are used as attack strings. These form the inputs for categories of inputs b) and d) described above. The strings generated by xeger using format information of assets along with some inputs from fuzzer after removing malicious part is used as input for categories a) and c) described above. Randomized inputs from fuzzer and xeger are used to avoid subjective judgement.

The equivalence classes involving the assets of the application considered for the case study include: a) numeric characters only fields, b) alpha characters only fields, c) alphanumeric

characters only fields, d) numeric characters with some allowed special characters (such as -, /, , etc.) that cannot result in malicious inputs, e) numeric characters with some allowed special characters that can result in malicious inputs, f) alpha characters with some allowed special characters that cannot result in malicious inputs, g) alpha characters with some allowed special characters that can result in malicious inputs, h) alphanumeric characters with some allowed special characters that cannot result in malicious inputs, and i) alphanumeric characters with some allowed special characters that can result in malicious inputs. Web pages involving assets from these equivalence classes are considered in the case study that will be representative of all assets of the application. IO for assets belonging to these classes are collected at three levels as depicted in Fig. 5.1 and fed to the designed monitor for processing and flagging IO.

5.1.4 Logic Linking the Data to the Propositions

The IO captured at three levels as mentioned should be marked as follows by the tool: a) All inputs that match blacklist patterns and do not match whitelist patterns defined for an asset under consideration should be marked as malicious, b) All inputs that do not match blacklist patterns and whitelist patterns defined for an asset under consideration should be marked as suspicious, c) All inputs that match both blacklist and whitelist patterns defined for an asset under consideration should be marked as potentially malicious, and d) All inputs that do not match blacklist patterns and match whitelist patterns defined for an asset under consideration should be marked as compliant.

The following two comparisons are performed to determine the effectiveness of the application in identifying, filtering, and/or sanitizing XSS-related malicious IO: a) A comparison is performed between the flagged application unprocessed IO (user inputs to application) and flagged application processed IO (requests from application to server), and b) A comparison is performed between the flagged application unprocessed IO (user inputs to the application) and flagged application processed IO (responses from server to the application).

The number of application-processed IO marked as malicious, potentially malicious, and suspicious provides a tally of how well IO validation is performed by the designed tool in comparison to the application under consideration.

The percentage of IO that is not malicious and contains allowed character-set but is marked as malicious, potentially malicious, or suspicious gives an estimate of the false positive generated by the tool. Similarly, the percentage of IO that is malicious but is not marked as malicious or potentially malicious gives an estimate of the false negatives generated by the tool.

5.1.5 Criteria for Interpreting the Findings

Analytical generalization is performed. The false positives and false negatives of the misuse detection approach of the method are determined as follows: a) the flag values of IO that are known to be malicious are checked to determine how many of those are marked as compliant which represents the number of false positives, and b) the flag values of IO that are known to be compliant are checked to determine how many of those are marked as malicious or potentially malicious which represents the number of false negatives. Similar check is performed for determining false positives and false negatives of the anomaly detection approach of the method.

5.2 CASE STUDY SETUP

The related-real world objects information of OPM along with other necessary information about assets that needs to be captured as described by the template in Chapter 3 is presented in Table 5.2. The table depicts the information about all assets that belong to the Project object which is stored in the ontology as application-specific information. This information is used by the anomaly detection part of the approach and supports customization of blacklist patterns.

The “add project” Web page captures: a) project name, b) project description, c) due date of the project, and d) budget of the project. The inputs for all these assets were generated in two ways: (Process a) inputs for all assets were generated by xeger using the format information of the assets, and (Process b) inputs for project name, due date, and budget w generated using format

information for these assets and the input for project description comes from the ZAP's XSS file fuzzer generated strings.

A total of 150 project-related non-malicious inputs were generated using (Process a) and 167 project-related malicious inputs were generated using (Process b). All the 317 inputs for "add project" were saved as "input-log" files. The requests corresponding to 150 non-malicious project-related inputs were not intercepted, and the network traffic was recorded. The requests corresponding to 167 malicious project-related inputs were intercepted using the Burp suite tool, and the requests (processed user inputs) were changed back to original user inputs and passed to the server and the network traffic is recorded. A similar setup was used for conducting the case study on the latest version of OPM. A total of 166 project-related malicious inputs were generated using (Process b).

All the recorded network traffic was processed and for each of the projects added two files were created: a) a file with the request information along with the time stamp, and b) a file with corresponding response information along with time stamp.

All the "input-log" files, request files, and response files were then processed by the XSSMon tool. The XSSMon tool processed (performs anomaly and misuse detection) each of the asset IO and flags them. The flag value of each of the asset IO was then written to the corresponding files.

5.3 Discussion of Results

This section presents all the results of performing the case study as described in Sections 5.1 and 5.2. The processed and flagged "input-log" files (unprocessed user input) for 150 projects using XSSMon tool can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Input/Whitelist. The processed and flagged "input-log" files for 167 projects can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Input/Blacklist. The processed and flagged request (processed user input by application) and response (server processed user input) files for 150 projects using XSSMon tool for the first run of the case study can be found

at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Whitelist/First%20Run. The processed and flagged request and response files for 167 projects using XSSMon tool for the first run can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Blacklist/First%20Run. The processed and flagged request and response files for 150 projects using XSSMon tool for the second run of the case study can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Whitelist/Second%20Run. The processed and flagged request and response files for 167 projects using XSSMon tool for the second run of the case study can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/Blacklist/Second%20Run. The processed and flagged request and response files for 166 projects using the XSSMon tool for the latest version of OPM can be found at the link: https://github.com/bgurijala/ProcessedFiles/tree/master/LatestVersion/Blacklist.

A summary of the results is provided in Table 5.3.1. The results presented in Table 5.3.1 correspond to the case study results using XSSMon version 1 tool (with incomplete ontology knowledge as described in Section 4.3). As shown in Table 5.3.1, a total of 12 requests were successfully identified/filtered/sanitized out of the known 167 malicious inputs, which implies that the client-side of OPM original version application has a success rate of 7.19%. The server-side of OPM original version identified/filtered/sanitized 4 out of 167 malicious inputs that it received yielding a success rate of 2.4 percent. The XSSMon version 1 tool identified 130 out of 155 (167 – 12 sanitized) malicious inputs in requests with a success rate of 83.87 percent. It identified 151 out of 163 (167 – 4 sanitized) malicious inputs in responses with a success rate of 92.63%. The combined malicious inputs identified in both request and responses by the OPM original version Web application is 16 out of a total of 334 malicious inputs, yielding a success rate of 4.79 percent. In comparison, the XSSMon version 1 tool identified a total of 281 out of 318 malicious request and response inputs resulting in a success rate of 88.36 percent. The statistical significance of the results is discussed later.

Table 5.3.1: Summary of XSS detection effectiveness results for OPM original version and XSSMon tool version 1.

Processed by	**Malicious Inputs Total**	**Successfully Identified/Filtered /Sanitized Requests**	**Percent Success**	**Total Malicious Inputs**	**Successfully Identified/Filtered/ Sanitized Responses**	**Percent Success**
OPM Original Version Client-Side	167	12	7.19	N/A	N/A	N/A
OPM Original Version Server-Side	167	N/A	N/A	167	4	2.4
XSSMon version 1	167 – 12 = 155	130	83.87	167 – 4 = 163	151	92.63

Table 5.3.2: Breakdown of flagged malicious requests and responses identified by XSSMon tool version 1.

Input # N = 167	**Compliant**	**Potentially Malicious**	**Malicious**	**Suspicious**
Requests	37	92	17	21
Responses	16	115	12	24
Total = 334	53	207	29	45

The summary of the flagged malicious inputs in requests and responses by XSSMon tool version 1 is given in Table 5.3.2. The 37 requests that were flagged as compliant are the requests that were sanitized by the client-side of the Web application and the 16 responses that were flagged as compliant are the responses that were sanitized by the server-side of the Web application. Some of the requests and responses are flagged due to incomplete/missing knowledge in the ontology. We consider these as false positives; thus, at least 25 out 167 requests and 12 out of 167 responses are incorrectly flagged as compliant. This implies the false-positives for requests is 14.97 percent and the false-positives for responses is 7.19 percent. The overall false-positives of the XSSMon version 1 tool is 11.07 percent. All the requests and responses that are known to be compliant are

correctly flagged as compliant. This implies that the tool did not yield any false negatives in this case study.

Table 5.3.3: Summary of flagged malicious input-log files by XSSMon version 1 tool.

Inputs #	Compliant	Potentially Malicious	Malicious	Suspicious
167	14	116	17	20

The summary of the flagged malicious inputs in the input-log files (unprocessed inputs) by the XSSMon version 1 tool is shown in Table 5.3.3. As mentioned previously, the 14 inputs are flagged incorrectly (as compliant) due to lack of knowledge in the ontology about various possible malicious representations. Thus, the false positives are 8.38 percent. None of the known compliant inputs in the input-log files were incorrectly flagged for the first run of the case study resulting in zero percent false-negatives.

The results depicted in the tables of this section use formats of the assets, i.e., the white list, that were generally defined, that is to say the formats were defined to include any possible sequence of allowed character-set. In practical situations, if the formats are defined to accept only certain possible sequence of allowed characters based on the need, then the results are expected to be more promising as the approach described in Chapter 3 uses both anomaly and customized misuse detection approaches to enhance security of the Web application. The formats for the assets are defined generally on purpose to avoid bias and subjective judgement.

Further tests were run using XSSMon version2. The results are depicted in Table 5.3.4. The number of malicious inputs for XSSMon tool version 2 were obtained by deducting number of request and responses sanitized by the OPM original version because the sanitized requests and responses are no longer malicious. The XSSMon version 2 tool identified and flagged a total of 155 requests as non-compliant out of 155 un-sanitized requests and 163 were flagged as non-compliant out of 163 responses. This means the tool has a 100 percent success rate in identifying XSS-related attacks. The XSSMon tool identified and flagged all malicious inputs that were not

sanitized by the OPM original version application (both client-side and server-side). The percentage of false-positives and false-negatives were zero.

Table 5.3.4: Summary of XSS detection effectiveness results for OPM original version and XSSMon tool version 2.

Processed by	**Malicious Inputs Total**	**Successfully Identified/Filtered/Sanitized Requests**	**Percent Success**	**Total Malicious Inputs**	**Successfully Identified/Filtered/Sanitized Responses**	**Percent Success**
OPM Original Version Client-Side	167	12	7.19	N/A	N/A	N/A
OPM Original Version Server-Side	167	N/A	N/A	167	4	2.4
XSSMon Tool	167 – 12 = 155	155	100	167 – 4 = 163	163	100

Table 5.3.5: Breakdown of flagged malicious requests and responses identified by XSSMon tool version 2.

Inputs # N = 167	**Compliant**	**Potentially Malicious**	**Malicious**	**Suspicious**
Requests	12	112	21	22
Responses	4	124	14	25
Total = 334	16	236	35	47

The summary of the flagged malicious inputs in requests and responses by the XSSMon version 2 tool is given in Table 5.3.5. The 12 requests that are flagged as compliant are the requests that are sanitized by the client-side of the Web application and 4 responses that are flagged as compliant are the responses that are sanitized by the server-side of the Web application.

Table 5.3.6: Summary of flagged malicious input-log files by XSSMon version 2 tool.

Inputs #	**Compliant**	**Potentially Malicious**	**Malicious**	**Suspicious**
167	0	125	23	19

The summary of the flagged malicious inputs in the input-log files (unprocessed inputs) by the XSSMon tool version 2 is given in Table 5.3.6. After updating knowledge in the ontology none of fuzzer generated malicious inputs were flagged as *Compliant* as expected.

The statistical significance of the results obtained from the case study was determined by computing binomial proportion confidence interval for: a) the number of malicious inputs sanitized or filtered by OPM original version (the Web application), and b) the number of malicious inputs identified by XSSMon tool versions 1 and 2. If the two computer binomial proportion confidence intervals do not overlap, we can conclude that there is statistically significant difference between OPM original version and XSSMon tool in identifying XSS attacks at the given level of confidence. The binomial proportion confidence interval calculator used to computer the intervals used the Clopper-Pearson (exact) method. There are a total of 334 files (167 request and 167 response files) with malicious inputs. The binomial proportion confidence intervals are computed for both versions of XSSMon. The values used for computing are as shown in Table 5.3.7.

Table 5.3.7: Summary of flagged malicious Requests and Responses by OPM original version and XSSMon tool for versions 1 and 2.

Processed by	**Inputs #**	**Sanitized/Filtered**	**Unidentified**
OPM Original Version	334	16	318
XSSMon version 1	318	281	37
XSSMont version 2	318	318	0

The binomial proportion confidence interval with 99% confidence level for OPM original version is between 0.022 and 0.086. The binomial proportion confidence interval with 99% confidence level for XSSMon version 1 is between 0.83 and 0.93. The binomial confidence interval with 99% confidence level for XSSMon version 2 is between 0.98 and 1. The confidence intervals of the XSSMon tool for both versions do not overlap with the confidence interval of OPM original version. In addition, the confidence interval for both versions of the XSSMon tool is

higher than the OPM original version. Thus, both versions of the XSSMon tool detected XSS attacks at a higher probability of success than the OPM original version.

To further substantiate the findings, the latest version of OPM Web application, which is currently not documented as having XSS vulnerability, was compared to XSSMon version 2. The results are given in Table 5.3.8, which shows that the latest version of OPM is still vulnerable to XSS attacks. The use of HTMLPurifier, which was incorporated into the latest version of OPM, helped in identifying, filtering, and/or sanitizing XSS attacks better than its previous version. However, the XSSMon version 2 tool identified all XSS-related malicious inputs that were unidentified by the Web application. The OPM latest version client-side identified, filtered, or sanitized only 11 out of 166 requests correctly which results in success of 6.63 percent. The OPM latest version server-side identified, filtered, or sanitized 82 out of 166 responses correctly which resulted in a success of 49.4 percent. The XSSMon version 2 tool identified and flagged a total of 155 as non-compliant out of 155 un-sanitized requests and 84 are flagged as non-compliant out of 84 responses. This means XSSMon version 2 had a 100% success rate in identifying XSS-related attacks. The XSSMon tool identified and flagged all malicious inputs that were not sanitized by the OPM latest version application (both client-side and server-side). The percentage of false-positives and false-negatives is zero.

Table 5.3.8: Summary of OPM latest version and XSSMon tool version 2 detection effectiveness.

Processed by	**Malicious Inputs Total**	**Successfully Identified/Filtered/Sanitized Requests**	**Percent Success**	**Total Malicious Inputs**	**Successfully Identified/Filtered/Sanitized Responses**	**Percent Success**
OPM Latest Version Client-Side	166	11	6.63	N/A	N/A	N/A
OPM Latest Version Server-Side	166	N/A	N/A	166	82	49.4
XSSMon version 2 Tool	166 – 11 = 155	155	100	166 – 82 = 84	84	100

The statistical significance of the results obtained from the case study is determined using the computing binomial proportion confidence interval as described earlier. There are a total of 332 files (166 request and 166 response files) with malicious inputs. The binomial proportion confidence intervals were computed for both OPM latest version and XSSMon version 2 tool. The values used for the computation are given in Table 5.3.9.

Table 5.3.9: Summary of flagged malicious requests and responses by OPM latest version and XSSMon version 2 tool.

Processed by	**Inputs #**	**Sanitized/Filtered**	**Unidentified**
OPM Latest Version	332	93	239
XSSMon version 2	239	239	0

The binomial proportion confidence interval with 99 percent confidence level for OPM latest version is between 0.22 and 0.35. The binomial confidence interval with 99 percent confidence level for second run of the tool is between 0.98 and 1. The confidence intervals of the XSSMon version 2 tool does not overlap with confidence interval of OPM latest version, and the confidence interval for the XSSMon version 2 tool is higher than OPM latest version. This confirms that the XSSMon tool can detect XSS attacks with a higher probability of success than OPM latest version.

According to the 2015 EdgeScan report [184], XSS density is 4.87 per Web application. The likelihood of XSS vulnerability being discovered in Web applications is 52% [184]. The most vulnerable application had 220 instances of XSS [185].

5.4 THREATS TO VALIDITY

This section presents the threats to validity of the case study performed as described in Section 5.1. There are four primary types of validity: construct validity, internal validity, external validity, and reliability. The construct validity issues arise when there are errors in measurement. Errors in measurement are not an issues in the case study since xeger and ZAP's XSS file fuzzer are used to automatically generate random inputs for the assets of the Web application under

consideration. The internal validity applies to explanatory and causal type of case studies; thus, it is not applicable to this case study. The external validity issues may arise from the fact that all the data of the case study is from one Web application. Extending the case study to other Web applications can strengthen the results of the case study. The reliability aspect of the validity of the case study involves demonstrating that the operations of the case study can be repeated with the same results. A prerequisite for reliability is documented procedures for the case study. Another aspect of the validity of case study is addressed by detailed and comprehensive documentation of the case study research design with all the five parts of the case study: a) research questions, b) propositions, c) units of analysis, d) logic linking the data to the propositions, and e) criteria for interpreting the findings. All the files used and produced by the case study are stored in the github and dropbox and the links for the files are provided in corresponding sections of this chapter.

Table 5.2: Related Real-World Objects along with other necessary information.

Related Real-World Object	Fields	Type	Length	Character-set	Format	Range	Sensitivity Level	Script	Called /Inline	Server	Script Name	IO Loc	Users
Projekte	ID	Int	10	Numeric	[0-9]{1,10}	0 to 4,294,967,295	2	No	N/A	N/A	N/A	N/A	
	Name	Varchar	255	Alphanumeric, Apostrophe, period, colon	[[((alp)||(ALP)||(num)||(ws)||(')||(.)||(:))+]]		4	No	N/A	N/A	N/A	N/A	
	Desc	Text	500	Alphanumeric, all special characters, images, links	[[((alp)||(ALP)||(num)||(ws)||(<)||(>)||(!)||(@)||(#)||($)||(%)||(^)||(&)||(*)||(()||())||({)||(})||(|)||(\)||(/)||(;)||(:)||(')||(")||(-)||(_)||(=)||(+)||(,		4	Yes	Both	X, Y, Z *	Name *	A, B, C *	

Related Real-World Object	Fields	Type	Length	Character -set	Format	Range	Sensitivity Level	Script	Called /Inline	Server	Script Name	IO Loc	Users
					)‖(.)‖(?)+]]								
	Start	Varchar	10	Numeric, /, ., and -	[[(num){1,2}]][[(.){1,1}]][[(num){1,2}]][[(.){1,1}]][[(num){4,4}]]		4	No	N/A	N/A	N/A	N/A	
	End	Varchar	10	Numeric, /, ., and -	[[(num){1,2}]][[(.){1,1}]][[(num){1,2}]][[(.){1,1}]][[(num){4,4}]]		4	No	N/A	N/A	N/A	N/A	
	Status	Tinyint	1				2	No	N/A	N/A	N/A	N/A	
	Budget	Float	9	Numeric, decimal point, comma	[[(num){1,3}]][[(,){0,1}]][[(num){3,3}]]	0 to +3.4E +38	8	No	N/A	N/A	N/A	N/A	
* Place holder. Actual values of allowed script name, server locations and IO locations are not mentioned													

5.5 GENERALIZATION OF CASE STUDY

To understand the effectiveness of the approach and the XSSMon tool, it is essential to be able to generalize the conclusion of the case study. Two approaches are applicable to the case study: analytical generalization and analogical generalization.

Analytical generalization requires a two-step process: a) the first step involves a conceptual claim whereby investigators show how their case study findings bear upon a particular theory, theoretical construct, or theoretical sequence of events, and b) the second step involves applying the same theory to implicate other, similar situations where analogous events also might occur [186]. Smaling [187] points out that analogical generalization, which is based on analogical reasoning, may be more suitable when research results obtained from one case study are to be generalized to another. Analogical generalization is plausible when there are solid arguments that, when a particular researched case has characteristics that are relevant for the research conclusions, another case that has not been researched also has these relevant characteristics. The six quality criteria for analogical reasoning include: a) the relative degree of similarity, b) the relevance for the conclusion, c) support by other, similar cases, d) support by means of variation, e) the relative plausibility of the conclusion on its own, and f) empirical and theoretical support. When all these six qualities have been fulfilled analogical reasoning can be stronger than inductive reasoning based on statistical representative sample, theory-carried generalization, or variation-based generalization.

To apply analytical generalization or analogical generalization and generalize the results to similar cases, it is important to identify the characteristics of the case study that influenced the results. The work uses anomaly and misuse detection to enhance security. The characteristics of the case study that mainly influence the results is the allowed character-set of an asset. This character-set is used in retrieving applicable blacklist patterns and customizing them. The main aim of the work is to detect XSS-attacks therefore; the identification of potential attack strings is of highest priority. The other characteristics that influence the results are: formats of the assets,

range, and other information of application-specific asset information that is used for anomaly detection. These asset-specific characteristics influence the overall flag value assigned but do not impact the detection of malicious or potentially malicious strings. Therefore, the character-set of an asset is the main characteristic that influences the results and conclusions. The character-set equivalence classes considered as part of the case study, as described in Section 5.1.3, include: a) numeric only fields, b) alpha only fields, c) alphanumeric only fields, d) numeric with some allowed special characters (such as -, /, , etc.) that cannot result in malicious inputs, e) numeric with some allowed special characters that can result in malicious inputs, f) alpha with some allowed special characters that cannot result in malicious inputs, g) alpha with some allowed special characters that can result in malicious inputs, h) alphanumeric with some allowed special characters that cannot result in malicious inputs, and i) alphanumeric with some allowed special characters that can result in malicious inputs. These equivalence classes considered as part of the case study are representative of many inputs that can be tampered to launch attacks. That said, the results of the malicious inputs detection presented in the case study can be extended to other Web applications that accept similar equivalence class as inputs on Web pages.

