UNIVERSITY OF CALIFORNIA
Santa Barbara

Automatic Detection and Repair of Input Validation and Sanitization Bugs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Muath Abdullah Alkhalaf

Committee in Charge:

Professor Tevfik Bultan, Chair

Professor Ben Hardekopf

Professor Christopher Kruegel

June 2014

The Dissertation of
Muath Abdullah Alkhalaf is approved:

Professor Ben Hardekopf

Professor Christopher Kruegel

Professor Tevfik Bultan, Committee Chairperson

June 2014

Automatic Detection and Repair of Input Validation and Sanitization Bugs

Copyright © 2014

by

Muath Abdullah Alkhalaf

To my parents. Thanks for your love and patience. God bless you.

To my dear and lovely wife. Without you I wouldn't have made it through the PhD.

To Abdullah and Deem. Thanks for filling my heart with joy.

Acknowledgements

I am most thankful for my advisor and mentor Tevfik Bultan. Tevfik gave me rigorous theoretical background in computer science and software verification. At the same time, he taught me the correct way to conduct scientific experiments to test and validate my ideas. He devoted considerable time and energy discussing my ideas and polishing my communication skills by revising my research papers, my presentations and this dissertation and giving me countless number of comments and suggestions.

I would like to thank Ben Hardekopf and Christopher Krugel for their detailed corrections and suggestions for this dissertation. I also would like to thank my colleagues, Fang Yu and Abdulbaki Aydin, who helped me developing and testing the string analysis library and verification tools in this dissertation. And I would like to thank Shauvik Roy Choudhary, Mattia Fazzini and Alessandro Orso for their help with the extraction and differential analysis of Java web applications.

I would like to thank King Saud University for the generous financial support for me and my family throughout my PhD.

My gratitude to Julia Bryson, Salman Haq and Vince Newman for their great medical treatment and advice which helped me to cope up with my complex health issues and made it possible for me to get back on my feet again and finish the PhD program.

Finally, Special thanks to my dear wife Banan Alsalhi who did tremendous effort to take care of me and our family throughout the PhD especially while I was sick in the past two years

and to my mother Norah Alghofaily and my father Abdullah Alkhalaf who overwhelmed me with their passion and love and helped me navigating through this life in the best way possible.

PHONE +18057089101 • E-MAIL MUATH@CS.UCSB.EDU, MOATH2@YAHOO.COM

MUATH ABDULLAH ALKHALAF

PERSONAL INFORMATION

- Nationality: Saudi Arabia
- Languages: Arabic, English, little German

EDUCATION

[2008-Now] UCSB Santa Barbara

PhD candidate in Computer Science Department

- Thesis Title: **Automatic Detection and Repair of Input Validation and Sanitization Bugs**
- Advisor: Prof. Tevfik Bultan.

[2006-2008] UCSB Santa Barbara

MS in Computer Science

- Project Title: **Automated web service testing using interface grammars**
- Advisor: Prof. Tevfik Bultan

[1999-2003] King Saud University Riyadh

Bachelor Degree in Computer and Information Sciences

- Graduated as the first of class with 4.81/5.0 GPA and first degree of honor.

PUBLICATIONS

- **Muath Alkhalaf**, Abdulbaki Aydin and Tevfik Bultan. "*Semantic Differential Repair for Input Validation and Sanitization.*" To appear in the Proceedings of 2014 International Symposium on Software Testing and Analysis (ISSTA 2014).
- Abdulbaki Aydin, **Muath Alkhalaf** and Tevfik Bultan. "*Automated Test Generation from Vulnerability Signatures.*" To appear in the Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST 2014).
- Fang Yu, **Muath Alkhalaf**, Tevfik Bultan, Oscar H. Ibarra. "*Automata-Based Symbolic String Analysis for Vulnerability Detection.*" Formal Methods in System Design, volume 44, number 1, pages 44-70, 2014
- **Muath Alkhalaf**, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso and Christopher Kruegel. "*ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies.*" Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), pages 56-66, Minneapolis, USA, July 15-20, 2012.
- **Muath Alkhalaf**, Tevfik Bultan, and Jose L. Gallegos. "*Verifying Client-Side Input Validation Functions Using String Analysis.*" Proceedings of the

34th International Conference on Software Engineering (ICSE 2012) pages 947-957, Zurich, Switzerland, June 2-9, 2012.

- Fang Yu, **Muath Alkhalaf** and Tevfik Bultan. "*Patching Vulnerabilities with Sanitization Synthesis.*" Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pages 251-260, Waikiki, Honolulu , Hawaii, USA, May 21-28, 2011.
- Fang Yu, **Muath Alkhalaf** and Tevfik Bultan. "*Stranger: An Automata-based String Analysis Tool for PHP.*" Tool paper. Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010), LNCS 6015, pages 154-157, Paphos, Cyprus, March 20-28, 2010.
- Sylvain Halle, Tevfik Bultan, Graham Hughes, **Muath Alkhalaf** and Roger Villemaire. "*Runtime Verification of Web Service Interface Contracts.*" IEEE Computer, volume 43, number 3, pages 59-66, March 2010.
- Sylvain Halle, Graham Hughes, Tevfik Bultan, and **Muath Alkhalaf**. "*Generating Interface Grammars from WSDL for Automated Verification of Web Services.*" Proceedings of the 7th International Conference on Service Oriented Computing (ICSOC 2009), pp. 516-530, Stockholm, Sweden, November 24-27, 2009.
- Fang Yu, **Muath Alkhalaf** and Tevfik Bultan. "*Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses.*" Short paper. Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pp. 605-609, Auckland, New Zealand, November 16-20, 2009.
- Graham Hughes, Tevfik Bultan and **Muath Alkhalaf**. "*Client and Server Verification for Web Services Using Interface Grammars.*" Proceedings of the Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB 2008), pp. 40-46, Seattle, Washington, July 21, 2008.

ACADEMIC EXPERIENCE

- Member of Verification Lab at UCSB
- Presented the following paper at **ISSTA 2012 conference**:
  - **Muath Alkhalaf**, Shauvik Roy Choudhary, Mattia Fazzini, Tevfik Bultan, Alessandro Orso and Christopher Kruegel. "*ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies.*" Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), pages 56-66, Minneapolis, USA, July 15-20, 2012.
- Presented the following paper at **ICSE 2012 conference**:
  - **Muath Alkhalaf**, Tevfik Bultan, and Jose L. Gallegos. "*Verifying Client-Side Input Validation Functions Using String Analysis*" Proceedings of the 34th International Conference on Software Engineering (ICSE 2012) pages 947-957, Zurich, Switzerland, June 2-9, 2012.
- Presented the following paper at **TACAS 2010 conference**:
  - Fang Yu, **Muath Alkhalaf** and Tevfik Bultan. "*Stranger: An Automata-based String Analysis Tool for PHP.*" Tool paper. Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS

2010), LNCS 6015, pages 154-157, Paphos, Cyprus, March 20-28, 2010.

- Presented the following paper at **SoCal Programming Languages workshop 2013**:
  - **Muath Alkhalaf**, Abdulbaki Aydin and Tevfik Bultan. " *Differential Patching of Input Validation in Web Applications*".
- Presented the following paper at **SoCal Programming Languages workshop 2011**:
  - **Muath Alkhalaf**, Tevfik Bultan, and Jose L. Gallegos. "*Verifying Client-Side Input Validation Functions Using String Analysis*".
- Received an NSF travel grant to attend the **Summer Formal 2011: First Summer School on Formal Techniques,** a one-week summer school which was organized by SRI International**.**
- Member of UCSB hackers team.
- Teaching Assistant for summer, fall and spring semesters in 2005/2006 in Computer Science Department at King Saud University. I have tutored four courses:
  - Introduction to programming with C++.
  - Introduction to programming with Java.
  - Advanced programming with C++.
  - Computer Arabization.
- Participated in 2001 summer Linux Arabization Project held by King Abdulaziz City for Sciences and Technology and written a paper entitled "Linux Arabization".

VERIFICATION & SECURITY TOOLS

- Lead developer for **SemRep**, which is a semantic differential repair tool for input validation and sanitization code. The tool can be found at (https://github.com/vlab-cs-ucsb/SemRep).
- Lead developer for **Stranger**, which is an open source PHP analysis tool to detect web vulnerabilities in PHP web applications such as XSS and SQLI using symbolic string analysis. The tool can be found at (http://www.cs.ucsb.edu/~vlab/stranger/).
- Co-developer for **LibStranger**, which is an open source automata-based symbolic string analysis library. The library can be found at (https://github.com/vlab-cs-ucsb/Stranger).
- Co-developer for **Linux Alarm** which is a Linux personal firewall implemented as undergraduate graduation project. The project is available in Sourceforge under http://sourceforge.net/projects/linuxalarm.

PROFESSIONAL EXPERIENCE

[June 2003- June 2005] Al-Elm Information Security Company Riyadh
Information Security Engineer

- Worked at Al-Elm Company as a security engineer.
- Earned the SANS GIAC Security Essentials Certification GSEC. As

part of the certification program, I have written a paper titled "*Introduction to Java Cryptography*" which was published on SANS' Information Security Reading Room, accessible through Internet at www.sans.org/rr/.

HONORS & AWARDS

- The Best student of King Saud University 2002/2003. This honor is given to students who can balance between their high academic achievements and their effective contributions to university activities.
- Won King Abdulaziz Saudi National Award for Scientific Creativity for my undergraduate graduation project "Linux Alarm" which is a Linux Personal Firewall.
- "Linux Alarm" was chosen among the best three graduation projects in the competition held at King Saud University, College of Computer and Information Sciences at year 2003.
- Won the best open source project for year 2004 from Saudi Linux Users Group for building "Linux Alarm".
- Represented Collage of Computer and Information Sciences in King Saud University Cultural Competition where we won the second place among colleges.
- Represented my Home City Al-Rass in the Saudi National Competition where we won the second place among Saudi's cities.

Abstract

Automatic Detection and Repair of Input Validation and Sanitization Bugs

Muath Abdullah Alkhalaf

A crucial problem in developing dependable web applications is the correctness of the input validation and sanitization. Bugs in string manipulation operations used for validation and sanitization are common, resulting in erroneous application behavior and vulnerabilities that are exploitable by malicious users. In this dissertation, we investigate the problem of automatic detection and repair of validation and sanitization bugs both at the client-side (JavaScript) and the server-side (PHP or Java) code.

We first present a formal model for input validation and sanitization functions along with a new domain specific intermediate language to represent them. Then, we show how to extract input validation and sanitization functions in our intermediate language from both client and server-side code in web applications. After the extraction phase, we use automata-based static string-analysis techniques to automatically verify and fix the extracted functions. One of our contributions is the development of efficient automata-based string analysis techniques for frequently used, complex string operations.

We developed two basic approaches to bug detection and repair: 1) policy-based, and 2) differential. In the policy-based approach, input validation and sanitization policies are expressed using two regular expressions, one specifying the maximum policy (the upper bound

for the set of strings that should be allowed) and the other specifying the minimum policy (the lower bound for the set of strings that should be allowed). Using our string analysis techniques we can identify two types of errors in an input validation and sanitization function: 1) it accepts a set of strings that is not permitted by the maximum policy (i.e., it is under-constrained), or 2) it rejects a set of strings that is permitted by the minimum policy (i.e., it is over-constrained).

Our differential bug detection and repair approach does not require any policy specifications. It exploits the fact that, in web applications, developers typically perform redundant input validation and sanitization in both the client and the server-side since client-side checks can be by-passed. Using automata-based string analysis, we compare the input validation and sanitization functions extracted from the client- and server-side code, and identify and report the inconsistencies between them. Finally, we present an automated differential repair technique that can repair client and server-side code with respect to each other, or across applications in order to strengthen the validation and sanitization checks. Given a reference and a target function, our differential repair technique strengthens the validation and sanitization operations in the target function based on the reference function by automatically generating a set of patches.

We experimented with a number of real world web applications and found many bugs and vulnerabilities. Our analysis generates counter-example behaviors demonstrating the detected bugs and vulnerabilities to help the developers with the debugging process. Moreover, we automatically generate patches that can be used to mitigate the detected bugs and vulnerabilities until developers write their own patches.

Contents

Acknowledgements **v**

Abstract **xi**

List of Figures **xvi**

List of Tables **xix**

1 Introduction **1**
- 1.1 Three-Tier Architecture of Web Applications . . . 2
- 1.2 Input Validation and Sanitization in Web Applications . . . 3
- 1.3 Overview of the Modular Verification Process . . . 6
  - 1.3.1 Extraction of Input Validation and Sanitization Functions . . . 8
  - 1.3.2 Policy-Based Detection and Repair of Vulnerabilities . . . 9
  - 1.3.3 Differential Bug Detection and Repair . . . 12
  - 1.3.4 Tools and Related Work . . . 14
- 1.4 Summary of Contributions . . . 14

2 Modeling and Extraction of Sanitizer Functions **17**
- 2.1 Formal Modeling of Sanitizer Functions . . . 18
  - 2.1.1 Input Validation vs. Sanitization in practice . . . 21
- 2.2 Input Validation and Sanitization Language . . . 22
  - 2.2.1 Using IVSL to Validate and Sanitize Inputs . . . 25
- 2.3 Composing Sanitizer Functions . . . 26
- 2.4 Extracting Sanitizer Functions . . . 27
- 2.5 Dynamic Extraction of Client-Side JavaScript Code . . . 32
  - 2.5.1 How JavaScript Validates and Sanitizes Input . . . 33
  - 2.5.2 Why Dynamic Extraction? . . . 34
  - 2.5.3 Technique Overview . . . 34
  - 2.5.4 Rhino and HtmlUnit Testing Framework . . . 36

2.5.5 JavaScript Memory Management in Rhino . . . 38
2.5.6 Tracking Memory Locations . . . 41
2.5.7 Memory Tracker . . . 43
2.5.8 Code Generation . . . 44
2.6 Static Extraction of Server-Side PHP Code . . . 55
2.6.1 Dependency Graphs . . . 56
2.7 Static Extraction of Server-Side Java Code . . . 59
2.7.1 Web Deployment Descriptor . . . 59
2.7.2 Server-Side Extraction . . . 60

3 Analyzing Sanitizer Functions Using String Analysis 62
3.1 Post- and Pre-Image of a Sanitizer . . . 63
3.1.1 Post-Image . . . 63
3.1.2 Pre-Image . . . 64
3.1.3 Negative Pre-Image . . . 66
3.1.4 Negative Post-Image . . . 66
3.2 Automata-Based Symbolic String Analysis . . . 68
3.2.1 Symbolic DFA . . . 68
3.2.2 Non-Determinism in Symbolic DFA . . . 73
3.2.3 Symbolic vs. Explicit DFA . . . 75
3.2.4 Analysis Lattice and Termination. . . . 75
3.2.5 Forward Analysis . . . 78
3.2.6 Backward Analysis . . . 79
3.2.7 Negative Backward Analysis . . . 83
3.2.8 Transfer Functions . . . 86
3.3 Specialized Replace Algorithms . . . 95

4 Policy-Based Bug Detection and Repair 122
4.1 Vulnerability Detection and Repair for Server-Side PHP Input Sanitization . . . 122
4.1.1 Example . . . 124
4.1.2 Policy-Based Repair Problem . . . 125
4.1.3 Policy-Based Repair Algorithm . . . 126
4.1.4 Empirical Evaluation . . . 132
4.1.5 Analyzing and Patching Open Source Applications: . . . 136
4.2 Verifying Client-Side Input Validation Against Minimum and Maximum Policies 140
4.2.1 Minimum and Maximum Validation policies . . . 141
4.2.2 Min Max Policy Conformance Problem . . . 143
4.2.3 Min Max Policy Conformance Algorithm . . . 143
4.2.4 Empirical Evaluation . . . 146
4.2.5 Verifying Stand-Alone Input Validation Functions . . . 146
4.2.6 Verifying Input Validation in Deployed Web Applications . . . 150

5 Differential Bug Detection and Repair **154**
5.1 Discovering Client- and Server-Side Input Validation and Sanitization Inconsistencies . . . 160
5.1.1 Mapping Input Validation and Sanitization Functions at the Client- and the Server-Side . . . 160
5.1.2 Inconsistency Identification Problem . . . 162
5.1.3 Inconsistency Identification Algorithm . . . 162
5.1.4 Empirical Evaluation . . . 165
5.2 Semantic Differential Repair For Input Validation and Sanitization . . . 171
5.2.1 Extracting and Mapping Sanitizers From the Client- and/or the Server-side . . . 180
5.2.2 Differential Repair Problem . . . 181
5.2.3 Differential Repair Algorithm . . . 182
5.2.4 Empirical Evaluation . . . 192

6 Tools **195**
6.1 LIBSTRANGER . . . 195
6.2 STRANGER . . . 196
6.2.1 Architecture . . . 197
6.3 SEMREP . . . 199
6.3.1 Architecture . . . 199

7 Related Work **202**
7.1 Bug Detection in Web Applications . . . 202
7.2 String Analysis . . . 206
7.2.1 Static String Analysis . . . 206
7.2.2 Hybrid String Analysis . . . 211
7.2.3 String Constraint Solvers . . . 213
7.2.4 Relationship to Model Checking, Abstract Interpretation and Pre- and Post-Condition Computation . . . 215
7.3 Differential Analysis and Repair . . . 218

8 Conclusions **221**

Bibliography **224**

List of Figures

1.1 Interaction between a user and a three-tier architectured web application. . . . 2
1.2 Java server-side validation code snippet. 4
1.3 Malicious user can bypass the client-side validation code. 5
1.4 The percentages of vulnerabilities in web applications among all disclosed vulnerabilities in 2010-2013 [51]. 6
1.5 An example of the complexity of input validation and sanitization code. . . . 7
1.6 A server-side PHP sanitization code snippet. 8
1.7 A client-side JavaScript email validation code snippet. 11
1.8 JavaScript client-side validation code snippet that corresponds to Java server-side code in Figure 1.2. 12

2.1 An example of a JavaScript pure validator. 18
2.2 An example of a PHP pure sanitizer function. 19
2.3 An example of a PHP validating sanitizor function. 20
2.4 The abstract grammar for IVSL, the intermediate language used to represent sanitizers. 23
2.5 An example of how a sanitizer is extracted on the client-side and the server-side. 30
2.6 Example of an HtmlUnit testcase. 37
2.7 (a) JavaScript code along with (b) corresponding Rhino Virtual Machine bytecode. 38
2.8 Structure of the two Java `ScriptableObject`s that Rhino uses to store the two JavaScript nested objects `obj` and `obj.p3`. 41
2.9 Structure of bytecode that corresponds to *if/else* and *while loop* statements. . . 46
2.10 JavaScript code along with corresponding bytecode and template AST. 51
2.11 IVSL code and its AST that correspond to JavaScript and bytecode in Figure 2.10. 52
2.12 JavaScript code along with corresponding IVSL code that is extracted. 53
2.13 Inlining function calls when dynamically extracting JavaScript code. 54
2.14 A PHP sanitizer and its dependency graph. 56

3.1 Example of *post-image* (shaded areas) for a sanitizer function F_1 assuming input to be Σ^* where $\Sigma = \{a, b\}$. 64
3.2 Example of *pre-image* (the shaded area on the left) of sanitizer function F_1 given a subset of the co-domain of F_1 (shaded area on the right). 65
3.3 Example of a *negative pre-image* (the shaded area on the left) of sanitizer function F_1 which is mapped by F_1 to $\perp$ (i.e., rejected). 67
3.4 Symbolic representation of a DFA using MBDD. 70
3.5 Σ and corresponding $\Sigma_{\mathcal{B}}$ for the sample symbolic DFA along with MONA transition labels and their corresponding explicit transitions. 71
3.6 A symbolic DFA on the right with $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^2$ simulating non-determinism in the NFA on the left using 2 extra bits. 73
3.7 Core automata operations used in our analysis. 75
3.8 Definition of post- and pre-images of the two most common string functions namely *concatenation* and *replace*. 89
3.9 Example of applying POSTESCAPE to DFA M_1 where $L(M_1) = \Sigma^3$ to escape the characters `.9513.6` and `.9513.6` with character `.9513.6`. 98
3.10 Example of double escaping that happens if an escape character e on transition $(1 \xrightarrow{e} 2)$ is escaping c and being escaped by another e. 100
3.11 Example of POSTTRIMLEFT. 104
3.12 Example of PRETRIMLEFT. 107
3.13 Example of POSTTRIMRIGHT. 110
3.14 Example of PRETRIMRIGHT. 111
3.15 Example of replacing a transition on char `.9513.6` between two states S_0 and S_1 with a path on string "`foo`" while computing POSTREPLACECHAR. 113
3.16 Example of running post-image computation for `htmlspecialchars` on an input DFA M where $\mathcal{L}(M) = \Sigma^2$. 119
3.17 Time and memory performance for generic replace and optimized/specialized replace algorithms. 121

4.1 A sanitizer with an XSS vulnerability extracted from server-side code in a PHP web application. 124
4.2 Post-image of sanitizer in Figure 4.1. 128
4.3 Intersection of post-image of sanitizer in Figure 4.2 and attack pattern $\Sigma^* < \Sigma^*$. 129
4.4 Vulnerability signature DFA M_{vs} for example in Figure 4.1 given attack pattern $\Sigma^* < \Sigma^*$. The dotted line shows the mincut for this vulnerability signature DFA. 129
4.5 Input matching overhead using `stranger_match` to simulate vulnerability signature DFA. 139
4.6 An over-constrained validator that corresponds to Javascript function in Figure 1.7. 140
4.7 Maximum input validation policies. 141

4.8 Minimum input validation policies. 141

5.1 JavaScript and HTML code snippets for client-side validation. 156
5.2 Java server-side validation code snippet. 156
5.3 High-level view of differential analysis technique. 160
5.4 A small, but illustrative example, showing a target function to be repaired based on a reference function. 172
5.5 The repaired function that is generated by our differential repair algorithm for the target function shown in Figure 5.4. 173
5.6 The repaired function is the composition of three automatically generated patches and the original target function. 174
5.7 The validation patch automaton M^V for the example in Figure 5.4. The validation patch F^V blocks the strings accepted by this automaton. 185
5.8 The length patch automaton M^L for the example in Figure 5.4. The length patch F^L rejects the strings accepted by this automaton. 188
5.9 The mincut automaton M_{mc} for the example in Figure 5.4. The dotted line shows the mincut edges with the corresponding mincut alphabet $\{<\}$. 190

6.1 The architecture of 12_. 197
6.2 The architecture of 12_. 200

List of Tables

2.1 Example of translation from JavaScript and PHP string operations to IVSL code. 26

4.1 Vulnerability analysis performance for benchmarks. 134
4.2 Signature generation performance for benchmarks. 134
4.3 Minimum edge and alphabet cuts. 135
4.4 The sizes of analyzed applications. 136
4.5 XSS vulnerability analysis results. 137
4.6 SQLI vulnerability analysis results. 137
4.7 Results of our analysis on input validation functions collected from JavaScript books and tutorials. 147
4.8 Results of our analysis on deployed websites. 151

5.1 Web applications used in our empirical evaluation. 166
5.2 Relevant data on the input validation modeling step of the technique. 168
5.3 Relevant data on the input validation extraction step of the technique. 169
5.4 Data on the inconsistency identification step of the approach and overall results. 170
5.5 Number of patches generated. 193
5.6 Sanitization patch results. 193
5.7 Time and memory performance of analysis. 194

Chapter 1

Introduction

Web applications have become a crucial part of commerce, entertainment and social interaction. They are rapidly replacing desktop applications. In the near future, they are likely to play critical roles in national infrastructures such as healthcare, national security, and the power grid. At the early days of web application development, most of the application logic was implemented at the server side. The client side consisted merely of an HTML web page that was rendered on the web browser. In the case where the application needed some data from the user, an HTML form was used where the data would be sent immediately to the server side without any processing or validation. In recent years, in order to improve efficiency and usability, web applications have started to migrate many of the computational tasks to the client-side code using new JavaScript-based development frameworks such as Ajax. This makes applications more responsive by reducing the need to send a request to the web server from the user's machine and wait for the response.

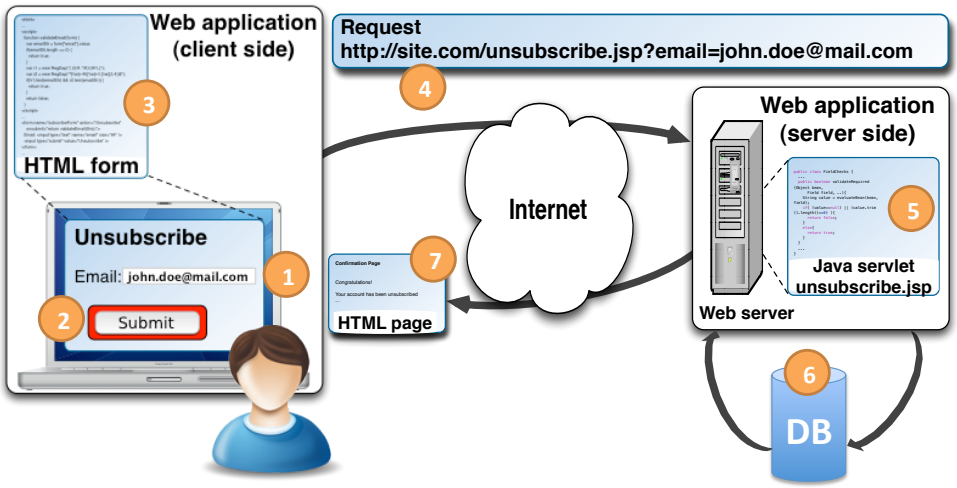

Figure 1.1: Interaction between a user and a three-tier architectured web application.

1.1 Three-Tier Architecture of Web Applications

Figure 1.1 shows a high-level view of the way web applications work by showing the typical scenario for an interaction between a user and a web application to unsubscribe from a mailing list. The web application has a three-tier architecture which consists of front-end *client-side* code (executing at the user's machine that is running the browser), back-end *server-side* code (executing at the web server) and the backend database (storing the persistent data on a separate database server). (1) First, the user opens a web page in his browser and enters his email address in the HTML form field labeled "Email". Filling out text fields in HTML forms is one of the main forms of interaction between a user and a web application. (2) Then, the user clicks the submit button to send the input email to the web application. (3) The input is checked and processed by the client-side JavaScript code and prepared to be sent to the server. (4) After that, the browser packages the input into an HTML request and sends it to the server-side of

the web application (written using Java Servlet Framework in our example). (5) The server-side logic checks the input data again, processes it and (6) then sends a query to the database to delete the user's email from the mailing list stored their. (7) Finally, the server-side replies back to the user that his email address has been removed.

A web application, such as our example, typically expects the user input to be in a certain format for many text fields (such as username, email, zip code, etc.). Since the user input can contain typing errors, or may be purposefully written (by a malicious user) to violate the expected format, the web application has to validate the user input using input validation operations such as regular expression matching. Furthermore, the web application may need to modify the input to put it in the expected format using sanitization operations such as trimming white spaces from the beginning and the end of an input or escaping problematic characters. Figure 1.2 shows an example of a validation and sanitization function written in Java to validate and sanitize email addresses in a real web application called JGOSSIP (`http://sourceforge.net/projects/jgossipforum/`). Line 5 checks that the email address value is not `null` or empty after trimming space characters. Lines 6 and 7 validate the input by matching against some regular expression.

1.2 Input Validation and Sanitization in Web Applications

Web application developers often introduce redundant input validation and sanitization code in the client and server-side code of a web application as is the case with our example

```
1 public class Validator {
2   public boolean validateEmail(Object bean, Field f, ..) {
3     String val = ValidatorUtils.getValueAsString(bean, f);
4     Perl5Util u = new Perl5Util();
5     if (!(val == null || val.trim().length == 0)) {
6       if ((!u.match("/( )|(@.*@)|(@\\.)/", val))
7         && u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/",
8                    val)) {
9         return true;
10      } else {
11        return false;
12      }
13    }
14    return true;
15  }
16 ...
17 }
```

Figure 1.2: Java server-side validation code snippet.

application. In step (3) in our example application, the entered email address gets validated and/or sanitized by the client-side code (written usually in JavaScript) that is executing on the user's machine to make sure it is in the correct format. One of the benefits of validating user input on the client-side (instead of doing it exclusively on the server-side) is that it improves usability and responsiveness of the application by preventing unnecessary communication with the server and reduces the server load at the same time. In step (5) in our example, the server-side code validates and/or sanitizes the input email address again. One of the reasons for redundant validation is that a malicious user can bypass the client-side validation as shown in Figure 1.3 by manually crafting the HTML request with malicious input.

If input validation or sanitization is not used, inputs that violate the expected format can easily cause an application to crash since the user input becomes the input parameter of the action that is executed based on the user request. Moreover, during action execution, user input can be passed as a parameter to security sensitive operations such as sending a query to the

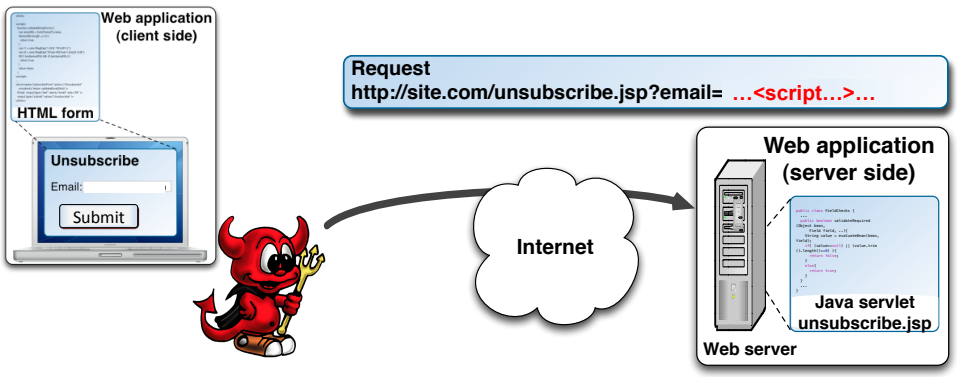

Figure 1.3: Malicious user can bypass the client-side validation code.

back-end database. In order to ensure the security of the application, the user inputs that flow into sensitive functions must be correctly validated and sanitized.

Due to global accessibility of web applications, malicious users all around the world can exploit a vulnerable application, so any existing vulnerability in a web application is likely to be exploited by some malicious user somewhere. Given the significance of this security threat, one would expect web application developers to be extremely careful in writing input validation and sanitization functions. Unfortunately, web applications are notorious for security vulnerabilities such as SQL injection and cross-site scripting (XSS) that are due to improper input validation and sanitization. In fact, according to IBM X-Force Trend and Risk Report [51] (which provides statistical information about all aspects of threats that affect Internet security), around 40% of disclosed vulnerabilities in years 2010-2014 are found in web applications (see Figure 1.4).

In this dissertation, *we introduce new techniques to automatically detect and repair bugs and vulnerabilities in input validation and sanitization code in web applications*. These tech-

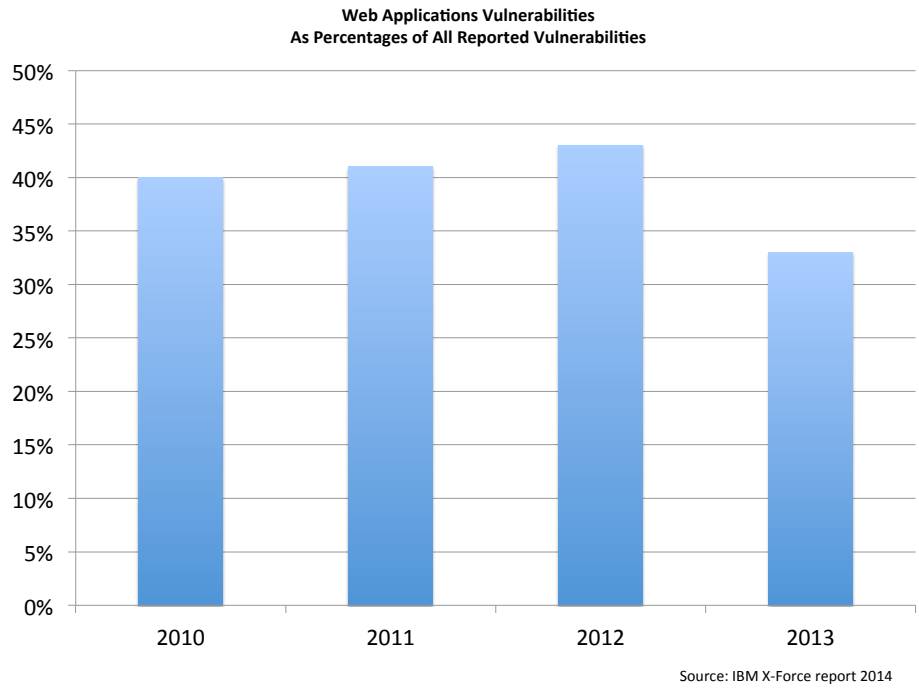

Figure 1.4: The percentages of vulnerabilities in web applications among all disclosed vulnerabilities in 2010-2013 [51].

niques are based on the automata-based symbolic static string analysis that we discuss in Chapter 3.

1.3 Overview of the Modular Verification Process

A big obstacle when analyzing input validation and sanitization code is its complexity. Figure 1.5 shows part of a typical input validation and sanitization code used to validate a form in a google website. The reader may observe that this code (1) mixes the input validation and sanitization of multiple HTML form fields at the same time, (2) mixes the actual code that does the input validation and sanitization with other parts of the program that do error reporting and

```
function validate() {
...
  switch(type) {
    case "time":
      var highlight = true;
      var default_msg = "Please enter a valid time.";
      time_pattern = /^[1-9]\:[0-5][0-9]\s*(\AM|PM|am|pm?)\s*$/;
      time_pattern2 = /^[1-1][0-2]\:[0-5][0-9]\s*(\AM|PM|am|pm?)\s*$/;
      time_pattern3 = /^[1-1][0-2]\:[0-5][0-9]\:[0-5][0-9]\s*(\AM|PM|
                        am|pm?)\s*$/;
      time_pattern4 = /^[1-9]\:[0-5][0-9]\:[0-5][0-9]\s*(\AM|PM|
                        am|pm?)\s*$/;
      if (field.value != "") {
        if (!time_pattern.test(field.value)
           && !time_pattern2.test(field.value)
           && !time_pattern3.test(field.value)
           && !time_pattern4.test(field.value)) {
             error = true;
        }
      }
      break;
    case "email":
      error = isEmailInvalid(field);
      var highlight = true;
      var default_msg = "Please enter a valid email address.";
      break;
    case "date":
      var highlight = true;
      var default_msg = "Please enter a valid date.";
      date_pattern = /^(\d{1}|\d{2})\/(\d{1}|\d{2})\/(\d{2}|\d{4})\s*$/;
      if (field.value != "")
        if (!date_pattern.test(field.value)||!isDateValid(field.value))
          error = true;
      break;
...
  if (alert_msg == "" || alert_msg == null) alert_msg = default_msg;
  if (error) {
    any_error = true;
    total_msg = total_msg + alert_msg + "|";
  }
  if (error && highlight) {
    field.setAttribute("class","error");
    field.setAttribute("className","error");    // For IE
  }
  ...
}
```

Figure 1.5: An example of the complexity of input validation and sanitization code.

```
1 <?php
2    $www = $_GET["www"];
3    $l_otherinfo = "URL";
4    $www = preg_replace( "/[^A-Za-z0-9 .-@://]/", "", $www );
5    echo $l_otherinfo . ": " . $www ;
6 ?>
```

Figure 1.6: A server-side PHP sanitization code snippet.

event handling. To deal with this complexity, we divide the problem of detecting and repairing bugs in input validation and sanitization code into three phases [2–4, 103]. In the first phase (Chapter 2), given a web application, we extract the relevant input validation and sanitization code into individual functions. In the second phase (Chapter 3), we analyze these functions using automata-based static string analysis. In the third phase (Chapters 4 and 5), we use the result from string analysis to find and repair bugs and vulnerabilities in input validation and sanitization functions.

1.3.1 Extraction of Input Validation and Sanitization Functions

In Chapter 2, we start explaining the extraction phase by formally specifying what we mean by input *validation* and *sanitization* functions. Then, we introduce a new intermediate language called Input Validation and Sanitization Language (IVSL) that allows us to have a unified framework for analyzing different input validation and sanitization functions extracted from different programming languages. After that, we explain static and dynamic techniques that we developed to extract input validation and sanitization code into an IVSL function.

1.3.2 Policy-Based Detection and Repair of Vulnerabilities

Let us give some examples of erroneous input validation and sanitization code to give a high level idea of what kind of bugs we deal with and how we deal with them. Figure 1.6 shows a simplified version of a vulnerable PHP sanitization code that is taken from a web application called MyEasyMarket [9]. The code starts with assigning the user input provided in the `$_GET` array to the `$www` variable in line 2. Then, in line 3, it assigns a string constant to the `$l_otherinfo` variable. Next, in line 4, the user input is sanitized using the `preg_replace` command. This replace command gets three arguments: the match pattern, the replace string and the target string. It finds all the substrings of the target string that match the match pattern and replaces them with the replace string. In the replace command shown in line 4, the match pattern is the regular expression `[^A-Za-z0-9 .-@://]`, the replace string is the empty string (which corresponds to deleting all the substrings that match the match pattern), and the target string is the value of the variable `$www`. After the sanitization step, the PHP code outputs the concatenation of the variable `$l_otherinfo`, the string constant `": "`, and the variable `$www`.

The replace operation in line 4 contains an error that leads to a XSS vulnerability. The error is in the match pattern of the replace operation: `[^A-Za-z0-9 .-@://]`. The goal of the programmer was to eliminate all the characters that should not appear in a URL. The programmer implements this by deleting all the characters that do not match the characters in the regular expression `[A-Za-z0-9 .-@://]`, i.e., eliminate everything other than alphanumeric characters, and the ASCII symbols `.`, `-`, `@`, `:`, and `/`. However, the regular expression

is not correct. First, there is a harmless error. The subexpression `//` can be replaced with `/` since repeating the symbol `/` twice is unnecessary. More serious error is the following: The expression `.-@` is the union of all the ASCII symbols that are between the symbol `.` and the symbol `@` in the ASCII ordering. The programmer intended to specify the union of the symbols `.`, `-`, and `@` but forgot that symbol `-` has a special meaning in regular expressions when it is enclosed with symbols `[` and `]`. The correct expression should have been `.\-@`. This error leads to a vulnerability because the symbol < (which can be used to start a script to launch a XSS attack) falls between the symbol `.` and the symbol `@` in the ASCII ordering. So, the sanitization operation fails to delete the < symbol from the input, leading to a XSS vulnerability.

Using string replace operations, such as the one in this example, to sanitize user input is common practice in web applications. However, this type of sanitization is error prone due to complex syntax and semantics of regular expressions. In the first half of Chapter 4, we discuss how to statically detect and remove common vulnerabilities such as XSS and SQLI from web applications [102–105]. Briefly, we statically approximate the output of input validation and sanitization code using static string analysis techniques from Chapter 3. Then we compare this output against some manually written security policies called *attack patterns* that characterize the set of bad and malicious strings. If we find similarities (i.e., shared values) between the two then we report a possible vulnerability. Furthermore, we generate a patch that will either block or sanitize bad inputs that may have caused the vulnerability but were missed by the original validation and/or sanitization.

```
1  function isValidEmail(emailField) {
2     var email = emailField.value.trim();
3     emailField.value = email;
5     EMAIL_REGEXP =
6          /^[a-z0-9!#$%&'*+/=?^_`{|}~-]+
7          (?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@
8          (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
9          [a-z0-9](?:[a-z0-9-]*[a-z0-9])$/;
10    if(!EMAIL_REGEXP.test(email)) {
11         alert("Please enter a correct email address.");
12         emailField.focus();
13         return false;
14    }
15    return true;
16 }
```

Figure 1.7: A client-side JavaScript email validation code snippet.

Figure 1.7 shows a JavaScript email validation function taken from a telecommunication company website (`www.stc.co.sa`). This function has a different problem than the previous one. The previous PHP function was under constrained and accepts bad inputs while this function is over constrained and rejects good inputs. In line 10, the input email address is validated against the complex regular expression in lines 5-9. The problem is that the language of this regular expression does not contain email addresses with capital letters. Although this problem is present in the client-side of the web application, it will affect the application's correctness since it will prevent some valid emails with capital letters to reach the server. To detect this type of problems, we extend the policy based verification in Chapter 4 to verify against two manually specified input policies instead of one, a *minimum* policy and a *maximum policy* [3]. If one of these two policies is violated, then we report this violation along with a counter example. We use regular expressions as the specification language for the minimum and maximum policies.

```
<html>
...
<script>
function validateEmail(form) {
  var emailStr = form["email"].value;
  if(emailStr.length == 0) {
    return true;
  }
  var r1 = new RegExp("( )|(@.*@)|(@\\.)");
  var r2 = new RegExp("^[\\w]+@([\\w]+\\.[\\w]{2,4})$");
  if(!r1.test(emailStr) && r2.test(emailStr)) {
    return true;
  }
  return false;
}
</script>
...
<form name="subscribeForm" action="/Unsubscribe"
    onsubmit="return validateEmail(this);">
  Email: <input type="text" name="email" size="64" />
  <input type="submit" value="Unsubscribe" />
</form>
...
</html>
```

Figure 1.8: JavaScript client-side validation code snippet that corresponds to Java server-side code in Figure 1.2.

1.3.3 Differential Bug Detection and Repair

Effectiveness of policy-based bug detection and repair depends on the correctness and precision of the written policies in characterizing good and bad string values. It is often possible, for instance, to encode well-known attacks into security policies (in the form of attack patterns) and write down policies for common input fields such as email address and zip code. In other cases, however, the checks to be performed on the inputs are specific to the functionality of the web application, and the input validation may be tightly coupled with and dependent on the application logic. Because they are specific to individual applications, there are no pre-specified policies that can be used to assess these types of input checks. In these cases, to make sure that

the input validation is adequate, it would be necessary to specify a different policy for each different application, which is a tedious and error-prone task.

In Chapter 5, we address this problem by presenting a semantic differential analysis and repair technique that eliminates the need to write manual specifications [2, 4]. As we have mentioned before, developers of web applications write redundant input validation and sanitization code both on the client and server-side of a web application. Figure 1.8 shows the JavaScript client-side input validation function in JGOSSIP web application that corresponds to the Java server-side one in Figure 1.2. Although both functions validate the same HTML input field that is used to input email address, they return different results for the same input. On one hand, the client-side validation function rejects a sequence of one or more white space characters (e.g., "␣␣" where ␣ means a white space character), for which the condition on line 6 evaluates to false and the regular expression check on line 11 fails, thereby resulting in the function returning false. However, for the same input, the second condition on line 5 of the server-side validation function (Figure 1.2) evaluates to false, due to the `trim` function call, and the string is therefore accepted by the server. Accepting white spaces as email addresses by the server might lead to failures. In Chapter 5 we show how to detect and repair this bug by automatically finding the difference between the client and server-side functions above and then repairing this difference.

1.3.4 Tools and Related Work

In Chapter 6 we present a string analysis library called LIBSTRANGER [103] that implements our automata-based symbolic string analysis algorithms along with two web application's verification tools called STRANGER and SEMREP. STRANGER [103] can be used to statically and automatically find and eliminate string-related security vulnerabilities in PHP applications. SEMREP is a language agnostic semantic differential repair tool that, given two input validation and sanitization functions a reference and a target, automatically repairs the target against the reference by producing a set of patch functions.

Finally, in Chapter 7 we give an overview of other work in the areas of strings analysis and automated program repair for input validation and sanitization code.

1.4 Summary of Contributions

The main contributions of this dissertation can be summarized as follows:

1. We present a new modular verification framework for analyzing and repairing input validation and sanitization code in web applications. The framework simplifies the verification process by separating the extraction, modeling and verification of input validation and sanitization functions.

2. We present a new formal specification for string related input validation and sanitization functions that are used in web applications.

3. We present a new Input Validation and sSanitization Language called IVSL that is capable of capturing the semantics of string related input validation and sanitization operations in different programming languages.

4. We present a new dynamic technique to extract input validation and sanitization functions from JavaScript into IVSL.

5. We present an automata-based symbolic string analysis framework for IVSL programs to 1) conservatively compute all possible output values assuming any input value, 2) conservatively compute all possible input values given a preferred set of output values and 3) conservatively compute the set of possibly rejected inputs.

6. We present a set of novel language-based algorithms to model string-related branch conditions in input validation and sanitization functions.

7. We present a set of novel language-based replace algorithms to efficiently and precisely model specialized string replace operations that are commonly used in input sanitization in web applications.

8. We present new policy-based techniques to automatically discover, generate counter examples and repair bugs and vulnerabilities in web applications.

9. We present a new semantic differential analysis and repair algorithm that is capable of automatically detecting and repairing differences between client-client, client-server and/or server-server input validation and sanitization code in a web application.

10. We extend LIBSTRANGER which is an efficient symbolic automata manipulation library and present two automatic bug detection and repair tools for web applications called STRANGER and SEMREP that are available online along with their source code.

Chapter 2

Modeling and Extraction of Sanitizer Functions

In this chapter we explain the first phase of our verification process which is the extraction phase. We start by formally specifying what we mean by input *validation* and *sanitization* functions. Then, we introduce a new intermediate language called Input Validation and Sanitization Language (IVSL) that allows us to have a unified language-agnostic framework for analyzing different input validation and sanitization functions extracted from different programming languages. After that, we show static and dynamic techniques that we developed to extract input validation and sanitization code, that is written in JavaScript, PHP and Java, from web applications into IVSL functions.

```
1 function validateEmail(inputField, helpText){
2    if (!/.+/.test(inputField.value)) {
3        return false;
4    }
5    else {
6      if (
        !/^[a-zA-Z0-9\.-_\+]+@[a-zA-Z0-9-]+(\.[a-zA-Z0-9]
        {2,3})+$/.test(inputField.value)) {
7        return false;
8      }
9      else {
10       return true;
11     }
12   }
13 }
```

Figure 2.1: An example of a JavaScript pure validator.

2.1 Formal Modeling of Sanitizer Functions

Input validation and sanitization operations in web applications can be characterized using three types of functions: 1) *pure validator*, 2) *pure sanitizer* and 3) *validating-sanitizer* functions [2]. Each of these three types of functions can further be characterized as either a single-input or multi-input functions. We first define the single-input version of each of the three function types then generalize the definition to multi-input functions.

A single-input pure validator is a total function:

$$F_v : \Sigma^* \rightarrow \{\bot, \top\}$$

that takes a string $s \in \Sigma^*$ and returns either $\top$ indicating that the string is valid and should be accepted or $\bot$ indicating the string is not valid and should be rejected.

A multi-input pure validator is a total function:

$$F_v : (\Sigma^*)^n \rightarrow \{\bot, \top\}$$

```
1  function escape($x){
2    $x = preg_replace('/"/', '\"', $x);
3    return $x;
4  }
```

Figure 2.2: An example of a PHP pure sanitizer function.

that takes a tuple of strings $(s_1, s_2, \ldots, s_n) \in (\Sigma^*)^n$ and returns either $\top$ indicating that all these strings are valid and should be accepted or $\bot$ indicating one of the strings s_i is not valid and hence the tuple $(s_1, s_2, \ldots, s_n)$ should be rejected.

Note that, a pure validator does not change the value of the input string, it either accepts or rejects it as it is. Figure 2.1 shows a JavaScript single-input pure validator that validates email addresses. The function makes sure that the email address is not empty (line 2) and that it matches the regular expression for valid email addresses (line 6). If these two conditions are satisfied then it accepts the input by returning true (line 10) otherwise it rejects it by returning false (lines 3,7). Notice that the email address value is not modified by the function.

A single-input pure sanitizer is a total function:

$$F_s : \Sigma^* \rightarrow \Sigma^*$$

that maps an input string $s \in \Sigma^*$ to an output string $s' \in \Sigma^*$.

A multi-input pure sanitizer is a total function:

$$F_s : (\Sigma^*)^n \rightarrow \Sigma^*$$

that maps an input tuple of strings $(s_1, s_2, \ldots, s_n) \in (\Sigma^*)^n$ to an output string $s' \in \Sigma^*$.

Note that, a pure sanitizer does not reject any input string, however, it may modify some of the input strings. Figure 2.2 shows a PHP single-input pure sanitizer function. The function

```
1  function reference_function($x){
2    if (strlen($x) > 4)
3      exit();
4    else {
5      $x = preg_replace('/</', '', $x);
6      if ($x == '')
7        exit();
8      else
9        return $x;
10   }
11 }
```

Figure 2.3: An example of a PHP validating sanitizor function.

modifies its input by escaping each " character with a \ character (line 2) then it returns the new modified value. Notice that the function does not reject any invalid input that contains the character ".

A single-input validating-sanitizer is a function:

$$F_{vs} : \Sigma^* \rightarrow \{\perp\} \cup \Sigma^*$$

that takes an input string $s \in \Sigma^*$ and either returns $\perp$ indicating that s is invalid or maps s to output string $s' \in \Sigma^*$.

A multi-input validating-sanitizer is a function:

$$F_{vs} : (\Sigma^*)^n \rightarrow \{\perp\} \cup \Sigma^*$$

that takes a tuple of strings $(s_1, s_2, \ldots, s_n) \in (\Sigma^*)^n$ and either returns $\perp$ indicating that one or more of the string values s_i is invalid or maps $(s_1, s_2, \ldots, s_n)$ to output string $s' \in \Sigma^*$ by modifying and/or combining one or more of the components s_i of the input tuple.

Note that, a validating-sanitizer may reject some inputs and modify some others. For the rest of the dissertation we call a validating-sanitizer function a sanitizer for short. We model

all input validation and sanitization operations in web applications as sanitizers. Note that, one can simulate a pure validator using a sanitizer: If an input is rejected by the validator, it is rejected by the sanitizer and if it is accepted by the validator it is returned without modification by the sanitizer. Obviously, any pure sanitizer is also a sanitizer that never rejects an input. Hence, by just focusing on sanitizers we are able to analyze all three types of behavior.

Figure 2.3 shows an example of a PHP single input validating-sanitizer function. The function validates the length of the input on line 2. Then, it sanitizes the input by deleting the character < on line 5. Finally, the function validates the result again to make sure it is not empty on line 6. This shows how input validation and sanitization operations are mixed together in a validating-sanitizer.

2.1.1 Input Validation vs. Sanitization in practice

Some examples of validation operations in practice include functions such as PHP function `preg_match`, JavaScript function `indexof` and Java function `contains` which are utilized usually through branch conditions. Examples of sanitization operations are JavaScript and Java `replace` functions and PHP functions `trim`, `addslashes` and `htmlspecialchars`.

Based on our observation of input validation and sanitization in web applications, we noticed a relationship between data read and write operations and the usage of either input validation or input sanitization. In case of a data read operation that will not change the backend database, input sanitization can be used to convert malicious user inputs into benign ones.

This should not affect the database since these sanitized values will be only used to query the database but not to change its state.

In case of a data write operation that will change the backend database, the use of validation vs. sanitization depends on weather or not the input value is used later to query the database. If the value is going to be used later to query the database, then input validation is used to make sure that the input is in a correct format that matches the format of the data type expected by the database. For example, when signing up in a website, input fields such as *username* are usually validated only and not sanitized. The reason is that, when a sanitizer modifies a *username* value during signup without the user knowledge, the user may not be able to use the original value s/he signed up with to login. Preventing attack strings that may come through these fields is done by validation operations. On the other hand, input fields for contents, such as messages in a forum, are only sanitized even when they are entered into the database since they are not used to query the database later on.

2.2 Input Validation and Sanitization Language

In this section we will define an intermediate representation called Input Validation and Sanitization Language (IVSL). This language is used to represent sanitizer functions—as defined in 2.1—which are then analyzed by our string analysis algorithms in 3 in a programming language independent way. Before analyzing input validation and sanitization code for a given input field(s) in a web application, we first extract such code as an IVSL program.

Sanitizer	→	**sanitizer** (*Var [, Var]∗*) { *Block* }
Var	→	<identifier>
Block	→	*Stmt [; Stmt]∗*
Stmt	→	*Var* := *Exp* | **return** *Var* | **reject**
	|	**if** (*Pred*) { *Block* }*[* ***else*** *{ Block }]*
	|	**while** (*Pred*) {*Block* }
Exp	→	"<string-literal>" | *Var* | ⋆
	|	*StringFunc*
Pred	→	*Pred* && *Pred* | *Pred* || *Pred* | !*Pred* | ⋆ | (*Pred*) |
	|	*Var RelOp* "<string-literal >"
	|	*Var* **matches** *RegExp*
	|	*StringFunc RelOp* "<string-literal >"
	|	*IntFunc RelOp* <integer-literal>
RelOp	→	< | <= | > | >= | == | !=
StringFunc	→	replace(*RegExp*, ("<string-literal>" | *Var*), *Var*)
	|	concat(("<string-literal>" | *Var*), ("<string-literal>" | *Var*))
	|	trim(*Var*, '<char >'*[*, '*<char>*'*]*∗)
	|	addslashes(*Var*)
	|	htmlspecialchars(*Var*)
	|	substring(*Var*, *[<integer-literal >]*, *[<integer-literal >]*)
IntFunc	→	length(*Var*) *RelOp* <integer-literal>
	|	indexof(*Var*, "<string-literal>"*[*, "*<string-literal >"]*)
RegExp	→	/*[^] UnionExp [*$*]*/
UnionExp	→	*InterExp* | *UnionExp*
	|	*InterExp*
InterExp	→	*ConcatExp* & *InterExp*
	|	*ConcatExp*
ConcatExp	→	*RepeatExp ConcatExp*
	|	*RepeatExp*
RepeatExp	→	*RepeatExp* ? | *RepeatExp* ∗ | *RepeatExp* +
	|	*RepeatExp* { <integer-literal >*[*, *<integer-literal >]* }
	|	*ComplExp*
ComplExp	→	~ *ComplExp*
	|	*CharClassExp*
CharClassExp	→	[*CharClasses*] | [^*CharClasses*]
	|	*SimpleExp*
CharClasses	→	*CharClass CharClasses* | *CharClass*
CharClass	→	<char> − <char> | <char>
SimpleExp	→	<char> | . | (*UnionExp*) | ϵ

Figure 2.4: The abstract grammar for IVSL, the intermediate language used to represent sanitizers.

Figure 2.4 shows the syntax for IVSL. Keywords and operators are written in **bold**, non-terminals in *italic*, terminals are surrounded by < and >, and the `typewriter` font is used for the built-in functions.

An IVSL program has only one single main function called **sanitizer** which represents one single-input or multi-input sanitizer function as defined in 2.1. **sanitizer** is a function that takes one or more string variables as input and either rejects or returns a string value as output. In IVSL, variables can be declared and defined simply by assigning them values. Only string variables are allowed and the ASCII encoding is the encoding that is currently supported.

<string-literal> represents a string literal (i.e., a string constant) where characters `"` and `\` should be escaped properly using character `\`. <char> represents a single ASCII character constant that is properly escaped depending on the context. If it appears outside a regular expression then only `'` and `\` should be escaped using `\`. If it appears inside a regular expression then, in addition to `'` and `\`, all regular expression reserved characters such as `/` and `?` should also be escaped. <integer-literal> represents an integer constant number along with the `-` sign if the number is negative. Integer literals are allowed only as parameters to functions, in regular expressions to allow for repetition or in predicates to represent variable length or indices within a string variable or value. Syntax for variable <identifier> follows rules for PHP identifiers.

The language allows conditional statements, loops and assignment statements with string operations. Assigning a variable $\star$ represents assignment of an arbitrary string value $s \in \Sigma^*$. This allows for translation into IVSL from other languages when right hand side expressions of non-string type are present. The operator **`matches`** is the language membership operator

which returns *true* if a variable string value is an element in the regular language defined by the regular expression. The comparison operators such as **<** and **!=** refer to lexicographical ordering when applied to string expressions. We use $\star$ to indicate non-deterministic branch conditions. Since we only allow string expressions in the language, $\star$ can be used to represent non-string predicates such as predicates on boolean or integer expressions. **matches** and comparison operators have the highest precedence followed by the parentheses then the logical operators.

The language does not allow user-defined functions. It provides two types of built-in functions: (1) string functions which return string values and (2) integer functions which return integer values. There are three core built-in functions which are `concat, replace` and `length`. These functions can be used to model a wide range of string manipulation operations in different programming languages. Table 2.1 shows some examples for translating some PHP and JavaScript string operations into IVSL code. Notice that the translation for the same operation may differ depending on the context where this operation has been used. In addition, the language provides a number of specialized built-in functions which are functions that allow for more precise modeling of builtin library string functions in PHP, JavaScript and Java.

2.2.1 Using IVSL to Validate and Sanitize Inputs

An IVSL program has only one *accepting* sink which is the `return` *`Var`* which returns the value of the string variable *Var*. The input string(s) are validated using branch conditions

Lang.	Original Operation	IVSL Code
JS	`if (v1.match(/foo/))`	`if(v1 matches /foo/)`
JS	`v2 = v1.match(/bar/)`	`v2 = ⋆`
JS	`if (/foo/.test(v1))`	`if (v1 matches /foo/)`
JS	`v2 = /bar/.test(v1)`	`v2 = ⋆`
PHP	`$v2 = nl2br($v1)`	`v2 = replace(/\\n/," ", v1);` `v2 = replace(/\\r/, " ", v2);` `v2 = replace(/\\r\\n/, " ", v2);` `v2 = replace(/\\n\\r/, " ", v2);`

Table 2.1: Example of translation from JavaScript and PHP string operations to IVSL code.

that test if a set of validation constraints are satisfied. For example, all string values for variable `s` of length greater than or equal to 10 will be filtered by the following branch condition: `length(s) < 10`. If a string value is not valid, then it will be rejected by executing the `reject` statement which halts the execution and exits the program. I.e., the `reject` statement corresponds to the `exit()` statement in PHP. Unlike `return` *Var*, we allow multiple `reject` statements since a string may get rejected based on many validation constraints.

Input sanitization is carried out either through core string manipulation operations `concat` and `replace` or through specific operations such as `trim`, `addslashes` and `htmlspecialchars`.

2.3 Composing Sanitizer Functions

Sanitizer functions can be composed together to produce a new sanitizer function. Here we will consider the composition of single-input sanitizers only. We formally define the sanitizer

composition as follows: given two single-input sanitizer functions F_1 and F_2, their composition, $F_1 \circ F_2 : \Sigma^* \to \Sigma^* \cup \{\bot\}$, is a sanitizer function defined as:

$$F_1 \circ F_2(x) = \begin{cases} \bot & \text{if } F_2 = \bot \\ F_1(F_2(x)) & \text{if } F_2(x) \neq \bot \end{cases}$$

In IVSL, two single-input sanitizers F_1 and F_2 are composed as $F_1 \circ F_2$ by inlining the first one F_1 into the second one F_2 which guarantees that the first one is going to run after the second one. The inlining is done as following:

- Replace the **return** *Var* statement in the second sanitizer F_2 with all the statements from the first sanitizer F_1.
- Renaming the input variable in the first sanitizer F_1 with the name of the output variable in the second sanitizer F_2.
- Renaming all other variables in the first sanitizer F_1 in a way that resolves any naming conflicts with variables in the second sanitizer F_2.

2.4 Extracting Sanitizer Functions

Input validation and sanitization is used ubiquitously in all types of computational tasks. In this dissertation, *we limit the scope of our verification techniques to the domain of web applications*. The first step in the verification process is to extract the input validation and

sanitization code in a given web application that we want to analyze and repair into an IVSL sanitizer function. Web applications receive their input through HTML form input fields (we will refer to them from now on as input fields) where each input field represents an input variable. The values entered into these fields are validated and sanitized by the JavaScript code on the client-side before they are sent in an HTTP request to the server-side where they get validated and sanitized again.

Client-side checks are mainly introduced for performance reasons, as they can save one network round-trip and the additional server-side processing that would be incurred when invalid input is sent to and subsequently rejected by the web application. Therefore, to improve the user experience and provide instant feedback, many web applications validate inputs at the client side before making the actual request to the server. On the other hand, since client-side validation can often be circumvented by malicious users, the server cannot trust the inputs coming from the client side, and all input checks performed on the client side must be repeated on the server side before user input is processed and possibly passed to security sensitive functions. In this dissertation, *we consider input validation and sanitization both on the client-side JavaScript code and the server-side PHP or Java code*.

Extraction phase consists of three steps:

- Finding sources where input values come from and the corresponding sinks where output values are sent out.

- Extracting input validation and sanitization code that flow from sources to sinks into IVSL sanitizer functions.

- Mapping two IVSL sanitizer functions to each other in the case of differential analysis.

We will describe how first and second steps are carried out for each language that we deal with while leaving the discussion of the third step to Chapter 5.

The first step in extraction of input validation and sanitization code is to locate a set of data *sources* and the corresponding *sinks*. A data *source* is the source where web application's input data come from. A *sink* is an output function that sends data from the web application to a database (through an SQL query) or to a user of the web application (through an HTTP response). A web application works by receiving input values from one or more sources, processing these values, then outputting through one of the sinks. Based on this view of how a web application works, we say that input values in a web application flow from one or more sources into a sink. In our verification of input validation and sanitization in web applications, we extract and verify all input validation and sanitization operations that are applied on the input values while they are flowing from one or more sources into a sink.

However, due to having input validation and sanitization both on the client- and the server-side of a web application, we need to locate sources and corresponding sinks both on the client-side and on the server-side. On one hand, sources on the client-side are the same as the sources of the web application itself and sinks on the server-side are the same as the sinks of the web application itself. On the other hand, sinks on the client-side are *intermediate* sinks

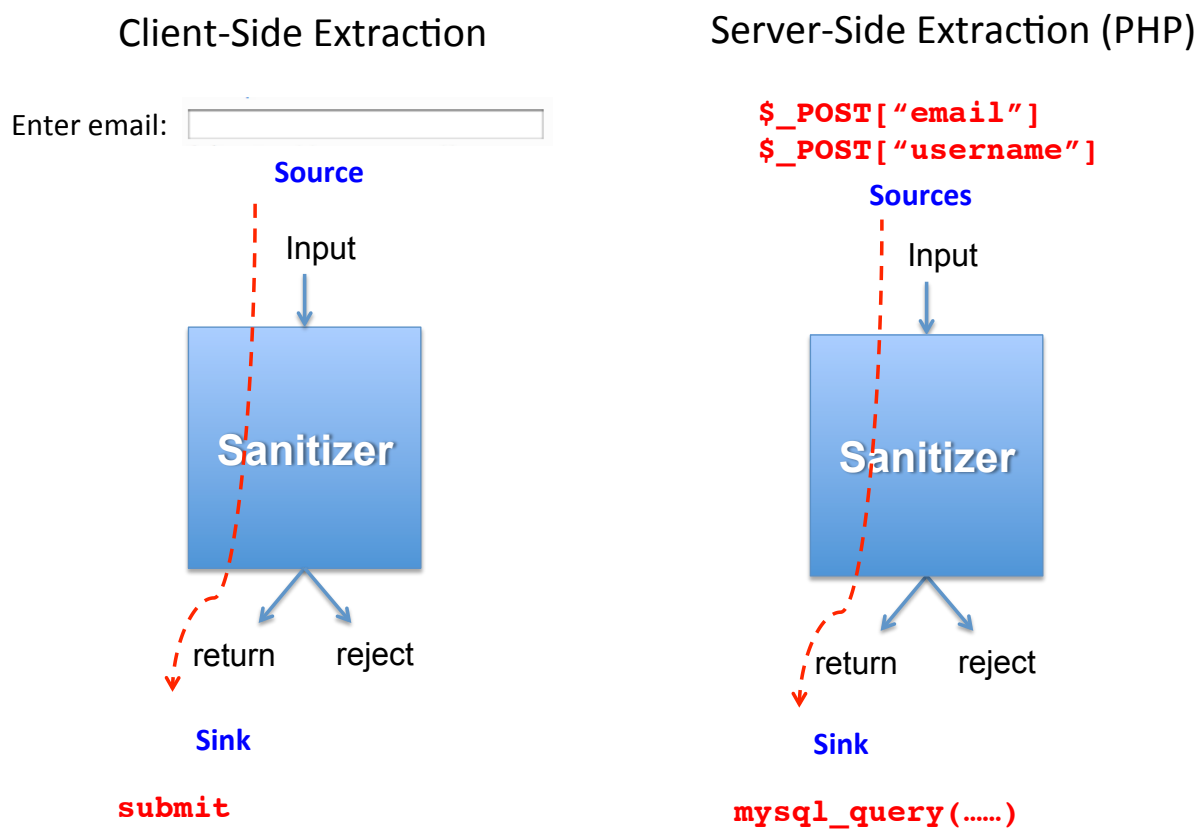

Figure 2.5: An example of how a sanitizer is extracted on the client-side and the server-side.

that send output to the server-side and sources on the server-side are *intermediate* sources that receive input from client-side. In other words, each *intermediate* sink on the client-side outputs (i.e., sends) one or more input values to the server-side which will result in one or more *intermediate* sources on the server-side. Based on this observation, we (1) first locate sources and corresponding sinks on the client-side. Then (2) we extract input validation and sanitization operations between these sources and sinks. Then (3) we locate sources and corresponding sinks on the server-side. Finally, (4) we extract input validation and sanitization operations between server-side sources and sinks.

On client-side, input values flow from input fields (after a user submits a form) to the HTTP request that is generated and sent to the server. The HTTP request is generated and sent either automatically by the browser (when a `submit` event is received) or manually by the JavaScript code (through `XMLHttpRequest.send()`). We only extract and verify validation and sanitization operations on the client-side that are carried out during the period between submitting the form by the user and sending the HTTP request[1]. We extract one single-input sanitizer function per each pair of an input field and a sink. The extracted function includes all the validation and sanitization operations that are applied on that particular field before its value reaches the sink and sent to the server. Notice that `submit` is not an actual sink (i.e., an output function) but rather a browser/JavaScript DOM event. In this case, we extract operations that are executed by the JavaScript event handler of the `submit` event. On the left side of Figure 2.5 we show how a sanitizer function is extracted from JavaScript code for an *email* input field.

On the server-side code, input values flow from HTTP request into output functions. Input values in an HTTP request are received and stored (on the server-side) in elements of PHP arrays `$_POST` and `$_GET` and fields of Java `HttpServletRequest` object. Examples of output functions are functions that send queries to database such as PHP function `mysql_query` and functions that send HTML response to the user such as PHP's `echo`[2] and Java method `HttpServletResponse.getWriter().println()`. We extract one single-input sanitizer func-

[1] We restricted our work in this dissertation to these validation and sanitization scenarios due to limitations of our extraction techniques. When combined with appropriate extraction techniques, our input validation and sanitization language, along with our verification techniques that we introduce here, can be used to verify and repair any string-related validation and sanitization code.

[2] Although `echo` looks like a function, in fact it is a construct in PHP language.

tion per each pair of an input in an HTTP request and a sink. The extracted function includes all the validation and sanitization operations that are applied on the value of that particular input before it reaches the sink.