CHAPTER 6: RELATED WORK

Various approaches have been presented to detect and prevent XSS attacks. A total of 131 research papers on techniques for detection and prevention of XSS-related attacks and vulnerabilities were identified and surveyed. Detailed description of the surveyed papers and their categorization based on the classification given in Chapter 2 is presented in Appendix A.

This section presents other work more closely related to the dissertation work. None of the four examples listed below use the reliable knowledge sources of CAPEC and CWE to drive the detection of XSS. Other differences are the use of provenance and ontologies to reason and infer new knowledge, as well as the ability to update the knowledge bases as new attacks are identified. Furthermore, the XSSMon tool is platform and programming language independent.

Venkatakrishnan et al. [83] described WebAppArmor, a framework that incorporates static and dynamic analysis techniques, symbolic execution, and execution monitoring to prevent XSS, SQL injection, and CSRF on existing (legacy) Web applications.

A monitor embedding framework DESERVE (DEtecting program SEcurity Vulnerability Exploitations) [120] identifies exploitable statements from source code based on static backward slicing and embeds necessary code to detect attacks. DESERVE can be used for detecting buffer overflow, SQL injection, and XSS attacks.

Li and Wang [137] presented FIRM, a system that embeds inline reference monitor (IRM) in Web pages hosting Flash content and protects it by controlling DOM methods and randomizing variables with sensitive data to prevent XSS attacks.

Ruse and Basu [113] described a two-phase technique that uses both static and dynamic analysis to detect XSS vulnerabilities and prevent XSS attacks. The static analysis phase includes application translation and concolic unit testing techniques whereas runtime monitoring phase monitors relevant variables in the application.

All the described work in Appendix A are geared towards: server-side, client-side, or generic, if it is not specified whether the work focuses or server-side, client-side, or both. The

dissertation work has the ability to detect both server-side and client-side XSS attacks. In addition, none of the approaches surveyed are using the vast knowledge base available and maintained by NVD, which includes CAPEC, CWE, and CVE. The dissertation work describes an approach that uses the vast CAPEC and CWE knowledge to drive the detection of XSS attacks.

CHAPTER 7: CONCLUSIONS

7.1 SUMMARY

The rapid growth in technology has resulted in the use of computers and the Web for managing large amounts of data, increasing the need to address confidentiality, integrity, and availability of computer-based applications and information. Cyber-attacks are placed among top five risks the world is likely to face over the next decade [7, 8]. XSS, which is one of the most common types of injection attacks exploiting input validation vulnerability [13], poses one of the main challenges to ensure cyber-security. XSS has consistently stayed in the top 3 risks for Web-applications since 2002 [2-6], emphasizing how risky and difficult such attacks are to detect and prevent.

To enhance cyber-security, the Intrusion Detection Approach (IDA) proactively prevents XSS-attacks by documenting knowledge and defining software processes that support incorporation of security concerns into Web application by customizing documented knowledge. The customization of security knowledge to any Web application promotes reuse of knowledge and allows software engineers to focus on business use-cases and functionality of the application. The approach combines and uses cyber-enhanced technologies such as ontology, provenance, formalizations, and security-related domain knowledge to specify application-specific patterns that drive monitoring of the Web application particularly, preventing XSS-attacks. Whenever there are new attacks or new ways of launching attacks, the mere update of new attack information into the knowledge-base helps monitor, identify, and prevent such new attacks without any changes to the Web application itself. The approach supports both prevention and detection of XSS-attacks by using latest security information. The IDA provides similar benefits as MDD approaches coupled with benefits of maintaining reusable security domain knowledge.

The case study showed the effectiveness of the XSSMon tool in identifying, detecting, and flagging XSS attacks. The tool fared better than the considered Web application, i.e., XSSMon had a higher success rate in identifying XSS-related attacks than the Web application's inbuilt security. The overall success rate of the original Web application under study was 4.79 percent

and 28.01 percent for the latest version in comparison to the success rate of 88.36 percent of XSSMon version 1 and 100 percent in detecting XSS attacks by XSSMon version 2. The XSSMon tool relies on the completeness of knowledge that comes from various sources, such as CAPEC, CWE, and OWASP cheat sheets. Keeping the knowledge base current and having accurate representation of the application assets is essential for the effectiveness of the tool.

One of the important findings of the case study is that even if input is sanitized or filtered by the Web application and a request is constructed, it is essential to process at the Web server before constructing the response or saving data as it is possible that the sanitized or filtered request is intercepted and malicious input is injected in the network. Similarly, it is essential to process the response received from server before rendering the response. Therefore, it is imperative that the IO is processed at both the client-side and server-side of the Web application to enhance security further. The XSSMon tool performs processing both while sending and receiving, which enhances the security of the Web applications. The results of the case study can be extended to other Web applications that accept similar equivalence classes of IO.

7.2 SIGNIFICANCE

The dissertation addresses two important high-level problems: a) lack of properly documented security domain knowledge that software engineers can use to develop and maintain Web-applications, and b) lack of well-defined software engineering processes that support prevention of vulnerabilities and threats. To this end, the work defines structures needed for storing and deriving XSS-related knowledge and its provenance systematically and formally. This knowledge drives the IDA for preventing successful XSS-attacks by helping build a layer between Web-applications and its users to monitor and prevent XSS attacks. The work provides a continuous focus on identification of vulnerabilities and threats allowing software engineers to focus of the business use-cases of the Web applications. The work uses the vast source of knowledge about attack patterns and weaknesses continuously and regularly maintained by NVD. The work allows reuse of security knowledge by customizing it to various Web applications

independent of the platform and programming language used. The work has the potential to support security testing of the Web application as well. The IDA separates security concerns from the business logic of the Web applications thereby, allowing customization of security to the latest and most up-to-date knowledge without having to change/maintain the Web application frequently to keep it abreast with ever growing security needs.

The use of cyber-enhanced technologies and techniques, such as ontologies and provenance provides a way to capture information from various sources and infer knowledge from them along with providing a way to customize and use security knowledge based on application-specific needs. The IDA provides similar benefits as model-driven development (MDD) approaches along with the benefits of maintaining reusable security domain knowledge.

The contributions of the dissertation work include the following:

1. A way to separate security concerns and maintaining higher level of security of Web applications independent of platforms and programming languages used against XSS attacks.
2. An IDA ontology, which is published through http://ontology.cybershare.utep.edu/security/ida-o, that captures XSS-attack specific security domain knowledge along with provenance information on vulnerability and threats.
    a. Automated loading, parsing, and populating of relevant information from user-specified version of CAPEC and CWE files.
    b. Automated application-specific customization of general XSS knowledge.
    c. Formalization of asset-specific whitelist.
3. A template to systematically capture all necessary application-specific information to provide customized and enhanced security for the Web application.
4. Processes and algorithms for deriving XSS-related application-specific customized patterns for monitoring and processing IO.

5. Reusable formalized generic XSS knowledge and customized application-specific formalized XSS knowledge.
6. Processes for customization based on provenance and prioritization of customized attack patterns based on sensitivity of asset, and severity and likelihood of attack patterns.

7.3 FUTURE WORK

In the IDA, the derivation of blacklist patterns based on the information from threats, weaknesses, vulnerabilities, and other online example sources is performed manually. This is not only tedious, but may also be error-prone and requires the knowledge or expertise in creation of patterns. The automation of the derivation of blacklist patterns using the information would be a candidate for future work. However, to automate this process AI technique(s) will likely need to be introduced.

Although CWE is comprehensive by aggregating weakness categories from many different vulnerability taxonomies, software technologies and products, and categorization perspectives, using its highly tangled web of weakness information and categories is a daunting task. Several manual and semi-automated approaches are described that can be used to extract useful information from sources such as CWE and CAPEC. The IDA uses partial information from CWE and CAPEC. It may be interesting and worthwhile to explore how other readily available information can be used to enhance security of Web applications.

The IDA predominantly considers and uses sources such as CWE, CAPEC, and OWASP; however, it would be intriguing to explore other reliable sources for security information and merge all information to create a robust security knowledge base. Another approach is to include sources that may not be as reliable, e.g., published information on the Web. With the use of provenance, it will be possible for the user to determine the level of trust to be placed on the source.

REFERENCES

[1] G. B. W. Art Conklin, "e-Government and Cyber Security: The Role of Cyber Security Exercises," in *Proceedings of the 39th Hawaii International Conference on System Sciences*, 2006, p. 8.

[2] IBM, "IBM X-Force Threat Intelligence Quarterly 1Q 2014," February 2014 2014.

[3] IBM, "IBM X-Force Threat Intelligence Quarterly 2Q 2014," 2014.

[4] OWASP, "OWASP Top 10 - 2013 The Ten Most Critical Web Application Security Risks," 2013.

[5] OWASP, "OWASP Top 10 - 2010 rc1 The Ten Most Critical Web Application Security Risks," 2010.

[6] R. A. M. Steve Christey. (2007, 07/10/2014). *Vulnerability Type Distributions in CVE*. Available: http://cwe.mitre.org/documents/vuln-trends/index.html

[7] "Global Risks 2012," World Economic Forum2012.

[8] "Global Risks 2014," World Economic Forum2014.

[9] D. M. P. Jayamsakthi Shanmugam, "Cross Site Scripting-Latest developments and solutions: A survey," *Int. J. Open Problems Compt. Math,* vol. 1, 2008.

[10] D. Watson, "Web application attacks," *Network Security,* vol. 2007, pp. 10-14, 2007.

[11] B. Warneck, "Defeating SQL Injection IDS Evasion," *SANS Institute Information Security Reading Room,* 2007.

[12] W. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA*, 2006, pp. 13-15.

[13] L. W. Yonghee Shin, "Toward A Taxonomy of Techniques to Detect Cross-site Scripting and SQL Injection Vulnerabilities," North Carolina State University TR-2008-4, 2008.

[14] J. Knutson. (2014, 07/10/2014). *Cross-Site Scripting (XSS)*. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[15] D. Wichers. (2013, 07/12/2014). *Types of Cross-Site Scripting*. Available: https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting

[16] A. Klein. (2005, 07/01/2014). *DOM Based Cross Site Scripting or XSS of the Third Kind*. Available: http://www.webappsec.org/projects/articles/071105.shtml

[17] A. Wiegenstein, M. Schumacher, X. Jia, and F. Weidemann, "The Cross Site Scripting Threat," *Virtual Forge Whitepaper,* 2007.

[18] P. H. Meland, D. G. Spampinato, E. Hagen, E. T. Baadshaug, K.-M. Krister, and K. S. Velle, "SeaMonster: Providing tool support for security modeling," *Norsk informasjonssikkerhetskonferanse, NISK,* 2008.

[19] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 2008, pp. 171-180.

[20] H. B. K. T. Lwin Khin Shar, "Automated Removal of Cross Site Scripting Vulnerabilities in Web Applications," *Information and Software Technology,* p. 12, 2012.

[21] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 47-53.

[22] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging user interactions for in-depth testing of web applications," in *Recent Advances in Intrusion Detection*, 2008, pp. 191-210.

[23] P. Bathia, B. R. Beerelli, and M.-A. Laverdière, "Assisting Programmers Resolving Vulnerabilities in Java Web Applications," in *Advanced Computing*, ed: Springer, 2011, pp. 268-279.

[24] A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 585-594.

[25] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Usenix Security*, 2005, pp. 18-18.

[26] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 432-441.

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

[28] 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*, 2004, pp. 40-52.

[29] 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*, 2003, pp. 148-159.

[30] R. Priyadarshini, D. Jagadiswaree, A. Fareedha, and M. Janarthanan, "A cross platform intrusion detection system using inter server communication technique," in *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, 2011, pp. 1259-1264.

[31] T. S. Barhoom and S. N. Kohail, "A New Server-Side Solution for Detecting Cross Site Scripting Attack," *Int. J. Comput. Inf. Syst,* vol. 3, pp. 19-23, 2011.

[32] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, 2008, pp. 335-344.

[33] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 33-39.

[34] S. Sundareswaran and A. C. Squicciarini, "XSS-Dec: a hybrid solution to mitigate cross-site scripting attacks," in *Data and Applications Security and Privacy XXVI*, ed: Springer, 2012, pp. 223-238.

[35] H. Shahriar and M. Zulkernine, "S2XS2: A Server Side Approach to Automatically Detect XSS Attacks," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 7-14.

[36] R. Sekar, "An Efficient Black-box Technique for Defeating Web Application Attacks," in *NDSS*, 2009.

[37] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "AProSec: An aspect for programming secure web applications," in *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, 2007, pp. 1026-1033.

[38] H. Shahriar and M. Zulkernine, "Injecting comments to detect JavaScript code injection attacks," in *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, 2011, pp. 104-109.

[39] B. Benjamin, F. Oladeji, C. Okolie, H. Alakiri, and O. Olisa, "S2MXS2: server side approach to mitigating XSS attacks using regular expression," *Journal of Emerging Trends in Engineering and Applied Sciences,* vol. 4, pp. 875-882, 2013.

[40] S. F. Hidhaya and A. Geetha, "Intrusion Protection against SQL Injection and Cross Site Scripting Attacks Using a Reverse Proxy," in *Recent Trends in Computer Networks and Distributed Systems Security*, ed: Springer, 2012, pp. 252-263.

[41] T. S. Mule, A. S. Mahajan, S. Kamble, and O. Khatavkar, "Intrusion Protection against SQL Injection And Cross Site Scripting Attacks Using a Reverse Proxy," *International Journal of Computer Science and Information Technologies,* 2014.

[42] E. Chin and D. Wagner, "Efficient character-level taint tracking for Java," in *Proceedings of the 2009 ACM workshop on Secure web services*, 2009, pp. 3-12.

[43] Y. Song, A. D. Keromytis, and S. Stolfo, "Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic," in *Network and Distributed System Security Symposium 2009: February 8-11, 2009, San Diego, California: Proceedings*, 2009, pp. 121-135.

[44] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest, "Learning DFA representations of HTTP for protecting web applications," *Computer Networks,* vol. 51, pp. 1239-1255, 2007.

[45] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 251-261.

[46] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Computer Networks,* vol. 48, pp. 717-738, 2005.

[47] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *NDSS*, 2006.

[48] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, "Preventing input validation vulnerabilities in web applications through automated type analysis," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, 2012, pp. 233-243.

[49] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ed: Springer, 2008, pp. 23-43.

[50] J. Garcia-Alfaro and G. Navarro-Arribas, "Prevention of cross-site scripting attacks on current web applications," in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, ed: Springer, 2007, pp. 1770-1784.

[51] M. Van Gundy and H. Chen, "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks," in *NDSS*, 2009.

[52] R. Mui and P. Frankl, "Preventing web application injections with complementary character coding," in *Computer Security–ESORICS 2011*, ed: Springer, 2011, pp. 80-99.

[53] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense," in *NDSS*, 2009.

[54] Y. Cao, V. Yegneswaran, P. Porras, and Y. Chen, "Poster: a path-cutting approach to blocking XSS worms in social web networks," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 745-748.

[55] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 921-930.

[56] F. Kerschbaum, "Simple cross-site attack prevention," in *Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. Third International Conference on*, 2007, pp. 464-472.

[57] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 587-600.

[58] W. K. Robertson and G. Vigna, "Static Enforcement of Web Application Integrity Through Strong Typing," in *USENIX Security Symposium*, 2009, pp. 283-298.

[59] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, *Automatically hardening web applications using precise tainting*: Springer, 2005.

[60] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in *Computer Security Applications Conference, 21st Annual*, 2005, pp. 9 pp.-311.

[61] J. Shanmugam and M. Ponnavaikko, "Behavior-based anomaly detection on the server side to reduce the effectiveness of Cross Site Scripting vulnerabilities," in *Semantics, Knowledge and Grid, Third International Conference on*, 2007, pp. 350-353.

[62] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 601-610.

[63] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with BEK," in *Proceedings of the 20th USENIX conference on Security*, 2011, pp. 1-1.

[64] M. T. Gebre, K.-S. Lhee, and M. Hong, "A robust defense against content-sniffing xss attacks," in *Digital Content, Multimedia Technology and its Applications (IDC), 2010 6th International Conference on*, 2010, pp. 315-320.

[65] A. Barth, J. Caballero, and D. Song, "Secure content sniffing for web browsers, or how to stop papers from reviewing themselves," in *Security and Privacy, 2009 30th IEEE Symposium on*, 2009, pp. 360-371.

[66] J. Shanmugam and M. Ponnavaikko, "Xss application worms: New internet infestation and optimized protective measures," in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, 2007, pp. 1164-1169.

[67] M. Johns, "SessionSafe: Implementing XSS immune session handling," in *Computer Security–ESORICS 2006*, ed: Springer, 2006, pp. 444-460.

[68] J. Shanmugam and M. Ponnavaikko, "Risk mitigation for cross site scripting attacks using signature based model on the server side," in *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on*, 2007, pp. 398-405.

[69] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 31-44.

[70] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing Confidentiality and Integrity in Web Applications," in *USENIX Security*, 2007.

[71] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications," in *NDSS*, 2010.

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

[73] N. Li, T. Xie, M. Jin, and C. Liu, "Perturbation-based user-input-validation testing of web applications," *Journal of Systems and Software,* vol. 83, pp. 2263-2274, 2010.

[74] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," in *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, 2004, pp. 145-151.

[75] Q. Zhenyu, X. Jing, L. Baoguo, and T. Fang, "MBDS: Model-based detection system for Cross Site Scripting," in *Wireless, Mobile and Sensor Networks, 2007.(CCWMSN07). IET Conference on*, 2007, pp. 849-852.

[76] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *Computer Security–ESORICS 2011*, ed: Springer, 2011, pp. 150-171.

[77] Q. Zhang, H. Chen, and J. Sun, "An execution-flow based method for detecting cross-site scripting attacks," in *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*, 2010, pp. 160-165.

[78] D. Arulsuju, "Hunting malicious attacks in social networks," in *Advanced Computing (ICoAC), 2011 Third International Conference on*, 2011, pp. 13-17.

[79] H. Bojinov, E. Bursztein, and D. Boneh, "XCS: cross channel scripting and its impact on web applications," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 420-431.

[80] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 330-337.

[81] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, "Client-side cross-site scripting protection," *computers & security,* vol. 28, pp. 592-604, 2009.

[82] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *NDSS*, 2007.

[83] V. Venkatakrishnan, P. Bisht, M. Ter Louw, M. Zhou, K. Gondi, and K. T. Ganesh, "WebAppArmor: a framework for robust prevention of attacks on web applications," in *Information Systems Security*, ed: Springer, 2011, pp. 3-26.

[84] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," in *Proceedings of the 23rd USENIX security symposium*, 2014.

[85] S. Tang, C. Grier, O. Aciicmez, and S. T. King, "Alhambra: a system for creating, enforcing, and testing browser security policies," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 941-950.

[86] 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*, 2009, pp. 331-346.

[87] G. Iha, "An implementation of the binding mechanism in the web browser for preventing XSS attacks: introducing the bind-value headers," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, 2009, pp. 966-971.

[88] V. S. Chandra and S. Selvakumar, "BIXSAN: browser independent XSS sanitizer for prevention of XSS attacks," *ACM SIGSOFT Software Engineering Notes,* vol. 36, pp. 1-7, 2011.

[89] S. Tiwari, R. Bansal, and D. Bansal, "Optimized client side solution for cross site scripting," in *Networks, 2008. ICON 2008. 16th IEEE International Conference on*, 2008, pp. 1-4.

[90] S. Shalini and S. Usha, "Prevention Of Cross-Site Scripting Attacks (XSS) On Web Applications In The Client Side," *International Journal of Computer Science Issues (IJCSI),* vol. 8, 2011.

[91] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen, "SessionShield: Lightweight protection against session hijacking," in *Engineering Secure Software and Systems*, ed: Springer, 2011, pp. 87-100.

[92] B. Livshits and Ú. Erlingsson, "Using web application construction frameworks to protect against code injection attacks," in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007, pp. 95-104.

[93] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web (TWEB),* vol. 1, p. 11, 2007.

[94] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *ACM SIGPLAN Notices*, 2007, pp. 237-249.

[95] E. Galán, A. Alcaide, A. Orfila, and J. Blasco, "A multi-agent scanner to detect stored-XSS vulnerabilities," in *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, 2010, pp. 1-6.

[96] Y. Wang, Z. Li, and T. Guo, "Program slicing stored xss bugs in web application," in *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*, 2011, pp. 191-194.

[97] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 199-209.

[98] A. Avancini and M. Ceccato, "Towards security testing with taint analysis and genetic algorithms," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, 2010, pp. 65-71.

[99] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, 2011, pp. 85-94.

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

[101] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier, "XSS vulnerability detection using model inference assisted evolutionary fuzzing," *Software Testing, Verification and Validation (ICST 2012),* pp. 815-817, 2012.

[102] H. Al-Amro and E. El-Qawasmeh, "Discovering security vulnerabilities and leaks in ASP. NET websites," in *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on*, 2012, pp. 329-333.