There are some verification tasks where we need to consider validation and sanitization operations that are applied to all inputs in an HTTP request at the same time. An example of such verification task is looking for XSS and SQLI vulnerabilities (Chapter 4). In this case, we need to consider, simultaneously, how values from multiple inputs in an HTTP request (i.e., multiple sources) flow into a single output (i.e., a sink). To do this, we extract one multi-input sanitizer function per each pair of one or more inputs in an HTTP request and a sink. The sanitizer function is extracted such that it characterizes validation and sanitization operations that are used to validate all sources that flow into this sink on all execution paths between these sources and the sink. On the right side of Figure 2.5 we show how a multi-input sanitizer function is extracted from PHP code for two input values in an HTTP request and a sink.

2.5 Dynamic Extraction of Client-Side JavaScript Code

We use dynamic analysis technique to extract sanitizer functions from JavaScript [3]. We extract one sanitizer for a given HTML form input field so the output of the extraction technique is a single-input sanitizer. We start with a brief discussion of validation and sanitization of HTML forms using JavaScript, then we explain the extraction technique that we developed.

2.5.1 How JavaScript Validates and Sanitizes Input

The first step in form validation and sanitization with JavaScript is to register an event handler for some event of the input form or its fields. The event handler is then used to call the JavaScript validation code for some or all of the form fields. Based on the result returned by this validation code, either the form will be submitted or error messages will be shown to the user. Submission of the form data is done either by the browser itself or a JavaScript issued XHR (`XMLHttpRequest`) request when Ajax is used. The default event for handling form validation code is `onsubmit` event of the form itself. In the basic case, the browser will execute the `onsubmit` event handler (if found) when a user tries to submit an HTML form by clicking on an HTML element of type `submit`. If the handler returns true (or if there is no handler for `onsubmit`) then the browser will submit the form data using an ordinary HTTP get/post request. In this approach all the validation code goes inside the `onsubmit` handler and the functions it calls.

In websites that use Ajax and XHR to submit the forms, the situation is different. First of all, the `onsubmit` handler should return false when the element used to submit the form is of type `submit` (so that the browser does not submit the form itself). Furthermore, since the form will be submitted from within the JavaScript code, the element the user is supposed to click to submit the form does not have to be of type `submit`, and there is a large number of events besides `onsubmit` that can be linked to form submission such as `onclick`, `onmousedown`, and `onmouseup`. Finally, due to the capturing and bubbling of DOM events, it is possible to do the validation in an event handler for one of the events of one of the ancestors of the element

used to submit the form. This happens especially when the area the user clicks on to submit the form consists of multiple elements overlaid on top of each other.

2.5.2 Why Dynamic Extraction?

It is not feasible to statically find and extract the code that does the client-side input validation with acceptable precision. First, it is difficult to find the event handlers that contain the form validation code due to the variety and complexity of the input validation process discussed above. Even in the basic case, where the `onsubmit` event handler is used, sometimes this event handler is registered dynamically from within the JavaScript code that loads the page instead of being statically linked in the HTML code of the webpage. Second, even if we succeed in statically locating and extracting the event handling code, the code itself is large and full of event handling, error handling and error message rendering functions which are hard to separate statically. Furthermore, the validation code contains all the validation functions for all form fields mixed together instead of having one function per input field. Third, JavaScript is notoriously difficult to analyze statically due to its highly dynamic and loosely-typed nature [78]. Due to these reasons, we use a dynamic extraction technique in which we trade off soundness to gain acceptable level of precision.

2.5.3 Technique Overview

We use semi-automatic crawling to find input fields and extract their validation and sanitization code. Given a starting page and a seed of URLs (to help with URLs that are hard to

reach automatically within an application), the crawling process tries to find as many pages as possible. If it hits a webpage with a form, it tries to fill out the form using a set of pre specified profiles. A profile is a map from a set of keys of type string to a set of string values. To fill out a field in a form, the crawling process tries to match the name of the field to one of the keys in the profile. We also designed the crawling such that it can take a list that maps certain URLs to form profiles to get more precision especially when filling out usernames and passwords. The crawling process is statefull, meaning that it keeps the session when it logs into an application and uses the session handler to crawl further through the application. The crawler is built on top of HtmlUnit web testing framework and Rhino JavaScript interpreter which we discuss later in this section.

During crawling, we use dynamic extraction technique in which we handle a single target input field at a time. We only consider input fields that hold string values. In the validation and sanitization code for a target input field i, we identify all and only the statements that operate on i, directly or indirectly, such as string manipulation operations on i and conditional statements affected by i's value. The rest of the code is disregarded because it is irrelevant for the validation and sanitization of i.

Specifically, we (1) execute all of the validation and sanitization code associated with i and (2) extract statements that operate on i on the fly by instrumenting the execution. The statements that operate on i are extracted into an IVSL sanitizer function. We perform these two steps (1) and (2) several times using different values for i chosen from a pool of representative values, generated using heuristics that are based on the type of the input field. We instrument

the execution using a modified version of the JavaScript interpreter Rhino. In particular, the modified interpreter converts all accesses to objects and arrays to accesses to specific memory locations (as we explain later), which avoids imprecision due to the use of objects, arrays, and aliasing. During the dynamic extraction, we handle internal function calls by inlining the code of the callees,[3] while external calls are either treated as uninterpreted functions or mapped to an IVSL built-in function or to a series of function calls to `replace`, `concat` and/or `length` functions. At the end of extraction, we get different sanitizer functions that result from different runs. We combine these functions by analyzing each one of them separately using automata-based string analysis (as described in Chapter 3) and then combining their analysis result using the automata *union* operation.

2.5.4 Rhino and HtmlUnit Testing Framework

Rhino [73] is a JavaScript interperter that is written in Java. HtmlUnit [34] is a GUI-less Java web browser that is used along with Rhino to provide a unit testing framework for web applications. Figure 2.6 shows a simple HtmlUnit testcase. The testcase first loads a web page with an Html form, fills out field *userid* in the form then submits the form.

To run JavaScript code, Rhino either: (1) transforms JavaScript code into Java bytecode then executes the bytecode using Java Virtual Machine (JVM) or (2) compiles the JavaScript code into its own bytecode and uses its own stack-based virtual machine, which we call Rhino

[3] Although inlining can be problematic in the case of recursion and in the presence of deep call graphs, validation and sanitization code tend to be simple and not affected by these issues. If this were not the case, it would always be possible to stop the inlining at a given depth and introduce approximations.

```
@Test
public void submittingForm() throws Exception {
    final WebClient webClient = new WebClient();

    // Get the first page
    final HtmlPage page1 = webClient.getPage("http://some_url");

    // Get the form that we are dealing with and within that form,
    // find the submit button and the field that we want to change.
    final HtmlForm form = page1.getFormByName("myform");

    final HtmlSubmitInput button = form.getInputByName("submitbutton");
    final HtmlTextInput textField = form.getInputByName("userid");

    // Change the value of the text field
    textField.setValueAttribute("root");

    // Now submit the form by clicking the button and get back the second page.
    final HtmlPage page2 = button.click();

    webClient.closeAllWindows();
}
```

Figure 2.6: Example of an HtmlUnit testcase.

Virtual Machine (RVM), to execute the code[4]. When instrumenting JavaScript code, we always use the second option. Figure 2.7 shows a simple JavaScript code along with the corresponding RVM bytecode. The bytecode for each function is stored as an instruction array. The numbers on the left of each instruction in the figure represent the index of the instruction in the bytecode array. Note that some instructions such as `BINDNAME` and `SETNAME` correspond to complex read and write operations that involve traversal of the prototype chain of JavaScript objects. Rhino uses an execution (call) stack where, for each JavaScript function under execution, a stack frame is used to store the instructions array along with the scope object and other runtime information related to this function. Note that this stack is different than the Rhino Virtual Machine (RVM) stack that is used to store arguments for instructions along with their results.

[4]We could not find an official documentation for RVM so the information provided here is based on our own observations while working with Rhino.

```
------------------
(a) JavaScript code
------------------

var x = "foo";
var y = x + "bar";

----------------
(b) RVM bytecode
----------------

[0] LINE : 1
[3] REG_STR_C0
[4] BINDNAME            //push value of x onto stack
[5] REG_STR_C1
[6] STRING              //push string "foo" onto stack
[7] REG_STR_C0
[8] SETNAME             //pop value of x and "foo", store "foo" into x then push value of x
[9] POP                 //pop value of x
[10] LINE : 2
[13] REG_STR_C2
[14] BINDNAME           //push value of y onto stack
[15] REG_STR_C0
[16] NAME               //push value of x onto stack
[17] REG_STR_C3
[18] STRING             //push "bar" onto stack
[19] ADD                //pop value of x and "bar", concatenate them then push result
[20] REG_STR_C2
[21] SETNAME            //pop result and value of y, store result into y then push value of y
[22] POP                //pop value of y
[23] RETURN_RESULT
```

Figure 2.7: (a) JavaScript code along with (b) corresponding Rhino Virtual Machine bytecode.

2.5.5 JavaScript Memory Management in Rhino

Our extraction technique tracks memory locations dynamically during JavaScript program execution. Tracking memory locations is done to allow our analysis to (1) extract only the statements that operate on the input field that we are interested in, (2) to deal with aliasing and (3) to give unique names to different memory locations. To understand how memory locations tracking works, we need first to explain briefly how JavaScript stores values during program execution and how Rhino implements this using Java.

There are three classes of values that a JavaScript program reads and writes during execution: (1) *primitive* values which are `String`, `Number` and `Boolean`; (2) *special* values which are `Undefined` which is a single-valued type that has one value `undefined` that is returned when reading an uninitialized variable or a non-existent object property and `Null` which is a single-valued type that has one value `null` assigned to variables and object properties that do not have a valid value of any of the other types; and (3) *composite* values which are `Array` and `Object`. A JavaScript object is a one to one mapping from a set of strings (property names) to a set of values (primitive, special and/or composite). JavaScript treats arrays as objects but with a special internal `length` property that is updated automatically by the language itself. JavaScript object model contains *intrinsic* object types such as `Object` (which is the supertype of all composite types), `Function`, `Array`, `Date`, ... etc, along with *user defined* object types. A third class of object types, called *host* objects, is added to JavaScript when it runs inside a web browser. These object types are not part of the JavaScript language specification but are provided by all modern browsers. These include the DOM tree objects (along with DOM event model) and the `Window` object that represents the browser window.

Since Rhino is implemented in Java, it has to map these data types into Java data types. (1) Rhino maps primitive types into Java types `String`, `double` and `boolean`. (2) Rhino maps JavaScript object model into a Java class hierarchy with the Java class `ScriptableObject` as its root (which correponds to the JavaScript type `Object`). HtmlUnit extends this hierarchy by providing *host* objects. (3) (I) the special single-valued JavaScript type `Undefined` is mapped to Java class `Undefined` with a singleton instance `Undefined.instance` that

represents the JavaScript value `undefined` and (II) the special single-valued type `Null` is mapped to `null` in Java.

During a JavaScript program execution, all memory reads and writes operate on either primitive values, the two special values `undefined` and `null` or object references. These are read from/written to either (1) variables or (2) object properties. JavaScript binds global variables to the global scope and local variables to the scope of the function where these variables are defined[5]. A JavaScript function's scope is stored in the function frame on the execution (call) stack. Finally, objects and their properties are stored in the heap. How does Rhino implement this? Let us first explain how Rhino maps each JavaScript object into an instance of a Java `ScriptableObject`. `ScriptableObject` stores properties of a JavaScript object in a hashtable. For each property, the hash value of the property name (which is of type `String`) is used as the key. Property value (which is either a primitive value, one of the two special values or an object reference) is stored in the *value* field of a `Slot` object. Figure 2.8 shows how Rhino stores two nested JavaScript objects into two instances of class `ScriptableObject`. For example, the value of the property `p1` of object `obj` which is the string value `foo` is stored in the Java object field `value` of the first `Slot` object that is indexed by h(`p1`) where h is the hash function and `p1` is of type `String`.

In the case of values that are stored in variables, Rhino stores these values in `Slot` objects in the hashtable of the special object `scope` (of type `ScriptableObject`) which represents the scope of some JavaScript function `func`. In this case, the hash values of the local

[5]In our extraction, we currently do not handle `with` statement.

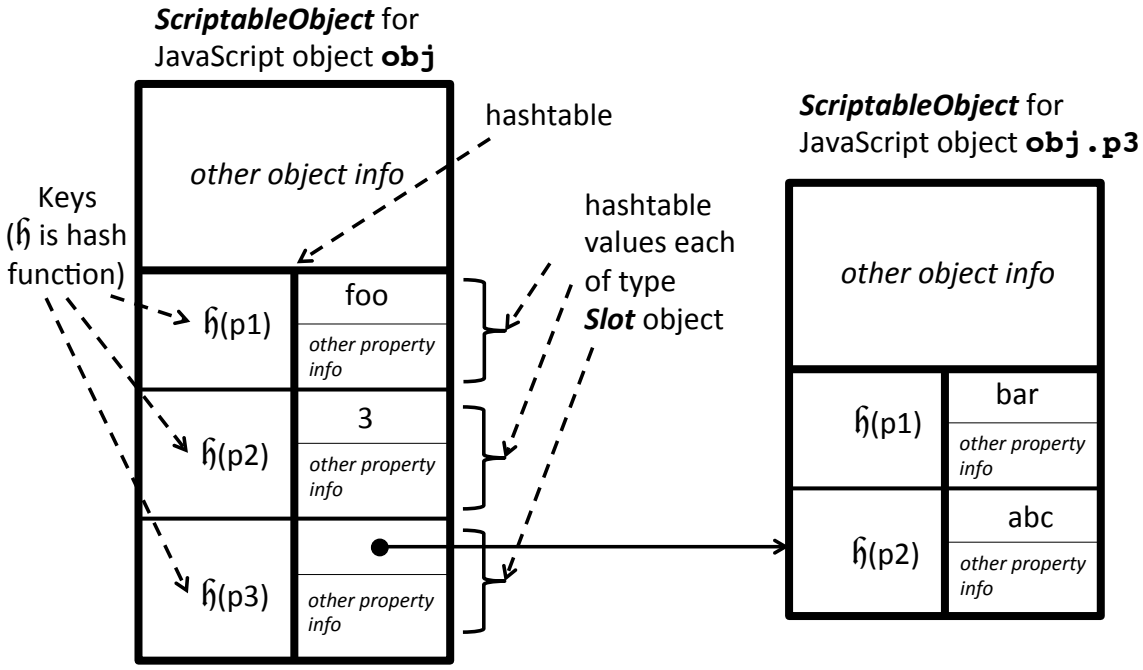

Figure 2.8: Structure of the two Java `ScriptableObject`s that Rhino uses to store the two JavaScript nested objects `obj` and `obj.p3`.

variables' names are used as keys. The `scope` object for some function `func` is stored in `func`'s stack frame in the execution stack. The top scope object (i.e., the scope object of the top code) where global variables are stored is the `Window` object.

2.5.6 Tracking Memory Locations

Unlike C language where the dynamic memory (i.e., heap) is modeled as a byte array and pointers can be used to directly access any memory location i.e., any byte in the array, in Java, dynamic memory (i.e., heap) is modeled as a graph of objects where each object is accessible

either through a reference stored in a local variable on the stack or through a reference stored in a field in another object. Since Rhino is implemented in Java we should expect the smallest unit of memory that we can track dynamically to be a Java object.

In fact, we define a *memory location* as a Java object of type `Slot` that is stored in some `ScriptableObject` and has the value of some JavaScript variable or object property. We track memory locations by keeping track of references to `Slot` objects. When tracking memory locations, we only track locations that store string values. Tracking `Slot` objects allows us to track memory locations that store primitive values (more specifically string values) since, as we discussed before, each JavaScript primitive value and object reference is stored in the `value` field of some `Slot` object that corresponds to some JavaScript variable or object property. For example, to track an HTML form input field of type text, we track the `Slot` object for the `text` property in the JavaScript DOM object that corresponds to the input field.

As we said before, in our extraction, we extract all input validation and sanitization operations that operate on a certain HTML from input field i. This means that we need to track the memory location that corresponds to this input field along with all other memory locations that are assigned the value of this input field during program execution. To do this we need first to find the memory location that stores the value of the input field itself which is done using HtmlUnit. When filling out a form using HtmlUnit (see Figure 2.6), it allows us to access DOM objects that store the values of the HTML input fields in this form. This in turn allows us to intiate the memory tracking in Rhino by feeding the `Slot` object for target HTML input

field to the MEMORYTRACKER component of the extraction framework as the initial memory location.

2.5.7 Memory Tracker

MEMORYTRACKER is the first component of our instrumentation framework. MEMORYTRACKER tracks various memory locations and stores their information. The only type of memory locations that are tracked are memory locations that store string values. Each memory location that is tracked corresponds to a `Slot` object in Java. The information that we keep for a memory location is an auto-generated name along with a reference to the corresponding `Slot` object. We use the word `mloc` along with a unique integer number to get a unique name for a memory location. MEMORYTRACKER stores memory locations' information in a hashtable indexed by Java `hashCode` of `Slot` objects (the reader should not confuse the Java `hashCode` of the `Slot` object that is used by MEMORYTRACKER and the hashcode that indexes a `Slot` object in a `ScriptableObject`). Since Rhino guarantees that two different JavaScript memory locations (i.e., two different variables or object properties) will have different `Slot` objects, indexing the hashtable with `hashCode` of `Slot` objects guarantees different names in the output generated code for different memory locations regardless of the identifier names and aliasing[6].

Temporary Memory Locations. Some arguments of bytecode instructions are temporary and do not correspond to any actual JavaScript variable or object property. As an example,

[6]We use Java default `hashCode` method and the current implementation of Java VM that we use guarantees unique `hashCode` upto 2^{32} objects

instruction `ADD` (number 19) in Figure 2.7 stores its result in a temporary location on the RVM stack which is then used by the instruction `SETNAME`. Since we generate one IVSL statement per each assign instruction (as we explain later), we need to give unique names for such *temporary memory locations* when generating the output code. The problem is further complicated by the fact that such instruction may reside in a loop which means that we need to remember the names that we give to temporary memory locations.

To solve this problem, MEMORYTRACKER keeps its own execution (call) stack frame for each JavaScript function. In this frame, it keeps an information array for memory locations where each entry in this array corresponds to an entry in the bytecode instructions array. In this array, information about memory locations accessed by each instruction is kept. If the accessed memory location corresponds to a `Slot` object, then we keep a reference to the `Slot` object information in the MEMORYTRACKER hashtable. If the accessed memory location is a temporary memory location, then we keep the assigned name for this location. This array allows us to use the same names for RVM stack temporary memory locations used by instructions when these instructions are executed multiple times within loops. Remember that these temporary memory locations do not have corresponding `Slot` objects and, thus, are not stored in the MEMORYTRACKER hashtable.

2.5.8 Code Generation

The CODEGENERATOR component of our JavaScript extraction framework is the one responsible for generating output code. When generating the output code, we generate three

types of IVSL statements: (1) assignment statement, (2) *if/else* conditional statement and (3) *while loop* statement. The question is how do we generate each of these three types of IVSL statements from the bytecode? We assume for now that we are dealing with a single JavaScript function when generating code. Later on, we explain, given the code generated for a JavaScript function, how to deal with interprocedural code generation (including passing arguments and returning results) through function inlining.

Control Statements

When extracting JavaScript code, we preserve the control structure of the JavaScript function by parsing the IF/GOTO control structure of the bytecode of the function into IVSL control statements (i.e., *if/else* or *while loop* statements). This results in an extracted code that contains conditional statements and loops as opposed to a flat execution trace. Preserving control structures such as loops allows us to avoid under-approximation that results from unrolling loops. Before explaining how this is done, we would like to point out that JavaScript does not allow arbitrary GOTO statements in its code. This allows our parsing algorithm to precisely detect the control structure of the bytecode of a single JavaScript function since the bytecode does not have arbitrary GOTO instructions. In fact, when Rhino compiles a JavaScript function code into RVM bytecode, it translates a JavaScript if/else control statement (including the condition and the body) systematically into a list of consecutive instructions where execution control (i.e., conditional jumps) is achieved by the two instructions `IFNE` and `GOTO` as shown in Figure 2.9. A similar thing is done for all loop statements i.e., while, do/while, for, and for/in

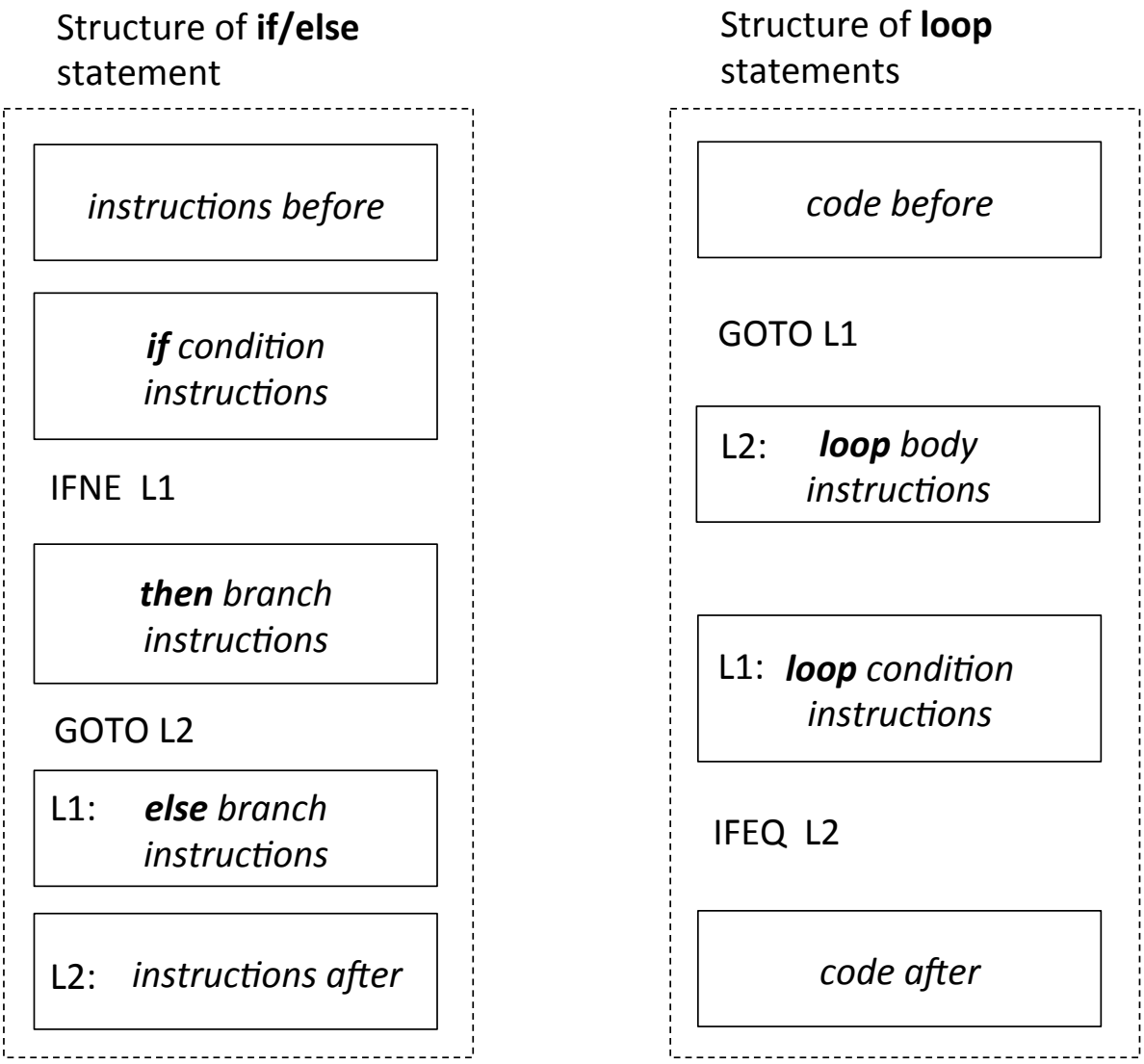

Figure 2.9: Structure of bytecode that corresponds to *if/else* and *while loop* statements.

where they are all translated into a list of consecutive instructions where execution control is achieved through the two instructions `GOTO` and `IFEQ` as shown in Figure 2.9.

Given this information, we can now proceed to the algorithm used to generate control statements. When a new JavaScript function is about to be executed, Rhino loads its instructions ar-

ray and creates a new execution stack frame. The CODEGENERATOR instruments this function loading phase by parsing the function bytecode and generating what we call *Template Abstract Syntax Tree* (Template AST) for the bytecode. A Template AST is an Abstract Syntax Tree that specifies the syntax of the control statements (i.e., the control structure) of a JavaScript function only without specifying the syntax of non-control statements. For non-control statements, *placeholder* nodes are used to reserve a place for the subtree that represents the syntax of that statement that will be generated later. Figure 2.10 shows the body of a JavaScript function along with the corresponding RVM bytecode and the Template AST. The nodes `INST0`, `INST1`, ...etc represent the placeholder nodes in the tree. To store the Template AST (along with other information), CODEGENERATOR uses its own execution (call) stack where it inserts a new frame for each new JavaScript function call and stores the Template AST (along with other information) in that frame.

The Template AST is built by parsing the IF/GOTO control structure of the bytecode into a more complex one that uses *if/else* and *while loop*. The parsing algorithm is recursive to allow for detecting nested control statements inside each other to any depth. The algorithm starts by receiving the bytecode array along with the indices of the first and last instructions that mark the region in the bytecode array that the algorithm is going to process. In the first call to the algorithm this region spans the whole bytecode array of the current JavaScript function that is being processed. For example, given the JavaScript function in Figure 2.10, the first call is going to scan and process the whole bytecode starting from instruction with index 0 (`[0] LINE : 1`) and ending with instruction with index 37 (`[37] RETURN_RESULT`).

In the consecutive calls, this region spans all instructions in the bytecode that correspond to the JavaScript control statement that is currently being processed. For example, in Figure 2.10, while scanning and analyzing the bytecode during the first call to the algorithm, when the algorithm reaches instruction 13, a recursive call is going to be initiated to process the instructions 13 to 34 that represent the while loop.

During each recursive call, a new subtree is generated as follows: (1) In the first call to the algorithm, it generates the Template AST for the current JavaScript function being processed. The Template AST will have a node of type `SANITIZER` that is also the root node along with another node of type `BLOCK` (see nodes `SANITIZER` and `BLOCK` in Figure 2.10). (2) In the consecutive recursive calls, a subtree is generated for either an IVSL if/else statement or an IVSL while loop statement. For an if/else statement, the subtree has an `IF` node as its root node and three subtrees under the `IF` node, (I) a condition subtree, (II) a **then** subtree and (III) an **else** subtree. For a loop statement, the subtree has a `WHILE` node as its root node and two subtrees under the `WHILE` node, (I) a condition subtree and (II) a loop body subtree. The subtree for either of the two types of control statements is inserted under the current parent tree by replacing the placeholder node for the first instruction corresponding to the control statement (i.e., `GOTO` for while loop statement and `IFNE` for if/else statement). The parent tree is either the AST itself, if the control statement is not nested in another control statement, or a subtree for an if/else or while loop statement in case of nested control statements (i.e., a loop nested inside a conditional statement, a loop nested inside another loop, a conditional

statement nested inside another conditional statement or a conditional statement nested inside a loop).

After the Template AST/subtree is created, the instructions within the current region of bytecode being processed are scanned and processed as follows: (1) If this is the first call to the algorithm, then scanning starts from the first instruction; (2) if this is a recursive call to process an if/ else statement then three regions are scanned, (I) the condition (which consists of the single instruction that precedes the `IFNE` instruction), (II) the **then** branch and (III) the **else** branch (see Figure 2.9); and (3) if this is a recursive call to process a loop statement then two regions are scanned: (I) the loop body and (II) the loop condition (see Figure 2.9). For example, in Figure 2.10, the algorithm during the first call is going to scan all instructions from instruction 0 to instruction 13. When it reaches instruction 13 it will initiate a recursive call to process a loop. This recursive call is going to first scan the body instructions i.e., instructions from 16 to 26 and then scan the condition instructions i.e., instructions from 29 to 33. Since there is no nested control statements, the algorithm will finish the recursive call and go back to process the instruction 37 and terminate.

While scanning the instructions, for instructions other than `GOTO`, `IFNE` and `IFEQ`, we generate a placeholder node for each instruction and insert it into the proper branch of current subtree based on the current region being processed. For example, if we are currently parsing instructions in the then branch region of an if/else statement, then we insert the node for the current instruction as a child of the then branch node of the if/else subtree. Notice that each node of an instruction is inserted as a right sibling of the node generated for the previous

instruction. When a `GOTO` instruction is found then we initiate a new recursive call to the algorithm to parse a loop statement. On the other hand, when an `IFNE` instruction if found, a new recursive call to the algorithm is initiated to parse an if/else statement.[7]

For each non-control instruction, the template AST keeps a placeholder node which will be replaced by the instruction's corresponding node or subtree after its execution. For instructions that are not executed or which are not relevant (not an assign or control instruction), their nodes will be removed at the end before the final IVSL code is generated from the AST. Figure 2.11 shows the AST after execution of the function along with generated IVSL code.

Assignment Statement

An IVSL assignment statement is generated per each assign bytecode instruction, An *assign* bytecode instruction is one that assigns a right hand side expression to a left hand side variable. An example of such instructions in Figure 2.7 are `SETNAME` (number 8) and `ADD` (number 19). There is a special type of assign instructions that may produce multiple IVSL statements since they correspond to multiple assignments. For example, `CALL` instruction, which is used to execute a function call, assigns zero or more arguments to zero or more parameters.

Dealing with Aliasing

Due to aliasing, an instruction may access different memory locations (i.e., object properties) during the same execution. For example, line 9 in JavaScript code in Figure 2.12 writes

[7] Currently we do not handle continue and break statements.

```
var x = "foo";
while (x == "foo")
    x = "bar";


---------


[0]   LINE : 1
[3]   REG_STR_C0
[4]   BINDNAME
[5]   REG_STR_C1
[6]   STRING
[7]   REG_STR_C0
[8]   SETNAME
[9]   POP
[10]  LINE : 2
[13]  GOTO 29
[16]  LINE : 3
[19]  REG_STR_C0
[20]  BINDNAME
[21]  REG_STR_C2
[22]  STRING
[23]  REG_STR_C0
[24]  SETNAME
[25]  POP_RESULT
[26]  LINE : 2
[29]  REG_STR_C0
[30]  NAME
[31]  REG_STR_C1
[32]  STRING
[33]  EQ
[34]  IFEQ 16
[37]  RETURN_RESULT
```

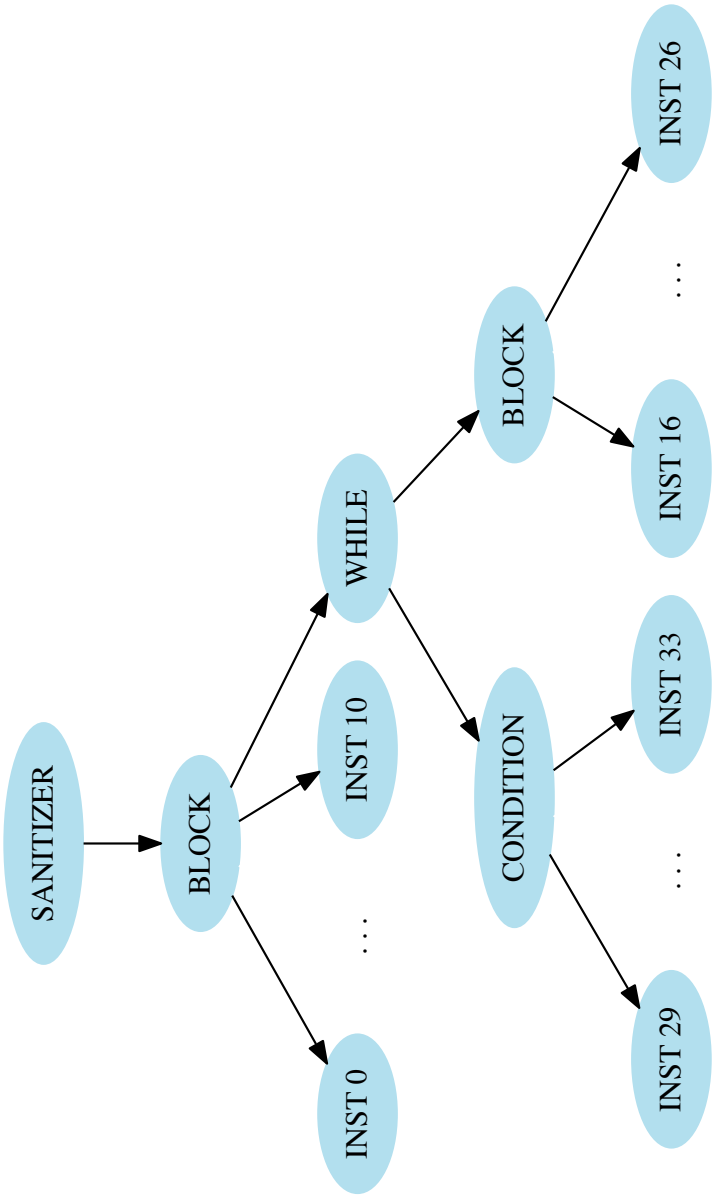

Figure 2.10: JavaScript code along with corresponding bytecode and template AST.

to two different memory locations, `x.p1` and `y.p1`, in different iterations of the while loop. Line 10 reads from two different memory locations which are `x.p1` and `y.p1`. We deal with this by having a number of IVSL statements per each bytecode instruction with read/write ac-

```
sanitizer(mloc0){
  mloc0 := "foo";
  while( mloc0 == "foo") {
    mloc0 := "bar";
  }
  return "";
}
```

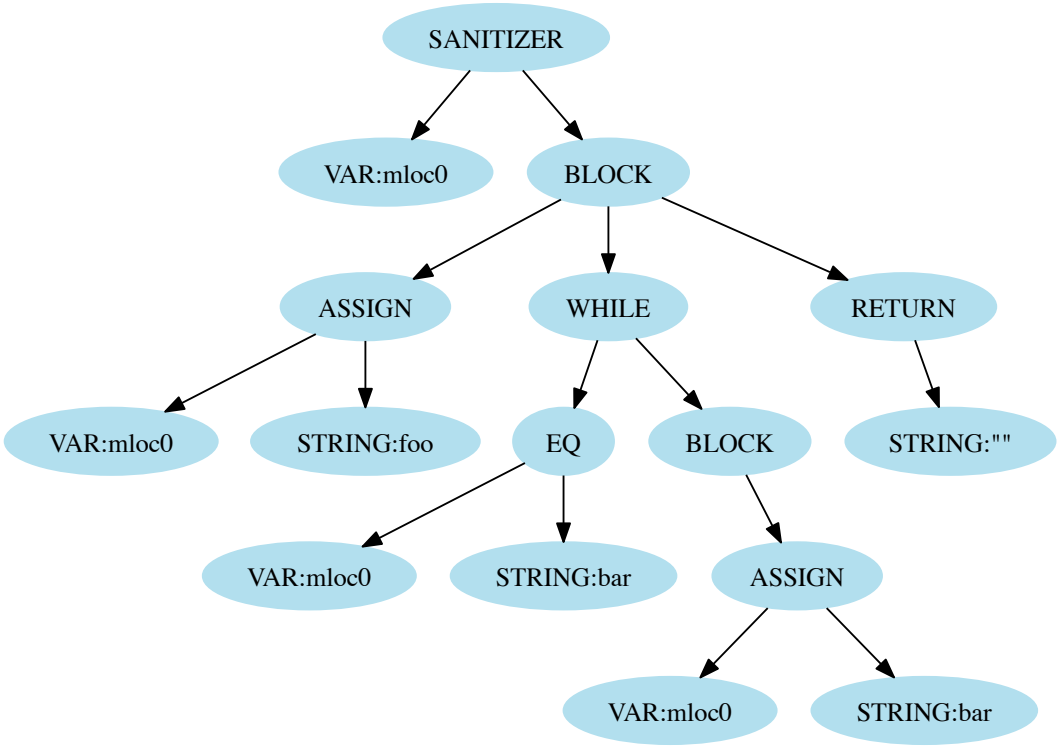

Figure 2.11: IVSL code and its AST that correspond to JavaScript and bytecode in Figure 2.10.

cess that involves aliasing. We wrap these instructions with a non-deterministic conditional statement. Lines 4-8 in the IVSL code in Figure 2.12 correspond to line 9 in JavaScript code while lines 9-13 of IVSL code correspond to line 10 of JavaScript code. Notice that lines 5-8

of the JavaScript code are ignored since they deal with non-string memory locations (`x`, `y` and `z` are object references while `i` is a number).

```
1    var x = {p1:"foo"};
2    var y = {p1:"bar"};
3    var w, i = 0, a;
4    while (i < 5){
5      if (i < 2)
6        w = x;
7      else
8        w = y;
9      w.p1 = "abc";
10     a = w.p1;
11     i++;
12   }
```

```
1   mloc0 := "foo";
2   mloc1 := "bar";
3   while(*) {
4     if(*) {
5       mloc0 := "abc";
6     }else {
7       mloc1 := "abc";
8     }
9     if(*) {
10      mloc2 := mloc0;
11    }else {
12      mloc2 := mloc1;
13    }
14  }
```

Figure 2.12: JavaScript code along with corresponding IVSL code that is extracted.