[103] G. Agosta, A. Barenghi, A. Parata, and G. Pelosi, "Automated security analysis of dynamic web applications through symbolic code execution," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, 2012, pp. 189-194.

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

[105] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, 2006, pp. 27-36.

[106] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *Tools and Algorithms for the Construction and Analysis of Systems*, ed: Springer, 2010, pp. 154-157.

[107] J. Choi, H. Kim, C. Choi, and P. Kim, "Efficient malicious code detection using n-gram analysis and SVM," in *Network-Based Information Systems (NBiS), 2011 14th International Conference on*, 2011, pp. 618-621.

[108] R. Komiya, I. Paik, and M. Hisada, "Classification of malicious web code by machine learning," in *Awareness Science and Technology (iCAST), 2011 3rd International Conference on*, 2011, pp. 406-411.

[109] P. M. Pérez, J. Filipiak, and J. M. Sierra, "LAPSE+ Static Analysis Security Software: Vulnerabilities Detection in Java EE Applications," in *Future Information Technology*, ed: Springer, 2011, pp. 148-156.

[110] S. Van Acker, N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens, "FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 12-13.

[111] A. R. F. G. A. Di Lucca, M. Mastroianni, P. Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications," presented at the Telecommunications Energy Conference, 2004.

[112] F. Yu, T. Bultan, and B. Hardekopf, "String abstractions for string verification," in *Model Checking Software*, ed: Springer, 2011, pp. 20-37.

[113] M. E. Ruse and S. Basu, "Detecting Cross-Site Scripting Vulnerability Using Concolic Testing," in *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, 2013, pp. 633-638.

[114] J.-M. Chen and C.-L. Wu, "An automated vulnerability scanner for injection attack based on injection point," in *Computer Symposium (ICS), 2010 International*, 2010, pp. 113-118.

[115] L. Zhang, Q. Gu, S. Peng, X. Chen, H. Zhao, and D. Chen, "D-WAV: A Web Application Vulnerabilities Detection Tool Using Characteristics of Web Forms," in *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, 2010, pp. 501-507.

[116] Z. Tang, H. Zhu, Z. Cao, and S. Zhao, "L-WMxD: lexical based Webmail XSS discoverer," in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, 2011, pp. 976-981.

[117] 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*, 2008, pp. 31-43.

[118] P. Xiong, B. Stepien, and L. Peyton, "Model-based penetration test framework for web applications using TTCN-3," in *E-Technologies: Innovation in an Open World*, ed: Springer, 2009, pp. 141-154.

[119] K. Li, "Towards Security Vulnerability Detection by Source Code Model Checking," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 381-387.

[120] A. Mohosina and M. Zulkernine, "DESERVE: a framework for detecting program security vulnerability exploitations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, 2012, pp. 98-107.

[121] A. E. Nunan, E. Souto, E. M. dos Santos, and E. Feitosa, "Automatic classification of cross-site scripting in web pages using document-based and URL-based features," in *Computers and Communications (ISCC), 2012 IEEE Symposium on*, 2012, pp. 000702-000707.

[122] C. M. Frenz and J. P. Yoon, "XSSmon: a Perl based IDS for the detection of potential XSS attacks," in *Systems, Applications and Technology Conference (LISAT), 2012 IEEE Long Island*, 2012, pp. 1-4.

[123] L. Coppolino, S. D'Antonio, I. A. Elia, and L. Romano, "From intrusion detection to intrusion detection and diagnosis: An ontology-based approach," in *Software Technologies for Embedded and Ubiquitous Systems*, ed: Springer, 2009, pp. 192-202.

[124] A. Avancini and M. Ceccato, "Grammar based oracle for security testing of web applications," in *Automation of Software Test (AST), 2012 7th International Workshop on*, 2012, pp. 15-21.

[125] H. Shahriar and M. Zulkernine, "Trustworthiness testing of phishing websites: A behavior model-based approach," *Future Generation Computer Systems,* vol. 28, pp. 1258-1271, 2012.

[126] T. Matsuda, D. Koizumi, and M. Sonoda, "Cross site scripting attacks detection algorithm based on the appearance position of characters," in *Communications, Computers and Applications (MIC-CCA), 2012 Mosharaka International Conference on*, 2012, pp. 65-70.

[127] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing Web vulnerability scanning tools for SQL injection and XSS attacks," in *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, 2007, pp. 365-372.

[128] K. Petkov, "Overcoming programming flaws: indexing of common software vulnerabilities," in *Proceedings of the 2nd annual conference on Information security curriculum development*, 2005, pp. 127-134.

[129] M. Johns, C. Beyerlein, R. Giesecke, and J. Posegga, "Secure code generation for web applications," in *Engineering Secure Software and Systems*, ed: Springer, 2010, pp. 96-113.

[130] N. Juillerat, "Enforcing code security in database web applications using libraries and object models," in *Proceedings of the 2007 Symposium on Library-Centric Software Design*, 2007, pp. 31-41.

[131] K. Sivakumar and K. Garg, "Constructing a "Common Cross Site Scripting Vulnerabilities Enumeration (CXE)" Using CWE and CVE," in *Information Systems Security*, ed: Springer, 2007, pp. 277-291.

[132] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, 2009, pp. 47-60.

[133] R. Putthacharoen and P. Bunyatnoparat, "Protecting cookies from cross site script attacks using dynamic cookies rewriting technique," in *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, 2011, pp. 1090-1094.

[134] P. Sharma, R. Johari, and S. Sarma, "Integrated approach to prevent SQL injection attack and reflected cross site scripting attack," *International Journal of System Assurance Engineering and Management,* vol. 3, pp. 343-351, 2012.

[135] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 601-614.

[136] S. Nanda, L.-C. Lam, and T.-c. Chiueh, "Dynamic multi-process information flow tracking for web application security," in *Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*, 2007, p. 19.

[137] Z. Li and X. Wang, "FIRM: Capability-based inline mediation of Flash behaviors," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 181-190.

[138] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All your clouds are belong to us: security analysis of cloud management interfaces," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, 2011, pp. 3-14.

[139] R. B. Brinhosa, C. M. Westphall, and C. B. Westphall, "Proposal and development of the web services input validation model," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 643-646.

[140] Y. Sun and D. He, "Model Checking for the Defense against Cross-Site Scripting Attacks," in *Computer Science & Service System (CSSS), 2012 International Conference on*, 2012, pp. 2161-2164.

[141] R. Grabowski, M. Hofmann, and K. Li, "Type-based enforcement of secure programming guidelines—code injection prevention at SAP," in *Formal Aspects of Security and Trust*, ed: Springer, 2012, pp. 182-197.

[142] J. Shanmugam and M. Ponnavaikko, "A solution to block cross site scripting vulnerabilities based on service oriented architecture," in *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, 2007.

[143] L. K. Shar and H. B. K. Tan, "Auditing the XSS defence features implemented in web application programs," *iET Software,* vol. 6, pp. 377-390, 2012.

[144] L. K. Shar and H. B. K. Tan, "Auditing the defense against cross site scripting in web applications," in *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, 2010, pp. 1-7.

[145] J. Stuckman and J. Purtilo, "A testbed for the evaluation of web intrusion prevention systems," in *Security Measurements and Metrics (Metrisec), 2011 Third International Workshop on*, 2011, pp. 66-75.

[146] S.-J. Wang, Y.-H. Chang, W.-Y. Chiang, and W.-S. Juang, "Investigations in Cross-site Script on Web-systems Gathering Digital Evidence against Cyber-Intrusions," in *Future Generation Communication and Networking (FGCN 2007)*, 2007, pp. 125-129.

[147] M. R. Faghani and H. Saidi, "Social networks' XSS worms," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, 2009, pp. 1137-1141.

[148] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, 2012, pp. 310-313.

[149] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 1293-1296.

[150] W. Z. Junhua Wang, Fengling He, Ying Wang, "A New Formal Description of Ontology Definition and Ontology Algebra," presented at the Second International Symposium on Knowledge Acquisition and Modeling, 2009.

[151] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing knowledge about information systems," *ACM Transactions on Information Systems (TOIS),* vol. 8, pp. 325-362, 1990.

[152] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, pp. 11-33, 2004.

[153] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Computing Surveys (CSUR),* vol. 26, pp. 211-254, 1994.

[154] J. Undercoffer, A. Joshi, and J. Pinkston, "Modeling computer attacks: An ontology for intrusion detection," in *Recent Advances in Intrusion Detection*, 2003, pp. 113-135.

[155] L. Viljanen, "Towards an ontology of trust," in *Trust, Privacy, and Security in Digital Business*, ed: Springer, 2005, pp. 175-184.

[156] D. Geneiatakis and C. Lambrinoudakis, "An ontology description for SIP security flaws," *Computer Communications,* vol. 30, pp. 1367-1374, 2007.

[157] G. Denker, L. Kagal, T. Finin, M. Paolucci, and K. Sycara, "Security for daml web services: Annotation and matchmaking," in *The Semantic Web-ISWC 2003*, ed: Springer, 2003, pp. 335-350.

[158] G. Denker, S. Nguyen, and A. Ton, "OWL-S semantics of security web services: A case study," in *The Semantic Web: Research and Applications*, ed: Springer, 2004, pp. 240-253.

[159] G. Denker, L. Kagal, and T. Finin, "Security in the Semantic Web using OWL," *Information Security Technical Report,* vol. 10, pp. 51-58, 2005.

[160] A. Kim, J. Luo, and M. Kang, *Security ontology for annotating resources*: Springer, 2005.

[161] A. Vorobiev and J. Han, "Security attack ontology for web services," in *Semantics, Knowledge and Grid, 2006. SKG'06. Second International Conference on*, 2006, pp. 42-42.

[162] A. Ekelhart, S. Fenz, M. Klemen, and E. Weippl, "Security ontologies: Improving quantitative risk analysis," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, 2007, pp. 156a-156a.

[163] A. A. Assali, D. Lenne, and B. Debray, "Ontology development for industrial risk analysis," in *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, 2008, pp. 1-5.

[164] G. Dobson and P. Sawyer, "Revisiting ontology-based requirements engineering in the age of the semantic web," in *Proceedings of the International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs*, 2006.

[165] B. Tsoumas and D. Gritzalis, "Towards an ontology-based security management," in *Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference on*, 2006, pp. 985-992.

[166] M. Karyda, T. Balopoulos, S. Dritsas, L. Gymnopoulos, S. Kokolakis, C. Lambrinoudakis, and S. Gritzalis, "An ontology for secure e-government applications," in *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, 2006, p. 5 pp.

[167] D. G. Firesmith, "A taxonomy of safety-related requirements," in *International Workshop on High Assurance Systems (RHAS'05)*, 2005.

[168] H. Mouratidis, P. Giorgini, and G. Manson, "An ontology for modelling security: The tropos approach," in *Knowledge-Based Intelligent Information and Engineering Systems*, 2003, pp. 1387-1394.

[169] F. Massacci, J. Mylopoulos, F. Paci, T. T. Tun, and Y. Yu, "An extended ontology for security requirements," in *Advanced Information Systems Engineering Workshops*, 2011, pp. 622-636.

[170] A. Herzog, N. Shahmehri, and C. Duma, "An ontology of information security," *International Journal of Information Security and Privacy (IJISP),* vol. 1, pp. 1-23, 2007.

[171] S. Fenz and A. Ekelhart, "Formalizing information security knowledge," in *Proceedings of the 4th international Symposium on information, Computer, and Communications Security*, 2009, pp. 183-194.

[172] M. Laclavik, M. Seleng, E. Gatial, Z. Balogh, and L. Hluchy, "Ontology based text annotation-OnTeA," *Frontiers in Artificial Intelligence and Applications,* vol. 154, p. 311, 2007.

[173] R. A. Gandhi, H. Siy, and Y. Wu, "Studying software vulnerabilities," *CrossTalk, September/October,* 2010.

[174] R. Gandhi, H. Siy, and Y. Wu, "Lightweight formal models of software weaknesses," in *Formal Methods in Software Engineering (FormaliSE), 2013 1st FME Workshop on*, 2013, pp. 50-56.

[175] B. Schneier, "Attack trees," ed, 2006.

[176] S. A. Camtepe and B. Yener, "Modeling and detection of complex attacks," in *Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. Third International Conference on*, 2007, pp. 234-243.

[177] R. Dewri, N. Poolsappasit, I. Ray, and D. Whitley, "Optimal security hardening using multi-objective optimization on attack tree models of networks," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 204-213.

[178] A. Morais, E. Martins, A. Cavalli, and W. Jimenez, "Security protocol testing using attack trees," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, 2009, pp. 690-697.

[179] Z. Ning, C. Xin-yuan, and X. Si-yuan, "Design and application of penetration attack tree model oriented to attack resistance test," in *Computer Science and Software Engineering, 2008 International Conference on*, 2008, pp. 622-626.

[180] J. H. A. Nancy R. Mead, Mark Ardis, Thomas B. Hilburn, Andrew J. Kornecki, Richard Linger, James McDonald, "Software Assurance Curriculum Project Volume I: Master of Software Assurance Reference Curriculum," CMU/SEI-2010-TR-005 2010.

[181] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, "Prov-o: The prov ontology," *W3C Recommendation, 30th April,* 2013.

[182] P. Hayati, N. Jafari, S. M. Rezaei, S. Sarenche, and V. Potdar, "Modeling input validation in uml," in *19th Australian Conference on Software Engineering (aswec 2008)*, 2008, pp. 663-672.

[183] S. Easterbrook and J. Aranda, "Case Studies for Software Engineers," *ICSE'06,* 2006.

[184] EdgeScan, "Edgescan 2015 Vulnerability Statistics Report," 2015.

[185] EdgeScan, "Edgescan 2014 Vulnerability Statistics Report," 2014.

[186] R. K. Yin, *Analytic Generalization. Encyclopedia of Case Study Research. SAGE Publications, Inc*. Thousand Oaks, CA: SAGE Publications, Inc.

[187] A. Smaling, "Inductive, analogical, and communicative generalization," *International Journal of Qualitative Methods,* vol. 2, pp. 52-67, 2003.

APPENDIX A

DETAILED SURVEY OF DETECTION AND PREVENTION OF XSS VULNERABILITIES AND ATTACKS

The results of literature review on the current state of art for detection and prevention of XSS-related vulnerabilities and attacks is classified into different categories as depicted Fig. A1. The summary of the classification in presented as tables in the document. The details of the literature review are presented in this appendix.

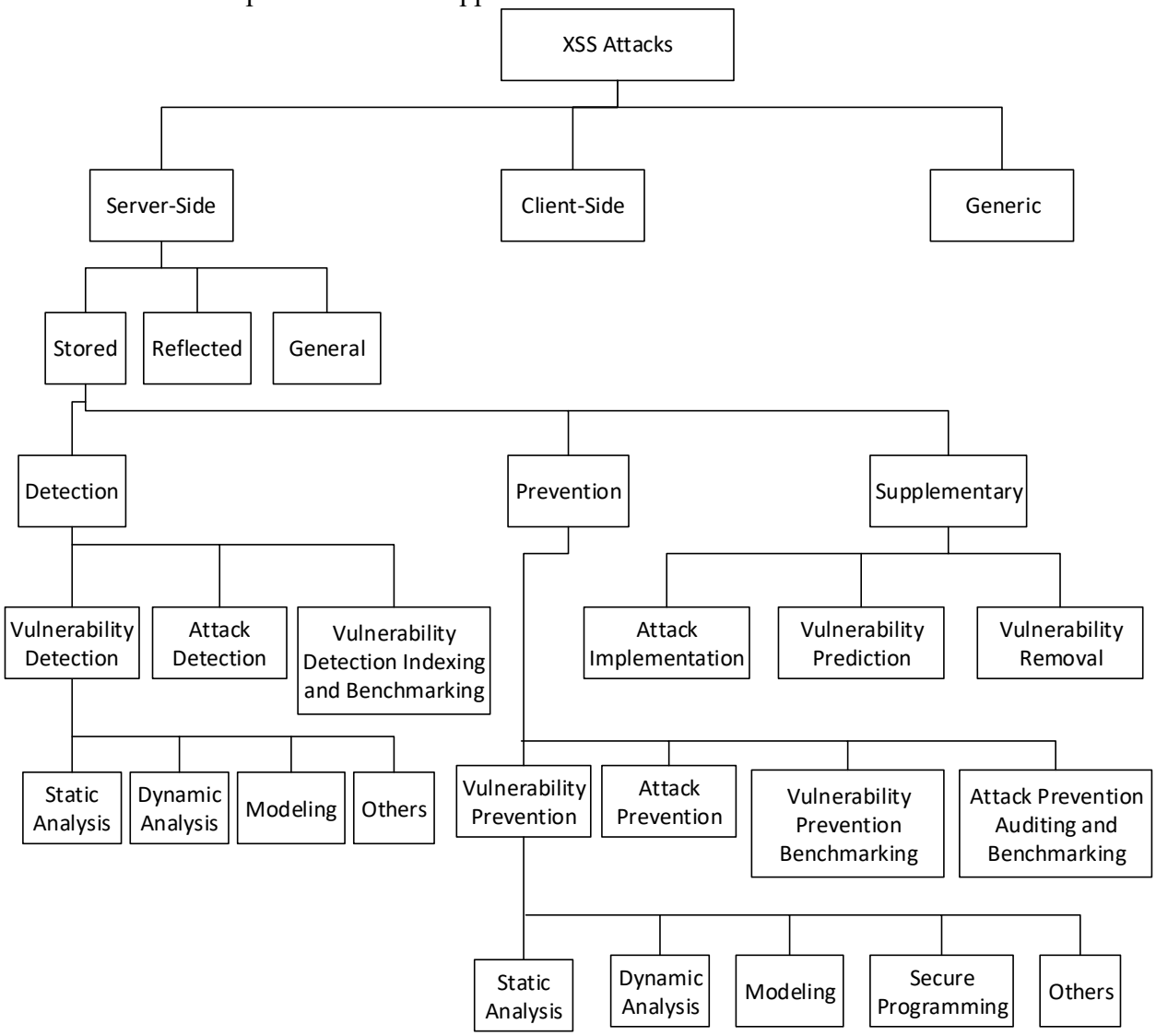

Figure A.1: Classification Categories for approaches to detect and prevent XSS-attacks.

A survey, which included review of over 130 research papers, was conducted, and the results were classified into the categories identified in Fig. A.1. The details of the literature review are summarized using the categories from Fig. A.1.

Server-Side

Stored XSS: Both Detection and Prevention: Dynamic Analysis

Barhoom and Kohail [31] described a server-side solution to detect and block stored XSS attacks using XML and XSD to enforce input type integrity by preventing untrusted user input from altering the structure of trusted code.

Reflected XSS: Prevention Methods/Approaches: Dynamic Analysis

Kerschbaum [56] presented a lightweight gateway solution at the server to prevent reflected XSS attacks by using Web page classification, referrer string, and cookies techniques. It ensures that input to a Web-site originated in the users' browser and has not been forged by an attacker by following a link.

Both Stored and Reflected: Detection Methods/Approaches: Static Analysis

Wassermann and Su [19] present a server-side static analysis approach for finding XSS vulnerabilities that address weak or absent input validation.

Shar and Tan [20] present an approach based on static analysis and pattern matching techniques to identify potential XSS vulnerabilities in source code and to secure them with appropriate escape mechanisms. They also developed a tool, saferXSS.

Both Stored and Reflected: Detection Methods/Approaches: Dynamic Analysis

Johns et al. [32] presented XSSDS, a server-side XSS passive detection system that uses two novel detection approaches based on generic observation of XSS attacks and Web applications.

McAllister et al. [22] presented an automated testing tool that can find reflected and stored XSS vulnerabilities in Web applications. The core of their system is a black-box vulnerability scanner.

Both Stored and Reflected: Prevention Methods/Approaches: Dynamic Analysis

Bisht and Venkatakrishnan [49] described XSS-Guard, a new framework that is designed as a prevention mechanism against XSS attacks on the server side.

Garcia-Alfaro and Navarro-Arribas [50] presented an approach to prevent stored and reflected XSS attacks by enforcing security policies defined at the server-side through the use of XACML and X.509 certificates.

Van Gundy and Chen [51] presented Noncespaces, a mechanism that allows server to identify untrusted content and reliably convey this information to the client to allow it to enforce security policy on untrusted content. Similar to BEEP, Noncespaces cannot handle client-side XSS.

Both Stored and Reflected: Prevention Methods/Approaches: Others

Stamm et al. [55] presented content restriction and content restriction scheme called Content Security Policy (CSP) for preventing XSS attacks. The content restriction rules are activated and enforced by supporting Web browsers when a policy provided for a site via HTTP.

Both Detection and Prevention: Dynamic analysis

Wurzinger et al. [33] described a server-side solution for detecting and preventing XSS attacks by introducing SWAP (Secure Web Application Proxy).

General: Detection Methods/Approaches: Static Analysis

Bathia et al. [23] presented an approach to detect and correct security vulnerabilities in Java Web applications using program slicing and transformation. They implemented a prototype as an Eclipse plugin, leveraging the WALA library to fix XSS vulnerabilities in an interactive manner.

Chaudhuri and Foster [24] implement Rubyx, a symbolic executor that analyzes Ruby-on-Rails Web applications for security vulnerabilities such as XSS, CSRF, insufficient authentication and access control, and leaks of secret information. Rubyx specifications are built on assertions, assumptions, and object invariants.

Livshits and Lam [25] propose a static analysis approach to detect vulnerabilities based on points-to analysis for analyzing byte-code of Java Web applications based on binary decision diagrams. They use a high-level declarative language Program Query Language (PQL) for specifying information flow and automating information flow analysis.

Minamide [26] presented a static string analyzer for PHP that detects XSS vulnerabilities using context-free grammar to approximate dynamic Web pages generated by a program.

General: Detection Methods/Approaches: Dynamic Analysis

Shahriar and Zulkernine [35] described a server-side XSS-attack detection technique based on the concept of boundary injection and policy generation. They presented another server-side approach in [38] to detect XSS attacks by distinguishing injected JavaScript code from legitimate JavaScript code via injecting comments for legitimate JavaScript code that encode them in terms of method definition and call signatures.

Sekar [36] presented an approach for detecting injection attacks using taint inference technique that operates by intercepting requests and reposes from Web application. Their approach has significant low overhead compared to other taint-based approaches.

Hermosillo et al. [37] presented a security aspect called AProSec through the use of aspect-oriented programming (AOP) for detecting SQL injection and XSS attacks.

Huang et al. [29] described WAVES, a testing framework for assessing the security of Web applications by injecting attack vectors.

General: Detection Methods/Approaches: Combination of static and dynamic analysis

Balzarotti et al. [27] described an approach that combines static and dynamic analysis techniques for analyzing the sanitization process and detecting vulnerabilities that stem from incorrect or incomplete sanitization. The approach is implemented as a tool, Saner.

General: Detection Methods/Approaches: Modeling

Song et al. [43] presented Spectrogram, a machine learning based statistical anomaly detection sensor for detecting Web-layer code-injection attacks. It uses a mixture of Markov chains based on n-gram transitions.

Ingham et al. [44] described a method for detecting anomalous Web requests via Deterministic Finite Automata (DFA). The method is used in combination with rules for reducing variability among requests and heuristics for filtering and grouping anomalies.

Kruegel and Vigna [45] and Kruegel et al. [46] presented one of the first works that uses different anomaly detection techniques to detect attacks against Web servers and Web-based applications. Web-server log files in common log format (CLF) is provided as input from which multiple statistical models to characterize different features of normal Web requests such as attribute length, character distribution, and attribute order are derived. A decision is made by taking into account all features of Web requests based of statistical models.

General: Detection Methods/Approaches: Others

Robertson et al. [47] presented an approach that addresses the limitations of anomaly-based intrusion detection system to detect Web attacks by using generalization and characterization techniques, which reduce the time required to pinpoint the location, make decisions about nature of anomalies and their criticality.

General: Prevention Methods/Approaches: Static Analysis

Scholte et al. [48] presented IPAAS, technique for preventing exploitation of XSS and SQL injection vulnerabilities based on automated data type detection of input parameters.

Robertson and Vigna [58] presented a Web application framework that leverages existing work on strong type systems to statistically enforce a separation between the structure and content of both Web documents and database queries generated by a Web application to prevent introduction of server-side XSS.

General: Prevention Methods/Approaches: Dynamic Analysis

Nguyen-Tuong et al. [59] presented a fully automated approach to securely hardening Web applications based on precisely tracking taintedness of data and checking specifically for malicious content only in parts of commands and output that came from untrustworthy sources.

Halder et al. [60] presented a framework for tagging, tracking, and preventing malicious user inputs at runtime. The described technique applies to Java class files and does not require source code.

Shanmugam and Ponnavaikko [61] presented a behavior-based anomaly detection approach on server-side to prevent XSS attacks.

Jim et al. [62] described a mechanism for preventing script injection called Browser-Enforced Embedded Policies (BEEP), that embeds a whitelist of known-safe scripts into each Web page and instructs the Web browser to filter suspicious scripts. BEEP cannot handle client-side XSS.

Gebre et al. [64] presented a server-side ingress filter that aims to provide defense against content-sniffing XSS attacks by protecting vulnerable browsers that treat non-HTML files as HTML files.

General: Prevention Methods/Approaches: Combination of static and dynamic analysis

Samuel et al. [57] presented a context-sensitive auto-sanitization engine based on type qualifiers that can be bolted onto existing Web template frameworks to prevent XSS attacks.

General: Prevention Methods/Approaches: Modeling

Shanmugam and Ponnavaikko presented a thread-based solution for efficient process utilization of Web server and to prevent XSS threats [66]. They described signature based misuse

detection approach in [68] as a security layer on top of the Web application so that existing Web application remains unchanged whenever a new threat is introduced that demands new security mechanisms.

Johns [67] identified Web application's characteristics responsible for enabling single attack methods which include: availability of session tokens via JavaScript, pre-knowledge of application's URL, and implicit trust relationship between Webpages of same origin. Three server-side techniques deferred loading, one-time URLs, and sub-domain switching were proposed to prevent session hijacking attacks. The author also implements SessionSafe that is a combination of the described approaches.

General: Prevention Methods/Approaches: Secure Programming

Chong et al. [70] presented Servlet Information Flow (SIF), a Web application framework based on security-typed language Jif. SIF is able to label user input, track information flow and enforce annotated security policies on them at both compile time and run-time. Parallel work by Chong et al. [69] presented Swift framework that automatically and securely partitions Jif source code into server-side and client-side code and enforces end-to-end information flow policies over code at both sides. SIF is applicable to Web applications whose functionality is mainly implemented as server-side code.

General: Both Detection and Prevention: Static Analysis

Huang et al. [28] presented WebSSARI (Web application Security by Static Analysis and Runtime Inspection), an extension system to language's type system that automatically inserts runtime guards in potentially insecure sections of code determined by static analysis.

General: Both Detection and Prevention: Dynamic Analysis

Chin and Wagner [42] described character-level taint-tracking approach for Java Web applications via instrumentation of Java library classes and Java Servlet to detect and block XSS attacks.

Benjamin et al. [39] designed and implemented S2MXS2 as a server-side XSS attack detection and prevention system using regular expression.

Hidhaya and Geetha [40] presented a server-side solution to mitigate XSS and SQL injection attacks using MD5 algorithm and grammar expressions manipulated in a reverse proxy.

Mule et al. [41] presented a policy-based proxy agent that classify user requests as scripted or query-based request and detects SQL injection and XSS-attacks. The user inputs are sanitized by the sanitizing application placed in the reverse proxy server.

Priyadarshini et al. [30] described an intrusion detection system that detects vulnerabilities including XSS and prevents attacks exploiting them using inter-server communication techniques.

Client-Side

Stored XSS: Prevention Methods/Approaches: Dynamic Analysis

Bojinov et al. [79] presented SiteFirewall which is a client-side defense against cross channel scripting (XCS), a form of stored XSS that affects embedded devices by injecting malicious scripts through ftp, p2p, or file logs.

Reflected XSS: Prevention Methods/Approaches: Combination of static and dynamic analysis

Vogt et al. [82] described a novel client-side solution that combines static and dynamic analysis techniques to prevent XSS attacks by tracking flow of sensitive information inside the Web browser.

Both Stored and Reflected: Detection Methods/Approaches: Dynamic Analysis

Saxena et al. [71] presented a dynamic analysis technique to systematically discover client-side XSS vulnerabilities and implemented a prototype tool FLAX that incorporated their described technique. Saxena et al. [72] described a system for exploring the execution space of JavaScript code using symbolic execution and built an automatic end-to-end tool Kudzu for detecting client-side XSS vulnerabilities. Kudzu automatically generates a test-suite that explores the execution space.

Both Stored and Reflected: Prevention Methods/Approaches: Dynamic Analysis

Kirda et al. [80] presented Noxes, a client-side solution for preventing XSS attacks. Noxes acts as a Web proxy and uses both manual and automatically generated rules mitigate XSS attacks. Kirda et al. [81] extended their previous work by dynamically enhancing protection mechanism and presented an approach against attacks based on n-ary alphabets.

Both Stored and Reflected: Both Detection and Prevention: Dynamic analysis

Zhang et al. [77] present an execution-flow based method at client-side for detecting XSS attacks by modeling the client-side behavior under normal execution as finite-state automate (FSA). The system is deployed in proxy mode that analyzes the execution flow of client-side JavaScript to prevent from potential malicious scripts that do not conform to FSA.

General: Detection Methods/Approaches: Static Analysis

Li et al. [73] presented an approach called perturbation-based interactive user-input-validation testing (PIUIVT) that improves the effectiveness of vulnerability scanners for user-input-validation (UIV) testing of Web applications by generating test inputs that detect XSS and other vulnerabilities.

Arulsuju [78] described XHunter, automata-based detection tool that employs forward and backward symbolic string analysis to detect XSS, SQL injection, and malicious file execution in Web applications.

General: Detection Methods/Approaches: Modeling

Zhenyu et al. [75] presented a client-side system that automatically detects XSS vulnerability by manipulating either primitive or advanced models.

Weinberger et al. [76] described a model of the Web browser and characterized the challenges of XSS sanitization provided by frameworks quantifying the gap between what frameworks provide and what applications require.

General: Prevention Methods/Approaches: Dynamic Analysis

Stock et al. [84] described a filter design for DOM-based XSS that utilizes run-time taint tracking and taint aware parsers to stop the parsing of attacker controlled syntactic content.

Tang et al. [85] presented Alhambra, a browser-based system designed to enforce and test Web browser security policies. They also present two security policies, one that uses taint-tracking engine to prevent XSS attacks and other that uses browsing history to create policies that restrict the contents of document and prevent the inclusion of malicious content.

Ter Louw and Venkatakrishnan [86] presented the design and implementation of BLUEPRINT, a robust prevention approach to XSS attacks by addressing browser inconsistencies in parsing Web contents. Parse tree generated from untrusted HTML with precautions taken to ensure absence of dynamic content on application server is conveyed to client browser.

Iha and Doi [87] described and implemented a binding mechanism in the Web browser for preventing XSS attacks.

Chandra and Selvakumar [88] presented BIXSAN, a Browser Independent XSS SANitizer for prevention of XSS attacks by filtering out harmful HTML content.

Tiwari et al. [89] described a client-side solution that uses step-by-step approach to detect XSS without much degrading of users' browsing experience. Similar step-by-step solution was presented by Shalini and Usha [90] to protect Web applications against XSS attacks without degrading users' browsing experience.

Nikiforakis et al. [91] presented SessionShield, a lightweight client-side protection mechanism against session hijacking, which is one of the main attack vectors of XSS.

Livshits and Erlingsson [92] presented a toolkits that help in preventing injection vulnerabilities including XSS in applications built on AJAX framework by refining the same-origin policy (SOP) in the browser.

Reis et al. [93] presented a vulnerability-driven filtering approach for static and dynamic content implemented into BrowserShield, a general framework that rewrites HTML pages and any embedded scripts to enforce policies on run-time behavior.

Yu et al. [94] presented CoreScript, an operational semantics of a subset of JavaScript useful for understanding and preventing sophisticated exploits that employ higher-order scripts. Edit automata is used to express security policies.

General: Prevention Methods/Approaches: Combination of dynamic and static analysis

Venkatakrishnan et al. [83] described WebAppArmor, a framework that incorporates static and dynamic analysis techniques, symbolic execution, and execution monitoring to prevent XSS, SQL injection, and CSRF on existing (legacy) Web applications.

General: Both Detection and Prevention: Dynamic Analysis

Ismail et al. [74] presented a client-side system that automatically detects and collects XSS vulnerabilities by manipulating either request or server response using user side local proxy servers and uses the information to prevent XSS attacks.

Combination of server-side and client-side

Both Stored and Reflected: Detection Methods/Approaches: Static Analysis

Shahriar and Zulkernine [21] described a mutation-based technique to generate adequate test data sets for detecting XSS vulnerabilities in Web applications that use PHP and JavaScript code. The approach is implemented into MUTEC, a mutation-based testing tool to automatically generate mutants.

Both Stored and Reflected: Detection Methods/Approaches: Dynamic Analysis

Sundareswaran and Squicciarini [34] present a hybrid client-server solution. The proxy-based solution leverages the strengths of both anomaly detection and control flow analysis to provide accurate detection.

Both Stored and Reflected: Prevention Methods/Approaches: Dynamic Analysis

Mui and Frankl [52] presented complementary character coding, a new approach to character level dynamic tainting that allows efficient and precise taint propagation across the

boundaries of server components and also between servers and clients over HTTP. This approach offers precise protection against stored XSS attacks.

Nadji et al. [53] presented an approach that models XSS as privilege escalation vulnerability rather than a sanitization problem by combining randomization of Web application code and runtime tracking of untrusted data both on server and the browser to combat XSS attacks. The technique ensures fundamental integrity property document structure integrity (DSI) that prevents untrusted data from altering the structure of trusted code throughout the execution lifetime of the Web application.

Cao et al. [54] described a new approach to blocking the two main propagation paths of JavaScript worms, DOM access to different view and unauthorized HTTP requests to the server.

Both Stored and Reflected: General: Prevention Methods/Approaches: Combination of dynamic analysis and modeling

Hooimeijer et al. [63] presented BEK, a language and a compiler for writing, analyzing string manipulation routines, and converting them to general-purpose languages. BEK can determine if a target string is a valid output of a sanitizer.

Barth et al. [65] presented an upload filter for Web sites to protect from content-sniffing attacks based on the models extracted from browsers using string-enhanced white-box exploration of binaries. They also presented a two-principled security enhanced browser content-sniffing algorithm that helps to avoid privilege escalation and to use prefix disjoint signatures to prevent content-sniffing attacks.

Generic

Stored: Detection Methods/Approaches: Static Analysis

Galan et al. [95] presented a multi-agent system for automatic scanning of Web-sites to detect the presence of XSS vulnerabilities exploitable by stored XSS-attacks.

Wang et al. [96] described a static analysis algorithm integrated with program slicing method to detect stored XSS vulnerabilities. The slices are composed of two parts, threat injection

and threat release, which reconstruct a stored XSS attack scenario. They also present a prototype implementation of their technique by extending Pixy [104], a tool that performs data flow analysis on PHP code to detect stored XSS vulnerabilities.

Supplementary: Attack Implementation: Modeling

Faghani and Saidi [147] suggested a general model of propagation of XSS worms in virtual social network along with simulation of propagation to verify conformation of described model.

Reflected: Detection Methods/Approaches: Static Analysis

Avancini and Ceccato [98, 99] integrated static analysis with genetic algorithms to suggest candidate false positives reported by static analysis and provide input vectors that expose actual vulnerabilities.

Reflected: Detection Methods/Approaches: Dynamic Analysis

Kals et al. [100] developed SecuBat, a generic and modular security vulnerability scanner that automatically scans Websites to find SQL injection and XSS vulnerabilities.

Reflected: Detection Methods/Approaches: Modeling

Duchene et al. [101] presented an approach to detect reflected XSS vulnerabilities by generating test inputs using a combination of model inference and evolutionary fuzzing. The knowledge about application behavior obtained from model inference is used by genetic algorithm to generate inputs. Kameleon-Fuzz is a work in progress implementation of their described approach.

Reflected: Prevention Methods/Approaches: Modeling

Sharma et al. [134] described an integrated model to prevent reflected XSS and SQL injection attacks in PHP Web applications.

Both Stored and Reflected: Detection Methods/Approaches: Dynamic Analysis

Kieyzun et al. [97] presented a technique for creating SQL injection and XSS attacks in Web applications and an automated tool Ardilla that implements the technique for PHP. The

technique is based on input generation, dynamic taint propagation, and input mutation to find a variant of the input that exposes a vulnerability. The tool is capable of detecting stored second-order XSS.

Both Stored and Reflected: Detection Methods/Approaches: Combination of dynamic and static analysis

Similar to work in [96], work [120] presents a monitor embedding framework DESERVE (DEtecting program SEcurity Vulnerability Exploitations) that identifies exploitable statements from source code based on static backward slicing and embeds necessary code to detect attacks. DESERVE can be used for detecting buffer overflow, SQL injection, and XSS attacks.

Both Stored and Reflected: Prevention Methods/Approaches: Dynamic Analysis

Putthacharoen and Bunyatnoparat [133] presented dynamic cookie rewriting technique implemented in a Web proxy to render the cookie useless for stored and reflected XSS attacks.

Both Stored and Reflected: Prevention Methods/Approaches: Combination of dynamic and static analysis

Phung et al. [132] described a method that combines static and dynamic analysis to prevent or modify inappropriate behavior caused by malicious scripts during JavaScript execution. The approach is based on modifying the code so as to make it self-protecting.

Both Stored and Reflected: Prevention Methods/Approaches: Modeling

Shar and Tan presented code-auditing approach that recovers the defense model implemented in source code and suggests guidelines for checking the adequacy of recovered model against XSS attacks [143]. After extracting all defenses implemented based on possible implementations patterns of defensive coding methods, tainted-information flow graph, a variant of control flow graph is introduced as a model to audit the adequacy of XSS defense artefacts.

General: Detection Methods/Approaches: Static Analysis

Al-Amro and El-Qawasmeh [102] described an algorithm that scans ASP.NET programs for the detection of XSS and SQL injection security vulnerabilities.

Agosta et al. [103] present a methodology and tool for vulnerability identification based on symbolic code execution exploiting static taint analysis to improve the efficiency of the analysis.

Jovanovic et al. [104] described a static source code analysis technique that includes flow-sensitive, inter-procedural, and context-sensitive data flow analysis combined with literal analysis and alias analysis for PHP-based Web applications that can be used to detect SQL injection, XSS, or command injection. They also implemented Pixy, an open source tool for detecting XSS attacks. They also presented an approach to detect Web vulnerabilities based on static source code analysis that uses precise alias analysis targeted at unique reference semantics commonly found in scripting languages [105].

Yu et al. [106] presented an automata-based string analysis tool, Stranger for finding and eliminating string-related security vulnerabilities in PHP applications. It uses symbolic backward and forward reachability analysis to compute possible values the string expression can take during program execution. It is capable of identifying XSS, SQL injection, and malicious file execution (MFE) vulnerabilities in the program.

Choi et al. [107] presented an approach that enables the detection of XSS and SQL injection vulnerabilities through the use of N-Gram analysis to extract malicious code and Support Vector Machines (SVM) to classify it as XSS or SQL injection.

Nunan et al. [121] presented a classification of XSS attacks using Web document-based and URL-based features. They also presented a method that employs machine learning techniques to detect XSS attacks in Web pages.

Komiya et al. [108] described an approach based on machine learning techniques to learn patterns of existing malicious codes and uses the information to classify new code as either malicious or not that helps detecting XSS and SQL injection vulnerabilities.

Perez et al. [109] presents LAPSE+, an enhanced version of Eclipse plugin LAPSE along with its command-line version making it independent of Eclipse IDE which is based on static analysis for detecting XSS vulnerabilities.

General: Detection Methods/Approaches: Dynamic Analysis

Chen and Wu [114] implemented an automated vulnerability scanner for injection attacks in particular SQL injection and XSS capable of detecting vulnerability based on injection point using black-box testing and crawling technique.

Zhang et al. [115] presented an approach to detect Web application vulnerabilities like XSS and SQL injection by analyzing characteristics of Web form and assigning test values to each of the fields for generating test suites, given URL. They implement D-Wav, a tool that incorporates their approach.

Tang et al. [116] described L-WMxD (Lexical based Webmail XSS Discoverer), a Webmail XSS fuzzer that uses lexical-based mutation engine to discover XSS vulnerabilities in Webmail applications.

Frenz and Yoon [122] presented XSSmon, a perl-based an intrusion detection system for XSS that captures potential client-side executable content and its hashing and later reprocessed for any difference that will indicate XSS attack.

General: Detection Methods/Approaches: Combination of static and dynamic analysis

Van Acker et al. [110] presented FlashOver, a system to automatically scan Rich Internet Applications for XSS vulnerabilities using a combination of static and dynamic code analysis.

Lucca et al. [111] described an approach that combines static and dynamic analysis. They define static analysis based criteria is used to detect potential vulnerabilities in server pages and dynamic analysis to detect actual vulnerabilities in a Web application.

General: Detection Methods/Approaches: Combination of dynamic analysis and modeling

Coppolino et al. [123] presented an ontology-based intrusion detection approach that includes diagnostic features and is used to detect XSS and SQL injection attacks.

General: Detection Methods/Approaches: Combination of static analysis and modeling

Yu et al. [112] present a set of abstractions for string and string operations that allow efficient and precise verification of string manipulating programs in particular, properties that involve implicit string relations among string variables.

General: Detection Methods/Approaches: Modeling

Avacini et al. [124] described a security oracle for XSS vulnerabilities based on safe model, which is abstraction of the parse trees of HTML code resulting from safe executions.

Shahriar and Zulkernine [125] describe trustworthiness testing of XSS-based phishing Websites based on behavioral model described using notion of finite state machines (FSM). They implement PhishTester, a tool to automate the testing process.

Xiong et al. [118] presented a model-based testing (MBT) approach using data model that describes relationship between Web security knowledge, business domain knowledge, and test case development for penetration testing using TTCN-3.

Li [119] described a model checking technique using Java source code model checker Bandera to check whether some security programming guidelines are followed and correspondingly detect related security vulnerabilities.

General: Detection Methods/Approaches: Others

Matsuda et al. [126] presented a new XSS detection algorithm considering the appearance frequency and position of characters in an input string.

Fonseca et al. [127] propose a method to evaluate and benchmark automatic Web vulnerability scanners using software fault injection techniques.