Function Inlining

CODEGENERATOR dynamically inlines functions when they are extracted. Each `CALL` instruction has its own subtree in the template AST. When executing a `CALL` instruction the following happens:

1. If caller passes string arguments then a number of IVSL assignment statement subtrees are generated, one subtree per each passed string argument. Each subtree represents the assignment of an argument to a callee function's parameter or to an element in the arguments array. The argument-passing subtrees are inserted next to each other under the node of the `CALL` instruction (which represents the call site).

```
1  function f(a){
2    var z = a;
3    z = arguments[1];
4    return z;
5  }
6  var x = "foo";
7  var y = f(x, x);
8  y = f(y, y);
```

```
1   mloc0 := "foo"; //x = "foo"
2   mloc1 := mloc0; //pass x to arguments[0]
3   mloc2 := mloc0; //pass x to arguments[1]
4   mloc3 := mloc0; //pass x to a
5   mloc4 := mloc3; //z = a
6   mloc4 := mloc2; //z = arguments[1]
7   mloc5 := mloc4; //return z into y
8   mloc6 := mloc5; //pass y to arguments[0]
9   mloc7 := mloc5; //pass y to arguments[1]
10  mloc8 := mloc5; //pass y to a
11  mloc9 := mloc8; //z = a
12  mloc9 := mloc7; //z = arguments[1]
13  mloc5 := mloc9; //return z into y
```

Figure 2.13: Inlining function calls when dynamically extracting JavaScript code.

2. The current CODEGENERATOR stack frame is pushed along with the current template AST.

3. A new CODEGENERATOR stack frame is generated along with a new template AST generated by parsing the callee function.

After the callee function returns from execution CODEGENERATOR does the following:

1. It pops the top stack frame, takes the AST of the callee and inserts all subtrees under top BLOCK node of the AST as subtrees of the `CALL` instruction node in the caller AST. These are inserted after the subtrees related to passing of arguments.

2. If the caller returns a string value then an IVSL assignment statement subtree is generated that represents the return of the value and inserted as the last subtree of the `CALL` instruction.

At the end of a function execution, the subtrees under each `CALL` instruction node in that function are moved as children of the parent of the `CALL` instruction's node. They are inserted

as right siblings of the subtree for the instruction before the `CALL` instruction and as left siblings of the subtree for the instruction after the `CALL` instruction. Figure 2.13 shows some JavaScript code along with the extracted IVSL code in which function `f` was inlined twice.

2.6 Static Extraction of Server-Side PHP Code

We use the front-end of PIXY—a vulnerability analysis tool for PHP that is based on taint analysis [55]—to extract sanitizer functions from PHP code [2, 103–105]. Unlike the case with forward extraction technique of JavaScript code, the extraction process here starts from a *sensitive sink* and goes backwards all the way to the input variables that may flow into this sink. A *sensitive sink* is a sink (i.e., an output function) for which all input values have to be sanitized before reaching it to avoid a vulnerability such as XSS or SQLI (see 4.1). Two of the most commonly used sensitive output functions in PHP are `print` and `mysql_query`.

By default, PIXY extracts a multi-input sanitizer per each sensitive sink. This sanitizer includes an input variable for each value in the HTTP request that may flow into this sink. A multi-input sanitizer is used for server-side policy-based repair that we discuss in 4.1. We augmented PIXY to add path sensitivity and support for PHP5. An IVSL sanitizer function extracted by PIXY is represented using a *dependency graph* instead of a control flow graph.

```
sanitizer(x){
  if (length(x) > 4)
    reject;
  else {
    x = replace('/</', "", x);
    if (x == '')
      reject;
    else
      return x;
  }
}
```

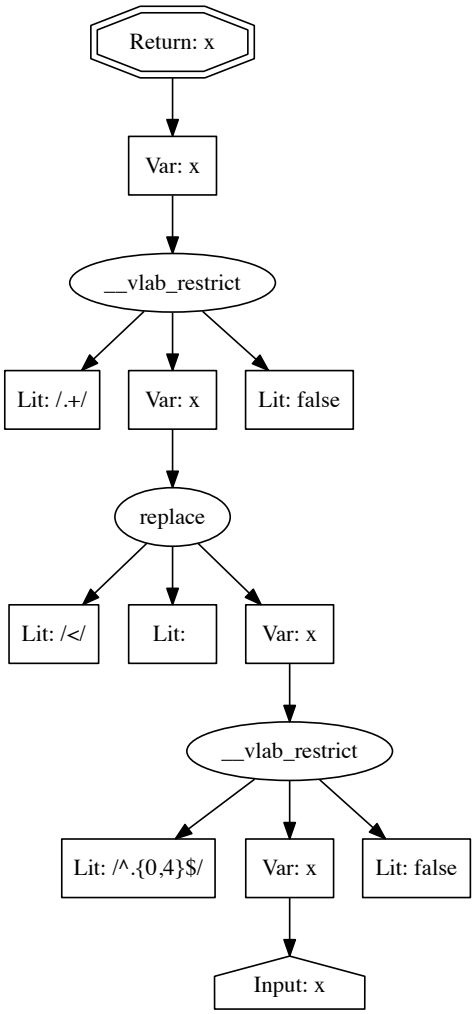

Figure 2.14: A PHP sanitizer and its dependency graph.

2.6.1 Dependency Graphs

A dependency graph specifies the data and control flow in the IVSL program. Formally speaking, a dependency graph $G = \langle N, E \rangle$ is a directed graph, where N is a finite set of nodes and $E \subseteq N \times N$ is a finite set of directed edges. An edge $(n_i, n_j) \in E$ identifies that the value of n_i depends on the value of n_j. For $n \in N$, $Succ(n) = \{n' \mid (n, n') \in E\}$ is the set of successors of n. $Pred(n) = \{n' \mid (n', n) \in E\}$ is the set of predecessors of n. Each node

$n \in N$ can be: (1) a *data* node including `input`, `literal`, `variable`, and `return`, or (2) an *operation* node including string operations such as the two string operations `concat` and `replace`.

An `input` node identifies the data from untrusted parties, e.g., an input from web forms, that is received as input to the sanitizer. A `literal` node is associated with a constant string value, a regular expression value (which is delimited by the symbol `/` at the beginning and the end) or the special value `false`. Both nodes have no successors. In other words, defining a leaf node as $Leaf(G) = \{n \mid Succ(n) = \emptyset\}$, each of these two types of nodes is always a leaf node. `return` node is the root defined as $Root(G) = \{n \mid Pred(n) = \emptyset\}$. It represents the positive sink at which the sanitizer returns its output. For each sanitizer only one `return` node is allowed.

An *operation* node represents a string manipulation operation such as `concat` and `replace`. This type of nodes has one or more successors which represent its parameters. For example, a `concat` node n has two successors labeled as the prefix node ($n.p$) and the suffix node ($n.s$), and stores the concatenation of any value of the prefix node and any value of the suffix node in n. If n is a `concat` node, $Succ(n) = \{n.p, n.s\}$. A `replace` node has three successors labeled as the target node ($n.t$), the match node ($n.m$), and the replacement node ($n.r$). For the example in Figure 2.14, $n.m$ has the regular expression `/</` as its value, $n.r$ has the string value "" (i.e., the empty string ϵ) and $n.t$ represents the variable `x`. It performs the following operations for each value of $n.t$: (1) identifies all the matches, i.e., any value of $n.m$, that

appear in $n.t$, (2) replaces all these matches in $n.t$ with any value of $n.r$, and (3) stores the replaced result in n. If n is a `replace` node, $Succ(n) = \{n.t, n.m, n.r\}$.

Furthermore, *operation* nodes include the special string operation `_vlab_restrict` which is used to represent control dependencies on branch conditions. A `_vlab_restrict` node has three successors labeled as the condition node ($n.c$), the target node ($n.t$), and the negation node ($n.g$). The condition node $n.c$ is a regular expression representing the constraint enforced by the branch condition. We have a simple converter to convert some types of constraints such as length constraints into a regular expression (as shown in the example in Figure 2.14). The converter leverages our regular expression syntax which allows for the intersection operator `&` in a regular expression (along with the standard union operator `|`) which returns the intersection of two regular languages encoded by two regular expressions. The intersection operator along with the union operator | allow to model the logical AND and OR operators in a branch condition. The negation node is used to decide if the dependency on the branch condition comes from the true branch or the false branch. A value `true` means that we should restrict with the negation of the language of the regular expression $n.c$ while a value of `false` means that we should restrict with the language of the regular expression $n.c$.

When doing differential verification discussed in Chapter 5, we need to extract a single-input sanitizer for each html input field i. In this case we need to deal with the fact that the value of the input field i may flow into more than one sensitive sink. Fortunately, since we use dynamic crawling to map client-side and server-side code to each other, we can utilize this dynamic analysis (as we explain in Chapter 5) to pinpoint which sink, among the sensitive

sinks that this input field i may flow into, we are considering. When the sink is specified, we statically extract the dependency graph—as explained above previously—starting from this sink and going backwards to all variables that may flow into it. Given the dependency graph, we do a forward phase to extract the validation and sanitization code for this variable up to the first string operation or branch condition where the variable interacts with another input variable.

Notice that here we use a sound static extraction algorithms which guarantees that all input validation and sanitization code for a given variable(s) is included.

2.7 Static Extraction of Server-Side Java Code

We used the Java static extraction technique from VIEWPOINTS [4][8] to extract Java input validation and sanitization code.

2.7.1 Web Deployment Descriptor

Each Java web application must provide a Web deployment descriptor file, `web.xml`, as specified in the Java EE specification [25]. In this first step, extraction technique analyzes this file to understand and store references to the different components used within the web application, along with the paths to various library and framework configuration files. It then performs framework specific analysis of this information to discover how input fields of appli-

[8] VIEWPOINTS is a collaborative work with Shauvik Roy Choudhary, Mattia Fazzini and Alessandro Orso who implemented the static extraction of Java code.

cation forms are validated and sanitized on the server side. Upon discovery of this information, our technique gets a reference to the server-side validation functions and proceeds with the validation code extraction. Our current implementation handles two popular J2EE frameworks: Struts and Spring MVC. Based on our experience, it could be extended to handle additional frameworks with relatively low effort.

2.7.2 Server-Side Extraction

Once the technique knows the specific server-side Java functions that are used to validate each input (i.e., form field), it accesses the corresponding class files using the Soot framework (`http://www.sable.mcgill.ca/soot/`) in order to analyze such validation functions. Because of the limitations of our current implementation, we had to apply several semi-automated transformations to the validation functions before being able to analyze them in isolation. Here is the list of transformations that our tool applies to each validation method (note that most of this transformations could be eliminated with further engineering) :

Input parameter re-writing: The function is transformed to remove all the formal arguments that are not of interest for our analysis. This allows us to have a simpler function and ignore many of the indirections introduced by the framework.

Function inlining and modeling: Our extraction technique inlines string operations performed by J2EE framework library functions.

Parameter inlining: Some of the validation routines might use parameters that are read from a configuration file. For example, the developer can specify a regular expression in the configuration that the validation function matches against the input. For such functions, the technique plugged the value of the parameter at all the corresponding uses inside the function.

After the above transformations are performed, our technique invokes the constant propagation and dead code elimination phases from the Soot framework to obtain a concise Java CFG for the validation function under analysis. These CFGs have two kinds of exit nodes—one that returns `true`, leading to the successful validation of the input, and the other one that returns `false`, leading to the rejection of the input. Our technique first uses this CFG to compute control and data dependences, which are then used to synthesize the PDG for the function. Upon the creation of the PDG, forward slicing is performed from the variable representing the input parameter being validated. This static slice contains all of the operations that are performed on the input variable and marks those that are string operations. Then, our technique performs backward slicing (on this forward slice) starting from the accepting nodes (i.e., `return true` statements) to capture the string operations involved in the successful validation. The resulting slice is a CFG with only string operations performed on the validated input and is transformed into an IVSL CFG graph for analysis.

Chapter 3

Analyzing Sanitizer Functions Using String Analysis

After extracting a sanitizer function, the next step is to anlyze this function using automata-based string analysis. The analysis result is used later to verify the correctness of the function and repair it in case a bug was found. The analysis computes the post- and pre-images of the function as deteministic finite automata using symbolic string analysis.

We first give the formal definition of the pre- and post-image of a sanitizer [2] and then show the string analysis algorithms that we use for image computation.

3.1 Post- and Pre-Image of a Sanitizer

3.1.1 Post-Image

Assuming a given input, we call the set of strings returned by a sanitizer function its *post-image* (which is the set of strings that reach the sink, i.e., the return statement). Formally speaking, given a sanitizer F with n input variables, the set of strings returned by F when the input language for each input variable v_i is restricted to L_i where $L_i \subseteq \Sigma^*$ is defined as:

$$\textsc{Post}(F, (L_1, \ldots, L_n)) = \{s \mid \exists (s_1, \ldots, s_n) \in L_1 \times \cdots \times L_n : \exists s \in \Sigma^* : F(s_1, \ldots, s_n) = s\}$$

We call this set the post-image of sanitizer F with respect to $L_1 \times \cdots \times L_n$. We compute the post-image of a sanitizer using automata-based forward symbolic string analysis algorithm in 3.2.5. In general, we can not precisely compute $\textsc{Post}(F, (L_1, \ldots, L_n))$ due to undecidability of string analysis. Instead we compute an over-approximation of this set, namely, $\textsc{Post}^+(F, (L_1, \ldots, L_n)) \supseteq \textsc{Post}(F, (L_1, \ldots, L_n))$. This means that, we may conclude that certain strings are accepted and returned by F when they are not. Since we are using automata-based symbolic string analysis, the result of the post-image computation is an automaton that accepts the language $\textsc{Post}^+(F, (L_1, \ldots, L_n))$, and we denote this automaton as $\mathcal{A}(\textsc{Post}^+(F, (L_1, \ldots, L_n)))$.

Figure 3.1 shows a sanitizer function F_1 along with Venn Diagrams illustrating its domain and co-domain. Function F_1 represents a single-input sanitizer function $F_1 : \Sigma^* \rightarrow \Sigma^* \cup \{\perp\}$ where $\Sigma = \{a, b\}$.

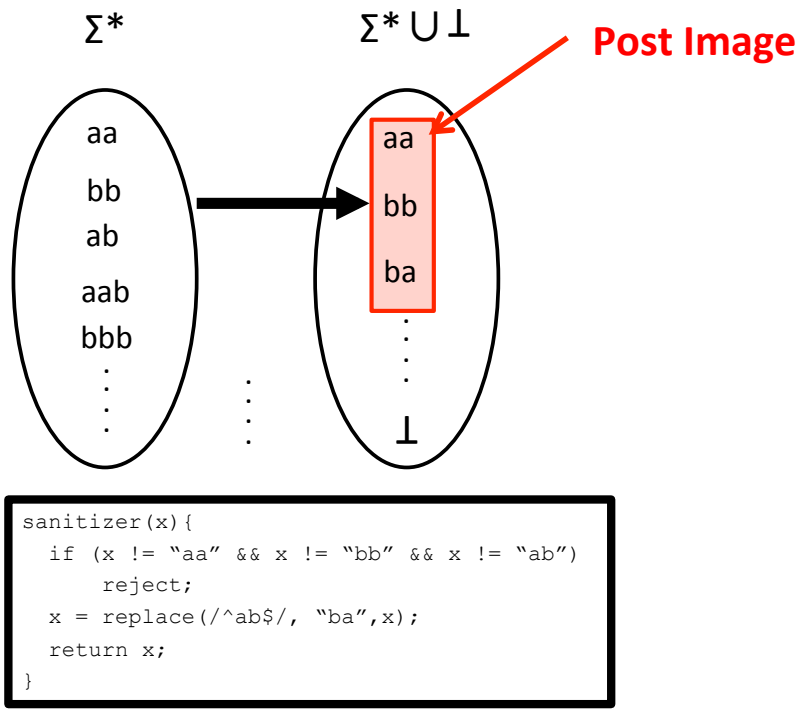

Figure 3.1: Example of *post-image* (shaded areas) for a sanitizer function F_1 assuming input to be Σ^* where $\Sigma = \{a, b\}$.

Assuming Σ^* as input, the function's *post-image* $\textsc{Post}(F_1, \Sigma^*) = \{aa, bb, ba\}$ (notice that we always exclude $\perp$ from *post-image* as it does not represent a returned string value).

3.1.2 Pre-Image

Given a sanitizer function F with n number of input variables and a set of strings $L \subseteq \Sigma^*$ in the co-domain of F, we call the set of input tuples of strings that is mapped by F to L the *pre-image* of F with respect to L and we define it as:

$$\textsc{Pre}(F, L) = \{(s_1, \ldots, s_n) \mid \exists s \in L : F(s_1, \ldots, s_n) = s\}$$

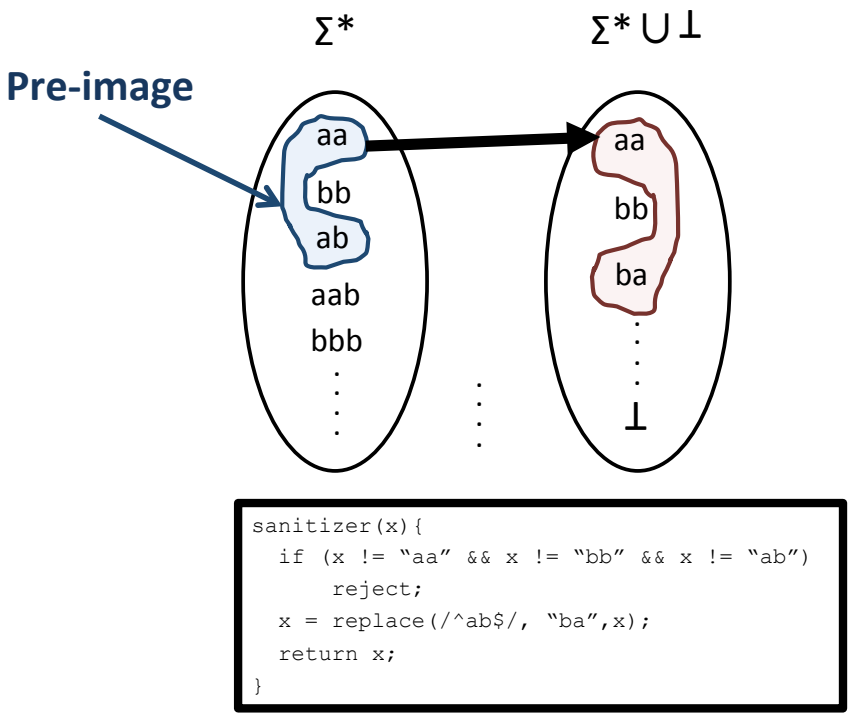

Figure 3.2: Example of *pre-image* (the shaded area on the left) of sanitizer function F_1 given a subset of the co-domain of F_1 (shaded area on the right).

We use automata-based backward symbolic string analysis algorithm in 3.2.6 to compute the pre-image of a sanitizer. Again, due to over-approximation, we compute the set $\text{PRE}^+(F, L) \supseteq \text{PRE}(F, L)$. Since the string analysis algorithm that we use does not consider the relations between different variables in a sanitizer (i.e., relational string analysis [110] is not used), $\text{PRE}^+(F, L)$ for a sanitizer F with more than one input variable (i.e., number of input variables $n > 1$) will always be the set $(\Sigma^*)^n$. In other words, when computing the pre-image of a sanitizer with more than one input variable, for all practical purposes the result is useless due to loss of all precision.

Figure 3.2 shows the sanitizer function F_1 along with its *pre-image*. Given the set $\{aa, ba\}$, the *pre-image* $\text{PRE}(F_1, \{aa, ba\}) = \{aa, ab\}$.

3.1.3 Negative Pre-Image

We call the set of strings that are rejected by a sanitizer F the *negative pre-image* of F. For a given sanitizer function F, this set is defined as:

$$\text{PRE}_{\perp}(F) = \{(s_1, \ldots, s_n) \mid F(s_1, \ldots, s_n) = \perp\}$$

We use a slightly different automata-based backward symbolic string analysis algorithm which we show in 3.2.7 to compute an over approximation of the negative pre-image, $\text{PRE}^{+}_{\perp}(F)$, where $\text{PRE}^{+}_{\perp}(F) \supseteq \text{PRE}_{\perp}(F)$. This means that, we may conclude that certain strings are rejected by F when they are not. On the other hand, since we are computing an over-approximation, any string that is rejected by F is guaranteed to be in $\text{PRE}^{+}_{\perp}(F)$. Since we are using automata-based symbolic string analysis, the result of the negative pre-image computation is an automaton that accepts the language $\text{PRE}^{+}_{\perp}(F)$, and we denote this automaton as $\mathcal{A}(\text{PRE}^{+}_{\perp}(F))$.

Figure 3.3 shows the *negative pre-image* of sanitizer F_1. $\text{PRE}_{\perp}(F_1) = \Sigma^* \setminus \{aa, bb, ab\}$.

3.1.4 Negative Post-Image

This set is a special set in the sense that it does not characterize a subset of the input or the output of a sanitizer. Given a sanitizer F and a set of possible input values L, the *negative*

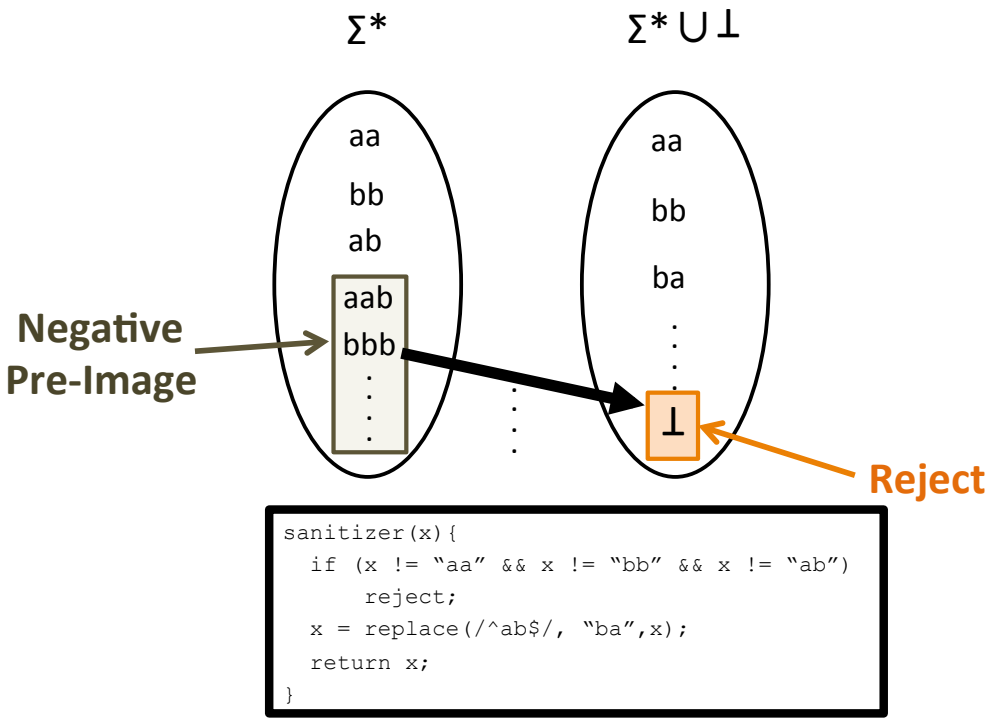

Figure 3.3: Example of a *negative pre-image* (the shaded area on the left) of sanitizer function F_1 which is mapped by F_1 to $\perp$ (i.e., rejected).

post-image of F with respect to L, $\textsc{Post}_{\perp}(F, L)$, is the set of strings that reach the negative sinks (i.e., reach the `reject` statements) in F.

As is the case with previous images, in general, we can not precisely compute $\textsc{Post}_{\perp}(F, L)$ due to undecidability of string analysis. Instead we compute an over-approximation of this set, namely, $\textsc{Post}^{+}_{\perp}(F, L) \supseteq \textsc{Post}_{\perp}(F, L)$. This means that, we may conclude that a string can reach a negative sink when it does not. Since we are using automata-based symbolic string analysis, the result of the negative post-image computation is an automaton that accepts the language $\textsc{Post}^{+}_{\perp}(F, L)$, and we denote this automaton as $\mathcal{A}(\textsc{Post}^{+}_{\perp}(F, L))$.

3.2 Automata-Based Symbolic String Analysis

To compute pre and post-images of a sanitizer function, we use automata-based flow-sensitive and path-sensitive symbolic string analysis. The string analysis technique we use is a forward/backward reachability computation that uses Deterministic Finite Automata (DFA) as a symbolic representation. We iteratively compute an over approximation of the least fixpoint that corresponds to the reachable values of the string expressions. In each iteration, given the current state DFA for a variable, we compute the next/previous state DFA. We use algorithms for next/previous state computation for common string operations such as concatenation and language-based replacement as well as specialized string sanitization operations such as `trim` and `addslashes`.

We use the symbolic DFA representation provided by the MONA DFA library [12], in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs) [33]. MONA also provides the implementation for automata operations union, intersection, negation and projection for symbolic DFA.

3.2.1 Symbolic DFA

Given $\mathcal{B} = \{0, 1\}$, a symbolic DFA M is a tuple $\langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ where:

- Q is a finite set of states.
- q_0 is the initial state.

- Instead of using a regular alphabet Σ, where each character c is a single ASCII printable symbol such as `a` and `b`, we use a symbolic binary alphabet $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^k$ where $k = log_2(\lceil |\Sigma| \rceil)$ and each alphabet symbol α is a k-bit string $\alpha \in \mathcal{B}^k$. In our discussion, we will use *character* to refer to a non-symbolic alphabet character $c \in \Sigma$ and we will use *alphabet symbol* to refer to a symbolic alphabet symbol $\alpha \in \Sigma_{\mathcal{B}}$. Each character $c \in \Sigma$ is mapped to one and only one corresponding alphabet symbol $\alpha_c \in \Sigma_{\mathcal{B}}$ and vice versa.

- $F \subseteq Q$ is finite set of accepting states.

- $\delta : Q \times \Sigma_{\mathcal{B}} \rightarrow Q$ is the transition relation.

Following our definition of Σ and $\Sigma_{\mathcal{B}}$, we define a non-symbolic string w of length n as a sequence of characters $\langle c_0, c_1, \ldots, c_{n-1} \rangle$ where each character $c_i \in \Sigma$ and its corresponding symbolic string $w_{\mathcal{B}}$ as a sequence of alphabet symbols $\langle \alpha_0, \alpha_1, \ldots, \alpha_{n-1} \rangle$ where each alphabet symbol $\alpha_i \in \Sigma_{\mathcal{B}}$. The special string w of length 0 $|w| = 0$ is called ϵ. Let us define the relation $\delta^* : Q \times \Sigma^*_{\mathcal{B}} \rightarrow Q$ for the symbolic DFA M as following:

given a string $w_{\mathcal{B}} = \langle \alpha_0, \alpha_1, \ldots, \alpha_{n-1} \rangle$ where each character $\alpha_i \in \Sigma_{\mathcal{B}}$

$\delta^*(q_i, \epsilon) = q_i$ and $\delta^*(q_i, w_{\mathcal{B}}) = q_j$ if there exists a sequence $\langle q_i, q_{i+1}, q_{i+2}, \ldots, q_{i+n} \rangle \in Q^{n+1}$ such that:

(1) $q_{i+n} = q_j$ (2) $\forall l \geq 0 : \ \delta(q_{i+l}, \alpha_l) = q_{i+l+1}$

A state q of M is a *sink* state if $\forall \alpha \in \Sigma_{\mathcal{B}}, \delta(q, \alpha) = q$ and $q \notin F$. In the following discussion, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a $sink$ state. When

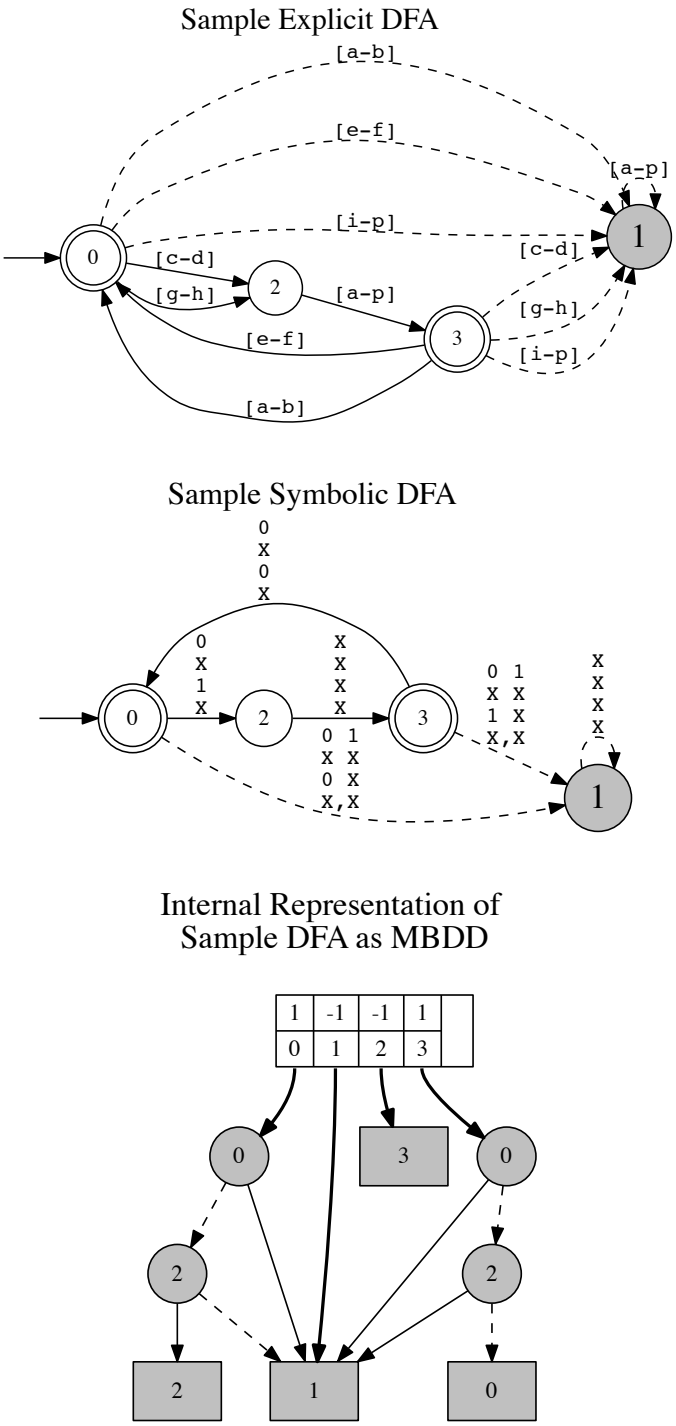

Figure 3.4: Symbolic representation of a DFA using MBDD.

```
a = 0000     b = 0001          0X0X = a, b, e, f = [a-b], [e-f]
c = 0010     d = 0011          0X1X = c, d, g, h = [c-d], [g-h]
e = 0100     f = 0101          1XXX = i, j, k, l, m, n, o, p = [i-p]
g = 0110     h = 0111          XXXX = a, b, c, d, e, f, g, h,
i = 1000     j = 1001                 i, j, k, l, m, n, o, p = [a-p]
k = 1010     l = 1011
m = 1100     n = 1101
o = 1110     p = 1111
```

Figure 3.5: Σ and corresponding $\Sigma_{\mathcal{B}}$ for the sample symbolic DFA along with MONA transition labels and their corresponding explicit transitions.

visualizing a DFA we omit the $sink$ state and the transitions that lead to a sink state. We say that a string $w_{\mathcal{B}}$ is accepted by M if $\delta^*(q_0, w_{\mathcal{B}}) \neq sink$. The language of M or $\mathcal{L}(M) \subseteq \Sigma_{\mathcal{B}}$ is the set of strings $w_{\mathcal{B}}$ that are accepted by M. For two states q_i and q_j in M, a transition between q_i and q_j on an alphabet symbol α is a tuple (q_i, α, q_j) where $\delta(q_i, \alpha) = q_j$ and we write it like this $(q_i \xrightarrow{\alpha} q_j)$. For two states q_i and q_j in M, we say there is a path $q_i, q_{i+1}, \ldots, q_j$ of length n between q_i and q_j if there is a string $w_{\mathcal{B}} \in \Sigma_{\mathcal{B}}^*$ of length n such that $\delta^*(q_i, w_{\mathcal{B}}) = q_j$ and we write it like this $(q_i \xrightarrow{w_{\mathcal{B}}}{}^{*} q_j)$.