Petkov [128] identified 15 vulnerability categories of programming flaws and indexed existing vulnerabilities in OSVDB and CVE databases into those categories.

General: Prevention Methods/Approaches: Dynamic Analysis

Saxena et al. [135] propose ScriptGard, a system that employs dynamic analysis approach to detect and auto-correct context-inconsistency errors in sanitization.

Nanda et al. [136] presented the design, implementation, and evaluation of a dynamic checking compiler called WASC that automatically adds checks into Web applications used in three-tier Internet services to protect them from SQL injection and XSS attacks. WASC compiler uses accurate dynamic information flow tracking and comprehensive HTML/SQL parsing to effectively stop SQL injection and XSS. Extends GIFT compiler.

Li and Wang [137] presented FIRM, a system that embeds inline reference monitor (IRM) in Web pages hosting Flash content and protects it by controlling DOM methods and randomizing variables with sensitive data to prevent XSS attacks. Somorovsky et al. [138] devise a black-box analysis methodology for public cloud interfaces.

General: Prevention Methods/Approaches: Modeling

Brinhosa et al. [139] presented a way to implement an input validation model for Web services to prevent XSS and SQL injection through use of predefined models that specify valid inputs. Their described model consists of an XML schema, an XML specification, and a module for performing input validation according to the schema.

Sun and He [140] presented a model checking method for defense against XSS-attacks. They propose an automatic modeling algorithm for HTML code.

Shar and Tan [144] described an approach for extracting XSS defense features in code to facilitate thorough examination and auditing of code to defend against XSS attacks.

General: Prevention Methods/Approaches: Secure Programming

Grabowski et al. [141] presented a type system for a Java-like language that is refined with extended string types, output effects, and polymorphic method types to automatically verify adherence to programming guidelines.

Johns et al. [129] presented a general approach to outfit modern programming languages with mandatory means for explicit and secure code generation providing strict separation between data and code using embedded language encapsulation type (ELET).

Juillerat [130] presented how a properly designed library can be used to enforce security resulting in creation of robust and secure applications. The author uses "Stones" Java library to validate his approach.

General: Prevention Methods/Approaches: Others

Sivakumar and Garg [131] organized errors into taxonomy and mapped common vulnerability exposures (CVE) with common weakness enumeration (CWE) of Mitre Corp, to construct a common XSS vulnerability enumeration (CXE) that can help security practitioners in recognizing common coding patterns leading to XSS vulnerability along with helping developers identify and rectify existing errors in application. Stuckman and Purtilo [145] propose a benchmarking test-bed for automatic evaluation of Web intrusion prevention systems in a standardized and reproducible way.

Shanmugam and Ponnavaikko [142] described a solution to block XSS attacks independent of languages used to develop Web applications and addresses XSS vulnerabilities that arise from other interfaces. The solution is based on service-oriented architecture (SOA) and makes use of XML and XSD for inter operations of the services. The solution involves generating an XML document based on all form controls submitted by the user which will be validated against a schema at the server side. Wang et al. [146] provide a scheme to collect evidence after suffering XSS attacks from network systems along with management strategies to prevent XSS attacks from network intrusions.

General: Both Detection and Prevention: Combination of static and dynamic analysis

Ruse and Basu [113] presented a two-phase technique that uses both static and dynamic analysis to detect XSS vulnerabilities and prevent XSS attacks. The static analysis phase includes application translation and concolic unit testing techniques whereas runtime monitoring phase monitors relevant variables in the application.

General: Both Detection and Prevention: Combination of dynamic analysis and modeling

Martin and Lam [117] presented QED, a goal-directed model-checking system that automatically generates attacks exploiting taint-based vulnerabilities like XSS and SQL injection.

General: Neither Detection nor Prevention: Modeling

Shar and Tan built vulnerability predictors from static code attributes that characterize input sanitization and validation code patterns for predicting SQL injection and XSS attacks [148]. The authors classify various input sanitization methods and propose a set of static code attributes to represent these types and later use data mining methods to predict SQL injection and XSS vulnerabilities in Web applications [149].

APPENDIX B

TEST PLAN

This test plan documents the unit test plan for the methods in the implementation.

secKMgmt Package

DataParser Class Methods

Table B-1: Test Plan for parseCAPEC method.

Test #	Input	Expected Output	Actual Output	Comments
PA-1	userInput.getCapecTestXmlLoc() = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecParsing0.xml	capecPattern.size = 0	The output matched the expected output.	The passed xml file has no attack pattern tags. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCAPEC/capecParsingTest0.xml
PA-2	userInput.getCapecTestXmlLoc() = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecParsingTest1.xml	capecPattern.size = 1 The contents of the resulting ArrayList should match the contents specified in file that can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest1.txt	The output matched the expected output.	The passed xml file has exactly 1 attack pattern tags. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCAPEC/capecParsingTest1.xml

Test #	Input	Expected Output	Actual Output	Comments
PA-3	userInput.getCapecTestXmlLoc() = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecParsingTest2.xml	capecPattern.size = 5 The contents of the resulting ArrayList should match the contents specified in file that can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt	The output matched the expected output.	The passed xml file has exactly 5 attack pattern tags. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCAPEC/capecParsingTest2.xml
PA-4	userInput.getCapecTestXmlLoc() = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecFile.xml	FileNotFoundException	The output matched the expected output.	The xml file is not found in the specified location.
PA-5	userInput.getCapecTestXmlLoc() = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecParsingTest3.xml	XMLStreamException	The output matched the expected output.	The passed xml file is malformed. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCAPEC/capecParsingTest3.xml

Table B-2: Test Plan for parseCWE method.

Test #	Input	Expected Output	Actual Output	Comments
PW-1	userInput.getCweTestXmlLoc() =	Cwe.Enum.size = 0	The output matched the expected output.	The passed xml file has no weakness tags. The xml file

Test #	Input	Expected Output	Actual Output	Comments
	C:\\\\Users\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweTest0.xml			used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCWE/cweParsingTest0.xml
PW-2	userInput.getCweTestXmlLoc() = C:\\\\Users\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweTest1.xml	cweEnum.size = 1 The contents of the resulting ArrayList should match the contents specified in file that can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest1.txt	The output matched the expected output.	The passed xml file has exactly 1 weakness tag. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCWE/cweParsingTest1.xml
PW-3	userInput.getCweTestXmlLoc() = C:\\\\Users\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweTest2.xml	Cwe.Enum.size = 5 The content of the ArrayList should match the contents specified in file that can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest2.txt	The output matched the expected output.	The passed xml file has exactly 5 weakness tags. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCWE/cweParsingTest2.xml
PW-4	userInput.getCweTestXmlLoc() = C:\\\\Users\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Sour	FileNotFoundException	The output matched the expected output.	The xml file is not found in the specified location.

Test #	Input	Expected Output	Actual Output	Comments
	ce\\\\CWE\\\\cweFile.xml			
PW-5	userInput.getCweTestXmlLoc() = C:\\\\Users\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweParsingTest3.xml	XMLStreamException	The output matched the expected output.	The passed xml file is malformed. The xml file used as test input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/DataParser/parseCWE/cweParsingTest3.xml

***DataReader* Class Methods**

Table B-3: Test Plan for loadLatestCAPEC method.

Test #	Input	Expected Output	Actual Output	Comments
LA-1	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd	Files with names capecXml.xml and capecXsd.xsd are saved at the specified location. The contents of the saved xml file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataReader/loadLatestCAPEC/capecXml.xml The contents of the saved xsd file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataReader/loadLatestCAPEC/capecXsd.xsd	The output matched the expected output.	The capec xml and xsd files from the specified URLs are saved at the specified locations.

Test #	Input	Expected Output	Actual Output	Comments
LA-2	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data//ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd	IOException	The output matched the expected output.	The capec xml URL is correct but capec xsd URL is incorrect.
LA-3	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\Bhanukiran\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd	IOException	The output matched the expected output.	The capec xml and xsd URLs are correct but the xsd file save location is incorrect
LA-4	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data//ap_schema_v2.7.1.xsd	IOException	The output matched the expected output.	The capec xml URL is correct but xsd URL is incorrect. The

Test #	Input	Expected Output	Actual Output	Comments
	userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\SkyDrive\\\\Research\\\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd			location for saving capec xml is incorrect but location for saving xsd is correct.

Table B-4: Test Plan for loadLatestCWE method.

Test #	Input	Expected Output	Actual Output	Comments
LW-1	userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd	Files with names capecXml.xml and capecXsd.xsd are saved at the specified location. The contents of the saved xml file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataReader/loadLatestCWE/cweXml.xml The contents of the saved xsd file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataReader/loadLatestCWE/cweXsd.xsd	The output matched the expected output.	The cwe xml and xsd files from the specified URLs are saved in the specified locations.
LW-2	userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip	IOException	The output matched the expected output.	The cwe xml URL is correct but

Test #	Input	Expected Output	Actual Output	Comments
	userInput.getCweXsdUrl = http://cwe.mitre.org/data//cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd			cwe xsd URL is incorrect.
LW-3	userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\SkyDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd	IOException	The output matched the expected output.	The cwe xml and xsd URLs are correct but the xsd file save location is incorrect
LW-4	userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data//cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\Administrator\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml	IOException	The output matched the expected output.	The cwe xml URL is correct but xsd URL is incorrect. The location for saving cwe xml is incorrect but location for

Test #	Input	Expected Output	Actual Output	Comments
	userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd			saving xsd is correct.

***OntologyManager* Class Methods**

Table B-5: Test Plan for updateCAPEC method.

Test #	Input	Expected Output	Actual Output	Comments
UA-1	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest0.owl	The output matched the expected output.	The ontology file containing zero threat individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest0.owl The number of attack patterns resulting from parsing xml file is zero.
UA-2	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest1.owl	The output matched the expected output.	The ontology file containing zero threat individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest0.owl The number of attack patterns resulting from parsing xml file is one. The attack pattern information

Test #	Input	Expected Output	Actual Output	Comments
		The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.		used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest1.txt
UA-3	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest2.owl The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing zero threat individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest0.owl The number of attack patterns resulting from parsing xml file is five. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt
UA-4	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2	The resulting ontology after updating with capec information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit	The output matched the expected output.	The ontology file containing one threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest1.owl

Test #	Input	Expected Output	Actual Output	Comments
	015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest3.owl		The number of attack patterns resulting from parsing xml file is zero.
UA-5	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest4.owl The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing one threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest1.owl The number of attack patterns resulting from parsing xml file is one. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest1.txt
UA-6	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest5.owl	The output matched the expected output.	The ontology file containing one threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest1.owl The number of attack patterns resulting from parsing xml file is five. The attack pattern information

Test #	Input	Expected Output	Actual Output	Comments
		The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.		used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt
UA-7	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest6.owl	The output matched the expected output.	The ontology file containing one threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest2.owl The number of attack patterns resulting from parsing xml file is one. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest1.txt
UA-8	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/up	The output matched the expected output.	The ontology file containing one threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest2.owl The number of attack patterns resulting from parsing xml file is

Test #	Input	Expected Output	Actual Output	Comments
	userInput.getProvIRI = http://www.w3.org/ns/prov	dateCAPEC/updateCapecTest7.owl The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.		five. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt
UA-9	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with capec information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCAPEC/updateCapecTest8.owl	The output matched the expected output.	The ontology file containing five threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest3.owl The number of attack patterns resulting from parsing xml file is five. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt
UA-10	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_Ver5.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2	The resulting ontology after updating with capec information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20O	The output matched the expected output.	The ontology file containing five threat individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCAPEC/updateCapecTest4.owl

Test #	Input	Expected Output	Actual Output	Comments
	015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	utputs/OntologyManager/updateCAPEC/updateCapecTest9.owl The Retrieval, Organization, Person, Likelihood, Severity, Pattern and Threat Entity has the individuals with data and object properties created as shown in the expected output file.		The number of attack patterns resulting from parsing xml file is five. The attack pattern information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCAPEC/capecParsingTest2.txt

Discussion:

During the first pass of testing, test case UA-9 did not pass and "Null Pointer Exception" was thrown. This was because some of the fields being retrieved for the attack pattern did not have any value. This was fixed by checking if the value retrieved was not null before inserting that value as property of an individual in the ontology.

Table B-6: Test Plan for updateCWE method.

Test #	Input	Expected Output	Actual Output	Comments
UW-1	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19	The resulting ontology after updating with cwe information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest0.owl	The output matched the expected output.	The ontology file containing zero weakness individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest0.owl The number of weaknesses resulting from parsing xml file is zero.

Test #	Input	Expected Output	Actual Output	Comments
	userInput.getProvIRI = http://www.w3.org/ns/prov			
UW-2	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest1.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing zero weakness individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest0.owl The number of weaknesses resulting from parsing xml file is one. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest1.txt
UW-3	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest2.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as	The output matched the expected output.	The ontology file containing zero weakness individuals that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest0.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as attack pattern information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expe

Test #	Input	Expected Output	Actual Output	Comments
		shown in the expected output file.		cted%20Outputs/DataParser/parseCWE/cweParsingTest2.txt
UW-4	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest3.owl	The output matched the expected output.	The ontology file containing one weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest1.owl The number of weaknesses resulting from parsing xml file is zero.
UW-5	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest4.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing one weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest1.owl The number of weaknesses resulting from parsing xml file is one. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest1.txt

Test #	Input	Expected Output	Actual Output	Comments
UW-6	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest5.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing one weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest1.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest2.txt
UW-7	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest6.owl	The output matched the expected output.	The ontology file containing one weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest2.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest1.txt

Test #	Input	Expected Output	Actual Output	Comments
UW-8	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest7.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing one weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest2.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest2.txt
UW-9	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information remains unchanged. The resulting owl file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest8.owl	The output matched the expected output.	The ontology file containing five weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest3.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest2.txt

Test #	Input	Expected Output	Actual Output	Comments
UW-10	userInput.getCapecUpdSavOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\capecUpdSavedOnt.owl userInput.getSecOntIRI = http://www.semanticweb.org/bhanukiran/ontologies/2015/10/untitled-ontology-19 userInput.getProvIRI = http://www.w3.org/ns/prov	The resulting ontology after updating with cwe information with contents can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/OntologyManager/updateCWE/updateCweTest9.owl The Retrieval, Organization, Person, Likelihood, and Weakness Entity has the individuals with data and object properties created as shown in the expected output file.	The output matched the expected output.	The ontology file containing five weakness individual that is used as input can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/updateCWE/updateCweTest4.owl The number of weaknesses resulting from parsing xml file is five. The weakness information used is same as weakness information found at link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/DataParser/parseCWE/cweParsingTest2.txt

Discussion:

During the first pass of testing, test case UW-9 did not pass and "Null Pointer Exception" was thrown. This was because some of the fields being retrieved for the weakness did not have any value. This was fixed by checking if the value retrieved was not null before inserting that value as property of an individual in the ontology.

Table B-7: Test Plan for testOrgExists method.

Test #	Input	Expected Output	Actual Output	Comments
OE-1	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl	True	The output matched the expected output.	The owl ontology file contains "The Mitre Corporation" as an Organization name. The owl ontology file can be found at the link:

Test #	Input	Expected Output	Actual Output	Comments
	agentName = "Mitre Corporation"			https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
OE-2	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = MITRE Corporation	False	The output matched the expected output.	The owl ontology file contains "The Mitre Corporation" as an Organization name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
OE-3	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = MiTrE CoRpOrAtIoN	False	The output matched the expected output.	The owl ontology file contains "The Mitre Corporation" as an Organization name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
OE-4	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = " Mitre Corporation "	True	The output matched the expected output.	The owl ontology file contains "The Mitre Corporation" as an Organization name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl

Discussion:

During the first pass of testing, test case OE-4 did not result in expected output. If there were any preceding and/or succeeding spaces in the same name, they were getting treated as different names. This was fixed by trimming the string before inserting into ontology as well as trimming the string before comparison.

Table B-8: Test Plan for testPersonExists method.

Test #	Input	Expected Output	Actual Output	Comments
PE-1	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = "CAPECContentTeam"	True	The output matched the expected output.	The owl ontology file contains "CAPECContentTeam" as a Person name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
PE-2	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = CapecContentTeam	False	The output matched the expected output.	The owl ontology file contains "CAPECContentTeam" as a Person name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
PE-3	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = CAPECContentTEAM	False	The output matched the expected output.	The owl ontology file contains "CAPECContentTeam" as a Person name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
PE-4	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = " CAPECContentTeam "	True	The output matched the expected output.	The owl ontology file contains "CAPECContentTeam" as a Person name. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl

Discussion:

During the first pass of testing, test case PE-4 did not result in expected output. If there were any preceding and/or succeeding spaces in the same name, they were getting treated as different names. This was fixed by trimming the string before inserting into ontology as well as trimming the string before comparison.

Table B-9: Test Plan for testThreatExists method.

Test #	Input	Expected Output	Actual Output	Comments
TE-1	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = "CAPEC199"	True	The output matched the expected output.	The owl ontology file contains "CAPEC199" as name of Threat. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
TE-2	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = Capec 199	False	The output matched the expected output.	The owl ontology file contains "CAPEC199" as name of Threat. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
TE-3	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = Capec199	False	The output matched the expected output.	The owl ontology file contains "CAPEC199" as name of Threat. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl
TE-4	userInput.getsecInfOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\infSavedOnt.owl agentName = " CAPEC199 "	True	The output matched the expected output.	The owl ontology file contains "CAPEC199" as name of Threat. The owl ontology file can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/OntologyManager/testExists/testExists0.owl

Discussion:

During the first pass of testing, test case TE-4 did not result in expected output. If there were any preceding and/or succeeding spaces in the same name, they were getting treated as different names. This was fixed by trimming the string before inserting into ontology as well as trimming the string before comparison.

Table B-10: Test Plan for retrieveThreatPatterns method.

Test #	Input	Expected Output	Actual Output	Comments
RA-1	None.	resultList.size = 0	The output matched the expected output.	Considering, the number of individuals of Pattern entity in owl file = 0
RA-2	None.	resultList.size = 6	The output matched the expected output.	Considering, the number of individuals of Pattern entity in owl file = 3 and each individual has 2 hasEntityDescription DataProperty
RA-3	None.	resultList.size = 3	The output matched the expected output.	Considering, the number of individuals of Pattern entity in owl file = 3 and each individual has 1 hasEntityDescription DataProperty
RA-4	None.	resultList.size = 10	The output matched the expected output.	Considering, the number of individuals of Pattern entity in owl file = 4 and each individual has 1, 2, 3, and 4 hasEntityDescription DataProperty respectively.

Table B-11: Test Plan for queryOntology method.

Test #	Input	Expected Output	Actual Output	Comments
MR-1	None.	resultList.size = 0	The output matched the expected output.	Considering, the number of individuals of Characters entity that are part of asset character-set in owl file = 0
MR-2	None.	resultList.size = 6	The output matched the expected output.	Considering, the number of individuals of Characters entity that are part of asset character-set in owl file = 3 and each individual has 2 hasMaliciousRepresentations DataProperty

Test #	Input	Expected Output	Actual Output	Comments
MR-3	None.	resultList.size = 3	The output matched the expected output.	Considering, the number of individuals of Characters entity that are part of asset character-set in owl file = 3 and each individual has 1 hasMaliciousRepresentations DataProperty
MR-4	None.	resultList.size = 10	The output matched the expected output.	Considering, the number of individuals of Characters entity that are part of asset character-set in owl file = 4 and each individual has 1, 2, 3, and 4 hasMalicious Representations DataProperty respectively.

Table B-12: Test Plan for queryRetrieveAssetFormats method.

Test #	Input	Expected Output	Actual Output	Comments
AF-1	None.	resultList.size = 0	The output matched the expected output.	Considering, the number of formats defined by hasEntityDescription of Format individual associated with asset under consideration is zero.
AF-2	None.	resultList.size = 1	The output matched the expected output.	Considering, the number of formats defined by hasEntityDescription of Format individual associated with asset under consideration is one.
AF-3	None.	resultList.size = 3	The output matched the expected output.	Considering, the number of formats defined by hasEntityDescription of Format individual associated with asset under consideration is three.

Table B-13: Test Plan for queryRetrieveAssetLocations method.

Test #	Input	Expected Output	Actual Output	Comments
AL-1	None.	resultList.size = 0	The output matched the expected output.	Considering, the number of allowed locations defined by hasEntityDescription of Locations individual associated with asset under consideration is zero.
AL-2	None.	resultList.size = 1	The output matched the expected output.	Considering, the number of allowed locations defined by hasEntityDescription of Locations individual associated with asset under consideration is one.
AL-3	None.	resultList.size = 3	The output matched the expected output.	Considering, the number of allowed locations defined by hasEntityDescription of Locations individual associated with asset under consideration is three.

Table B-14: Test Plan for queryRetrieveAssetIOLocations method.

Test #	Input	Expected Output	Actual Output	Comments
IOL-1	None.	resultList.size = 0	The output matched the expected output.	Considering, the number of allowed IO locations defined by hasEntityDescription of IOLocations individual associated with asset under consideration is zero.
IOL-2	None.	resultList.size = 1	The output matched the expected output.	Considering, the number of allowed IO locations defined by hasEntityDescription of IOLocations individual associated with asset under consideration is one.
IOL-3	None.	resultList.size = 3	The output matched the expected output.	Considering, the number of allowed IO locations defined by hasEntityDescription of IOLocations individual associated with asset under consideration is three.

secKToRE Package

***RegexGenerator* Class Methods**

Table B-15: Test Plan for buildRegex method.