Example. Figure 3.4 shows an example symbolic DFA. At the top is the explicit DFA using an explicit representation that uses character ranges. In the middle is the symbolic DFA that uses binary alphabet symbols. In the bottom is the actual internal representation using a Multi-terminal Binary Decision Diagram (MBDD). The alphabet Σ and the corresponding symbolic alphabet $\Sigma_{\mathcal{B}}$ is shown in Figure 3.5. We have 16 characters in Σ which means that we need $log_2(\lceil 16 \rceil) = 4$ bits for the symbolic alphabet (i.e., $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^4$). Figure 3.5 shows each character $c \in \Sigma$ and its corresponding alphabet symbol $\alpha_c \in \Sigma_{\mathcal{B}}$. For example, $\alpha_a = 0000$ and $\alpha_l = 1011$.

The sample DFA in Figure 3.4 has 3 states $Q = \{S_0, S_1, S_2, S_3\}$. Our convention is to label each state with a number n and refer to it in the text with S_n. The shaded state S_1 is the sink state and dashed edges represent transitions that go to sink state. From now on we always omit sink states to simplify the figures (shaded states and dashed edges will be used for other purposes). In the sample symbolic DFA in the middle, we use MONA symbolic labels such as $\begin{smallmatrix}0\\X\\1\\X\end{smallmatrix}$ where the X symbol represents an option of either 0 or 1. For example, from state $S_0 \rightarrow S_2$, an edge labeled $\begin{smallmatrix}0\\X\\1\\X\end{smallmatrix}$ means that there are four transitions between these two states on alphabet $0010, 0011, 0110$ and 0111 (i.e., four transitions on characters c, d, g, h). Figure 3.5 shows each of the MONA labels used in sample symbolic DFA and its meaning.

MBDD. Figure 3.4 shows at the bottom the Multi-terminal Binary Decision Diagram [33] that is used as the actual internal representation of the sample symbolic DFA. The second row in table at the top |0|1|2|3| represents DFA states while the first row in that table |1|-1|-1|1| represents state types which are either accepting state 1 or rejecting state -1. The shaded nodes represent *BDD nodes*. Each circle-shaped node has a number n that represents its level i.e., which BDD variable n (in other words, which bit n in an alphabet symbol $\alpha \in \Sigma_{\mathcal{B}}$) it corresponds to. Each square-shaped node has a number n that represents the destination state S_n that the node corresponds to. Dashed line represents a BDD variable (bit) value of 0 while a regular line represents a BDD variable (bit) value of 1. The symbolic transition relation works as following: suppose that we are in state S_0 and given alphabet symbol $\alpha_c = 0010$. Let us see how we go from state S_0 to state S_2 on character c. We start from table cell |0| then go to a BDD node at level 0. Then, looking at value of bit-0 of α_c which is 0, we go to a BDD

node at level 2. Notice that since we are in a BDD node at level 2, we will look at 2nd BDD variable (i.e., bit-2) skipping value of bit-1 as it does not affect which destination state we will go to. Then we look at value of bit-2 which is 1 which means that we go to destination state S_2 (skipping value of bit-3).

Throughout the remaining of this text we use a representation of a DFA in a figure that is a mixture of the top two representations in Figure 3.4. On one hand, we will have one edge only between each two states in the DFA (as we did in the middle sample symbolic DFA). On the other hand, instead of labeling the edge with a MONA label like $\begin{smallmatrix}1\\X\\X\\X\end{smallmatrix}$ we will label it with either character ranges such as `[a-c]` or a character set such as $\{a, b\}$ and $\Sigma \setminus \{d, h, k\}$. In addition, we will always omit the sink state and all transitions that go to it.

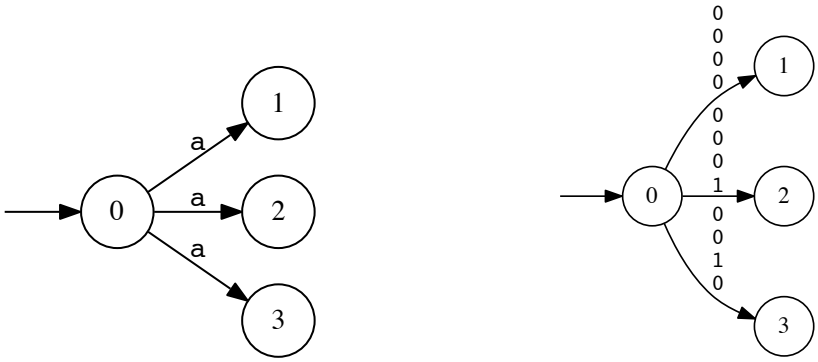

Figure 3.6: A symbolic DFA on the right with $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^2$ simulating non-determinism in the NFA on the left using 2 extra bits.

3.2.2 Non-Determinism in Symbolic DFA

The algorithms for next/previous state computation for string operations are implemented without using the standard constructions based on the ϵ-transitions, since the MBDD-based

automata representation used by MONA does not allow ϵ-transitions. Non-determinism is modeled by extending the alphabet with extra bits and then projecting them away using the on-the-fly subset construction algorithm provided by MONA. Projection is applied one bit at a time, and after projecting each bit away, the size of the resulting automaton is reduced using MBDD-based automata minimization.

Formally, the project and determinize operation, denoted as PROJECT(M, i), where $1 \leq i \leq k$, converts a DFA M recognizing a language L over the alphabet $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^k$, to a DFA M' recognizing a language L' over the alphabet $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^{k-1}$, where L' is the language that results from applying the tuple projection on the i^{th} bit to each symbol of the alphabet. The process consists of removing the i^{th} track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction. This operation is provided by MONA library. If we have a DFA M recognizing a language L over the alphabet $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^k$, and we want to add to M n non-deterministic transitions out from a state S on some character c, we need to extend $\Sigma_{\mathcal{B}}$ with $l = \lceil log_2(n) \rceil$ extra bits to get $\Sigma'_{\mathcal{B}} \subseteq \mathcal{B}^{k+l}$. Then we determinize by projecting the extra l bits one bit at a time.

Figure 3.6 shows on the left part of a Non-deterministic FA with three non-deterministic transitions on character a from state S_0 to states S_1, S_2 and S_3. On the right it shows the corresponding symbolic DFA where $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^2$ and $\alpha_a = 00$. To simulate non-determinism, we need to extend the alphabet $\Sigma_{\mathcal{B}}$ by adding 2 extra bits to represent 3 different characters a namely a_0, a_1 and a_2 where $\alpha_{a_0} = 0000$, $\alpha_{a_1} = 0001$ and $\alpha_{a_2} = 0010$. At the end, we determinize the DFA by projecting bit-3 then bit-2.

CONSTRUCT(regexp e)$\equiv \mathcal{A}(\{w \mid w \in \mathcal{L}(e)\})$.
CLOSURE(DFA M_1) $\equiv \mathcal{A}(\{w_1 w_2 \dots w_k \mid k > 0, \forall i, 1 \leq i \leq k, w_i \in \mathcal{L}(M_1)\})$.
UNION(DFA M_1, DFA M_2) $\equiv M$ where $\mathcal{L}(M) = \mathcal{L}(M_1) \cup \mathcal{L}(M2)$.
INTERSECT(DFA M_1, DFA M_2) $\equiv M$ where $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.
WIDENING(DFA M_1, DFA M_2) $\equiv M$ where $\mathcal{L}(M) \supseteq \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$.
EQUCHECK(DFA M_1, DFA M_2) $\equiv$ True if $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ otherwise.
EMPCHECK(DFA M) $\equiv$ True if $\mathcal{L}(M) = \emptyset$ otherwise False.
EMPTY() $\equiv M$ where $\mathcal{L}(M) = \emptyset$.
UNIVERSAL() $\equiv M$ where $\mathcal{L}(M) = \Sigma^*$

Figure 3.7: Core automata operations used in our analysis.

3.2.3 Symbolic vs. Explicit DFA

Symbolic DFA is more efficient in terms of memory than explicit DFA which means that, using symbolic DFA, our analysis consumes less memory. Although both explicit and BDD representation of Sample DFA shown in Figure 3.4 seem to use the same number of transitions, bear in mind that a BDD transition is labeled with a single bit while an explicit DFA transition is labeled with 2 characters using 4 bits to represent each one ($4 = log_2(|\Sigma|)$). In general, the difference between explicit and symbolic DFA in memory consumption becomes more obvious as size of alphabet $|\Sigma|$ grows such as the case with $|\Sigma_{ASCII}| = 256$ and $|\Sigma_{UNICODE}| = 65536$ [47].

3.2.4 Analysis Lattice and Termination.

Before discussing the analysis lattice, let us first introduce the automata operations $\cap, \cup, \setminus,$ — (which are implemented by MONA library [12]) that generate automata that accept the intersection, union, difference and complement of the languages of the given automata, respectively, the operation $\odot$ that generates an automaton that accepts the concatenation of the languages of

the given two automata and the $=, \subseteq$ operations which test the language equivalence and language inclusion between two automata. The function $\mathcal{A}(L)$ takes a regular language $L \subseteq \Sigma^*$ and returns a DFA that recognizes L. We also use $\mathcal{L}(M)$ to denote the language accepted by the automaton M. Figure 3.7 shows the definitions for these operations (for concatenation operation $\odot$ see POSTCONCAT in Figure 3.8).

Each element in string analysis lattice for a given variable is a symbolic DFA M that encodes a regular language $\mathcal{L}(M)$. $\mathcal{L}(M)$ is the set of possible values that a variable can take at a program point. The bottom element in the lattice is $\perp$ which is the empty set DFA $\mathcal{A}(\emptyset)$ while the top $\top$ element is the DFA $\mathcal{A}(\Sigma^*)$. Elements in the lattice are partially ordered using operator $\subseteq$ where $M_1 \preceq M_2 \Leftrightarrow M_1 \subseteq M_2 \Leftrightarrow \mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$. The join operator is $\cup$ and the meet operator is $\cap$. The lattice for the analysis is the cartisian product of the lattices for all the variables in the sanitizer function.

Although a DFA is finite structure, it may represent an infinite set of strings which means that the lattice for a variable (consequently the analysis lattice) has an infinite height. For example, given $\Sigma = \{a\}$, $\mathcal{A}(\emptyset) \subseteq \mathcal{A}(\{a\}) \subseteq \mathcal{A}(\{a, aa\}) \subseteq \mathcal{A}(\{a, aa, aaa\}) \cdots \subseteq \mathcal{A}(a^*)$ is an infinite chain in the lattice. Due to this reason, the fixpoint computations are not guaranteed to converge. To alleviate this problem, we use the automata widening operator ∇ proposed by Bartzis and Bultan [11] to compute an over-approximation of the least fixpoint. This widening operator merges those states belonging to the same equivalence class identified by certain conditions.

Algorithm 1 FWANALYSIS($CFG, M_1, \ldots, M_n$)

```
1:  initInputVars(v_1, ..., v_n, M_1, ..., M_n, getIN(CFG.entrynode));
2:  queue WQ := NULL;
3:  WQ.enqueue(CFG.entrynode);
4:  while (WQ ≠ NULL) do
5:      node := WQ.dequeue(); changed := false;
6:      IN := getIN(node); OUT := getOUT(node);
7:      for all (var in getVars()) do
8:          IN[var] := A(∅);
9:          for all (node' in getPredNodes(node)) do
10:             if (node' ≡ IF pred THEN or node' ≡ WHILE pred) then
11:                 if node = getSuccOnTrueBranch(node') then
12:                     OUT_node' := getOUTOnTrueBranch(node');
13:                 else
14:                     OUT_node' := getOUTOnFalseBranch(node');
15:                 end if
16:             else
17:                 OUT_node' := getOUT(node');
18:             end if
19:             IN[var]:= IN[var] ∪ OUT_node'[var];
20:         end for
21:     end for
22:     if (node ≡ IF pred THEN or node ≡ WHILE pred) then
23:         OUT_on_T := getOUTOnTrueBranch(node); OUT_on_F := getOUTOnFalseBranch(node);
24:         tmp_on_T := tmp_on_F := IN;
25:         if (numOfVars(pred) = 1) then
26:             var := getPredVar(pred);
27:             predVal := EVALPRED(pred, IN[var]);
28:             tmp_on_T[var] : = IN[var] ∩ predVal;
29:             tmp_on_F[var] : = IN[var] ∩ (A(Σ*) \ predVal);
30:         end if
31:         for all (var in getVars()) do
32:             tmp_on_T[var] := (tmp_on_T[var] ∪ OUT_on_T[var]) ∇ OUT_on_T[var];
33:             tmp_on_F[var] := (tmp_on_F[var] ∪ OUT_on_F[var]) ∇ OUT_on_F[var];
34:             if (tmp_on_T[var] ⋦ OUT_on_T[var]) then
35:                 OUT_on_T[var] := tmp_on_T[var]; OUT_on_F[var] := tmp_on_F[var];
36:                 changed := true;
37:             end if
38:         end for
39:         if (changed) then
40:             WQ.enqueue(getSuccOnTrueBranch(node)); WQ.enqueue(getSuccOnFalseBranch(node));
41:         end if
42:     else
43:         tmp := POSTTRANSFERFUNCTION(node, IN);
44:         for all (var in getVars()) do
45:             tmp[var] := (tmp[var]∪OUT[var]) ∇OUT[var];
46:             if (tmp[var] ⋦ OUT[var]) then
47:                 OUT[var] := tmp[var];
48:                 changed := true;
49:             end if
50:         end for
51:         if (changed) then
52:             WQ.enqueue(getSuccNode(node));
53:         end if
54:     end if
55: end while
56: var := getVarInReturnNode();
57: return getIN(CFG.returnnode)[var]
```

3.2.5 Forward Analysis

Algorithm 1 computes the least fixed point that over-approximates the possible values that string variables can take at any given program point [103, 105]. We use the algorithm to compute the post-image of a sanitizer function F with n inputs variables (i.e., $\textsc{Post}(F, L_1, \ldots, L_n)$). The algorithm takes as input the CFG of a sanitizer function F along with an array of DFAs $M_1 \ldots M_n$ that represents the assumed initial input i.e., the initial sets of values that we assume input variables $v_1 \ldots v_n$ can take. The algorithm is worklist based, that is, it keeps track of the CFG nodes that still needs to be processed in a worklist. Each statement is associated with two arrays of DFAs: `IN` and `OUT`. DFA `IN[var]` is a DFA that accepts all string values that a certain string variable var can take at the program point just before the execution of that statement. Similarly, `OUT[var]` is a DFA that accepts all string values that string variable var can take at the program point just after the execution of that statement. The `tmp` array is used to store the temporary values (i.e. DFAs) computed by the transfer function before joining these values with the previous ones.

Starting with the CFG entry node in the worklist (line 3), a CFG node is extracted from the worklist at each iteration of the algorithm (lines 4, 5). `IN` array for this node is computed as the union of `OUT` arrays from predecessor statements (lines 6-21) representing possible values that may flow from predecessor statements to this statement. Then the transfer function for the corresponding statement is computed (lines 22-50) as we describe later in this chapter. Briefly, the DFAs in the `tmp` array are updated based on the DFAs in the `IN` array and the transfer function of the statement associated with the current node (lines 24-29, 43). After

computing the transfer function, each value in the `tmp` array for the current statement is joined (unioned) and widened with the corresponding old value in the `OUT` array (lines 32, 33, 45). The analysis converges when the worklist becomes empty, which means that reevaluating the transfer functions would not change the values in the `OUT` array (lines 34-37, 46-49). After convergence, the `OUT[var]` value for the variable var that is returned at the `return var` statement (i.e., the statement that represents returning an output value by the sanitizer function) is the result of our analysis—a DFA that accepts an over-approximation of the set of values output by the sanitizer function (lines 56, 57).

Modified Forward Analysis Assuming a sanitizer F and a set of inputs L, we need to slightly modify the forward analysis algorithm to compute the negative post-image $\textsc{Post}^{+}_{\perp}(F, L)$. The modification is to compute an over-approximation of each of the sets of all possible strings that may reach a *negative* sink (i.e., a `reject` statement) instead of the *positive* sink (i.e., the `return var` statement). It is enough to change the line 56 and 57 to achieve this such that we return the union of the DFA values at each negative sink for the given variable (we only consider one variable here according to the definition of the negative post image).

3.2.6 Backward Analysis

Algorithm 2 is a worklist-based algorithm that, given a sanitizer function F and a set of string values $\mathcal{L}(M)$ that represents the preferred output (i.e., preferred subset of strings that are allowed to reach the program point after sink statement `return var`), it computes an over-approximation of the set (of input values) that is mapped by the sanitizer function F to

Algorithm 2 BWANALYSIS($CFG, M, FwdResult$)

```
1:  v_1 := getVarInReturnStmt();
2:  initVar(v_1, M);
3:  queue WQ := NULL;
4:  WQ.enqueue(CFG.returnnode);
5:  while (WQ ≠ NULL) do
6:      node := WQ.dequeue(); changed := false;
7:      OUT := getOUT(node); IN := getIN(node);
8:      for all (var in getVars()) do
9:          node' := getSuccNode(node);
10:         IN_node' := getIN(node');
11:         OUT[var] := IN_node'[var];
12:     end for
13:     if (node ≡ IF pred THEN or node ≡ WHILE pred) then
14:         OUT_on_T := getOUTOnTrueBranch(node); OUT_on_F := getOUTOnFalseBranch(node);
15:         tmp_on_T := OUT_on_T; tmp_on_F := OUT_on_F;
16:         if (numOfVars(pred) = 1) then
17:             var := getPredVar(pred);
18:             predVal := EVALPRED(pred, IN[var]);
19:             tmp_on_T[var] : = OUT[var] ∩ predVal;
20:             tmp_on_F[var] : = OUT[var] ∩ (A(Σ*) \ predVal);
21:         end if
22:         for all (var in getVars()) do
23:             tmp[var] := (tmp_on_T[var] ∪ tmp_on_F[var] ∪ IN[var]) ∇ IN[var];
24:             if (tmp[var] ⊈ IN[var]) then
25:                 IN[var] := tmp[var];
26:                 changed := true;
27:             end if
28:         end for
29:         if (changed) then
30:             WQ.enqueue(getPredNodes(node));
31:         end if
32:     else
33:         if FwdResult ≠ NULL then
34:             OUT_max := getOUT(FwdResult[node]);
35:         else
36:             OUT_max := POSTTRANSFERFUNCTION(node, [A(Σ*), ..., A(Σ*)]);
37:         end if
38:         for all (var in getVars()) do
39:             OUT[var] := OUT[var] ∩ OUT_max[var];
40:         end for
41:         tmp := PRETRANSFERFUNCTION(node, IN);
42:         for all (var in getVars()) do
43:             tmp[var] := (tmp[var]∪IN[var]) ∇IN[var];
44:             if (tmp[var] ⊈ IN[var]) then
45:                 IN[var] := tmp[var];
46:                 changed := true;
47:             end if
48:         end for
49:         if (changed) then
50:             WQ.enqueue(getPredNodes(node));
51:         end if
52:     end if
53: end while
54: return getOUT(CFG.entrynode)
```

Algorithm 3 NEGBWANALYSIS(CFG)

```
1:  v_1 := getVarInReturnStmt();
2:  initVar(v_1, A(∅));
3:  queue WQ := NULL;
4:  WQ.enqueue(CFG.returnnode);
5:  while (WQ ≠ NULL) do
6:      node := WQ.dequeue(); changed := false;
7:      OUT := getOUT(node); IN := getIN(node);
8:      for all (var in getVars()) do
9:          node' := getSuccNode(node);
10:         IN_node' := getIN(node');
11:         OUT[var] := IN_node'[var];
12:     end for
13:     if (node ≡ IF pred THEN or node ≡ WHILE pred) then
14:         OUT_on_T := getOUTOnTrueBranch(node); OUT_on_F := getOUTOnFalseBranch(node);
15:         tmp_on_T := OUT_on_T; tmp_on_F := OUT_on_F;
16:         if (numOfVars(pred) = 1) then
17:             var := getPredVar(pred);
18:             predVal := EVALPRED(pred, IN[var]);
19:             tmp_on_T[var] : = OUT[var] ∪ A(Σ*) \ predVal;
20:             tmp_on_F[var] : = OUT[var] ∪ (predVal);
21:         end if
22:         for all (var in getVars()) do
23:             tmp[var] := (tmp_on_T[var] ∪ tmp_on_F[var] ∪ IN[var]) ∇ IN[var];
24:             if (tmp[var] ⊈ IN[var]) then
25:                 IN[var] := tmp[var];
26:                 changed := true;
27:             end if
28:         end for
29:         if (changed) then
30:             WQ.enqueue(getPredNodes(node));
31:         end if
32:     else
33:         OUT_max := POSTTRANSFERFUNCTION(node, [A(Σ*), ..., A(Σ*)]);
34:         for all (var in getVars()) do
35:             OUT[var] := OUT[var] ∩ OUT_max[var];
36:         end for
37:         tmp := PRETRANSFERFUNCTION(node, OUT);
38:         for all (var in getVars()) do
39:             tmp[var] := (tmp[var]∪IN[var]) ∇IN[var];
40:             if (tmp[var] ⊈ IN[var]) then
41:                 IN[var] := tmp[var];
42:                 changed := true;
43:             end if
44:         end for
45:         if (changed) then
46:             WQ.enqueue(getPredNodes(node));
47:         end if
48:     end if
49: end while
50: return getOUT(CFG.entrynode)
```

$\mathcal{L}(M)$ [102, 104]. We use the algorithm to compute the pre-image of a sanitizer function F given a set of preferred outputs L (i.e., PRE(F, L)). The algorithm takes as input the CFG of the sanitizer function F along with a DFA M that represents the set of preferred output values that F is allowed to return. The difference between this algorithm and forward analysis algorithm (Algorithm 1) is that this algorithm goes in the opposite direction starting from the CFG node that represents the sink statement `return var`. For a given statement, the values in the `IN` array are computed based on the values in the `OUT` array. In other words, given a DFA `OUT[var]` that accepts all the possible values that a variable var can take after executing a statement, we compute the pre-image of `OUT[var]` which is the DFA `IN[var]` that accepts all possible values with which, evaluating the statement expression/predicate may result in `OUT[var]`.

For a given statement s and a variable var, the language accepted by the DFA `OUT[var]` that is computed by the post transfer function POSTTF$_s$ of statement s can not be larger than the language accepted by the DFA `OUT`$_{max}$`[var]` where `OUT`$_{max}$ is computed by the post transfer function of statement s over Σ^* i.e., `OUT`$_{max}$ = POSTTF$_s(\Sigma^*)$. The reasons are (1) Σ^* represents the largest set that may reach the program point before statement s and (2) the transfer functions that we use are monotonic i.e., for two regular sets X and Y, $X \subseteq Y \Rightarrow$ POSTTF$_s(X) \subseteq$ POSTTF$_s(Y)$. We can utilize this observation to add more precision to our analysis. Before computing the pre-image array `IN` of a statement, we intersect each DFA value in the post-image array `OUT` with the corresponding DFA from `OUT`$_{max}$ (lines 35-40). Since, for a given statement s and a variable var, `OUT`$_{max}$`[var]` represents the largest set of

possible values for var that may reach the program point after statement s, we can be even more precise if we have already computed the *post-image* of the sanitizer function F on the input $(\Sigma^*, \ldots, \Sigma^*)$ using forward analysis (Algorithm 1). In this case, the resulting `OUT` for a statement s represents a smaller, and hence more precise, upper bound (lines 33, 34, 38-40) for values that may reach the program point after s.

3.2.7 Negative Backward Analysis

Given a sanitizer F, we use negative backward analysis (Algorithm 3) to compute an over-approximation of the possible set of input values that are mapped by F to $\perp$ (i.e., $\textsc{Pre}^+_\perp(F)$). A sanitizer F partitions the set of all possible input strings into two sets, the set of rejected inputs I_R and its complement I_A which is the set of accepted inputs. Although this partitioning is exclusively the result of the filtering of invalid string values through validation operations (i.e., branch conditions), this does not mean that sanitization operations do not affect it. Rejecting an input string may not be the direct result of filtering its value v by a branch condition. The presence of sanitization operations complicates this since a sanitizer F, using sanitization operations, may map v to another value v' and then the filtering is applied onto v'. For example, the following sanitizer filters any input string that does not contain a non-space character, indirectly, by first mapping all input strings that consist of only space characters to ϵ and then blocking ϵ.

```
sanitizer(x){
  x = trim(x);
```

```
  if (x == "")
      reject;
  return x;
}
```

So the string "␣␣" which consists of two consecutive white space characters is rejected by first mapping it into ϵ then rejecting ϵ. This means that a sanitizer F maps I_R to the set S_n which is the set of string values that reach the *negative* sinks (i.e., `reject` statements) and maps I_A to the set S_p which is the set of string values that reach the *positive* sink (i.e., `return var` statement). Note that $S_p = \textsc{Post}(F, \Sigma^*)$ and $I_A = \textsc{Pre}(F, S_p)$.

Given this, a straight forward method to compute $I_R^+ \supseteq I_R$ can utilize previous forward and backward algorithms as following:

- First, assuming all possible inputs (i.e., Σ^*), we slightly modify forward analysis to compute an over-approximation of each of the sets of all possible strings that may reach a *negative* sink (i.e., a `reject` statement) instead of the *positive* sink (i.e., the `return var` statement). Let us call these sets of rejected strings $R_1^+, R_2^+, \ldots$. Now we have partitioned the strings that may reach the two types of sinks in the sanitizer function into two sets: the negative sinks set $S_n^+ = \bigcup_{i=1}^{n} R_i^+$ and the positive sink set $S_p^- = \Sigma^* \setminus S_n^+$ where $S_p^- \subseteq S_p$ (since we over-approximate S_n, S_p is under approximated).

- Then, given $R_1^+, R_2^+, \ldots$, we use slightly modified backward analysis that (1) starts from negative sinks instead of the positive sink and (2) uses $R_1^+, R_2^+, \ldots$ as its input, to com-

pute an over-approximation of the set of input strings that may result in each of the sets $R_1^+, R_2^+, \ldots$ reaching a negative sink.

To improve performance, we designed a faster one-phase algorithm instead of the two-phase algorithm above. We call this algorithm negative backward analysis (Algorithm 3) and it is a modified version of the backward analysis shown in Algorithm 2. The algorithm is based on the following insights:

- The set of strings that may reach the positive sink given the set of inputs I_R is $\emptyset$. So, without doing any forward analysis, we start the backward analysis from the positive sink assuming $\emptyset$ as the only set that reaches the program point after `return var` statement as a result of I_R.

- Let us call the set of strings that satisfy a branch condition B the language of B or $L(B)$. Given the set of inputs I_R, the reason that no string reaches the positive sink is because of the filtering by the branch conditions i.e., not satisfying $L(B)$ for some branch condition B. So during backward analysis, to get the value of `IN[var]` for a branch condition B we union the value in $\texttt{OUT}_{on\_T}\texttt{[var]}$ with the negation of $L(B)$ and union the value in $\texttt{OUT}_{on\_F}\texttt{[var]}$ with $L(B)$ (lines 14-21).

So, negative backward analysis algorithm starts with $\emptyset$ at the sink and then computes the input values that do not satisfy branch conditions on any of the computational paths to positive sink `return var` in F.

3.2.8 Transfer Functions

Below we describe the transfer functions used to compute the `OUT` DFA in forward analysis and `IN` DFA for backward analysis for the types of statements in a sanitizer function: *Assignment Statement*, *Halt Statements* and *Conditional Statement*.

Algorithm 4 POSTTRANSFERFUNCTION($node, IN$)

```
1:  OUT := IN;
2:  if (node ≡ VAR := EXP) then
3:      var1 := getVar(VAR);
4:      if (EXP ≡ "<string-literal>") then
5:          OUT[var1] := A({"<string-literal>"});
6:      else if (EXP ≡ VAR) then
7:          var2 := getVar(EXP);
8:          OUT[var1] := IN[var2];
9:      else if (EXP ≡ ⋆) then
10:         OUT[var1] := A(Σ*);
11:     else if (EXP ≡ STRINGFUNC) then
12:         OUT[var1] := POST(STRINGFUNC);
13:     end if
14: else if (node ≡ return Var or node ≡ reject) then
15:     for all var in getVars() do
16:         OUT[var] := A(∅);
17:     end for
18: end if
19: return OUT;
```

Algorithm 5 PRETRANSFERFUNCTION($node, OUT$)

```
1:  IN := OUT;
2:  if (node ≡ VAR := EXP) then
3:      var1 := getVar(VAR);
4:      if (EXP ≡ "<string-literal>" or EXP ≡ VAR or EXP ≡ ⋆) then
5:          IN[var1] := A(Σ*);
6:      else if (EXP ≡ STRINGFUNC) then
7:          IN[var1] := PRE(STRINGFUNC);
8:      end if
9:  else if (node ≡ return Var or node ≡ reject) then
10:     for all var in getVars() do
11:         IN[var] := A(Σ*);
12:     end for
13: end if
14: return IN;
```

Assignment Statement: In this type of statement, variable var on the left-hand side is assigned a value of an expression on the right-hand side. We use the Algorithm 4 (lines 1-13)

to compute the set of string values that an expression may produce in the forward analysis and Algorithm 5 (lines 1-8) to compute the set of string values that the expression may take to produce a certain value in the backward analysis. Algorithm 4 takes two inputs: the node which contains the expression on the right-hand side of the assignment and the `IN` DFA of the assignment statement where the expression is. Algorithm 5 takes two inputs: the node which contains the expression on the right-hand side of the assignment and the `OUT` DFA of the assignment statement where the expression is. The two algorithms evaluate the expressions on the right hand side as follows:

- *string-literal*: for post transfer function a singleton set that only contains the value of the *string-literal* is returned (i.e., a DFA that recognizes only the *string-literal*). For pre transfer function variable is assigned Σ^* since we do not know what its value was before the assignment.

- *Var*: for post transfer function we copy the set that represents all possible values for the right hand side variable into the left hand side variable. For pre transfer function left hand side variable is assigned Σ^* since we do not know what its value was before the assignment.

- $\star$: since our analysis is concerned only with analyzing string values, $\star$ is used to represent non-string expressions. In these cases we return $\top = \Sigma^*$ indicating that the value of the expression is not known.

- *StringFunc*: in this case we use the specific post and pre image computation algorithm for the function. Figure 3.8 shows a brief description of the post and pre images for the two common string functions **concatenate** and **replace**. For full details on the algorithms used to compute the post and pre images of these two functions see [106] [104].

  – **concatenate**(*expression1*, *expression2*): In this case, POSTCONCAT computes the concatenation of the regular languages resulting from evaluating *expression1* and *expression2* and returns it as the result (using the symbolic DFA concatenation operation discussed in [106]). PRECONCATPREFIX computes the prefix pre-image language while PRECONCATSUFFIX computes the suffix pre-image language using the pre-image computation algorithm discussed in [104].

  – **replace**(*pattern1, pattern2, s*): POSTREPLACE computes the result of replacing all string values in `IN` that match the *pattern1* (given as a regular expression) with all string values regular expression *pattern2*. There are two types of pattern matching: **partial match** and **full match**. The match operation used is chosen based on the *pattern1* value as follows. 1) If the value starts with symbol "^" and ends with symbol "$", a full match must be performed, that is, string in `IN` should be replaced only if it fully matches the regular expression in the *pattern1*. This is accomplished by taking the difference between the language in `IN` and the language $L(pattern1)$ and adding the language $L(pattern2)$ to the result. 2) In all other cases, a partial match is performed, where the result is computed by using the language-based

POSTCONCAT(DFA M_1, DFA M_2): $\equiv \mathcal{A}(\{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\})$.

POSTREPLACE(DFA M_1, DFA M_2, DFA M_3)$\equiv$
$\mathcal{A}(\{w_1c_1w_2c_2 \ldots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \ldots w_kx_kw_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\})$.

PRECONCATPREFIX(DFA M, DFA M_2) returns a DFA M_1 so that $M = \text{CONCAT}(M_1, M_2)$.

PRECONCATSUFFIX(DFA M, DFA M_1) returns a DFA M_2 so that $M = \text{CONCAT}(M_1, M_2)$.

PREREPLACE(DFA M, M_2, M_3) returns a DFA M_1 so that $M = \text{REPLACE}(M_1, M_2, M_3)$.

Figure 3.8: Definition of post- and pre-images of the two most common string functions namely *concatenation* and *replace*.

replacement algorithm described in [106]. PREREPLACE computes the pre-image for case 1 by adding the language $L(\textit{pattern1})$ to `IN` without removing the language $L(\textit{pattern2})$ since we are not sure that this was added by the replace operation. For case 2 pre-image is computed using the replace pre-image computation algorithm discussed in [104].

In addition to the general replace operation above, we also implemented a number of specialized automata replace operations to model some string sanitization operations such as `trim`, `htmlspecialchars`, `addslashes` and `mysql_real_escape_string`, The details for these operations are shown in 3.3.

Halt statements: for the two halt statements `return var` and `reject`, algorithm 4 (lines 14-17) returns $\bot = \emptyset$ for post transfer function since halting execution kills all values. On the other hand algorithm 5 (lines 9-13) returns $\top = \Sigma^*$ for pre transfer function. This indicates that, going backwards, a variable can have any possible string value as its value at the program point before a halt statement regardless of its possible values at the program point after the halt

statement. The reason is that a halt statement stops execution which means that the values at the program points after it are not the result of the values at the program point before it.

Algorithm 6 EVALPRED(*Pred*, M_{var})

```
1:  if Pred ≡ Pred1 && Pred2 then
2:      return EvalPred(Pred1) ∩ EvalPred(Pred2);
3:  else if Pred ≡ Pred1 || Pred2 then
4:      return EvalPred(Pred1) ∪ EvalPred(Pred2);
5:  else if Pred ≡ !Pred1 then
6:      return Σ* - EvalPred(Pred1);
7:  else if Pred ≡ (Pred ) then
8:      return EvalPred(Pred);
9:  else if Pred ≡ Var RelOp "<string-literal>" then
10:     return EvalVarRelOpLit(RelOp, "<string-literal>");
11: else if Pred ≡ Var matches RegExp then
12:     return EvalLangMembership(Pred);
13: else if Pred ≡ StringFunc RelOp "<string-literal>" then
14:     return StringFuncModelForPred(param1, param2, ..., M_var, RelOp, "<string-literal>");
15: else if Pred ≡ IntFunc RelOp <integer-literal> then
16:     return IntFuncModelForPred(param1, param2, ..., RelOp, <integer-literal>);
17: else
18:     return A(Σ*);
19: end if
```

Conditional Statement: Conditional statements consist of a predicate on variables and constants. Because they represent a branch in the program, unlike other statements, they are followed by two statements, one on the `ON_TRUE` branch and the other on the `ON_FALSE` branch. If the predicate evaluates to true, the execution will continue in the `ON_TRUE` branch. Otherwise, it will take the `ON_FALSE` branch. This behavior is represented in our analysis by having two `OUT` DFAs reflecting the possible future values on each of the two branches of execution. $\texttt{OUT}_{on\_T}$ represents the values for the `ON_TRUE` branch, and $\texttt{OUT}_{on\_F}$ represents the values for the `ON_FALSE` branch. In order to compute these DFAs, our analysis first computes, using EVALPRED algorithm 6, DFA_T—the DFA that accepts the set of string values that would make the predicate evaluate to true [3]. Then, the algorithm computes the $\texttt{OUT}_{on\_T}$ DFA by inter-

secting the IN DFA with DFA$_T$. Conversely, to compute the OUT$_{on_F}$, our algorithm intersects the IN DFA with the complement of DFA$_T$.

Algorithm 6 recursively traverses the predicate while constructing the DFA for each subexpression in the predicate. Logical operations are handled using automata union, intersection and complement operations, while all other expressions are mapped to regular expressions [3]. In case of nondeterministic predicate $\star$, if the predicate is alone of if it is disjoined with an-

Algorithm 7 EVALVARRELOPLIT(*RelOp, strlit*)

if *RelOp* $\equiv$ **==** **then**
  $M_r := \mathcal{A}(\{strlit\})$;
else if *RelOp* $\equiv$ **!=** **then**
  $M_r := \mathcal{A}(\Sigma^*) \setminus \mathcal{A}(\{strlit\})$;
else
  $M_r := \mathcal{A}(\emptyset)$;
  for $i := 0 \rightarrow len(strlit)$ **do**
    $M_{t_1} = \mathcal{A}(\{\epsilon\})$;
    for $j := 0 \rightarrow i - 1$ **do**
      $M_{t_1} := M_{t_1} \odot \mathcal{A}(\{strlit[j]\})$;
    end for
    $j := i$;
    $M_{t_2} := \mathcal{A}(\emptyset)$;
    for all $c \in \Sigma$ **do**
      if ((*RelOp* $\equiv$ **<** or *RelOp* $\equiv$ **<=**) and $c \prec strlit[j]$) or ((*RelOp* $\equiv$ **>** or *RelOp* $\equiv$ **>=**) and $c \succ strlit[j]$) **then**
        $M_{t_2} := M_{t_2} \cup \mathcal{A}(\{c\})$;
      end if
    end for
    $M_{t_1} := M_{t_1} \odot M_{t_2}$;
    if $i == len(strlit)$ **then**
      if *RelOp* $\equiv$ **<=** or *RelOp* $\equiv$ **>=** **then**
        $M_{t_1} := temp1 \odot \Sigma^*$
      else
        $M_{t_1} := M_{t_1} \odot \Sigma^+$;
      end if
    else
      $M_{t_1} := M_{t_1} \odot \Sigma^*$;
    end if
    $M_r := M_r \cup M_{t_1}$;
  end for
end if
return M_r;

Algorithm 8 EVALLANGMEMBERSHIP(*RegExp*)

if check_regexp(*RegExp*) = partial_match **then**
  return $\mathcal{A}(\Sigma^*) \odot \mathcal{A}(RegExp) \odot \mathcal{A}(\Sigma^*)$;
else
  return $\mathcal{A}(RegExp)$;
end if

other predicate then EVALPRED returns Σ^* indicating that we lose path sensitivity as we can not guarantee the condition on which we took a specific branch of execution. On the other hand, if the nondeterministic predicate is conjoined with another predicate then for the `ON_TRUE` branch we just return the result for the other predicate while for the `ON_FALSE` branch we re-

Algorithm 9 `substring`MODELFORPRED(*intlit_begin, intlit_end, RelOp, strlit*)

1: i := intlit_begin; j := intlit_end;
2: $M_{prefix} := \mathcal{A}(\Sigma^i)$;
3: **if** *RelOp* $\equiv$ **==** **then**
4: **if** len(strlit) $neqj - i$ **then**
5: $M_r := \emptyset$;
6: **else**
7: $M_r := M_{prefix} \odot \mathcal{A}(\{strlit\}) \odot \mathcal{A}(\Sigma^*)$;
8: **end if**
9: **else if** *RelOp* $\equiv$ **!=** **then**
10: $M_r := \mathcal{A}(\Sigma^*) \setminus (M_{prefix} \odot \mathcal{A}(\{strlit\}) \odot \mathcal{A}(\Sigma^*))$;
11: **else**
12: $M_r := \mathcal{A}(\emptyset)$;
13: **if** len(strlit) $\geq (j - i)$ **then**
14: $M_r := \emptyset$;
15: **for** $i := 0 \rightarrow j - i$ **do**
16: $M_{t_1} = \mathcal{A}(\{\epsilon\})$;
17: **for** $j := 0 \rightarrow i - 1$ **do**
18: $M_{t_1} := M_{t_1} \odot \mathcal{A}(\{strlit[j]\})$;
19: **end for**
20: $j := i$;
21: $M_{t_2} := \mathcal{A}(\emptyset)$;
22: **for all** $c \in \Sigma$ **do**
23: **if** ((*RelOp* $\equiv$ **<** or *RelOp* $\equiv$ **<=**) and $c \prec strlit[j]$) or ((*RelOp* $\equiv$ **>** or *RelOp* $\equiv$ **>=**) and $c \succ strlit[j]$) **then**
24: $M_{t_2} := M_{t_2} \cup \mathcal{A}(\{c\})$;
25: **end if**
26: **end for**
27: $M_{t_1} := M_{t_1} \odot M_{t_2}$;
28: **if** $i == len(strlit)$ **then**
29: **if** *RelOp* $\equiv$ **<=** or *RelOp* $\equiv$ **>=** **then**
30: $M_{t_1} := temp1 \odot \Sigma^*$
31: **else**
32: $M_{t_1} := M_{t_1} \odot \Sigma^+$;
33: **end if**
34: **else**
35: $M_{t_1} := M_{t_1} \odot \Sigma^*$;
36: **end if**
37: $M_r := M_r \cup M_{t_1}$;
38: **end for**
39: $M_r := M_{prefix} \odot M_r \odot \mathcal{A}(\Sigma^*)$
40: **end if**
41: **end if**
42: **return** M_r;

turn Σ^*. Recursive evaluation of a predicate stops when reaching one of the following 4 atomic predicates:

Algorithm 10 `length`MODELFORPRED(*RelOp, intlit*)

if *RelOp* $\equiv$ **==** **then**
  return $\mathcal{A}(\Sigma^{intlit})$;
else if *RelOp* $\equiv$ **>** **then**
  return $\mathcal{A}(\Sigma^*) \setminus \bigcup_{l=0}^{intlit} \mathcal{A}(\Sigma^l)$;
else if *RelOp* $\equiv$ **>=** **then**
  return $\mathcal{A}(\Sigma^*) \setminus \bigcup_{l=0}^{intlit-1} \mathcal{A}(\Sigma^l)$;
else if *RelOp* $\equiv$ **<** **then**
  return $\bigcup_{l=0}^{intlit-1} \mathcal{A}(\Sigma^l)$;
else if *RelOp* $\equiv$ **<=** **then**
  return $\bigcup_{l=0}^{intlit} \mathcal{A}(\Sigma^l)$;
else if *RelOp* $\equiv$ **>** **then**
  return $\mathcal{A}(\Sigma^*) \setminus \mathcal{A}(\Sigma^{intlit})$;
end if

Algorithm 11 `indexof`MODELFORPRED(*RelOp, intlit*)

strlit = <string-literal>;
if *RelOp* $\equiv$ **==** **then**
  if *intlit* $=$ -1 **then**
    return $\mathcal{A}((\Sigma \setminus \{strlit[0]\})^*)$;
  else if *intlit* ≥ 0 **then**
    return $\mathcal{A}(\Sigma^{intlit-1}) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  end if
else if *RelOp* $\equiv$ **>=** **then**
  if *intlit* $=$ -1 **then**
    return $\mathcal{A}(\Sigma^*) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  else
    return $\mathcal{A}((\Sigma \setminus \{strlit[0]\})^{intlit}) \odot \mathcal{A}(\Sigma^*) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  end if
else if *RelOp* $\equiv$ **>** **then**
  return $\mathcal{A}((\Sigma \setminus \{strlit[0]\})^{intlit+1}) \odot \mathcal{A}(\Sigma^*) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
else if *RelOp* $\equiv$ **<=** **then**
  if *intlit* = -1 **then**
    return $\mathcal{A}(\Sigma \setminus \{strlit[0]\})^*)$;
  else
    return $\bigcup_{l=0}^{intlit} \mathcal{A}(\Sigma^l) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  end if
else if *RelOp* $\equiv$ **<** **then**
  if *intlit* = 0 `or` *intlit* = -1 **then**
    return $\mathcal{A}((\Sigma \setminus \{strlit[0]\})^*)$;
  else
    return $\bigcup_{l=0}^{intlit-1} \mathcal{A}(\Sigma^l) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  end if
else if *RelOp* $\equiv$ **!=** **then**
  if *intlit* = -1 **then**
    return $\mathcal{A}(\Sigma^*) \odot \mathcal{A}(\{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  else if *intlit* = 0 **then**
    return $\mathcal{A}(\Sigma \setminus \{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  else
    return $\mathcal{A}(\Sigma^{intlit-1}) \odot \mathcal{A}(\Sigma \setminus \{strlit[0]\}) \odot \mathcal{A}(\Sigma^*)$;
  end if
end if

- *Var RelOp "<string-literal>"* (lines 9,10): this represents a lexicographical comparison between the string value of the left hand side variable and the value of <string-literal>. Algorithm 7 shows how to evaluate this predicate.

- *Var* **`matches`** *RegExp* (lines 11,12): this predicate tests if a variable is a member of the language of a regular expression. Algorithm 8 shows how to evaluate this predicate.

- *StringFunc RelOp "<string-literal>"* (lines 13,14): this predicate calls the specific algorithm for the given StringFunction. Algorithm 9 shows how to evaluate predicates that use `substring` function.

- *IntFunc RelOp "<integer-literal>"* (lines 15,16): this predicate calls the specific algorithm for the given IntFunction. Algorithms 10 and 11 show how to evaluate predicates that use `length` and `indexof` functions.

Notice that we only call EVALPRED to evaluate a predicate if it contains a single variable (see algorithm 1: lines 25-30, algorithm 2: lines 22-28, and algrotihm 3: lines 16-21). To handle branch conditions with multiple variables we need relational string analysis [110].

After evaluating the predicate we then compute post-image of the conditional statement in the case of forward analysis and pre-image in the case of backward analysis. In forward analysis (algorithm 1) we compute $\texttt{OUT}_{on\_T}\texttt{[var]}$ by intersecting result of EVALPRED with the value of `IN[var]` and compute value of $\texttt{OUT}_{on\_F}\texttt{[var]}$ by intersecting the complement of the result of EVALPRED with the value of `IN[var]`. This reflects the fact that the possible values that the variable var can take will be filtered (i.e., constrained by) the predicate on true

branch and its negation on false branch. In the case of backward analysis, there are two ways to compute the pre-image of conditional statements. In algorithm 2 `IN[var]` DFA gets the union of the values of $\texttt{OUT}_{on\_T}$ and $\texttt{OUT}_{on\_F}$ after they are *intersected* with the (complement of) result of EVALPRED. The reason here is that we are trying to compute the set of input string values that are mapped to a given set of output string values by the sanitizer function. This set is the set that is not mapped by the sanitizer function to $\perp$ but rather to $S_p \subseteq \Sigma^*$ which means that it satisfies all branch conditions on all paths to positive sink (i.e., `return var` statement) program point. For algorithm 3 the situation is different and we need to *union* with the (complement of) result of EVALPRED as we explained before.

3.3 Specialized Replace Algorithms

To increase the precision and performance of our analysis, we developed [2] a number of automata-based algorithms for computing the pre and post-images of frequently used string operations such as `trim`, `htmlspecialchars`, `addslashes`, `mysql_real_escape_string`, `tolower`, and `toupper`. These algorithms are more precise and more efficient than using the general replace algorithm to model these specialized operations [106]. The general replace algorithm consumes significantly more time and space since it relies on automata determinization which requires the use of subset construction algorithm which has an exponential complexity. On the other hand, the specialized algorithm we developed for the ESCAPE operation, for example,

runs in linear time and its result is precise without any over-approximation. In the experimental results we demonstrate the improvement that we gain from these specialized algorithms.

Below we describe the post-image of four of the specialized replace operations that we implemented.

Definitions

POSTESCAPE(DFA M_1, char e, charset E): this automata operation escapes characters in E along with the escape character e itself in all strings in $\mathcal{L}(M_1)$ using the escape character e. It returns a DFA M such that $\mathcal{L}(M) = \{\ w_1ec_1w_2ec_2\ \ldots\ w_kec_kw_{k+1} \mid k > 0, w_1c_1w_2c_2 \ldots w_kc_kw_{k+1} \in \mathcal{L}(M_1), \forall i : c_i \in E \cup \{e\}$ and $w_i \in \Sigma^* - (E \cup \{e\})^*\}$. An example of an escape function is PHP's `addslashes`. Notice that ESCAPE escapes all chars $c \in \{e\} \cup E$ without checking if they have already been escaped before. This may result in double escaping i.e. w_1eew_2 will become w_1eeeew_2.

POSTTRIMLEFT(DFA M_1, char s): this automata operation removes all the space characters s from the beginning of strings in $\mathcal{L}(M_1)$ up to the first character that is not equal to s. It returns a DFA M such that $\mathcal{L}(M) = \{c_1w \mid w_1c_1w \in \mathcal{L}(M_1), w_1 \in \{s\}^*$ and $w \in \Sigma^*$ and $c_1 \in \Sigma - \{s\}\}$.

POSTTRIMRIGHT(DFA M_1, char s): this automata operation removes all the s characters from the end of strings in $\mathcal{L}(M_1)$ going in backward (i.e., reverse) direction starting from the end up to the first character that is not equal to s. It returns a DFA M such that $\mathcal{L}(M) = \{wc_1 \mid wc_1w_1 \in \mathcal{L}(M_1), w_1 \in \{s\}^*$ and $w \in \Sigma^*$ and $c_1 \in \Sigma - \{s\}\}$.

POSTREPLACECHAR(DFA M_1, char r, String w): this automaton operation replaces a single char r with a string w in all strings in $\mathcal{L}(M_1)$. It returns a DFA M such that $\mathcal{L}(M) = \{w_1 w w_2 w \dots \quad w_k w w_{k+1} \quad | \quad k \quad > \quad 0, \quad w_1 r w_2 r \dots w_k r w_{k+1} \quad \in \quad \mathcal{L}(M_1), w_i \in (\Sigma - \{r\})^*\}$. This operation can be used to model string operations such as PHP's `htmlspecialchars` efficiently.

POSTREPLACECHARSETWITHCHARSET(DFA M_1, Relation $\mathcal{R}$): in this automaton operation we are given a relation (partial function) $\mathcal{R} \subseteq \Sigma \times \Sigma$ that maps a set of characters to another set of characters where a character does not appear at the same time in the domain and the range of the relation and each character can be mapped by $\mathcal{R}$ to one and only one character. Formally, $\forall c_1, c_2$ and $c_3 \in \Sigma$, $(c_1, c_2) \in \mathcal{R}$ and $(c_1, c_3) \in \mathcal{R} \Rightarrow c_2 = c_3$ and $\forall c$ and $c' \in \Sigma$, if $(c, c') \in \mathcal{R}$ then $\nexists c'' \in \Sigma$ such that $(c'', c) \in \mathcal{R}$.

This automaton operation replaces each character c in the domain of $\mathcal{R}$ with its counterpart in the range of $\mathcal{R}$ in all strings in $\mathcal{L}(M_1)$. It returns a DFA M such that $\mathcal{L}(M) = \{w_1 c'_1 w_2 c'_2 \dots w_k c'_k w_{k+1} \mid k > 0, w_1 c_1 w_2 c_2 \dots w_k c_k w_{k+1} \in \mathcal{L}(M_1), \forall i \leq k (c_i, c'_i) \in \mathcal{R}\}$. This can be used to model PHP operations such as `tolower` and `toupper` that change all characters in input string to lower or upper case.

Algorithms

POSTESCAPE(M_1, e, E): Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ the result DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ is constructed as follows: For each state q_i that has at least one out transition ($q_i \xrightarrow{c} q_j$) on a character $c \in \{e\} \cup E$ (1) we mark each transition ($q_i \xrightarrow{c} q_j$) out

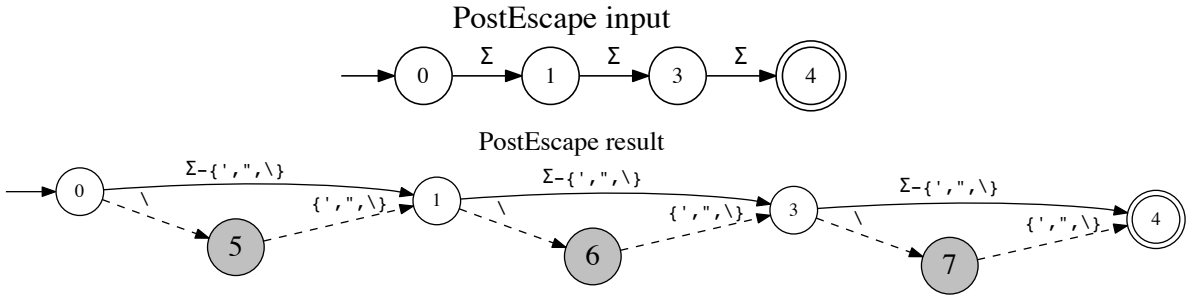

Figure 3.9: Example of applying POSTESCAPE to DFA M_1 where $L(M_1) = \Sigma^3$ to escape the characters `'` and `"` with character \.

from q_i to a state q_j, (2) we add a new state q'_k and a new transition ($q_i \xrightarrow{e} q'_k$) on escape character e, (3) we move each marked transition ($q_i \xrightarrow{c} q_j$) to become a transition ($q'_k \xrightarrow{c} q_j$). The resulting automaton does not have nondeterminism which means that we avoid the use of subset construction algorithm for determinization.

Formally, PostEscape-DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ can be constructed as follows:

- $Q = Q_1 \cup Q', Q' = \{q'_i \mid \forall i \geq 0 : \exists\, \alpha \in E_{\mathcal{B}} \cup \{\alpha_e\} : q_i \in Q_1$ and $\delta_1(q_i, \alpha) \neq sink\}$.
- $F = F_1$.
- $\forall q \in Q, \forall \alpha \in \Sigma_{\mathcal{B}} \setminus E_{\mathcal{B}} \cup \{\alpha_e\} : \delta(q, \alpha) = \delta_1(q, \alpha)$

  $\forall i \geq 0 : (\exists\, \alpha \in E_{\mathcal{B}} \cup \{\alpha_e\} : \delta_1(q_i, \alpha) \neq sink) \Rightarrow (\delta(q_i, \alpha_e) = q'_i$ and $\forall \alpha \in E_{\mathcal{B}} \cup \{\alpha_e\} :$

  $\delta(q'_i, \alpha) = \delta_1(q_i, \alpha))$.

Example. Figure 3.9 shows the result of applying POSTESCAPE to DFA M_1 where $\mathcal{L}(M_1) = \Sigma^3$ to escape the characters $E = \{$`'`, `"`$\}$ with escape character $e =$ \ which models the seman-

tics of function `addslashes` in PHP. The shaded states are the newly added states and dash lines are newly added transitions.

Notice that, given any two states, instead of drawing one edge between the two states for each transition (on a character in Σ) between the two states, we draw a single edge between these two states and label it with the set of characters that have transitions between the two states. In other words, each edge represents n transitions in the transition relation where n is the number of characters in the character set labeling the edge. For example, $(S_0 \xrightarrow{\Sigma} S_1)$ in M_1 represents 256 transitions between states S_0 and S_1, one transition per each ASCII character in Σ.

(1) First, we mark in M_1 transitions $(S_0 \xrightarrow{'} S_1)$, $(S_0 \xrightarrow{"} S_1)$, $(S_0 \xrightarrow{\backslash} S_1)$, $(S_1 \xrightarrow{'} S_3)$, $(S_1 \xrightarrow{"} S_3)$, $(S_1 \xrightarrow{\backslash} S_3)$, $(S_3 \xrightarrow{'} S_4)$, $(S_3 \xrightarrow{"} S_4)$, $(S_3 \xrightarrow{\backslash} S_4)$. (2) Then we add states S_5, S_6, S_7 and transitions $(S_0 \xrightarrow{\backslash} S_5)$, $(S_1 \xrightarrow{\backslash} S_6)$, $(S_3 \xrightarrow{\backslash} S_7)$. (3) Finally, we move transitions $(S_0 \xrightarrow{'} S_1)$, $(S_0 \xrightarrow{"} S_1)$, $(S_0 \xrightarrow{\backslash} S_1)$ to $(S_5 \xrightarrow{'} S_1)$, $(S_5 \xrightarrow{"} S_1)$, $(S_5 \xrightarrow{\backslash} S_1)$ and move transitions $(S_1 \xrightarrow{'} S_3)$, $(S_1 \xrightarrow{"} S_3)$, $(S_1 \xrightarrow{\backslash} S_3)$ to become $(S_6 \xrightarrow{'} S_3)$, $(S_6 \xrightarrow{"} S_3)$, $(S_6 \xrightarrow{\backslash} S_3)$ and move transitions $(S_3 \xrightarrow{'} S_4)$, $(S_3 \xrightarrow{"} S_4)$, $(S_3 \xrightarrow{\backslash} S_4)$ to become $(S_7 \xrightarrow{'} S_4)$, $(S_7 \xrightarrow{"} S_4)$, $(S_7 \xrightarrow{\backslash} S_4)$.

Notice that adding transitions $(S_0 \xrightarrow{\backslash} S_5)$, $(S_1 \xrightarrow{\backslash} S_6)$, $(S_3 \xrightarrow{\backslash} S_7)$ did not introduce nondeterminism since we moved transitions $(S_0 \xrightarrow{\backslash} S_1)$, $(S_1 \xrightarrow{\backslash} S_3)$, $(S_3 \xrightarrow{\backslash} S_4)$ from states S_0, S_1, and S_3. This shows how critical it is to have the escape e character being at the same time an escaped character.

PREESCAPE(M_1, e, E) implementation: Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ we first preprocess M_1 to partition all transitions $(q \xrightarrow{e} q')$ into two sets of transitions: Escaping transi-

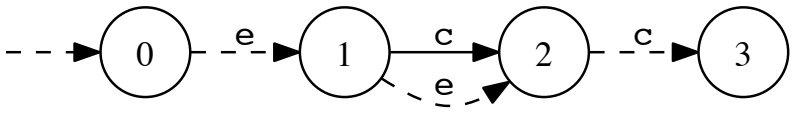

Figure 3.10: Example of double escaping that happens if an escape character e on transition $(1 \xrightarrow{e} 2)$ is escaping c and being escaped by another e.

tions T_g and escaped transitions T_e such that: $\forall q, q' \in Q_1 : (q \xrightarrow{e} q') \in T_e \Rightarrow \exists q'' \in Q_1$: $(q'' \xrightarrow{e} q) \in T_g$.

Notice that in M_1, a transition $(q \xrightarrow{e} q')$ can not be escaping and at the same time being escaped, i.e., $T_e \cap T_g = \emptyset$. Otherwise we will have a string $w_1 eecw_2 \in \mathcal{L}(M_1)$ where $w_1, w_2 \in \Sigma^*$ which contradicts the definition of POSTESCAPE(M_1, e, E). Figure 3.10 shows part of a DFA where character c has to be always escaped by e. Notice that $(1 \xrightarrow{e} 2)$ is escaping and at the same time being escaped. This gives us the path in dashed lines that would result in double escaping. We can formalize this condition as follows: There is no path $q_0, \ldots, q_{i-1}, q_i, q_{i+1}, q_{i+2}, \ldots, q_f$ in M_1 where $q_f \in F, \delta(q_{i-1}, e) = q_i$, $\delta(q_i, e) = q_{i+1}$ and $\delta(q_{i+1}, c) = q_{i+2}$ where $c \in \{e\} \cup E$. During the analysis, we enforce this condition by applying PREESCAPE on $M_1 \cap$ ESCAPE$(\mathcal{A}(\Sigma^*), e, E)$. Using the same reasoning we conclude that (1)$\forall (q \xrightarrow{e} q') \in T_g : q' \notin F$, (2) $\forall c \notin \{e\} \cup E,\ \forall (q \xrightarrow{e} q') \in T_g : \delta(q', c) = sink$, (3) $\forall q \in Q_1 (\forall c \in \{e\} \cup E : \delta(q, c) \neq sink \Leftrightarrow \forall c' \notin \{e\} \cup E : \delta(q, c') = sink)$, and (4)$\delta(q_0, e) \neq sink \Rightarrow (q_0 \xrightarrow{e} q') \in T_g$.

Using these results, we compute T_g and T_e using a depth first traversal starting from the start state q_0. We then compute the set of *escaped states* Q_e which is the set of states that has all input transitions in T_g. Due to the preconditions we stated earlier, all transitions on e

must be either coming into an escaped state or going out from it. Also all transitions on escaped characters $c \in \{E\}$ must be going out from an escaped state. Taking the result of POSTESCAPE in Figure 3.9 as an example input to PREESCAPE, $T_g = \{(S_0 \xrightarrow{\setminus} S_5), (S_1 \xrightarrow{\setminus} S_6), (S_3 \xrightarrow{\setminus} S_7)\}$, $T_e = \{(S_5 \xrightarrow{\setminus} S_1), (S_6 \xrightarrow{\setminus} S_3), (S_7 \xrightarrow{\setminus} S_4), (S_5 \xrightarrow{'} S_1), (S_6 \xrightarrow{'} S_3), (S_7 \xrightarrow{'} S_4), (S_5 \xrightarrow{"} S_1), (S_6 \xrightarrow{"} S_3), (S_7 \xrightarrow{"} S_4)\}$ and $Q_e = \{S_5, S_6, S_7\}$.

Finally, given Q_e, we construct the new DFA by removing all states $q_k \in Q_e$ such that: (1) all transitions $(q_i \xrightarrow{e} q_k)$ are removed, and (2) each transition $(q_k \xrightarrow{c} q_j)$ is added, as an out transition, to all states q_i where a transition $(q_i \xrightarrow{e} q_k)$ was removed. Based on the conditions we discussed above on M_1, this last step can be done without determinization and subset construction.

Finally PreEscape-DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ can be constructed as follows.

- $Q = Q_1 - Q_e$.
- $F = F_1$.
- $\forall q \in Q, \forall \alpha \in \Sigma_{\mathcal{B}} - E_{\mathcal{B}} \cup \{\alpha_e\} : \delta(q, \alpha) = \delta_1(q, \alpha)$.
- $\forall q \in Q, \forall \alpha \in \{\alpha_e\} \cup E_{\mathcal{B}} : \delta(q, \alpha) = q' \in Q$ if $\delta_1(q, \alpha_e) = q''$ and $\delta_1(q'', \alpha) = q'$.

The two algorithms above are precise as they do not over-approximate the result. Additionally, they are linear in the size of the input DFA and avoid subset construction which makes them very fast compared to the general replace algorithm.

POSTTRIMLEFT(M_1, s): Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ our goal is to construct a DFA that would accept w if $w_1 w \in L(M_1)$ where $w_1 \in s^*$. We start first by marking all states

q_i reachable from start state q_0 on a string $w \in s^*$ where s is the char to be trimmed. Given a DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$, a state $q_m \in Q$ and a char $s \in \Sigma$, let us define function $s\text{-}reach : Q \leftarrow 2^Q$ as following:

$$s\text{-}reach(q_m) = \{q_n\text{: there exists a path } q_m, q_{m+1}, \ldots, q_n \text{ such that}$$
$$\forall\, 0 \leq i,\ (q_{m+i}, s, q_{m+i+1}) \in \delta\}$$

Notice that $q_m \in s\text{-}reach(q_m)$ since q_m is reachable from itself on $\epsilon = s^*$. We say that a state q_n is s-reachable from a state q_m if $q_n \in s\text{-}reach(q_m)$. The first step in our algorithm is to mark all states that are s-reachable from start state q_0. Intuitively, we try to add to the language accepted by the new DFA all substrings w that (1) do not start with s and (2) can be accepted by M_1 starting from a state $q_m \in s\text{-}reach(q_0)$. We can achieve this by adding a new start state q_{init} and then for all transitions $(q_i \xrightarrow{c} q_j)$ where q_i is reachable from q_0 on s ($q_i \in s\text{-}reach(q_0)$) and c is not s ($c \in \Sigma \setminus \{s\}$), we copy that transition to get a new transition $(q_{init} \xrightarrow{c} q_j)$. Since adding two transitions to q_{init} on the same alphabet symbol α to two different states will introduce non-determinism, we need to simulate this non-determinism because symbolic automaton does not allow non-deterministic transitions (i.e., two transitions on the same character out from the same state) or ϵ-transitions. This simulation is done by adding extra bits to $\Sigma_{\mathcal{B}}$ as we explained before. In our case here, the worst case scenario is that we find an outgoing transition on a character $c \neq s$ out from each state $q_i \in s\text{-}reach(q_0)$ (i.e.$\exists c \in \Sigma \setminus \{s\}$ such that $\forall q_i \in s\text{-}reach(q_0) \exists q_j$ such that $(q_i \xrightarrow{c} q_j)$). Since we need to copy all these transitions as outgoing transitions from the new start state q_{init}, we need to simulate this non-determinism symbolically to differentiate between all of these new transitions on the same

character c (i.e., to let each pair of the new copied transitions have different alphabet symbol on it). This can be achieved by adding extra bits to each alphabet symbol as following: let n be the size of $s\text{-}reach(q_0)$ (i.e., $n = |s\text{-}reach(q_0)|$). Given $m = \lceil log_2(n) \rceil$, $\forall \alpha \in \Sigma_{\mathcal{B}}$, $\forall q \in s\text{-}reach(q_0)$ we define a new symbol αm_q where $m_q \in \mathcal{B}^m$. Adding extra bits to an alphabet symbol α_c allows us to have multiple versions $c_0, c_1, c_2, \ldots$ of the corresponding character c where we can use two different versions for each pair of transitions outgoing from the same state on c. We then construct an intermediate DFA $M' = \langle Q', q'_0, \Sigma'_{\mathcal{B}}, \delta', F' \rangle$ as follows:

- $\Sigma'_{\mathcal{B}} \subseteq \mathcal{B}^{k+m}$ where $m = \lceil log_2(n) \rceil$.
- $Q' = Q_1 \cup \{q_{init}\}$.
- $F' = F_1$ if $F_1 \cap s\text{-}reach(q_0) = \phi$ otherwise $F' = F_1 \cup \{q_{init}\}$.
- $\delta(q_{init}, \alpha m_q) = q' \in Q'$ if $\alpha \neq \alpha_s$ and $\exists q \in s\text{-}reach(q_0)$ such that $\delta_1(q, \alpha) = q'$.
- $\forall q \in Q_1 : \delta(q, \alpha m_0) = \delta_1(q, \alpha)$ where $m_0 = 0^m$.