Test #	Input	Expected Output	Actual Output	Comments
BR-1	None.	resultList.size = 1 and the dfa representation of the resulting regular expression can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildRegex/buildRegex0.txt	The output matched the expected output.	Considering, only < as symbol and allowed charcter-set to be alphanumeric, &, #, ;, and <
BR-2	None.	resultList.size = 1 and the dfa representation of the resulting regular expression can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildRegex/buildRegex1.txt	The output matched the expected output.	Considering, only > as symbol and allowed character-set to be alphanumeric, &, #, ;, and >
BR-3	None.	resultList.size = 2 and the dfa representation of resulting regular expression at 0^{th} position can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildRegex/buildRegex0.txt The dfa representation of resulting regular expression at 1^{st} position can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildRegex/buildRegex1.txt	The output matched the expected output.	Considering, both < and > as symbols and allowed character-set to be alphanumeric, &, #, ;, <, and >
BR-4	None.	resultList.size = 0	The output matched the expected output.	Considering no symbols

Table B-16: Test Plan for createRegex method.

Test #	Input	Expected Output	Actual Output	Comments
CR-1	<	resRegex = seq(&, l, t, ;)	The output matched the expected output.	Builds regular expression representing string passed by creating symbols and appending them one character at a time in the same order in which they appear
CR-2	<	resRegex = sym(<)	The output matched the expected output.	

Table B-17: Test Plan for getPatternRegex method.

Test #	Input	Expected Output	Actual Output	Comments
PR-1	builtRegex = result of test case 1.3 patterns[0] = [[LT{0,2}]][[(?i)s]][[((?i)script)+]][[(?i)c]][[((?i)script)*]][[(?i)r]][[((?i)script){1,2}]][[(?i)i]][[((?i)script)]][[(?i)p]][[((?i)script)*]][[(?i)t]][[((?i)script)*]][[GT{1,2}]][[LT?]][[GT+]][[LT*]]	resultRegex, whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/getPatternRegex/getPatternRegex0.txt	The output matched the expected output.	Considering the allowed character-set to be alphanumeric, &, #, ;, <, and >

Table B-18: Test Plan for buildSymRegex method.

Test #	Input	Expected Output	Actual Output	Comments
LT-1	ltReps = from test case 1 patString = [[LT*]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex0.txt	The output matched the expected output.	Considering allowed character-set includes alphanumeric, &, #, ;, and <

Test #	Input	Expected Output	Actual Output	Comments
LT-2	ltReps = from test case 1 patString = [[LT+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex1.txt	The output matched the expected output.	
LT-3	ltReps = from test case 1 patString = [[LT?]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex2.txt	The output matched the expected output.	
LT-4	ltReps = from test case 1 patString = [[LT{1, 2}]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex3.txt	The output matched the expected output.	
LT-5	ltReps = from test case 1 patString = [[LT{0, 2}]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex4.txt	The output matched the expected output.	
LT-6	ltReps = from test case 1 patString = [[LT{1, 1}]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildLTRegex/buildLTRegex5.txt	The output matched the expected output.	

Table B-19: Test Plan for buildBodyRegex method.

Test #	Input	Expected Output	Actual Output	Comments
BB-1	[[script]]	resultRegex = seq(s, c, r, i, p, t) whose resulting dfa representation can be found at the link:	The output matched the	No range; Case-sensitive

Test #	Input	Expected Output	Actual Output	Comments
		https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildBodyRegex/buildBodyRegex0.txt	expected output.	
BB-2	[[((?i)script){1,1}]]	resultRegex = seq((s|S), (c|C), (r|R), (i|I), (p|P), (t|T)) whose resulting dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildBodyRegex/buildBodyRegex1.txt	The output matched the expected output.	Exact; Case-insensitive
BB-3	[[script{1,2}]]	resultRegex = [seq(s, c, r, i, p, t) ‖ [seq(s, c, r, i, p, t, s, c, r, i, p, t)] whose resulting dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildBodyRegex/buildBodyRegex2.txt	The output matched the expected output.	Fixed; Case sensitive
BB-4	[[(script)*]]	resultRegex = star(seq(s, c, r, i, p, t)) whose resulting dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildBodyRegex/buildBodyRegex3.txt	The output matched the expected output.	Star; Case-Sensitive
BB-5	[[(script)+]]	resultRegex = plus(seq(s, c, r, i, p, t)) whose resulting dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildBodyRegex/buildBodyRegex4.txt	The output matched the expected output.	Plus; Case Sensitive

Table B-20: Test Plan for buildWhitelistRegex method.

Test #	Input	Expected Output	Actual Output	Comments
WR-1	formats.get(0) = [[((alp)‖(num))+]]	resultList.size = 1 and the dfa representation of the resulting regular expression can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildWhitelistRegex/buildWhitelistRegex0.txt	The output matched the expected output.	
WR-2	formats.get(0) = [[(num)*]]	resultList.size = 1 and the dfa representation of the resulting regular expression can be found at the link:	The output matched the	

Test #	Input	Expected Output	Actual Output	Comments
		https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildWhitelistRegex/buildWhitelistRegex1.txt	expected output.	
WR-3	formats.get(0) = [[((num)‖(alp))+]][[(<)+]]	resultList.size = 1 and the dfa representation of the resulting regular expression can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildWhitelistRegex/buildWhitelistRegex2.txt	The output matched the expected output.	
WR-4	formats.get(0) = [[((num)‖(alp))+]][[(<)+]] formats.get(0) = [[((num)‖(alp)+]][[(ALP)*]]	resultList.size = 2 and the dfa representation of the resulting first and second regular expressions can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildWhitelistRegex/buildWhitelistRegex3.txt https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/buildWhitelistRegex/buildWhitelistRegex4.txt	The output matched the expected output.	

Table B-21: Test Plan for createSymb method.

Test #	Input	Expected Output	Actual Output	Comments
CS-1	patString = <	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/createSymb/createSymb0.txt	The output matched the expected output.	
CS-2	patString = ws	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/createSymb/createSymb1.txt	The output matched the expected output.	

Table B-22: Test Plan for processOR method.

Test #	Input	Expected Output	Actual Output	Comments
POR-1	patString = [[((alp)‖(num))]{1,2}]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processOR/processOR0.txt	The output matched the expected output.	
POR-2	patString = [[((alp)‖(num)‖(ALP))+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processOR/processOR1.txt	The output matched the expected output.	
POR-3	patString = [[((alp)‖(num))*]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processOR/processOR2.txt	The output matched the expected output.	

Table B-23: Test Plan for processNoOR method.

Test #	Input	Expected Output	Actual Output	Comments
NOR-1	patString = [[(alp)+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processNoOR/processNoOR0.txt	The output matched the expected output.	
NOR-2	patString = [[(ALP)+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processNoOR/processNoOR1.txt	The output matched the expected output.	
NOR-3	patString = [[(Alpha)+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processNoOR/processNoOR2.txt	The output matched the expected output.	
NOR-4	patString = [[(num)+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processNoOR/processNoOR3.txt	The output matched the expected output.	
NOR-5	patString = [[(>)+]]	resultRegex whose dfa representation can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Expected%20Outputs/RegexGenerator/processNoOR/processNoOR4.txt	The output matched the expected output.	

Table B-24: Test Plan for getUserInput method.

Test #	Input	Expected Output	Actual Output	Comments
UI-1	capecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml capecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd capecXmlLoc = C:/Users/bhanu/OneDrive/Research/Ontology/Source/CAPEC/capecXml.xml capecXsdLoc = C:/Users/bhanu/OneDrive/Research/Ontology/Source/CAPEC/capecXsd.xsd cweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip cweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd cweXmlLoc = C:/Users/bhanu/OneDrive/Research/Ontology/Source/CWE/cweXml.xml cweXsdLoc = C:/Users/bhanu/OneDrive/Research/Ontology/Source/CWE/cweXsd.xsd secOntLoc = C:/Users/bhanu/OneDrive/Research/Ontology/secKOnt_TestVer6.owl	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_TestVer6.owl	The output matched the expected output.	The inputs prompted for include: a) URL of CAPEC xml, b) URL of CAPEC xsd, c) location for saving CAPEC xml, d) location for saving CAPEC xsd, e) URL of CWE xml, f) URL of CWE xsd, g) location for saving CAPEC xml, h) location for saving CWE xsd, and i) location of ontology file.

Test #	Input	Expected Output	Actual Output	Comments
UI-2	capecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml capecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd cweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip cweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml userInput.getCapecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_TestVer6.owl	The output matched the expected output.	Same as above.
UI-3	None are entered	userInput.getCapecXmlUrl = http://capec.mitre.org/data/xml/capec_v2.8.xml	The output matched the	Same as above.

Test #	Input	Expected Output	Actual Output	Comments
		userInput.getCapecXsdUrl = http://capec.mitre.org/data/xsd/ap_schema_v2.7.1.xsd userInput.getCapecXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXml.xml userInput.getCapecXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CAPEC\\\\capecXsd.xsd userInput.getCweXmlUrl = http://cwe.mitre.org/data/xml/cwec_v2.9.xml.zip userInput.getCweXsdUrl = http://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd userInput.getCweXmlLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXml.xml userInput.getCweXsdLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\Source\\\\CWE\\\\cweXsd.xsd userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDrive\\\\Research\\\\Ontology\\\\secKOnt_TestVer6.owl	expected output.	

Table B-25: Test Plan for processUserInput method.

Test #	Input	Expected Output	Actual Output	Comments
PUI-1	No Inputs.	userInput.getSecOntLoc = C:\\\\Users\\\\bhanu\\\\OneDriv	The output matched the	This method processes the values stored in userInput object and puts them in proper format for further use in the system.

Test #	Input	Expected Output	Actual Output	Comments
		e\\\\Research\\\\Ontology\\\\secKOnt_TestVer6.owl	expected output.	Processing of one of the valid user input stored is considered for unit testing. For instance, if userInput.getSecOntLoc = C:/Users/bhanu/OneDrive/Research/Ontology/secKOnt_TestVer6.owl
PUI-2	No Inputs.	userInput.getSecOntLoc = SomeInvalidPathInput	The output matched the expected output.	Processing of one of the invalid user input stored is considered for unit testing. For instance, userInput.getSecOntLoc = SomeInvalidPathInput

reToDFA Package

Alt Class Methods

Table B-26: Test Plan for mkNfa method.

Test #	Input	Expected Output	Actual Output	Comments
ALT-1	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=4; exit=5; transitions = {0=[-p-> 1], 1=[-null-> 5], 2=[-@-> 3], 3=[-null-> 5], 4=[-null-> 0, -null-> 2], 5=[]}	The output matched the expected output.	Both regular expressions contain only one symbol. In this case, p and @ are used as symbol for first and second regular expression respectively.
ALT-2	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=8; exit=9; transitions = {0=[-p-> 1], 1=[-null-> 2], 2=[-@-> 3], 3=[-null-> 9], 4=[-k-> 5], 5=[-null-> 6], 6=[-:-> 7], 7=[-null-> 9], 8=[-null-> 0, -null-> 4], 9=[]}	The output matched the expected output.	Both regular expressions contain have more than one symbol. In this case, first regular expression has p and @ symbols in sequence and second regular expression has k and : symbols in sequence.
ALT-3	Names of type Nfa.NameSource (integer	The expected Nfa has the following information: start=8; exit=9; transitions = {0=[-@-> 1], 1=[-null-> 9], 2=[-k-> 3], 3=[-null-> 7],	The output matched the expected output.	One regular expression contains only one symbol and other regular expression contains more than one symbol. In this case, first regular expression has @

Test #	Input	Expected Output	Actual Output	Comments
	counter for states)	4=[-:-> 5], 5=[-null-> 7], 6=[-null-> 2, -null-> 4], 7=[-null-> 9], 8=[-null-> 0, -null-> 6], 9=[]}		symbol and second regular expression has k and : symbols in alternate ways leading to accept state.

***Dfa* Class Methods**

Table B-27: Test Plan for matchDfa method.

Test #	Input	Expected Output	Actual Output	Comments
MD-1	The Dfa used by the method can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/Dfa/Dfa0.txt Input = sSCRIPT	false	The output matched the expected output.	The pattern considered is [[LT{0,2}]][[(?i)s]][[SCRIPT]] [[GT{1,2}]]. The allowed character-set is alphanumeric, &, #, ;, <, and >.
MD-2	Input = sSCRIPT>	True	The output matched the expected output.	
MD-3	Input = <SSCRIPT>&GT	True	The output matched the expected output.	
MD-4	Input = <<<<<<<&#60sSCRIPT&#62&#x3E	True	The output matched the expected output.	
MD-5	Input = sscript>	False	The output matched the expected output.	
MD-6	Input = &ltSsCrIpT&#62	False	The output matched the expected output.	

Table B-28: Test Plan for matchWhitelistFormat method.

Test #	Input	Expected Output	Actual Output	Comments
FM-1	The Dfa used by the method can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/Dfa/Dfa2.txt Input = abc12345zz	True	The output matched the expected output.	The pattern considered is [[((alp)‖(num))+]]. The allowed character-set is alphanumeric.
FM-2	The Dfa used by the method can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/Dfa/Dfa1.txt Input = xpa76run>	True	The output matched the expected output.	The pattern considered is [[((alp)‖(num))+]][[(>)+]]. The allowed character-set is alphanumeric, and >.
FM-3	The Dfa used by the method can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/Dfa/Dfa1.txt Input = < xpa76run	False	The output matched the expected output.	The pattern considered is [[((alp)‖(num))+]][[(>)+]]. The allowed character-set is alphanumeric, and >.
FM-4	The Dfa used by the method can be found at the link: https://github.com/bgurijala/TestFiles/blob/master/Unit%20Testing/Inputs/Dfa/Dfa3.txt Input = <<<<	True	The output matched the expected output.	The pattern considered is [[(<)+]]. The allowed character-set is <.

***Nfa* Class Methods**

Table B-29: Test Plan for toDfa method.

Test #	Input	Expected Output	Actual Output	Comments
DFA-1	N/A	The expected Dfa has the following information:	The output matched the	The Nfa that gets converted to Dfa is as follows: start=0; exit=1;

Test #	Input	Expected Output	Actual Output	Comments
		start=0 accept=[1] transitions = {0={$=1}, 1={}}	expected output.	transitions = {0=[-$-> 1], 1=[]}
DFA-2	N/A	The expected Dfa has the following information: start=1 accept=[0, 2, 3] transitions = {0={}, 1={$=3, :=2, k=0}, 2={}, 3={}}	The output matched the expected output.	The Nfa that gets converted to Dfa is as follows: start=8; exit=9; transitions = {0=[-$-> 1], 1=[-null-> 9], 2=[-k-> 3], 3=[-null-> 7], 4=[-:-> 5], 5=[-null-> 7], 6=[-null-> 2, -null-> 4], 7=[-null-> 9], 8=[-null-> 0, -null-> 6], 9=[]}
DFA-3	N/A	The expected Dfa has the following information: start=0 accept=[1, 2] transitions = {0={$=3}, 1={}, 2={}, 3={:=2, k=1}}	The output matched the expected output.	The Nfa that gets converted to Dfa is as follows: start=0; exit=7; transitions = {0=[-$-> 1], 1=[-null-> 6], 2=[-k-> 3], 3=[-null-> 7], 4=[-:-> 5], 5=[-null-> 7], 6=[-null-> 2, -null-> 4], 7=[]}
DFA-4	N/A	The expected Dfa has the following information: start=1 accept=[1, 2] transitions = {0={:=2}, 1={k=0}, 2={k=0}}	The output matched the expected output.	The Nfa that gets converted to Dfa is as follows: start=4; exit=4; transitions = {0=[-k-> 1], 1=[-null-> 2], 2=[-:-> 3], 3=[-null-> 4], 4=[-null-> 0]}

Seq **Class Methods**

Table B-30: Test Plan for mkNfa method.

Test #	Input	Expected Output	Actual Output	Comments
SEQ-1	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=0; exit=3; transitions = {0=[-c-> 1], 1=[-null-> 2], 2=[-%-> 3], 3=[]}	The output matched the expected output.	Both regular expressions contain only one symbol. In this case, c and % are used as symbol for first and second regular expression respectively.

Test #	Input	Expected Output	Actual Output	Comments
SEQ-2	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=0; exit=7; transitions = {0=[-c-> 1], 1=[-null-> 2], 2=[-%-> 3], 3=[-null-> 4], 4=[-a-> 5], 5=[-null-> 6], 6=[-b-> 7], 7=[]}	The output matched the expected output.	Both regular expressions contain have more than one symbol. In this case, first regular expression has c and % symbols in sequence and second regular expression has a and b symbols in sequence.
SEQ-3	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=0; exit=7; transitions = {0=[-c-> 1], 1=[-null-> 6], 2=[-y-> 3], 3=[-null-> 7], 4=[-z-> 5], 5=[-null-> 7], 6=[-null-> 2, -null-> 4], 7=[]}	The output matched the expected output.	One regular expression contains only one symbol and other regular expression contains more than one symbol. In this case, first regular expression has c symbol and second regular expression has y and z symbols in alternate ways leading to accept state.

***Star* Class Methods**

Table B-31: Test Plan for mkNfa method.

Test #	Input	Expected Output	Actual Output	Comments
ST-1	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=2; exit=2; transitions = {0=[-$-> 1], 1=[-null-> 2], 2=[-null-> 0]}	The output matched the expected output.	The regular expression contains only one symbol. In this case, $ is used as symbol.
ST-2	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=4; exit=4; transitions = {0=[-a-> 1], 1=[-null-> 2], 2=[-<-> 3], 3=[-null-> 4], 4=[-null-> 0]}	The output matched the expected output.	The regular expression contains more than one symbols in sequence. In this case a, < symbols are used in sequence.
ST-3	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=6; exit=6; transitions = {0=[-x-> 1], 1=[-null-> 5], 2=[-&-> 3], 3=[-null-> 5], 4=[-null-> 0, -null-> 2], 5=[-null-> 6], 6=[-null-> 4]}	The output matched the expected output.	The regular expression contains more than one symbols in alternate ways leading to accept state. In this case, x and & are symbols used.

***Sym* Class Methods**

Table B-32: Test Plan for mkNfa method.

Test #	Input	Expected Output	Actual Output	Comments
SY-1	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=0; exit=1; transitions = {0=[- -> 1], 1=[]}	The output matched the expected output.	The symbol set in the object is white-space.
SY-2	Names of type Nfa.NameSource (integer counter for states)	The expected Nfa has the following information: start=0; exit=1; transitions = {0=[-#-> 1], 1=[]}	The output matched the expected output.	The symbol set in the object is any character (including special symbols). In this case, # was used for testing.

APPENDIX C

CRC CARDS FOR XSSMON TOOL

The Class-Responsibility-Collaboration (CRC) cards for the classes of XSSMon as depicted in Fig. 4.1 can be found in Tables C-1 through C-13. Each table represents a CRC card for a class.

Table C-1: CRC Card for MonitorIO class.

Class Name: MonitorIO	
Super classes: None.	
Subclasses: None.	
Description: This class accepts, processes, and flags IO.	
Responsibilities: - Initialize - Customize - Flags IO	**Collaborations:** OntologyManager (OM1) RegexGenerator (RE1) DFA (D3)
Comments:	

Table C-2: CRC Card for UserInput class.

Class Name: UserInput	
Super classes: None.	
Subclasses: None.	
Description: This class acts as an interface between the user and the system and is responsible for accepting user inputs	
Responsibilities: *Knows Data Source Locations (UI1)* - Knows latest CAPEC xml and xsd file URLs - Knows latest CWE xml and xsd file URLs *Knows Source Files Saved Location (UI2)* - Knows location to save CAPEC xml and xsd files - Knows location to save CWE xml and xsd files *Knows Ontology Location (UI3)* - Knows location of security ontology - Knows location to save inferred ontology	**Collaborations:**

Knows DFA Saved Location (UI4) - Knows location to save DFAs	
Comments:	

Table C-3: CRC Card for Alt class.

Class Name: Alt	
Super classes: Regex	
Subclasses: None.	
Description: This class creates representation of NFA using two regular expressions that matches when either of the regular expressions occur once.	
Responsibilities: *Make NFA that matches when either of the regular expressions occur once Contract (R1)* *R2 Contract* - Constructor accepts two regular expressions for ORing	**Collaborations:** Nfa (N2)
Comments:	

Table C-4: CRC Card for Dfa class.

Class Name: Dfa	
Super classes:	
Subclasses:	
Description: This class creates representation of DFA.	
Responsibilities: - Knows location to save DFA - Knows DFA Contract (start state, accept state, and transitions) *Save DFA Contract (D1)* *Match Patterns Contract (D2)*	**Collaborations:** UserInput (UI4)
Comments:	

Table C-5: CRC Card for Nfa class.

Class Name: Nfa
Super classes:
Subclasses:
Description: This class creates representation of NFA and converts NFA to DFA. This class contains two nested classes: NameSource and Transition. NameSource class creates

distinctly names states when constructing a NFA. Transition class represents a transition from one state to another.	
Responsibilities: - Computes composite Dfa transitions - Computes epsilon closure - Renames composite state *Convert NFA to DFA Contract (N1)* *Knows NFA Contract (start state, accept states, and transitions) (N2)*	**Collaborations:** None
Comments:	

Table C-6: CRC Card for Regex class.