Finally we determinize the symbolic DFA M' by projecting all extra bits. Formally, we construct $M = \forall 1 \leq i \leq m$, PROJECT($M', k + i$).

Example. Figure 3.11 shows an example of running POSTTRIMLEFT on the input symbolic DFA shown on the top left corner. For each step of the algorithm, we show a symbolic DFA with symbolic alphabet along with another one with non-symbolic alphabet to give an idea of how we actually implement our algorithms using symbolic automata representation. In the following discussion we will explain the example using the symbolic DFAs. We will use the

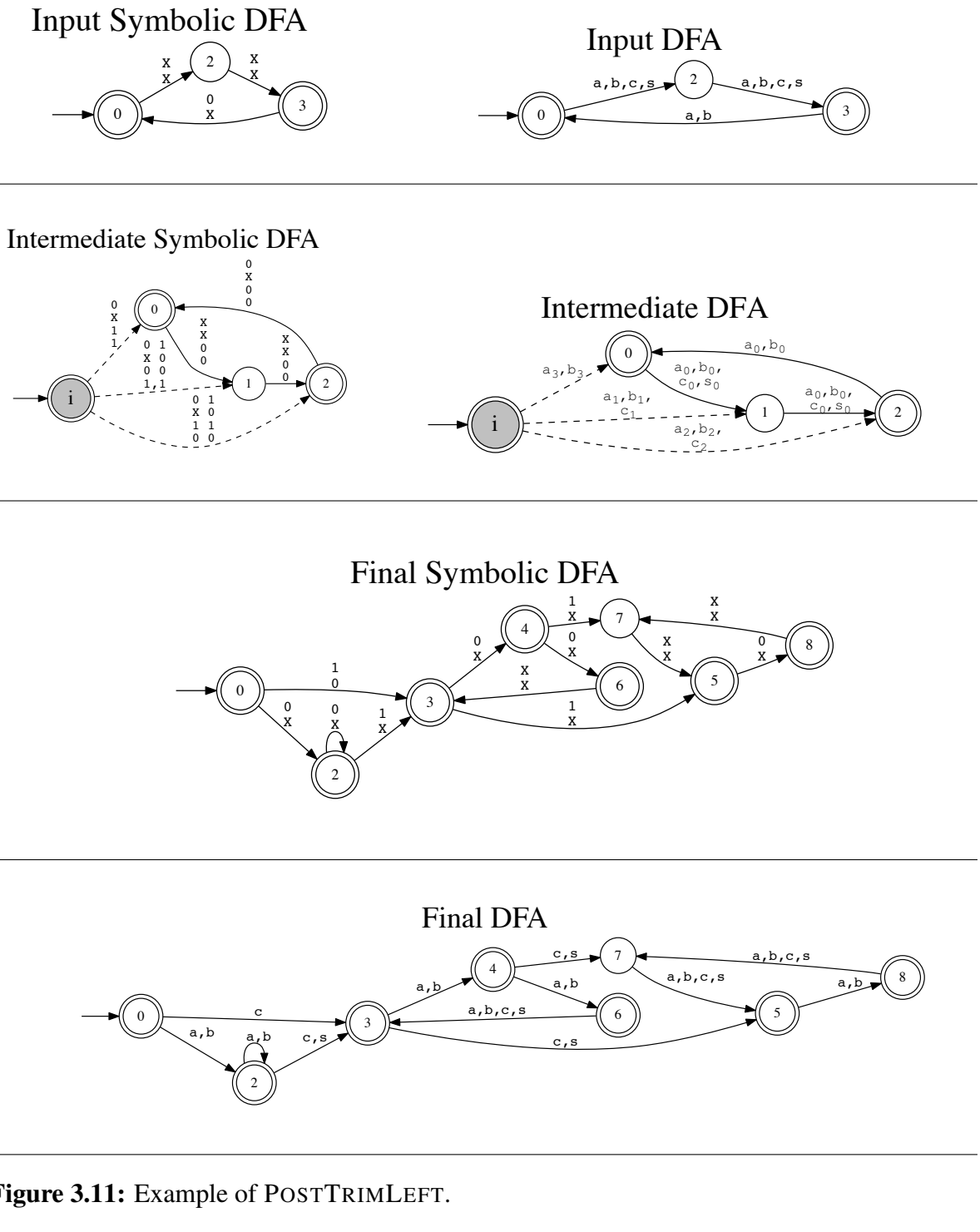

Figure 3.11: Example of POSTTRIMLEFT.

prefix S before each state number to refer to a certain state in one of DFAs in the example. For example, S_0 means state 0 (the initial state in input DFA on top left corner) while S_i means state i (the initial state in intermediate DFA on top right corner). Since we want to give a simple example to explain the algorithm, we used the small alphabet $\Sigma = \{a, b, c, s\}$ where each character $c \in \Sigma$ is represented by an alphabet symbol $\alpha_c \in (0|1)^*$ as following: $\alpha_a \Rightarrow 00, \alpha_b \Rightarrow 01, \alpha_c \Rightarrow 10, \alpha_s \Rightarrow 11$. An X symbol in the figure represents an option of either 0 or 1. For example, from state $S_0 \rightarrow S_1$ in input DFA, an edge labeled $\begin{matrix}\mathtt{X}\\\mathtt{X}\end{matrix}$ means that there are four transitions between these two states on alphabet $00, 01, 10$ and 11 (i.e., four transitions on characters a, b, c, s). Also, from state $S_2 \rightarrow S_0$ an edge labeled $\begin{matrix}0\\\mathtt{X}\end{matrix}$ means that there are two transitions between these two states on alphabet 00 and 01 (i.e., two transitions on characters a and b).

The first step in the algorithm is to compute $s\text{-}reach(S_0)$ (i.e., states that are reachable from state 0 on α_s^*). Since S_0 is reachable from itself on $\alpha_s^0 = \epsilon$, we add S_0. Then, starting from S_0 we have edge labeled $\begin{matrix}\mathtt{X}\\\mathtt{X}\end{matrix}$ to S_1 which means one of the transitions that it represents is on $\alpha_s = 11$. So we add S_1. Then checking transitions out from S_1 we have edge labeled $\begin{matrix}\mathtt{X}\\\mathtt{X}\end{matrix}$ to S_2 which means one of the transitions that it represents is on $\alpha_s = 11$. So we add S_2. Then checking transitions out from S_2 we have edge labeled $\begin{matrix}\mathtt{0}\\\mathtt{X}\end{matrix}$ to S_0 which means none of the transitions that it represents is on $\alpha_s = 11$. So the result is $s\text{-}reach(S_0) = \{S_0, S_1, S_2\}$. Then we calculate m which is the number of extra bits that we need to symbolically simulate non-determinism (i.e., number of extra bits to differentiate between different transitions on same

character that will go out from the new initial state that we will introduce). To calculate m we take the log of the size n of $s\text{-}reach(q_0)$ to base 2: $m = \lceil log_2(3) \rceil = 2$.

Next we build intermediate DFA (the DFA on the top right corner) by (1) copying original input DFA after adding the extra bits to its alphabet then (2) adding a new start state S_i then (3) copying all transitions out from $s\text{-}reach(S_0)$ states $\{S_0, S_1, S_2\}$ on a character not equal to s to new start state S_i and adding the extra bits to differentiate them. To appreciate the 2 extra bits that we need to add notice that we have 3 transitions on the two alphabet 00 and 01 out from new initial state S_i (see edges labeled with $\begin{smallmatrix}0\\ \mathtt{x}\\ 1\end{smallmatrix}$, $\begin{smallmatrix}0\\ \mathtt{x}\\ 0\end{smallmatrix}$ and $\begin{smallmatrix}0\\ \mathtt{x}\\ 1\end{smallmatrix}$ out from S_i) and we needed two extra bits to differentiate all these three transitions from each other. We have to point out that adding the extra bits introduced large number of new transitions that are not shown here since they all go to sink state which is also not shown here.

Finally, we project the two extra bit, bit-2 and bit-3 to determinize the intermediate DFA. The result is the DFA shown in the bottom. All strings accepted by this new DFA do not start with the left-trimmed symbol $\alpha_s = 11$.

PRETRIMLEFT(M, s): Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ our goal is to construct a DFA M where $M_1 =$ POSTTRIMLEFT(M, s). Notice that we do not know how many characters s have been left-trimmed from each string in $L(M)$ to get $L(M_1)$. For example, assuming $\Sigma = \{a, b, s\}$, left-trimming s in the following two languages $L_1 = \{sa, ssab\}$ and $L_2\{sab, ab, sssa\}$ will result in the same language $L_{lt} = \{a, ab\}$. In fact, left-trimming all

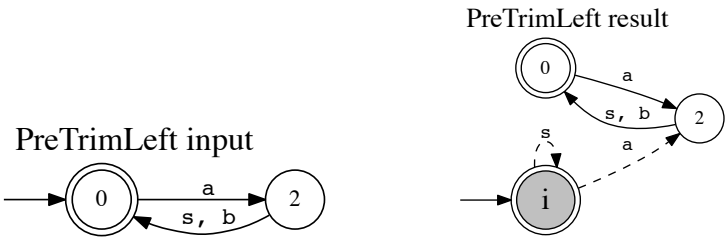

Figure 3.12: Example of PRETRIMLEFT.

languages $L \subseteq L_{max} = (s^*a|s^*ab)$ will result in L_{lt}. So we are going to construct the maximal DFA M such that $\forall$ DFA $M_i : M_1 = \text{POSTTRIMLEFT}(M_i, s) \Rightarrow L(M_i) \subseteq L(M)$.

It is tempting at the beginning to just add a self loop on s to the start state q_0. However, although this is a correct over-approximation, it is too imprecise since it may add the substrings s^* to all paths that have a back edge to start state q_0 on some $c \in \Sigma$ (i.e., $(q_i, \xrightarrow{c}, q_0)$). So a more precise way to do this is to copy q_0, along with all transitions out from it, into a new start state q_{init} and then add the self loop transition on s to this new state (i.e., $(q_{init}, \xrightarrow{s}, q_{init})$). Notice that there could not be a transition on s out from q_0 (i.e., $(q_0, \xrightarrow{s}, q_i)$) since we assume that M_1 is the result of POSTTRIMLEFT. This means that adding the self loop on s out from q_{init} will not introduce nondeterminism and there is no need for extra bits.

Formally, we will construct a DFA $M' = \langle Q', q_{init}, \Sigma_{\mathcal{B}}, \delta', F' \rangle$ as following:

- $Q' = Q_1 \cup \{q_{init}\}$.
- $F' = F_1$ if $q_0 \notin F_1$ otherwise $F' = F_1 \cup \{q_{init}\}$.
- $\forall q \in Q_1,\ \delta'(q, \alpha) = \delta_1(q, \alpha)$ and

  $\delta'(q_{init}, \alpha) = \delta_1(q_0, \alpha)$, $\delta'(q_{init}, \alpha_s) = q_{init}$

Example. Figure 3.12 shows an example of running PRETRIMLEFT. Notice that we did not need an intermediate DFA. We just added a new state S_i with a self loop on s and copied all transitions out from S_0 to new S_i.

POSTTRIMRIGHT(M_1, s): Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ our goal is to right trim s by constructing a DFA that would accept w if $ww_1 \in L(M_1)$ where s is that char to be right trimmed and $w_1 \in s^*$. We start first by marking, for each accepting state q_f in M_1, all states q_i, where q_f is reachable from q_i on s^*. To do this, let us first define the reverse of transition relation δ_1 as following:

$$\delta_1^{-1} = \{(q_j, c, q_i) \text{: where } (q_i, c, q_j) \in \delta_1\}$$

Notice that δ_1^{-1} is nondeterministic i.e., $\exists (q_i, c, q_j) \in \delta_1^{-1}$ and $(q_i, c, q_k) \in \delta_1^{-1}$ where $(q_j \neq q_k)$. Now given a DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$, a state $q_m \in Q$ and a char $s \in \Sigma$, let us define function $s\text{-}reach^{-1} : Q \leftarrow 2^Q$ as following:

$$s\text{-}reach^{-1}(q_m) = \{q_n \text{: there exists a path } q_m, q_{m+1}, \ldots, q_n \text{ such that}$$
$$\forall\, 0 \leq i,\ (q_{m+i}, s, q_{m+i+1}) \in \delta^{-1}\}$$

Now we can define the first step of the algorithm as computing $s\text{-}bw\text{-}reachable = \bigcup_{q_f \in F_1} s\text{-}reach^{-1}(q_f)$. To compute this set, we run depth first search multiple times on δ^{-1} starting each time from one of the accepting states. Intuitively, this step helps us to add to the language accepted by the new DFA all substrings w that (1) do not end with s and (2) by simulating M_1 on w starting from q_0 we will reach a state q_m that reachable from an accepting state by reverse transition relation δ^{-1}.

Given $s\text{-}bw\text{-}reachable$, we construct a new DFA M' from M as following: (1) add a new accepting state q_{final} and then (2) for each transition $(q_i \xrightarrow{c} q_j)$ where $q_j \in s\text{-}bw\text{-}reachable$ (i.e., reachable from an accepting state on s using δ^{-1} and $c \in \Sigma \setminus \{s\}$, we copy that transition to get a new transition $(q_i \xrightarrow{c} q_{final})$. Since adding this new transition from q_i on the same character c to a different state than q_j will introduce non-determinism, we need to add an extra bit to $\Sigma_{\mathcal{B}}$. $\forall \alpha \in \Sigma_{\mathcal{B}}$, we define two new symbols $\alpha 0$ and $\alpha 1$. Finally, if $q_0 \in s\text{-}bw\text{-}reachable$ then q_0 becomes an accepting state. We construct the intermediate DFA $M' = \langle Q', q_0, \Sigma'_{\mathcal{B}}, \delta', F' \rangle$ as follows:

- $\Sigma'_{\mathcal{B}} = \mathcal{B}^{k+1}$ where $\Sigma_{\mathcal{B}} = \mathcal{B}^k$.
- $Q' = Q_1 \cup \{q_{final}\}$.
- $F' = \{q_{final}, q_0\}$ if $q_0 \in s\text{-}bw\text{-}reachable$ otherwise $F' = \{q_{final}\}$.
- $\delta'(q_i, \alpha 0) = \delta(q_i, \alpha 0)$ and

  $\delta'(q_i, \alpha 1) = q_{final}$ if $\delta(q_i, \alpha) = q_j$ for a $q_j \in s\text{-}bw\text{-}reachable$ and $\alpha \neq \alpha_s$.

Finally we determinize the symbolic DFA M' by projecting the extrabit. Formally, final result is $M = \textsc{Project}(M', k+1)$.

Example. Figure 3.13 shows an example of running PostTrimRight on the input DFA. The first is computing $s\text{-}reach^{-1}(S_3)$ (i.e., states that can reach accepting state 3 on α_s^*). These states are = $\{S_0, S_2, S_3\}$. The second is building intermediate DFA (the DFA in the middle) by (1) copying original input DFA after adding an extra bit to its alphabet then (2) adding a

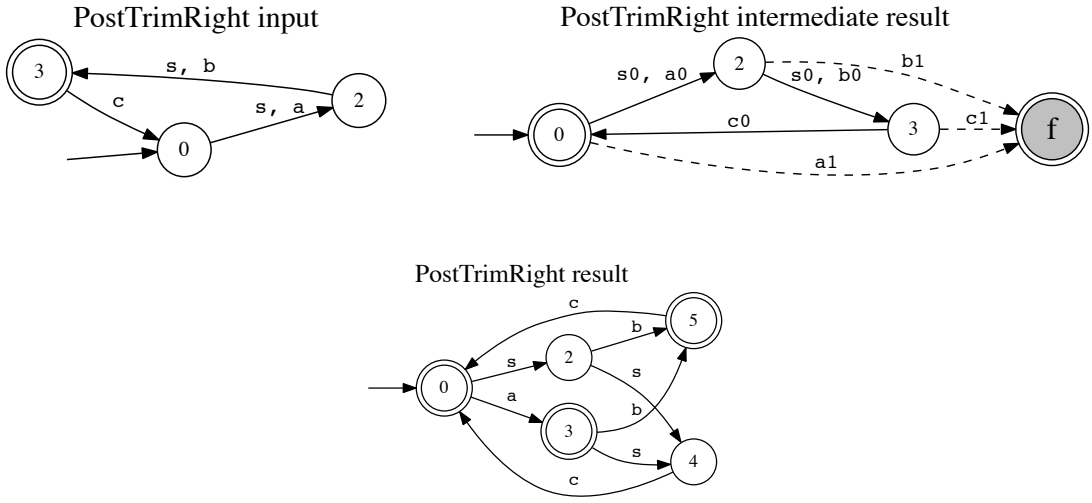

Figure 3.13: Example of POSTTRIMRIGHT.

new accepting state S_f (i.e., q_{final}) then (3) copying all transitions out from $s\text{-}reach^{-1}(S_3)$ states $\{S_0, S_2, S_2\}$ on a character not equal to s to accepting state S_f and adding an extra bit to differentiate them and finally (4) marking start state S_0 as an accepting state since it is in $s\text{-}reach^{-1}(S_3)$. The third and final step is to project the extra bit to determinize the intermediate DFA. All strings accepted by this new DFA do not end with the right-trimmed character s.

PRETRIMRIGHT(M, s): Given $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$ our goal is construct a DFA M where $M_1 =$ POSTTRIMRIGHT(M, s). Notice that we do not no how many characters s have been right-trimmed from each string in $L(M)$ to get $L(M_1)$. For example, assuming $\Sigma = \{a, b, s\}$, right-trimming s in the following two languages $L_1 = \{as, abss\}$ and $L_2\{absss, ab, ass\}$ will result in the same language $L_{lt} = \{a, ab\}$. In fact, right-trimming all languages $L \subseteq L_{max} = (as^*|abs^*)$ will result in L_{RT}. So we are going to construct the maximal DFA M such that $\forall$ DFA $M_i : M_1 =$ POSTTRIMRIGHT(M_i, s) $\Rightarrow L(M_i) \subseteq L(M)$.

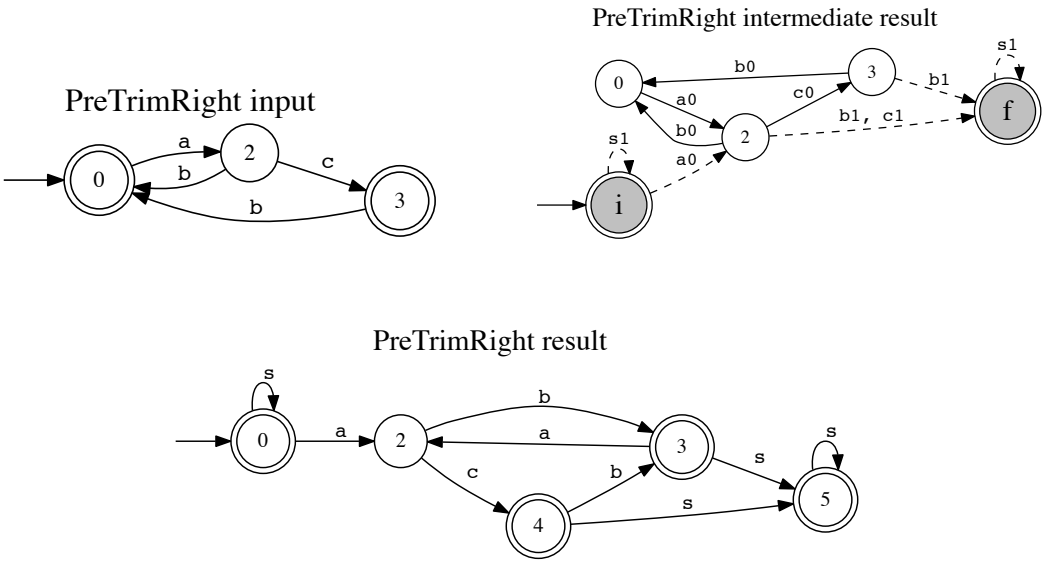

Figure 3.14: Example of PRETRIMRIGHT.

It is tempting at the beginning to just add a self loop on s to each of the accepting states $q_f \in F_1$. But, although this is a correct over-approximation, it is too imprecise as it may add the substrings s^* to all paths that have a back edge from an accepting state q_f on some $c \in \Sigma$ (i.e., $(q_f, \xrightarrow{c}, q_i)$). So a more precise way to do this is to (1) add a new accepting state q_{final}, and (2) copy any outgoing transition from a state q_i on a character c that goes into an accepting state $q_f \in F_1$ as a transition that goes from q_i into q_{final} ((i.e., $(q_i, \xrightarrow{c}, q_{final})$) and (3) add the self loop transition on s to this new state (i.e., $(q_{final}, \xrightarrow{s}, q_{final})$). Since in the second step, for each state q_i that has an outgoing transition to a final state $q_f \in F_1$ we will duplicate that transition to q_{final}, this will result in having two transitions on a character c out from the same state q_i. To deal with the nondeterminism we need to extend $\Sigma_{\mathcal{B}}$ by one more bit to differentiate these two transitions. Finally, if that start state q_0 is an accepting state (i.e., $q_0 \in F_1$) then we need to add a new start state q_{init}—in addition to q_{final}—in the same way that we did in PRETRIMLEFT.

Formally, we will construct an intermediate DFA $M' = \langle Q', q'_0, \Sigma'_{\mathcal{B}}, \delta', F' \rangle$ as following:

- $\Sigma'_{\mathcal{B}} = \mathcal{B}^{k+1}$ where $\Sigma_{\mathcal{B}} = \mathcal{B}^k$.
- $q'_0 = q_0$ if $q_0 \notin F_1$ otherwise

  $q'_0 = q_{init}$
- $Q' = Q_1 \cup \{q'_0, q_{final}\}$
- $F' = \{q_{final}\}$ if $q_0 \notin F_1$ otherwise

  $F' = \{q'_0, q_{final}\}$
- $\forall q \in Q_1,\ \delta'(q, \alpha 0) = \delta_1(q, \alpha)$ and

  $\forall q \in Q_1,\ \delta'(q, \alpha 1) = q_{final}$ if $\delta_1(q, \alpha) \in F_1$ and

  $\delta'(q_{final}, \alpha_s 1) = q_{final}$ and

  if $q_0 \in F_1$ then $\delta'(q_{init}, \alpha 0) = \delta_1(q_0, \alpha)$, $\delta'(q_{init}, \alpha_s 1) = q_{init}$.

Finally we determinize the symbolic DFA M' by projecting all extra bits. Formally, we construct $M = \textsc{Project}(M', k+1)$.

Example. Figure 3.14 shows an example of running PRETRIMRIGHT on the input DFA shown on the top. We built the intermediate DFA (the DFA in the middle) by (1) copying original input DFA after adding an extra bit to its alphabet then (2) adding a new start state S_i (i.e., q_{init}) since S_i is an accepting state and a new accepting state S_f (i.e., q_{final}) then (3) copying all transitions coming into accepting states $\{S_0, S_3\}$ to transition coming into S_f and adding an extra bit to differentiate them and finally (4) adding self loop transitions on new start and accepting states S_i

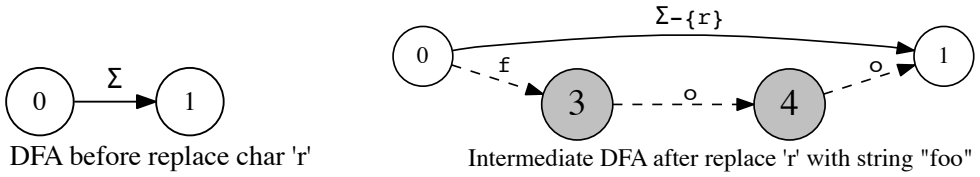

Figure 3.15: Example of replacing a transition on char `r` between two states S_0 and S_1 with a path on string "`foo`" while computing POSTREPLACECHAR.

and S_f. Finally, we project the extra bit to determinize the intermediate DFA. The result is the DFA shown in the bottom.

POSTREPLACECHAR(M_1, r, w): The algorithm takes as input a symbolic DFA $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$, a character $r \in \Sigma$ and a string w of length $n > 0$ that consists of a sequence of characters $w = \langle c_0, c_1, \ldots, c_{n-1} \rangle$ where each character $c_i \in \Sigma$. The result DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ is constructed as follows:
(1) if $|w| = 1$ (i.e., w consists of a single character c_0) we replace each transition $(q_i \xrightarrow{r} q_j)$ on replace character r with transition $(q_i \xrightarrow{c_0} q_j)$ (2) else we replace each transition $(q_i \xrightarrow{r} q_j)$ on replace character r with a new path $(q_i \xrightarrow{w}{}^{*} q_j)$ of length n by adding $n-1$ new states $q'_{i_0}, q'_{i_1}, \ldots, q'_{i_{n-2}}$ and n transitions $(q_i \xrightarrow{c_0} q'_{i_0})$, $(q'_{i_0} \xrightarrow{c_1} q'_{i_1})$, $\ldots$, $(q'_{i_{n-2}} \xrightarrow{c_{n-1}} q_j)$. Since q_i might have an outgoing transition on c_0, we need to add one extra bit to simulate non-determinism.

Figure 3.15 shows how to replace a marked transition on `r` between states S_0 and S_1 with a new path between the states on string `foo`. The shaded states are the new added states and the dashed transitions are the new added transitions.

The resulting intermediate automaton $M' = \langle Q', q_0, \Sigma'_{\mathcal{B}}, \delta', F' \rangle$ is constructed as follows:

- $\Sigma'_{\mathcal{B}} \subseteq \mathcal{B}^{k+1}$ where $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^k$.
- Q' = $Q_1 \cup Q'', Q'' = \{q'_{i_l} | \forall i \geq 0,\ q_i \in Q_1$ and $\delta_1(q_i, \alpha_r) \neq sink$ and $0 \leq l \leq n-2)\}$.
- $F' = F_1$.
- $\forall q \in Q, \forall \alpha \neq \alpha_r : \delta'(q, \alpha 0) = \delta_1(q, \alpha)$

  if $n = 1$ then $\forall i \geq 0 :\ (\delta_1(q_i, \alpha_r) \neq sink) \Rightarrow (\delta'(q_i 1, \alpha_{c_0} 1) = \delta_1(q_i, \alpha_r))$

  else $\forall i \geq 0 :\ (\delta_1(q_i, \alpha_r) \neq sink) \Rightarrow (\delta'(q_i, \alpha_{c_0} 1) = q'_{i_0}, \forall 0 \leq l \leq n-2 :$ $\delta'(q'_{i_l}, \alpha_{c_{l+1}} 1) = q'_{i_{l+1}}$ and $\delta'(q'_{i_{n-2}}, \alpha_{c_{n-1}} 1) = \delta_1(q_i, \alpha_r))$.

Then we get final DFA M by projecting the added bit, M = PROJECT($M', k+1$)

PREREPLACECHAR(M_1, r, w): The algorithm takes as input a symbolic DFA $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$, a character $r \in \Sigma$ and a string w of length $n > 0$ that consists of a sequence of characters $w = \langle c_0, c_1, \ldots, c_{n-1} \rangle$ where each character $c_i \in \Sigma$. The result is a DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ such that M_1 =POSTREPLACECHAR(M, r, w). M is constructed as following:

(1) for each state $q_i \in Q_1$ we check if there a path on w to a non-sink state q_j (i.e., ($q_i \xrightarrow{w}{}^* q_j$) or $\delta_1^*(q_i, w) = q_j$ where $q_j \neq sink$) (2) if so then we add a new transition from q_i to q_j on r (i.e., ($q_i \xrightarrow{r} q_j$)) if such transition does not exist before *without* removing the path ($q_i \xrightarrow{w}{}^* q_j$). Since q_i might have an outgoing transition on r to a state $q_k \neq q_j$, we need to add one extra bit to simulate non-determinism.

The resulting intermediate automaton $M' = \langle Q', q_0, \Sigma'_{\mathcal{B}}, \delta', F' \rangle$ is constructed as follows:

- $\Sigma'_{\mathcal{B}} \subseteq \mathcal{B}^{k+1}$ where $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^k$
- Q' = Q_1.
- $F' = F_1$.
- $\forall q \in Q, \forall \alpha \in \Sigma_{\mathcal{B}} : \delta'(q, \alpha 0) = \delta_1(q, \alpha)$

  $\forall q \in Q : \ \delta'(q, \alpha_r 1) = \delta_1^*(q, \alpha_r)$.

Then we get final DFA M by projecting the added bit, M = PROJECT($M', k+1$)

POSTREPLACECHARSETWITHCHARSET(M_1, $\mathcal{R}_{\mathcal{B}}$): The algorithm takes as input a symbolic DFA $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$, and a relation (partial function) $\mathcal{R}_{\mathcal{B}} \subseteq \Sigma_{\mathcal{B}} \times \Sigma_{\mathcal{B}}$ where $\forall \alpha_1, \alpha_2$ and $\alpha_3 \in \Sigma_{\mathcal{B}}$, $(\alpha_1, \alpha_2) \in \mathcal{R}_{\mathcal{B}}$ and $(\alpha_1, \alpha_3) \in \mathcal{R}_{\mathcal{B}} \Rightarrow \alpha_2 = \alpha_3$ and $\forall \alpha$ and $\alpha' \in \Sigma_{\mathcal{B}}$, if $(\alpha, \alpha') \in \mathcal{R}_{\mathcal{B}}$ then $\nexists \alpha'' \in \Sigma_{\mathcal{B}}$ such that $(\alpha'', \alpha) \in \mathcal{R}_{\mathcal{B}}$. For each pair of characters $(c_1, c_2) \in \mathcal{R}$, the algorithm replaces each occurance of the first character c_1 in all strings $w \in \mathcal{L}(M_1)$ with the second character c_2. The result DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, R \rangle$ is constructed as follows:

for each state q_i for each pair of characters $(c_1, c_2) \in \mathcal{R}$ we replace each outgoing transition ($q_i \xrightarrow{c_1} q_j$) on first character c_1 to some state q_j with a transition ($q_i \xrightarrow{c_2} q_j$). Since q_i might already have an outgoing transition on c_1, we need to add one extra bit to simulate non-determinism.

The resulting intermediate automaton $M' = \langle Q, q_0, \Sigma'_{\mathcal{B}}, \delta', F \rangle$ is constructed as follows:

- $\Sigma'_{\mathcal{B}} \subseteq \mathcal{B}^{k+1}$ where $\Sigma_{\mathcal{B}} \subseteq \mathcal{B}^{k}$.
- $Q = Q_1$.
- $F' = F_1$.
- $\forall q \in Q, \forall \alpha \in \Sigma_{\mathcal{B}}$: if $\exists \alpha'$ such that $(\alpha, \alpha') \in \mathcal{R}_{\mathcal{B}}$ then $\delta'(q, \alpha'1) = \delta_1(q, \alpha)$ and $\delta'(q, \alpha 0) = sink$ else $\delta'(q, \alpha 0) = \delta_1(q, \alpha)$

Then we get final DFA M by projecting the added bit, M = PROJECT($M', k+1$)

PREREPLACECHARSETWITHCHARSET(M_1, $\mathcal{R}_{\mathcal{B}}$): The algorithm takes as input a symbolic DFA $M_1 = \langle Q_1, q_0, \Sigma_{\mathcal{B}}, \delta_1, F_1 \rangle$, and a relation (partial function) $\mathcal{R}_{\mathcal{B}} \subseteq \Sigma_{\mathcal{B}} \times \Sigma_{\mathcal{B}}$ where $\forall \alpha_1, \alpha_2$ and $\alpha_3 \in \Sigma_{\mathcal{B}}$, $(\alpha_1, \alpha_2) \in \mathcal{R}_{\mathcal{B}}$ and $(\alpha_1, \alpha_3) \in \mathcal{R}_{\mathcal{B}} \Rightarrow \alpha_2 = \alpha_3$ and $\forall \alpha$ and $\alpha' \in \Sigma_{\mathcal{B}}$, if $(\alpha, \alpha') \in \mathcal{R}_{\mathcal{B}}$ then $\nexists \alpha'' \in \Sigma_{\mathcal{B}}$ such that $(\alpha'', \alpha) \in \mathcal{R}_{\mathcal{B}}$.. Then it computes a DFA $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, R \rangle$ where POSTREPLACECHARSETWITHCHARSET(M, $\mathcal{R}_{\mathcal{B}}$) = M_1. The pre image computation is very similar to the post image computation except that 1) we will compute the result M using $\mathcal{R}_{\mathcal{B}}^{-1}$ and 2) for all pairs $(c_1, c_2) \in \mathcal{R}$ we are not going to remove any transition ($q_i \xrightarrow{c_2} q_j$) on the second character c_2. The reason is that we do not know if c_2 was the result of replacing c_1 while doing the post-image computation or not. Since M_1 is supposed to be the result of POSTREPLACECHARSETWITHCHARSET, M_1 can not contain any character c_2 in a pair $(c_1, c_2) \in \mathcal{R}$. This means that we do not need extra bits when adding an outgoing transition ($q_i \xrightarrow{c_2} q_j$) on c_2.