Class Name: Regex (Abstract Class)	
Super classes: None.	
Subclasses: Alt, Seq, Star, Sym	
Description: This is an abstract class that creates representation of NFA for a regular expression.	
Responsibilities: *Make NFA Contract (R1)*	**Collaborations:** Nfa (N2)
Comments:	

Table C-7: CRC Card for Seq class.

Class Name: Seq	
Super classes: Regex	
Subclasses: None.	
Description: This class creates representation of NFA from two regular expressions that matches when the preceding regular expression is followed by the succeeding regular expression.	
Responsibilities: *Makes NFA that matches when first regular expression is followed by second regular expression Contract (R1)* *R2 Contract* - Constructor accepts two regular expressions and puts them in sequence	**Collaborations:** Nfa (N2)
Comments:	

Table C-8: CRC Card for Star class.

Class Name: Star	
Super classes: Regex	
Subclasses: None.	
Description: This class creates a representation of NFA from a regular expression that matches when the regular expression occurs 0 or more times.	
Responsibilities: *Makes NFA that matches when a regular expression occurs 0 to n times Contract (R1)* *R2 Contract* - Constructor accepts one regular expression to perform Star operation on it.	**Collaborations:** Nfa (N2)
Comments:	

Table C-9: CRC Card for Sym class.

Class Name: Sym	
Super classes: Regex	
Subclasses: None.	
Description: This class creates representation of NFA for a symbol under consideration.	
Responsibilities: *Make NFA for a symbol Contract (R1)* *R2 Contract* - Constructor accepts a symbol as a String.	**Collaborations:** Nfa (N2)
Comments:	

Table C-10: CRC Card for DataReader class.

Class Name: DataReader	
Super classes: None.	
Subclasses: None.	
Description: This class loads the latest files from data sources. Precisely, this class hosts methods for loading CAPEC and CWE (xml and xsd) files from the data sources.	
Responsibilities: - Knows URLs of attack pattern sources (xml and xsd files) - Knows URLs of weakness sources (xml and xsd sources) - Knows location to save attack pattern files (xml and xsd files) - Knows location to save weakness files (xml and xsd files)	**Collaborations:** UserInput (UI1, UI2)

Load attack pattern files Contract (xml and xsd files) (DR1) *Load weakness files (xml and xsd files) (DR2)*	
Comments:	

Table C-11: CRC Card for DataParser class.

Class Name: DataParser	
Super classes: None.	
Subclasses: None.	
Description: This class defines two other classes in its body: AttackPattern class and Weakness class. These classes define the structure for storing all relevant attack pattern information and weakness information retrieved from data source(s) CAPEC (Common Attack Pattern Enumeration and Classification) and CWE (Common Weakness Enumeration) respectively. This class parses the CAPEC and CWE xml files and retrieves useful information and stores them in defined structures for further use.	
Responsibilities: - Knows location of saved attack pattern files (xml and xsd files) - Knows location of weakness files (xml and xsd files) - Saves attack pattern information - Saves weakness information *Parse attack pattern file (xml) Contract (DP1)* *Parse weakness file (xml) Contract (DP2)*	**Collaborations:** UserInput (UI2)
Comments:	

Table C-12: CRC Card for OntologyManager class.

Class Name: OntologyManager	
Super classes: None.	
Subclasses: None.	
Description: This class manages all operations on the security ontology (knowledge-base). This class hosts methods for loading, updating, inferring, and querying the ontology.	
Responsibilities: - Tests if an Agent or Entity already exists in ontology - Knows location of ontology - Knows location to store inferred ontology - Update attack pattern information in ontology - Update weakness information in ontology	**Collaborations:** DataReader (DR1, DR2) DataParser (DP1, DP2) UserInput (UI3)

- Infer ontology - Save inferred ontology *Initialize Contract (OM1)* - Loads Ontology - Updates attack pattern information in ontology - Updates weakness information in ontology - Infers ontology - Save inferred ontology *Retrieve applicable malicious representations Contract (OM2)* *Retrieve applicable attack patterns Contract (OM3)*	
Comments:	

Table C-13: CRC Card for RegexGenerator class.

Class Name: RegexGenerator	
Super classes: None.	
Subclasses: None.	
Description: This class hosts methods for building customized application-specific regular expression using the knowledge retrieved from the ontology.	
Responsibilities: - Knows applicable malicious representations - Knows applicable attack patterns *Customize Contract (RE1)* - Formalization of application-specific knowledge - Builds regular expressions for special symbols - Builds regular expression for non-special symbols - Builds regular expression by merging regular expressions for both special and non-special characters - Builds customized regular expression based on generic applicable attack patterns	**Collaborations:** OntologyManager (OM2, OM3)
Comments:	

APPENDIX D

DETAILED DESIGN AND TEST STRATEGY FOR UNIT TESTING METHODS OF XSSMON TOOL

The pre and post conditions for each of the methods along with the test strategy for designing test suite is described below:

secKMgmt Package

***DataParser* Class Methods**

1. public ArrayList<secKMgmt.AttackPattern> parseCAPEC(UserInput userInput) throws FileNotFoundException,XMLStreamException

Method Description: This method parses the CAPEC xml file and retrieves attack pattern information that includes: ID, title, summary, likelihood, severity, submitter info (such as source, submitter name, organization, and date), and modification info (such as source, modifier name, organization, date, and change summary) for each of the attacks and saves it in the structure defined by AttackPattern class. The maximum size of the resulting ArrayList is restricted by the memory size available.

Parameter(s): Object of type UserInput, that holds the location of CAPEC xml file

Return: Object of type ArrayList<secKMgmt.AttackPattern>, referred to as capecPattern

Throws: FileNotFoundException, XMLStreamException

Pre-Condition: there exists xml file in the specified location AND the xml file is well-formed

Post-Condition: capecPattern.size is greater than or equal to 0 AND capecPattern.size is equal to number of attack patterns in the xml file AND ID, title, summary, likelihood, severity, submitter name, submitter source, submitter organization, submission date, modifier name, modification source, modifier organization, modification date, change summary fields of each of the array list elements match exactly one attack pattern information in the xml file.

Test Design Strategy: The test strategy is "Equivalence-class" testing. The equivalence classes are: no file presented; the xml file is ill-formed; no attack patterns are in the input file; one attack pattern is in the input file; and more than one attack pattern is in the input file. Because there are

523 attack patterns in the current version of CAPEC xml file, the test case included an input file with 110 attack patterns. See Appendix B, Table B-1 for the Test Plan.

2. public ArrayList<secKMgmt.Weakness> parseCWE(UserInput userInput) throws XMLStreamException, FileNotFoundException

Method Description: This method parses the CWE xml file and retrieves weakness information that includes: ID, name, summary, likelihood, submitter info (such as submitter name, source, date, and organization), modifier info (such as modifier name, source, organization, and date), and related attack patterns information for each of the weaknesses and saves it in the structure defined by Weakness class. The maximum size of the resulting ArrayList is restricted by the memory size available.

Parameter(s): Object of type UserInput, that holds the location of CWE xml file

Return: Object of type ArrayList<secKMgmt.Weakness>, referred to as cweEnum

Throws: FileNotFoundException, XMLStreamException

Pre-Condition: there exists xml file in the specified location AND the xml file is well-formed

Post-Condition: cweEnum.size is greater than or equal to 0 AND cweEnum.size is equal to number of weaknesses in the xml file AND ID, name, summary, likelihood, submitter name, submitter source, submitter organization, submission date, modifier name, modification date, modification source, modifier organization, and related patterns fields of each of the array list elements match exactly one weakness information in the xml file.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes are: no file presented; the xml file is ill-formed; no weaknesses are present in the input file; one weakness is in the input file; and more than one weakness is in the input file. Because there are 724 weaknesses in the current version of CWE xml file, the test case included an input file with 125 weaknesses. See Appendix B, Table B-2 for the Test Plan.

***DataReader* Class Methods**

1. public void loadLatestCAPEC(UserInput userInput) throws IOException

Method Description: This method reads-in all the contents of the user-specified version of CAPEC xml and xsd files from the data source using the URLs specified by the user and stores (saves) these files with user-specified name in the user-specified location.

Parameter(s): Object of type UserInput that holds the URLs of CAPEC files (xml and xsd), and the location information to save them.

Return: Void – does not return anything

Throws: IOException

Pre-Condition: specified save location for xml and xsd files are reachable AND specified URLs for xml and xsd file are reachable AND xml and xsd files are well-formed AND xml and xsd files are buffer and stream without any exceptions

Post-Condition: the xml and xsd files are saved in the specified location with the file name specified by the user.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) valid URLs (of xml and xsd files) and valid path for saving files (both xml and xsd), b) one invalid URL, c) one invalid path for saving file, d) one invalid URL and one invalid path for saving file. The mentioned equivalence classes subsume the testing for all possible exceptions that may be thrown by the method. See Appendix B, Table B-3 for the Test Plan.

2. public void loadLatestCWE(UserInput userInput) throws IOException

Method Description: This method reads-in all the contents of the user-specified version of CWE xml and xsd files from the data source using the URLs specified by the user and stores (saves) these files with user-specified name in the user-specified location.

Parameter(s): Object of type UserInput, that holds the URLs of CWE files (xml and xsd) and the location information to save them.

Return: Void – does not return anything

Throws: IOException

Pre-Condition: specified save location for xml and xsd files are reachable AND specified URLs for xml and xsd file are reachable AND xml and xsd files are well-formed AND xml and xsd files are buffer and stream without any exceptions

Post-Condition: the xml and xsd files are saved in the specified location with the name coming from the data source.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) valid URLs (of xml and xsd files) and valid path for saving files (both xml and xsd), b) one invalid URL, c) one invalid path for saving file, d) one invalid URL and one invalid path for saving file. The mentioned equivalence classes subsume the testing for all possible exceptions that may be thrown by the method. See Appendix B, Table B-4 for the Test Plan.

***OntologyManager* Class Methods**

1. public org.semanticweb.owlapi.model.OWLOntology updateCAPEC(UserInput userInput) throws org.semanticweb.owlapi.model.OWLOntologyCreationException, FileNotFoundException, XMLStreamException, org.semanticweb.owlapi.model.OWLOntologyStorageException

Method Description: This method updates the OWLOntology with information parsed from CAPEC xml file.

Parameter(s): One parameter: Object of type UserInput containing the name-space information of the ontology and imported ontology used within it

Return: Object of type org.semanticweb.owlapi.model.OWLOntology, referred to as ontology

Throws: org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, XMLStreamException, and FileNotFoundException

Pre-Condition: there exists owl file in the specified location AND the xml file is well-formed AND there exists location specified for saving updated ontology

Post-Condition: ontology with updated CAPEC information AND number of individuals of Threat entity created = capecPattern.size AND total number of individuals = number of individuals of Threat entity + capecPattern.size.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) ontology with zero threat individuals and parsed array with zero attack patterns, b) ontology with zero threat individuals and parsed array with 1 attack pattern, c) ontology with zero threat individuals and parsed array with 5 attack patterns, d) ontology with 1 threat individual and parsed array with zero attack patterns, e) ontology with 1 threat individual and parsed array with 1attack pattern not matching existing individual, f) ontology with 1 threat individual and parsed array with 5 attack patterns with none matching existing individual, g) ontology with 1 threat individual and parsed array with 1 attack pattern matching existing individual, h) ontology with 1 threat individual and parsed array with 5 attack patterns and one of them matching existing individual, i) ontology with 5 threat individuals and parsed array with 5 attack patterns that match all existing individuals, j) ontology with 5 threat individuals and parsed array with 5 attack patterns with few matching existing individuals in ontology and few distinct, and k) the strings used to create IRIs of ontology is incorrect (security ontology IRI and provenance IRI). The mentioned equivalence classes subsume the testing for all possible exceptions that may be thrown by the method. See Appendix B, Table B-5 for the Test Plan.

2. public org.semanticweb.owlapi.model.OWLOntology updateCWE(UserInput userInput) throws org.semanticweb.owlapi.model.OWLOntologyCreationException, FileNotFoundException, XMLStreamException, org.semanticweb.owlapi.model.OWLOntologyStorageException

Method Description: This method updates the OWLOntology with information parsed from CWE xml file.

Parameter(s): One parameter: Object of type UserInput containing the name-space information of the ontology and imported ontology used within it

Return: Object of type org.semanticweb.owlapi.model.OWLOntology, referred to as ontology

Throws: org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, XMLStreamException, and FileNotFoundException

Pre-Condition: the exists owl file in the specified location AND the xml file is well-formed AND there exists location specified for saving updated ontology

Post-Condition: ontology with updated CWE information AND number of individuals of Weakness entity created = cweEnum.size AND total number of individuals = number of individuals of Weakness entity + cweEnum.size.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) ontology with zero weakness individuals and parsed array with zero weakness information, b) ontology with zero weakness individuals and parsed array with 1 weakness information, c) ontology with zero weakness individuals and parsed array with 5 weakness information, d) ontology with 1 weakness individual and parsed array with zero weakness information, e) ontology with 1 weakness individual and parsed array with 1weakness information not matching existing individual, f) ontology with 1 weakness individual and parsed array with 5 weakness information with none matching existing individual, g) ontology with 1 weakness individual and parsed array with 1 weakness information matching existing individual, h) ontology with 1 weakness individual and parsed array with 5 weakness information and one of them matching existing individual, i) ontology with 5 weakness individuals and parsed array with 5 weakness information that match all existing individuals, j) ontology with 5 weakness individuals and parsed array with 5 weakness information with few matching existing individuals in ontology and few distinct, and k) the strings used to create IRIs of ontology is incorrect (security ontology IRI and provenance IRI). The mentioned equivalence classes subsume the testing for all possible exceptions that may be thrown by the method. See Appendix B, Table B-6 for the Test Plan.

3. public boolean testOrgExists(UserInput userInput, String agentName) throws FileNotFoundException

Method Description: This method checks whether or not an organization with passed agentName exists in the ontology and returns Boolean result.

Parameter(s): Two parameters: agentName of type String, and Object of type UserInput with location information of inferred ontology

Return: Boolean true is returned if an organization exists with input agentName and false otherwise.

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location AND agentName != ""

Post-Condition: true if agentName already exists in ontology file loaded, false otherwise

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) lower case strings that match and do not match, b) upper case strings that match and do not match, c) mixed case strings that match and do not match, and d) mixed case strings with leading and/or trailing spaces that match and do not match. See Appendix B, Table B-7 for the Test Plan.

4. public boolean testPersonExists(UserInput userInput, String agentName) throws FileNotFoundException

Method Description: This method checks whether or not a person with passed agentName exists in the ontology and returns Boolean result.

Parameter(s): Two parameters: agentName of type String, and Object of type UserInput with location information of inferred ontology

Return: Boolean true is returned if person exists with input agentName and false otherwise.

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location AND agentName != ""

Post-Condition: true if agentName already exists in ontology file loaded, false otherwise

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) lower case strings that match and do not match, b) upper case strings that match and do not match, c) mixed case strings that match and do not match, and d) mixed case strings with leading and/or trailing spaces that match and do not match. See Appendix B, Table B-8 for the Test Plan.

5. public boolean testThreatExists(UserInput userInput, String threatName) throws FileNotFoundException

Method Description: This method checks whether or not a threat with passed threatName exists in the ontology and returns Boolean result.

Parameter(s): Two parameters: threatName of type String, and Object of type UserInput with location information of inferred ontology

Return: Boolean true is returned if threat exists with input threatName and false otherwise.

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location AND threatName != ""

Post-Condition: true if threatName already exists in ontology file loaded, false otherwise

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) lower case strings that match and do not match, b) upper case strings that match and do not match, c) mixed case strings that match and do not match, and d) mixed case strings with leading and/or trailing spaces that match and do not match. See Appendix B, Table B-9 for the Test Plan.

6. public ArrayList<String> retrieveThreatPatterns() throws FileNotFoundException

Method Description: This method retrieves all threat patterns associated with each of the threat individuals and returns them as an ArrayList<String>.

Parameter(s): One parameter: Object of type UserInput with location information of inferred ontology.

Return: Object of type ArrayList<String>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: resultList.size = 0 iff #individuals of Pattern entity in owl ontology file = 0

Else resultList.size > 0 AND resultList.size = Summation of hasEntityDescription DataProperty of each of the individuals of Pattern entity.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) Pattern individuals with only one pattern, and b) Pattern individuals with multiple patterns. See Appendix B, Table B-10 for the Test Plan.

7. public List<String> queryOntology() throws FileNotFoundException

Method Description: This method queries on ontology and produces a result set of all malicious representations of allowed character-set of an asset.

Parameter(s): No parameters passed.

Return: Object of type List<String>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: If malicious representations retrieved from owl file != null then, resultList.size > 0 AND resultList.size = Summation of hasMaliciousRepresentations data property of all individuals of Entity Type Characters such that individuals are part of the asset character-set. iff malicious representations retrieved from owl file != null

Else resultList.size = 0

Test Design Strategy: The test strategy is "Equivalence class" testing. The test case design is performed as follows: a) Character individual with only one malicious representation, b) character

individual with multiple malicious representations, and c) character individuals with combination of one and multiple malicious representations. See Appendix B, Table B-11 for the Test Plan.

8. public ArrayList<String> queryRetrieveAssetFormats(String assetName) throws FileNotFoundException

Method Description: This method queries on ontology and produces a result set of all formats of an asset.

Parameter(s): One parameter: assetName of type String.

Return: Object of type ArrayList<String>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: If asset formats retrieved from owl file != null then, resultList.size > 0 AND resultList.size = Summation of hasEntityDescription data property of all individuals of Entity Type Format such that individuals are related to assetName by isDefinedBy Object Property.

Else resultList.size = 0

Test Design Strategy: The test strategy is "Equivalence class" testing. The test case design is performed as follows: a) Format individual with only one format defined, b) Format individual with multiple formats defined, and c) Format individual with no format defined. See Appendix B, Table B-12 for the Test Plan.

9. public ArrayList<String> queryRetrieveAssetLocations(String assetName) throws FileNotFoundException

Method Description: This method queries on ontology and produces a result set of all allowed and valid server locations that asset can use.

Parameter(s): One parameter: assetName of type String.

Return: Object of type ArrayList<String>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: If allowed locations retrieved from owl file != null then, resultList.size > 0 AND resultList.size = Summation of hasEntityDescription data property of all individuals of Entity Type Locations such that individuals are related to assetName by hasAllowedLocations Object Property.

Else resultList.size = 0

Test Design Strategy: The test strategy is "Equivalence class" testing. The test case design is performed as follows: a) Locations individual with only one allowed location defined, b) Locations individual with multiple allowed locations defined, and c) Locations individual with no allowed locations defined. See Appendix B, Table B-13 for the Test Plan.

10. public ArrayList<String> queryRetrieveAssetIOLocations(String assetName) throws FileNotFoundException

Method Description: This method queries on ontology and produces a result set of all allowed and valid locations from which an asset can load or use IO.

Parameter(s): One parameter: assetName of type String.

Return: Object of type ArrayList<String>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: If allowed locations retrieved from owl file != null then, resultList.size > 0 AND resultList.size = Summation of hasEntityDescription data property of all individuals of Entity Type IOLocations such that individuals are related to assetName by hasAllowedIOLocations Object Property.

Else resultList.size = 0

Test Design Strategy: The test strategy is "Equivalence class" testing. The test case design is performed as follows: a) IOLocations individual with only one allowed location defined, b)

IOLocations individual with multiple allowed locations defined, and c) IOLocations individual with no allowed locations defined. See Appendix B, Table B-14 for the Test Plan.

11. public void getInferredOntology(org.semanticweb.owlapi.model.OWLOntology secOnt, UserInput userInput) throws org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, FileNotFoundException

Method Description: This method runs reasoner on the passed OWLOntology and infers from it and saves the inferred ontology for further processing.

Parameter(s): Two parameters: secOnt of type org.semanticweb.owlapi.model.OWLOntology, and Object of type UserInput containing the location information for saving the inferred ontology

Return: Void – does not return anything

Throws: org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, FileNotFoundException

Pre-Condition: there exists owl file in the specified location AND there exists a reachable location specified for saving the inferred ontology

Post-Condition: The ontology with inferred sub-classes, property assertion, class assertion, data property, object property, class equivalency, inverse object properties is generated and saved in the specified location.

Test Design Strategy: N/A.

12. public org.semanticweb.owlapi.model.OWLOntology loadOntology(UserInput userInput) throws org.semanticweb.owlapi.model.OWLOntologyCreationException

Method Description: This method is used to load an OWLOntology instance

Parameter(s): Object of type UserInput that has the location of the security ontology.

Return: Object of type org.semanticweb.owlapi.model.OWLOntology, referred to as ont

Throws: org.semanticweb.owlapi.model.OWLOntologyCreationException

Pre-Condition: there exists owl file in the specified location

Post-Condition: returns ont that is a reference to the owl file loaded from file on specified location.

Test Design Strategy: N/A.

13. public com.hp.hpl.jena.ontology.OntModel loadOntologyModel(UserInput userInput) throws FileNotFoundException

Method Description: This method returns an instance of OntModel, which is a handle to inferred ontology.

Parameter(s): Object of type UserInput containing location information of inferred ontology.

Return: Object of type com.hp.hpl.jena.ontology.OntModel, referred to as model

Throws: FileNotFoundException

Pre-Condition: there exists owl ontology file in the specified location

Post-Condition: model is reference to owl ontology in the specified location

Test Design Strategy: N/A.

14. public com.hp.hpl.jena.query.ResultSet executeSelectQuery(com.hp.hpl.jena.ontology.OntModel model, String queryString)

Method Description: This method accepts an instance of OntModel and queryString as parameters, executes the query and returns the result set.

Parameter(s): Two parameters: model of type com.hp.hpl.jena.ontology.OntModel and queryString of type String

Return: Object of type com.hp.hpl.jena.query.ResultSet, referred to as resultSet

Throws: NONE

Pre-Condition: none.

Post-Condition: resultSet contains results of executing select query on owl ontology AND resultSet.size = number of triples satisfying the executed select query.

Test Design Strategy: N/A.

secKToRE Package

RegexGenerator Class Methods

1. public ArrayList<Regex> buildRegex() throws FileNotFoundException

Method Description: This method builds regular expressions for all possible malicious representations of allowed symbols (special-characters) and returns an ArrayList of regex, where each list represents all possible malicious representations for an allowed symbol of the character-set

Parameter(s): No parameters passed

Return: Object of type ArrayList<Regex>, referred to as resultList

Throws: FileNotFoundException

Pre-Condition: there exists owl file in the specified location.

Post-Condition: If there exist special symbols with malicious representations, resultList.size > 0 AND resultList.size = number of malicious special symbols with malicious representations (resultList contains a single regular expression for malicious representations of each of the special symbols). Else resultList.size = 0.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) only one symbol allowed by the character-set, b) two symbols allowed by the character-set, and c) no symbols allowed by the character-set. See Appendix B, Table B-15 for the Test Plan.

2. public Regex createRegex(String malVal)

Method Description: This method takes a String of malicious representation as parameter and builds a regular expression representing that string by appending one character at a time.

Parameter(s): One parameter: malVal of type String

Return: Object of type Regex, referred to as resRegex

Throws: NONE

Pre-Condition: malVal != “”

Post-Condition: resRegex is the result of building a single regular expression using all characters of the String malVal in the order in which they appear.

Test Design Strategy: The test strategy is “Equivalence class” testing. The equivalence classes identified are: a) input string with only one character, and b) input string with multiple characters. See Appendix B, Table B-16 for the Test Plan.

3. public ArrayList<Regex> getPatternRegex(ArrayList<Regex> builtRegex,
   ArrayList<String> patterns)

Method Description: This method accepts the customized ArrayList of regular expressions representing special symbols and arraylist of String patterns as parameter, and returns customized ArrayList of regular expressions based on allowed character-set

Parameter(s): Two parameters: builtRegex of type ArrayList<Regex> and patterns of type ArrayList<String>

Return: Object of type ArrayList<Regex>, referred to as resultRegexes

Throws: NONE

Pre-Condition: builtRegex != null AND patterns.size > 0

Post-Condition: resultRegexes.size = patterns.size AND (resultRegexes[i] contains a customized regular expression representing the pattern[i].

Test Design Strategy: The test strategy is “Equivalence class” testing. The equivalence classes identified are: a) pattern with symbols in a range including ?, *, +, zero and non-zero, b) pattern with case-sensitive and case-insensitive body, c) pattern with case-sensitive and case-insensitive body with range including *, +, zero and non-zero. See Appendix B, Table B-17 for the Test Plan.

4. public Regex buildSymRegex(Regex symReps, String patString)

Method Description: This method accepts regular expression representing a symbol (special character such as <, >, etc.) and formalized representation of attack pattern as String and returns customized application-specific regular expression for symbol based on attack pattern.

Parameter(s): Two parameters: symReps of type Regex and patString of type String

Return: Object of type Regex

Throws: NONE

Pre-Condition: symReps != null AND patString != ""

Post-Condition: resultRegex is a single resultant regular expression with all possible malicious representations for a symbol dictated by allowed character-set AND symbol related pattern information (like occurrence information {1,2} or * or ? or +).

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) symbol repeated zero to n times (star operation), b) symbol repeated one to n times (plus operation), c) symbol occurring zero or one time, d) symbol repeated within the specified range (non-zero range such as {2,4}), e) symbol repeated within specified range (including zero such as {0,3}), and f) symbol repeated for exact number of times specified by range (such as {2,2}). See Appendix B, Table B-18 for the Test Plan.

5. public Regex buildBodyRegex(String patString)

Method Description: This method accepts formalized pattern represented as String as parameter and builds a regular expression based on the pattern.

Parameter(s): One parameter: patString of type String

Return: Object of type Regex, referred to as resultRegex

Throws: NONE

Pre-Condition: patString != ""

Post-Condition: resultRegex, a single regular expression for non-special symbols dictated by the pattern information AND non-special symbol related pattern information (?i, or {1,2}, or *).

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) case-sensitive alphanumeric string without range, b) case-insensitive alphanumeric string without range, c) case-insensitive alphanumeric string with exact range (such as {2,2}), c) case-sensitive alphanumeric string with specified range (non-zero such as {1.3}), d) case-sensitive alphanumeric string with specified range (including zero such as {0,2}), e) case-sensitive alphanumeric string repeated zero to n times (star operation), and f) case-sensitive alphanumeric string repeated one or more times (plus operation). See Appendix B, Table B-19 for the Test Plan.

6. public ArrayList<Regex> buildWhitelistRegex(ArrayList<String> formats)

Method Description: This method accepts an array list of formats (represented as strings) of an asset and constructs a regular expression representing each of the format of an asset and returns array list of regular expressions representing all the formats representing an asset.

Parameter(s): One parameter: formats of type ArrayList<String>

Return: Object of type ArrayList<Regex>, referred to as resultRegexes

Throws: NONE

Pre-Condition: N/A

Post-Condition: resultRegexes.size = formats.size AND resultRegexes[i] contains regular expression representing formats[i].

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) Single Format string containing expression with OR symbol only, b) Single Format string containing expression without OR symbol, c) Single Format string containing expression with and without OR symbol, and d) Multiple Format strings containing expression with and without OR symbol. See Appendix B, Table B-20 for the Test Plan.

7. public Regex createSymb(String patString)

Method Description: This method accepts a string as input and creates a regular expression for the symbol represented by the passed string.

Parameter(s): One parameter: patString of type String

Return: Object of type Regex, referred to as resultRegex

Throws: NONE

Pre-Condition: N/A

Post-Condition: resultRegex, a single regular expression representing the special symbol passed as parameter.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) White space as symbol, and b) Special Symbol. See Appendix B, Table B-21 for the Test Plan.

8. public Regex processOR(String patString)

Method Description: This method receives the string that contains OR symbol as input and creates regular expression representation of the string.

Parameter(s): One parameter: patString of type String.

Return: Object of type ArrayList<Regex>, referred to as resultRegex.

Throws: NONE.

Pre-Condition: N/A.

Post-Condition: resultRegex, a single regular expression representing the string passed.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) String with only one OR symbol within specified range, b) String with multiple OR symbols with one or more repetitions, and c) String with multiple OR symbols with zero or more repetitions. See Appendix B, Table B-22 for the Test Plan.

9. public Regex processNoOR(String patString)

Method Description: This method receives the string that does not contain OR symbol as input and creates regular expression of the string.

Parameter(s): One parameter: patString of type String.

Return: Object of type ArrayList<Regex>, referred to as resultRegex.

Throws: NONE

Pre-Condition: N/A

Post-Condition: resultRegex, a single regular expression representing the string passed.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) String with lower-case alphabets, b) String with upper-case alphabets, c) String with both lower-case and upper-case alphabets, d) Strings with numeric, and e) Strings with special symbols. See Appendix B, Table B-23 for the Test Plan.

monitor Package

monitorIO Class Methods

1. public static void main(String[] args) throws IOException, org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, XMLStreamException, ParseException

Method Description: This method is the main method used to execute all the functionality of the system. This method makes a call to methods necessary for executing the functionality of the system.

Parameter(s): One parameter: args of type String array

Return: Void – does not return anything

Throws: IOException, org.semanticweb.owlapi.model.OWLOntologyCreationException, org.semanticweb.owlapi.model.OWLOntologyStorageException, XMLStreamException, and ParseException

inputManagement Package

UserInput Class Methods

1. public void getUserInput()

Method Description: This method prompts the user for input, accepts and saves the input into appropriate variables.

Parameter(s): NONE

Return: Void – does not return anything

Throws: NONE

Pre-Condition: N/A

Post-Condition: The userInput object is set with the user inputs provided after processing them.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) all prompted user inputs are entered, b) Only some prompted user inputs are entered, and c) None of the prompted user inputs are entered. See Appendix B, Table B-24 for the Test Plan.

2. public void processUserInput()

Method Description: This method processes the accepted user input and puts it in usable format.

Parameter(s): NONE

Return: Void – does not return anything

Throws: NONE

Pre-Condition: The user input is prompted and accepted.

Post-Condition: The user input received is prompted and the userInput object is updated with the processed object.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) valid inputs that require processing, and b) invalid inputs that cannot be processed. See Appendix B, Table B-25 for the Test Plan.

reToDFA Package

Alt Class Methods

1. public Nfa mkNfa(Nfa.NameSource names)

Method Description: This method creates NFA from two regular expressions representing two alternate paths from start state to end state.

Parameter(s): One parameter: names of type NameSource that belongs to Nfa class, which is an integer representing the state.

Return: returns object of type Nfa, referred to as nfa0

Throws: NONE

Pre-Condition: None

Post-Condition: If nfa1 has form s1s ----> s1e AND nfa2 has form s2s ----> s2e then,

nfa0 has form s0s -eps-> s1s ----> s1e -eps-> s0e

s0s -eps-> s2s ----> s2e -eps-> s0e

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) both regular expressions have only one symbol, b) both regular expressions have more than one symbols, and c) first regular expression has one symbol and second has more than one symbols. See Appendix B, Table B-26 for the Test Plan.

Dfa Class Methods

1. public boolean matchDfa(String input)

Method Description: This method performs pattern matching of IO strings based on DFA derived from regular expression.

Parameter(s): One parameter: input of type String

Return: Returns a Boolean value. True if the input has a pattern defined by dfa, false otherwise.

Throws: NONE

Pre-Condition: dfa != null AND input != "" AND there should be at least one path from start state to accept state in dfa

Post-Condition: true if the input takes dfa to one of its final states, false otherwise

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) matching DFA of a regular expression that has only symbol occurring at the start, b) matching DFA of a regular expression that has only one symbol occurring at the end, c) matching DFA of a regular expression that starts with one symbol and ends with another one, d) matching DFA of a regular expression that has symbols at the start and end and alphanumeric in between, and e) matching DFA of a regular expression that has combination of symbols and alphanumeric with some range, case-sensitive and case-insensitive. See Appendix B, Table B-27 for the Test Plan.

2. public boolean matchWhitelistFormat(String input)

Method Description: This method performs pattern matching of IO strings based on DFA derived from regular expression.

Parameter(s): One parameter: input of type String

Return: Returns a Boolean value. True if the input has a pattern defined by dfa, false otherwise.

Throws: NONE

Pre-Condition: dfa != null AND input != "" AND there should be at least one path from start state to accept state in dfa

Post-Condition: true if the input takes dfa to one of its final states, false otherwise

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) matching DFA of a regular expression that has OR symbol only, b) matching DFA of a regular expression that does not has OR symbol only, and c) matching DFA of a regular expression that has some symbols with OR and some without OR symbol. See Appendix B, Table B-28 for the Test Plan.

3. public void writeDot(String filename) throws IOException

Method Description: This method writes a file with the DFA created for each of the application-specific regular expressions for future use.

Parameter(s): One parameter: filename of type String

Return: Void – does not return anything

Throws: IOException

Pre-Condition: dfa object is not null AND dfa object has it attributes (start state, accept states, and transitions) set.

Post-Condition: The contents of the dfa object are written to the file.

Test Design Strategy: N/A.

***Nfa* Class Methods**

1. public void addTrans(Integer s1, String lab, Integer s2)

Method Description: This method adds a new transition from state s1 to state s2 on occurrence of lab string.

Parameter(s): Three parameters: two Integer parameters representing state and a String parameter that causes state transition from one state to another.

Return: Void – does not return anything

Throws: NONE

Pre-Condition: None.

Post-Condition: The transition from state s1 to state s2 is added on occurrence of symbol represented by lab String.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as a part of testing mkNfa method of Alt, Seq, and Star classes.

2. public void addTrans(Map.Entry<Integer,List<Nfa.Transition>> tr)

Method Description: This method adds all the transitions of a Nfa object (passed as parameter) to the Nfa object that calls this method.

Parameter(s): One parameter: tr of type Map.Entry<Integer, List<Nfa.Transition>>

Return: Void – does not return anything

Throws: NONE

Pre-Condition: The transitions parameter tr passed is not null.

Post-Condition: The calling Nfa object, referred to as resNfa.transitions = resNfa.transitions UNION tr.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as a part of testing mkNfa method of Alt, Seq, Star, and Sym classes.

3. public Dfa toDfa()

Method Description: This method converts NFA to DFA by constructing composite states using compositeDfaTrans and epsilonClose methods. Then, composite states are replaced by simple states using methods rename and mkRenamer.

Parameter(s): No parameters

Return: Object of type Dfa, referred to as resultDFA

Throws: NONE

Pre-Condition: The start state and transitions of the Nfa object are not null.

Post-Condition: resultDfa, which is equivalent representation of the Nfa object is created and returned.

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) Nfa representing regular expression with only one symbol, b) Nfa representing regular expression of sequence of symbols, c) Nfa representing regular expression of alternate symbols, and d) Nfa representing star operation on regular expression. See Appendix B, Table B-29 for the Test Plan.

4. **static** Map<Set<Integer>, Map<String, Set<Integer>>> compositeDfaTrans(Integer s0, Map<Integer, List<Transition>> trans)

Method Description: Create the epsilon-closure S0 (a HashSet of Integers) of the start state s0, and put it in a worklist. Create an empty DFA transition relation, which is a HashMap from a composite state (an epsilon-closed HashSet of Integers) to a HashMap from a label (a non-null String) to a composite state. Repeatedly choose a composite state S from the worklist. If it is not already in the keyset of the DFA transition relation, compute for every non-epsilon label lab the set T of states reachable by that label from some state s in S. Compute the epsilon-closure Tclose of every such state T and put it on the worklist. Then add the transition S -lab-> Tclose to the DFA transition relation, for every lab.

Parameter(s): Two parameters: s0 of type Integer and trans of type Map<Integer, List<Transition>>

Return: Object of type Map<Set<Integer>, Map<String, Set<Integer>>>, resDfa

Throws: NONE

Pre-Condition: None.

Post-Condition: resDfa, is the Dfa representation of Nfa with composite states.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as a part of testing toDfa method.

5. **static** Set<Integer> epsilonClose(Set<Integer> S, Map<Integer, List<Transition>> trans)

Method Description: Repeatedly choose a state s from the worklist, and consider all epsilon-transitions s -eps-> s' from s. If s' is in S already, then do nothing; otherwise add s' to S and the worklist. When the worklist is empty, S is epsilon-closed; return S.

Parameter(s): Two parameters: States S of type Set<Integer> and trans of type Map<Integer, List<Transition>>

Return: Object of type Set<Integer>, referred to as resSet

Throws: NONE

Pre-Condition: None.

Post-Condition: resSet is set of all states s' such that there is an epsilon transition from state s.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as part of testing of toDfa method.

6. **static** Map<Set<Integer>, Integer> mkRenamer(Collection<Set<Integer>> states)

Method Description: Given a Map from Set of Integer to something, create an injective Map from Set of Integer to Integer, by choosing a fresh Integer for every value of the map.

Parameter(s): One parameter: states of type Collection<Set<Integer>>

Return: Object of type Map<Set<Integer>, Integer>, referred to as refMap

Throws: NONE

Pre-Condition: None.

Post-Condition: resMap has a new integer assigned for every value of set of integers.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as part of testing toDfa method.

7. **static** Map<Integer, Map<String, Integer>> rename(Map<Set<Integer>, Integer> renamer, Map<Set<Integer>, Map<String, Set<Integer>>> trans)

Method Description: This method uses renamer to rename the composite (Set of Integer) states with simple (Integer) states in the transitions, which is a Map from Set of Integer to Map from String to Set of Integer. The result is a Map from Integer to Map from String to Integer.

Parameter(s): Two parameters: renamer of type Map<Set<Integer>, Integer> and trans of type Map<Set<Integer>, Map<String, Set<Integer>>>

Return: Object of type Map<Integer, Map<String, Integer>>, referred to as renamed

Throws: NONE

Pre-Condition: None.

Post-Condition: renamed contains all composite states of the transitions renamed to simple integer state.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as a part of testing toDfa method.

8. **static** Set<Integer> acceptStates(Set<Set<Integer>> states, Map<Set<Integer>, Integer> renamer, Integer exit)

Method Description: This method returns the set of accepted states for the Dfa created from Nfa.

Parameter(s): Three parameters: states of type Set<Set<Integer>>, renamer of type Map<Set<Integer>, Integer>, and exit of type Integer

Return: Object of type Set<Integer>, referred to as accStates

Throws: NONE

Pre-Condition: None.

Post-Condition: accStates = set of integers where each integer represents accept states of Dfa.

Test Design Strategy: The test strategy is "Equivalence class" testing. The testing of this method is performed as a part of testing toDfa method.

***Regex* Class Methods**

1. public abstract Nfa mkNfa(Nfa.NameSource names)

Method Description: This is an abstract method that will be implemented by classes that extend Regex class.

Parameter(s): One parameter: names of type NameSource of Nfa class

Return: Object of type Nfa

Throws: NONE

Pre-Condition: None.

Post-Condition: Returns Nfa object representing Nfa of regular expression.

Test Design Strategy: N/A.

***Seq* Class Methods**

1. public Nfa mkNfa(Nfa.NameSource names)

Method Description: This method creates Nfa that represents results a single path from start state to end state that is sequence of two regular expressions

Parameter(s): One parameter: names of type NameSource of Nfa class, which is an integer representing the state

Return: Object of type Nfa, referred to as nfa0

Throws: NONE

Pre-Condition: None

Post-Condition: If nfa1 has form s1s ----> s1e AND nfa2 has form s2s ----> s2e then,

nfa0 has form s1s ----> s1e -eps-> s2s ----> s2e

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) both regular expressions have only one symbol, b) both regular expressions have more than one symbols, and c) first regular expression has one symbol and second has more than one symbols. See Appendix B, Table B-30 for the Test Plan.

***Star* Class Methods**

1. public Nfa mkNfa(Nfa.NameSource names)

Method Description: This method creates Nfa that represents result of performing * (star) operation on a regular expression.

Parameter(s): One parameter: names of type NameSource of Nfa class, which is an integer representing state.

Return: Object of type Nfa, referred to as nfa0

Throws: NONE

Pre-Condition: None

Post-Condition: If nfa1 has form s1s ----> s1e then,

nfa0 has form s0s ----> s0s

s0s -eps-> s1s

s1e -eps-> s0s

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) regular expression with only one symbol, and b) regular expression with more than one symbols. See Appendix B, Table B-31 for the Test Plan.

***Sym* Class Methods**

1. public Nfa mkNfa(Nfa.NameSource names)

Method Description: This method creates Nfa for a symbol.

Parameter(s): One parameter: names of type NameSource of Nfa class, which is an integer representing state.

Return: Object of type Nfa, referred to as nfa0

Throws: NONE

Pre-Condition: None

Post-Condition: The resulting nfa0 has form s0s -sym-> s0e

Test Design Strategy: The test strategy is "Equivalence class" testing. The equivalence classes identified are: a) the symbol passed is white space, and b) the symbol passed is character (any character including special symbols). See Appendix B, Table B-32 for the Test Plan.

VITA

Bhanukiran Gurijala was born on November 25, 1983. The second child and only son of Narasimhulu Venkata Gurijala and Lavanya Gurijala, he earned his Bachelor of Engineering degree in Computer Science and Engineering from Rashtreeya Vidyalaya College of Engineering (RVCE) affiliated to Visvesvaraya Technological University (VTU) in 2005. He worked as Senior Software Engineer (SSE) in a software firm for 1.5 years in India. Later, he moved to USA for Masters and received Master of Science degree in Software Engineering from University of Houston – Clear Lake in 2008. After working for close to two years in the software field in different roles, he joined the doctoral program in Computer Science at The University of Texas at El Paso.

While pursing the doctoral degree, he worked as a teaching assistant, research associate, and instructor for the department of Computer Science. He co-taught a graduate-level course and an undergraduate-level course as instructor. He served as the head TA in the computer science department for close to 4 years. He served as an advisory board member at UTEP for Office of International Programs (OIP) and Masters' program in Software Engineering when it was initially introduced. He interned at HCSS in summer 2012 and 2013 and led the automation of regression testing efforts. Overall, he has over 4 years of industry experience handling different roles and positions, around 5 years of teaching experience in the roles as lecturer and teaching assistant, and over 2 years of research experience.

Bhanukiran Gurijala's dissertation, *A Unified Cyber-Enhanced Approach for Detecting Cross-Site Scripting Attacks on Web Applications*, was supervised by Dr. Ann Q. Gates.

Permanent address: 610 Prospect Street, Apt 5
El Paso, Texas, 79902-3770

This dissertation was typed by Bhanukiran Gurijala.