The resulting automaton $M = \langle Q, q_0, \Sigma_{\mathcal{B}}, \delta, F \rangle$ is constructed as follows:

- $Q = Q_1$.

- $F = F_1$.

- $\forall q \in Q, \forall \alpha \in \Sigma_{\mathcal{B}}$: if $\exists \alpha'$ such that $(\alpha', \alpha) \in \mathcal{R}_{\mathcal{B}}$ then $\delta(q, \alpha) = \delta_1(q, \alpha')$ else $\delta(q, \alpha) = \delta_1(q, \alpha)$

Computing Post and Pre-images of IVSL String Functions

We use the previous pre and post-image computation algorithms along with the algorithms for POSTCONCAT, POSTREPLACE, PRECONCATPREFIX, PRECONCATSUFFIX and PREREPLACE to compute pre and post-images (transfer functions) for String Functions in IVSL. Image computation for some functions such as `concat`, `replace` and `addslashes` map directly to image computation algorithms POSTCONCAT/PRECONCAT, POSTREPLACE/PREREPLACE and POSTESCAPE/PREESCAPE. For others this is not the case. Here are two examples.

`trim` : this function removes the chars ␣ , `\t`, `\n` and `\0` from beginning and end of its input string (␣ represents the white space char). Given M_{in} =`IN[var]` for some variable `var` and `trim(var)`, we can use compute post-image M_{out} =`OUT[var]` by multiple calls to POSTTRIMLEFT and POSTTRIMRIGHT as following:

$M_1 =$ POSTTRIMLEFT(M_{in}, ␣)

$M_2 =$ POSTTRIMLEFT(M_1, `\t`)

$M_3 =$ POSTTRIMLEFT(M_2, `\n`)

$M_4 =$ POSTTRIMLEFT(M_3, `\0`)

$M_5 =$ POSTTRIMRIGHT(M_4, ␣)

$M_6 =$ POSTTRIMRIGHT(M_5, \t)

$M_7 =$ POSTTRIMRIGHT(M_6, \n)

$M_{out} =$ POSTTRIMRIGHT(M_7, \0)

Assuming $M_{out} =$OUT[var] as input to pre-image computation, we substitute POSTTRIMLEFT with PRETRIMLEFT and POSTTRIMRIGHT with PRETRIMRIGHT going in reverse direction starting with $M_1 =$ PRETRIMRIGHT(M_{out}, \0) all the way to $M_{in} =$ PRETRIMLEFT(M_7, ␣).

htmlspecialchars : this function replaces some chars with their HTML encoding as following: < with "<", > with ">", ' with "'", " with """ and finally & with "&". Given $M_{in} =$ IN[var] for some variable var and htmlspecialchars(var) , we can use compute post-image $M_{out} =$ OUT[var] by multiple calls to POSTREPLACECHAR as following:

$M_1 =$ POSTREPLACECHAR(M_{in}, <, "\xfelt;")

$M_2 =$ POSTREPLACECHAR(M_1, >, "\xfegt;")

$M_3 =$ POSTREPLACECHAR(M_2, ', "\xfeapos;")

$M_4 =$ POSTREPLACECHAR(M_3, ", "\xfequot;")

$M_5 =$ POSTREPLACECHAR(M_4, &, "&")

$M_{out} =$ POSTREPLACECHAR(M_5, \xfe, &)

Notice that we used a temporary intermediate char \xfe that is internally reserved. This allows us to avoid replacing all chars & that could have been introduced by the first four replacement operations which will result in optimized more precise image computation that can handle

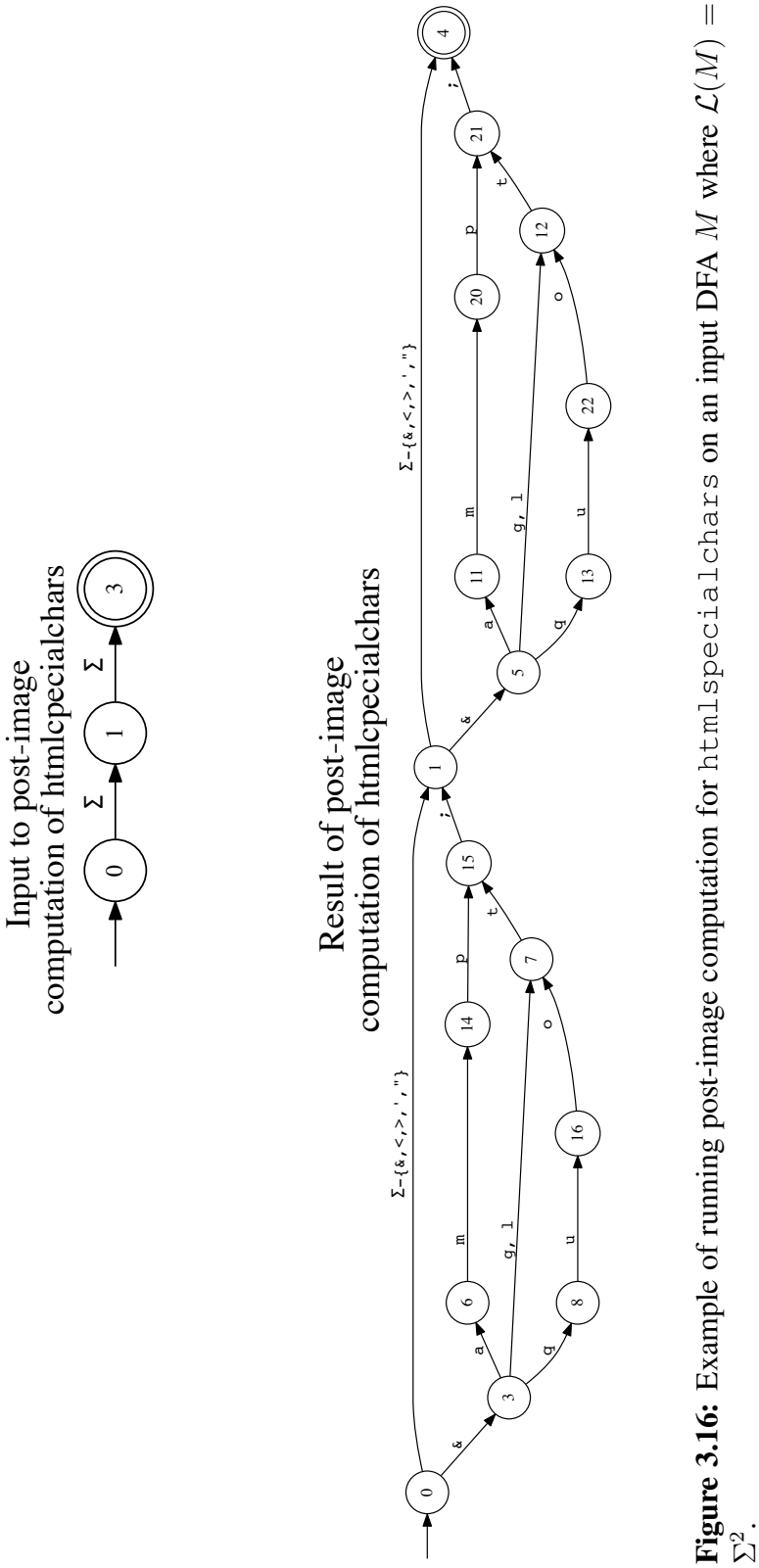

Figure 3.16: Example of running post-image computation for `htmlspecialchars` on an input DFA M where $\mathcal{L}(M) = \Sigma^2$.

`htmlspecialchars` in the presence of length constraints upto 1000. Without this optimization, the computation reaches MONA memory limit (see end of this Chapter) at around length 150.

Figure 3.16 shows an example of running post-image computation for `htmlspecialchars` on an input DFA M where $\mathcal{L}(M) = \Sigma^2$. Assuming M_{out} = `OUT[var]` as input to pre-image computation, we substitute POSTREPLACECHAR with PREREPLACECHAR going in reverse direction starting with M_1 = PREREPLACECHAR(M_{out}, `&`, "`&`") all the way to M_{in} = PREREPLACECHAR(M_4, <, "`<`") without the need for the special reserved char `\xfe`.

Performance of Special Replace Algorithms

Figure 3.17 shows a comparison in terms of time and memory (represented using the number of BDD nodes) between the performance of the generic replace algorithm and the specialized replace algorithms we presented here. In our setup we computed the post-image of the two PHP functions `addslashes` (which does 3 replace operations) and `htmlspecialchars` (which does 5 replace operations) on the language $\bigcup_{i=0}^{l} \Sigma^l$. This setup is identical to a sanitizer function that restricts the length of its input i to a certain value l through branch condition `len(i) <= l`, and then sanitizes the input that passes the branch condition using one of the two functions. The x axis shows the length while the y axis shows the time and memory and uses a log scale. We notice that the generic replace grows exponentially as we increase the length while the specialized ones do not. For the generic replace, the analysis reaches MONA

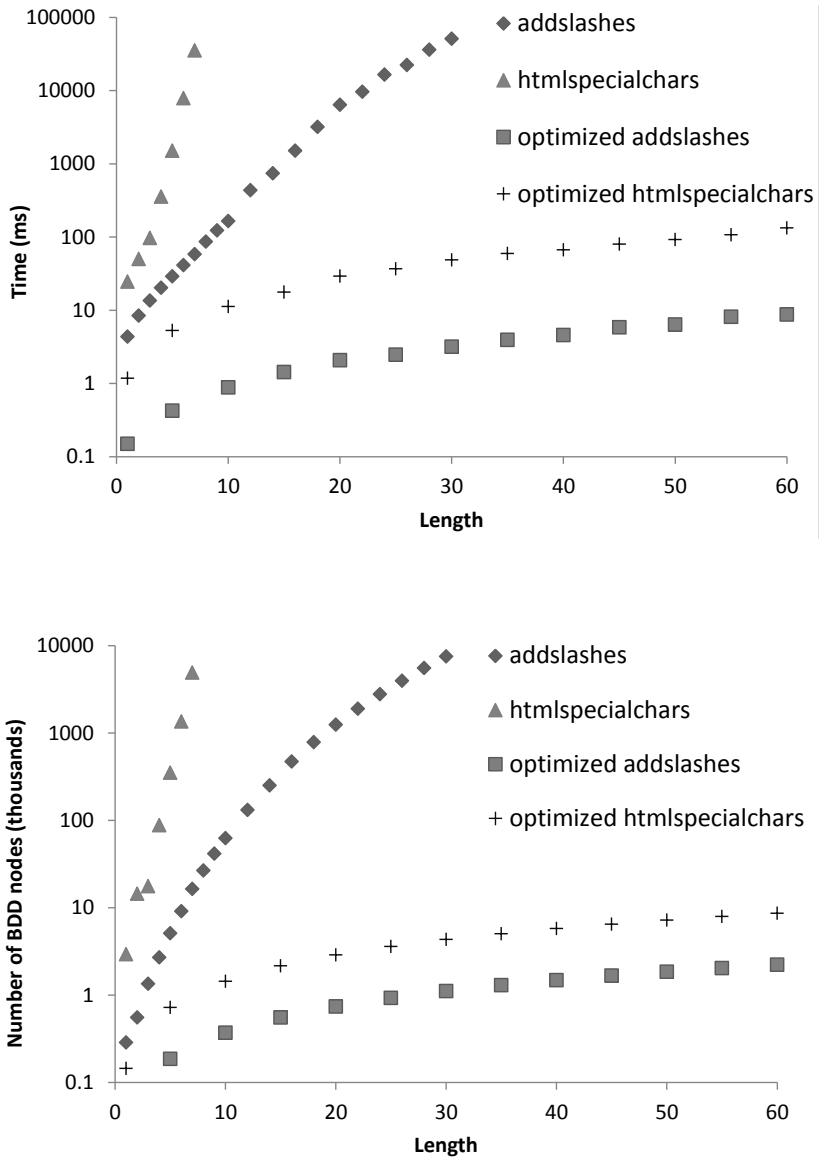

Figure 3.17: Time and memory performance for generic replace and optimized/specialized replace algorithms.

limit on BDD size[1] at length 31 for `addslashes` and at length 8 for `htmlspecialchars`. On the other hand, for the specialized replace operations it took less than 1 second to run with length 100 with negligible memory overhead.

[1]MONA has a hard limit on maximum number of BDD nodes which is 2^{24}

Chapter 4

Policy-Based Bug Detection and Repair

In this chapter we show how to verify extracted sanitizers against manually written policies. First we show how to detect and repair some vulnerabilities such as XSS and SQLI in PHP server-side code in web applications using security policies [3, 102–105]. Then we switch to client-side Javascript and show how we detect input validation errors by verifying validators against a minimum and a maximum policy to make sure that they are not over-constrained or under-constrained.

4.1 Vulnerability Detection and Repair for Server-Side PHP Input Sanitization

Web application development is error prone and results in applications that are vulnerable to attacks by malicious users. The global accessibility of Web applications makes this an ex-

tremely serious problem. According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious web application vulnerabilities, the top three vulnerabilities in 2007 [75] were: 1) Cross Site Scripting (XSS) and 2) Injection Flaws (such as SQL Injection). Even after it has been widely reported that web applications suffer from these vulnerabilities, the top two of the vulnerabilities listed in OWASP's top ten list in 2010 [76] were the same top two from 2007.

A *XSS vulnerability* results from the application inserting part of the user's input in the next HTML page that it renders. Once the attacker convinces a victim to click on a URL that contains malicious HTML/JavaScript code, the user's browser will then display HTML and execute JavaScript that can result in stealing of browser cookies and other sensitive data. An *SQL Injection vulnerability*, on the other hand, results from the application's use of user input in constructing database statements. The attacker can invoke the application with a malicious input that is part of an SQL command that the application executes. This permits the attacker to damage or get unauthorized access to data stored in a database.

As we satated earlier, all these vulnerabilities are caused by improper string manipulation in server-side code. Programs that propagate and use malicious user inputs with improper sanitization on the server-side are vulnerable to these well-known attacks. The attacks that exploit the vulnerabilities related to string manipulation can be characterized as *attack patterns*, i.e., regular expressions that specify potential attack strings. In this section we explain how to use these attack patterns as security policies against which we verifiy and repair vulnerabilities.

```
1 sanitizer(www){
2    l_otherinfo = "URL";
3    www = replace( "/[^A-Za-z0-9 .-@://]/", "", www );
4    temp1 = concat(l_otherinfo, ":");
5    temp2 = concat(temp1, www);
6    return temp2;
7 }
```

Figure 4.1: A sanitizer with an XSS vulnerability extracted from server-side code in a PHP web application.

4.1.1 Example

Let us look at the previous example from Figure 1.6. Figure 4.1 shows the corresponding sanitizer written in IVSL language and we have already explained how to extract such function from PHP code (see 2.6). The PHP input variable `$_GET["www"]` has been translated into the parameter `www` and the PHP sink function `echo` has been translated into the `return` statement. Computing the post-image of the IVSL sanitizer corresponds to computing the values that flow from input variable `$_GET["www"]` into the sink function `echo`. So to check if this PHP example server-side code is vulnerable i.e., to check if some vulnerable string values can flow into the sensitive sink `echo`, we need to check if the IVSL sanitizer can return such vulnerable values as its output.

We briefly describe the vulnerability in this example here as we have already explained the example in detail in the introduction. The problem with this sanitizer is in the replace operation in line 3. The goal of this replace operation is to remove any special characters from the input to prevent XSS attacks e.g., to prevent strings that contain the string constant `<script` from being returned as output. However, since the regular expression `[A-Za-z0-9 .-@://]`, contains

ASCII symbols that are between the symbol . and the symbol @, this leads to the symbol < not being deleted from the input, leading to a XSS vulnerability i.e., the possibility of the sanitizer to return a malicious output containing some javascript code in between <script> and </script> tags.

To detect and repair such vulnerability, we need to 1) check if the sanitizer is capable of returning bad output values and if so 2) we need to sanitize or block the input values that resulted in such bad output values. But in order to do this, we need to first specify what constitutes bad output values. For this purpose, we use a special type of security policies that we call *attack patterns*. An *attack pattern* is a regular expression that specifies potential attack strings specific to a certain vulnerability i.e., strings that appear in the exploits of such vulnerability. Based on these security policies, we introduce a new policy based algorithm to detect and repair XSS and SQLI vulnerabilities. First, the algorithm compares the sanitizer's possible output language (i.e., post-image) to the language of an XSS attack pattern. If the two langugaes intersect then we report a possible XSS vulenrability. After that, given the intersection result which characterizes the bad output, we compute the possible inputs that may have resulted in this bad output (i.e., the pre-image of the sanitizer and the bad output) using algorithms from 3.

4.1.2 Policy-Based Repair Problem

Let us define a security policy P (i.e., an attack pattern) as a regular expression that specifies all attack strings that may appear in the output of a vulnerable function. Given a potentially

vulnerable sanitizer function F and a a security policy P, the goal of policy-based repair is to automatically generate a new sanitizer function F^P, called a patch, such that when F is patched by composing it with F^P, the set of strings returned by the resulting repaired function are not in the language of the regular expression P. Formally, the policy-based repair problem is to automatically construct a patch F^P such that $\text{POST}(F \circ F^P, \Sigma^*) \cap L(P) = \emptyset$ where $L(P)$ is the language of the security policy (i.e., regular expression) P. This means when we compose F with F^P (see 2.3 for definition of sanitizers composition) we want to make sure that the result, $F \circ F^P$ does not return a string $s \in L(P)$ for any given input. We call this new composed function the *policy-based repair* F_{PR}, where $F_{PR} = F \circ F^P$.

4.1.3 Policy-Based Repair Algorithm

Algorithm 12 POLICYBASEDREPAIR($F, P, strategy$)

1: $M_p := \mathcal{A}(\mathcal{L}(P))$;
2: $M_1 := \mathcal{A}(\text{POST}^+(F, \Sigma^*, \Sigma^*, \ldots, \Sigma^*))$;
3: $M_b = M_1 \cap M_p$;
4: **if** ($\mathcal{L}(M_b) \neq \emptyset$) **then**
5: REPORT(`Vulnerable`);
6: **if** (F is single-input sanitizer) **then**
7: $M_{vs} := \mathcal{A}(\text{PRE}^+(F, \mathcal{L}(M_b)))$;
8: **if** ($strategy =$ `match-and-block`) **then**
9: $F^P :=$ GENERATEBLOCKINGSIMULATOR(M_{vs});
10: **else**
11: $\Sigma_{mc} :=$ MINCUT(M_{vs});
12: $F^P :=$ GENERATEMATCHINGSANITIZER(M_{vs}, Σ_{mc});
13: **end if**
14: **else**
15: $F^P :=$ GENERATESIMULATOR($\mathcal{A}(\emptyset)$);
16: **end if**
17: **else**
18: $F^P :=$ IDENTITYFUNCTION;
19: **end if**
20: $F_{PR} := F \circ F^P$;
21: **return** F_{PR};

Our policy-based repair algorithm is shown in Algorithm 12. The algorithm takes a single or multi-input sanitizer function F and a security policy (i.e., attack pattern) P as input and 1) reports if the sanitizer F is vulnerable and 2) if F is a vulnerable *single-input* sanitizer then it generates a sanitizer F_{PR} as output which corresponds to policy-based repair of F against P. The algorithm is based on automata based symbolic string analysis, and uses the post and pre-image computation algorithms from Chapter 3. In the algorithm, each variable that has a name starting with M represents a DFA, each variable with a name starting with F represent a sanitizer. The algorithm uses the DFA operations $\cap, \cup, \setminus, —, \mathcal{A}, \mathcal{L}$ as described in 3.2.4.

In line 1 we construct the DFA M_p that accepts the language of the security policy. Then in line 2 we compute an over-approximation of the post-image of the sanitizer F and the tuple $\Sigma^*, \Sigma^*, \ldots, \Sigma^*$ (see 3.1.1), i.e., all possible output values that F may return when each of its input variables is allowed to take any possible input value. In line 4 we check if there is a possiblity for F to output a bad value according to P i.e., a value that is in the language of the attack pattern P. If this is possible then we report a possible detected vulnerability in line 5.

Figure 4.2 shows the post-image for the example vulnerable sanitizer in figure 4.1[1]. In this figure and the next two, we use character ranges to indicate transitions on each character in this character range. For example, the self loop edge from state $S6$ to itself has `[.-Z]` which means that there is a transition from $S6$ to itself on each character in the ASCII table between the two characters `.` (i.e., `0x2E`) and `Z` (i.e., `0x5A`) including the two characters themselves. We use either special symbols or the decimal value for non-printable characters. For example, `\s`

[1]These DFAs are automatically generated by our tools

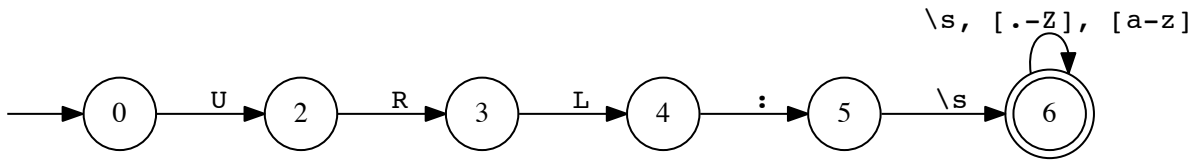

Figure 4.2: Post-image of sanitizer in Figure 4.1.

means white space character and `[NUL-253]` means range of ASCII characters from the first ASCII character `NULL` (i.e., `0x00`) to the character with the ASCII decimal number 253 (i.e., `0xFD`) (note that we reserve the two characters 254 and 255 for internal use). Figure 4.3 shows the intersection of the attack pattern $\Sigma^* < \Sigma^*$ with the post-image.

Notice that since we over-approximate the set of values that F may return when computing its post-image (see 3.1.1), we are not going to miss a vulnerability. However, this over-approximation may result in our algorithm reporting a false positive (i.e., spurious) vulnerability here. In other words, in this algorithm we compute $\textsc{Post}^+(F, \Sigma^*, \ldots, \Sigma^*) \supseteq \textsc{Post}(F, \Sigma^*, \ldots, \Sigma^*)$. If $\textsc{Post}^+(F, \Sigma^*, \ldots, \Sigma^*) \cap L(P) \neq \emptyset$, this does not mean that $\textsc{Post}(F, \Sigma^*, \ldots, \Sigma^*) \cap L(P) \neq \emptyset$ while the opposite is true i.e., $\textsc{Post}^+(F, \Sigma^*, \ldots, \Sigma^*) \cap L(P) = \emptyset$ guarantees that $\textsc{Post}(F, \Sigma^*, \ldots, \Sigma^*) \cap L(P) = \emptyset$.

If the input function F is a multi-input sanitizer then we stop at the detection phase of the algorithm and do not proceed to the patch generation phase. The reason is that pre-image computation for a multi-input sanitizer is not precise enough to produce acceptable results (one can use relational analysis here as we discussed in 3.1.2).

After that, in line 7, given the set of possible bad outputs that F may return (i.e., the set of strings that are in the post-image of F and at the same time are in the language of P),

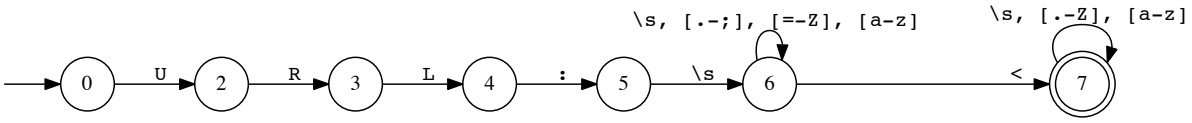

Figure 4.3: Intersection of post-image of sanitizer in Figure 4.2 and attack pattern $\Sigma^* < \Sigma^*$.

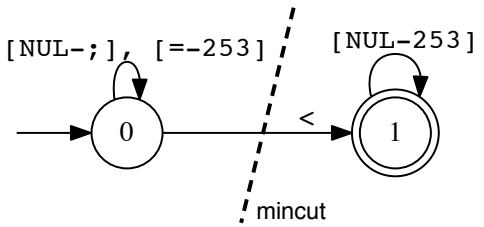

Figure 4.4: Vulnerability signature DFA M_{vs} for example in Figure 4.1 given attack pattern $\Sigma^* < \Sigma^*$. The dotted line shows the mincut for this vulnerability signature DFA.

we compute the vulnerability signature DFA (the DFA M_{vs}) using the pre-image algorithm (see 3.1.2). The vulnerability signature gives an over-approximation of all possible malicious input values that can exploit the vulnerability. Hence, if we do not allow input values that match the vulnerability signature then we can remove the vulnerability.

To proof this let us formally define what a *vulnerability signature* VS is. Given an sanitizer F and a security policy P, we formally define the *vulnerability signature* VS as following:

$$VS = \textsc{Pre}^+(F,\ \textsc{Post}^+(F, \Sigma^*) \cap L(P))$$

VS = $\textsc{Pre}^+(F,\ \textsc{Post}^+(F, \Sigma^*) \cap L(P)) \supseteq \textsc{Pre}(F,\ \textsc{Post}^+(F, \Sigma^*) \cap L(P))$

$\supseteq \textsc{Pre}(F,\ \textsc{Post}(F, \Sigma^*) \cap L(P))$ which is the actual set of malicious input values. Note that due to over-approximation, we may consider some input values as malicious while they are not. The extreme case is when the actual set of malicious input values is $\emptyset$ while $VS \neq \emptyset$. This

happens when we report a spurious vulnerability. Figure 4.4 shows the vulnerability signature for our example in Figure 4.1.

Given the vulnerability signature DFA M_{vs}, there are two strategies that we can follow to generate the patch function F^P: (1) *match-and-block* and (2) *match-and-sanitize*. In our *match-and-block* strategy (line 9) we generate a patch that simply checks if the input string matches the vulnerability signature DFA M_{vs} by simulating M_{vs}. If it does, it halts the execution without executing the rest of the code.

In our *match-and-sanitize* strategy, instead of blocking the execution, we modify the input in a minimal way to guarantee that the modified input cannot lead to any attack strings. We do this by analyzing the vulnerability signature DFA M_{vs}. Our goal is to find a minimal set of characters, such that if we remove those characters from a given string, the resulting string will not be accepted by the DFA. We use the mincut algorithm [104] to achieve this. The mincut algorithm takes the DFA M_{vs} as input, and produces a set of characters Σ_{mc}. Using Σ_{mc}, we generate a repair function F^P (line 12) that modifies a given input string by deleting a set of characters—using the `replace` function—such that the modified string is not accepted by the M_{vs}. In order to prevent extensive modification to the input, the set of characters to be deleted should be as small as possible. The question, then, is, how do we identify the set of characters to be deleted?

MinCut Algorithm

First, we formalize this problem in automata-theoretic terms [104]. We say $S \subseteq \Sigma$ is an *alphabet-cut* of M, if $\mathcal{L}(M) \cap L_{\bar{S}} = \emptyset$, where $L_{\bar{S}} = (\Sigma \setminus S)^*$ is the set of all strings that do not contain any character in S. The *min-alphabet-cut* problem is finding the alphabet-cut S_{min}, such that for any other alphabet-cut S, $|S_{min}| \leq |S|$.

The min-alphabet-cut problem can also be stated in graph-theoretic terms. Given a DFA M, an *edge-cut* of M is a set of transitions $E \subseteq \delta$ such that if the set of transitions in E are removed from the transition relation δ then none of the states in F are reachable from the initial state q_0. Let S_E denote the set of symbols of the transitions in E. If E is an *edge-cut* of M then S_E is an *alphabet-cut* of M. Hence, finding the min-alphabet-cut is equivalent to finding an edge-cut with minimum set of distinct symbols.

Note that, if M accepts the empty string then there will not be any edge (or alphabet) cut since the initial state would be an accepting state. For the rest of our discussion we assume that $\mathcal{L}(M) \neq \emptyset$ (we can easily handle the cases where it accepts the empty string by first testing if the input string is empty and then inserting a single character to the input if it is).

It has been shown that the min-alphabet-cut problem is NP-hard [104], so, rather than trying to find the optimum solution, we can consider using efficient heuristics that give a reasonably small cut that is not necessarily the optimum solution. One heuristic solution is to minimize the number of edges in a cut rather than the number of distinct alphabet symbols. Given a DFA M, a *min-edge-cut* of M is an edge-cut E_{min} such that for any other edge-cut E, $|E_{min}| \leq |E|$. Note that the min-edge-cut minimizes the number of edges in the edge-cut whereas the

min-alphabet-cut minimizes the set of symbols on the edges in the edge-cut. Interestingly, even though the min-alphabet-cut problem is intractable, there is an efficient algorithm for computing the min-edge-cut. We use the Ford-Fulkerson's max-flow min-cut algorithm [22] to find a min-edge-cut E_{min} where the complexity of the algorithm is $O(|\delta|^2)$. Note that $|S_{min}| \leq |E_{min}|$, i.e., the min-edge-cut provides and upper bound for the min-alphabet-cut. So if the min-edge-cut is small then the set of distinct symbols on the edges of the min-edge-cut will give us a good approximation of the S_{min}. Figure 4.4 shows the mincut of the vulnerability signature DFA for our example in Figure 4.1. The mincut alphabet is $\Sigma_{mc} = \{<\}$.

Once we compute an alphabet-cut Σ_{mc}, we generate the patch F^P (line 12) with a `replace` statement that deletes the symbols in Σ_{mc} from the input if it matches M_{vs}, making sure that the resulting string does not match M_{vs}. The function F^P is a sound repair that will guarantee that $\text{POST}(F \circ F^P, \Sigma^*) \cap \mathcal{L}(P) = \emptyset$.

4.1.4 Empirical Evaluation

We experimentally evaluate our approach for XSS vulnerabilities using five known vulnerabilities. Then we apply our analysis on three open source web applications looking for XSS and SQLI vulnerabilities. In our experiments we use an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We used 8 bits to encode each ASCII character. During extraction phase, for the XSS vulnerabilities the sinks for which we extracted sanitizers include the `printf` and `echo` functions. For SQLI vulnerabilities the sinks include the `mysql_query` function. We used the attack pattern Σ^* `<SCRIPT`Σ^* for XSS and reported a

vulnerability for each vulnerable sanitizer that can return an attack string (matching the attack pattern). We used the attack pattern Σ^* `' or 1=1` Σ^* for SQLI.

We first perform vulnerability analysis using forward analysis algorithm (Algorithm 1) to compute the post-images. Then for each detected vulnerability, we generate the corresponding vulnerability signature(s) using the backward analysis algorithm to compute the pre-images. Note that, during this analysis, we only used the general replace algorithm not the customized and optimized ones (Algorithm 2). Finally, we synthesize sanitization code based on the vulnerability signature DFAs that we computed. For match statements, we generate a C extension to PHP language that simulates the vulnerability signature automaton. For replace statements, we use the PHP function `preg_replace` to delete characters that are identified by the alphabet-cuts generated from the vulnerability signature automata.

Patching Known Vulnerabilities

We first analyzed five benchmarks manually extracted from (1) `MyEasyMarket-4.1` (a shopping cart program), (2) `BloggIT-1.0` (a blog engine), and (3) `proManager-0.72` (a project management system). Each benchmark represents a known XSS vulnerability [9] containing a single sink where the statements in the original program that are not related to this sink have been removed. The dependency graphs (see 2.6) for the extracted sanitizers from these benchmarks are rather small (around 20-30 nodes) but include loops, concatenations with large constants, and nested replacements (from customized or PHP built-in sanitization routines). We believe that they present typical string manipulations in PHP Web applications.

Vulnerability Analysis: The first step is to detect whether there is a vulnerability in the sanitizer function (lines 1-5 of Algorithm 12). Table 4.1 shows the performance of the vulnerability analysis. It shows the spent time and consumed memory by the analysis besides the number of states (#states) and the number of BDD nodes (#bdds) of the DFA M (the transition relation of the DFA is stored symbolically as a multi-terminal decision diagram) that represents the post-image of a sanitizer. For all five benchmarks, $\mathcal{L}(M)$ is not an empty set and we conclude that there is a vulnerability in all benchmarks. (#inputs indicates the number of `input` variables.)

	time(s)	mem(kb)	res.	#states / #bdds	#inputs
1	0.08	2599	vul	23/219	1
2	0.53	13633	vul	48/495	1
3	0.12	1955	vul	125/1200	2
4	0.12	4022	vul	133/1222	1
5	0.12	3387	vul	125/1200	1

Table 4.1: Vulnerability analysis performance for benchmarks.

Signature Generation: The next step is to generate their vulnerability signatures (line 7 of Algorithm 12). Table 4.2 summarizes the performance of vulnerability signature generation. The last column shows the size of the vulnerability signature DFA (since benchmark 3 contains multiple (two) inputs, we ignore it). It can be seen that most of them are computed within seconds except for benchmark 2. Taking a closer look, we found that it consists of several nested replacement operations that cause the pre-image computations to blow-up.

	time(s)	mem(kb)	#states /#bdds
1	0.46	2963	9/199
2	41.03	1859767	811/8389
4	2.33	32035	91/1127
5	5.02	14958	20/302

Table 4.2: Signature generation performance for benchmarks.

Sig.	1	2	4	5
#edges	1	8	4	4
mincut-alphabet	$\{<\}$	$\{S,',"\}$	$\{<,',"\}$	$\{<,',"\}$

Table 4.3: Minimum edge and alphabet cuts.

Finally, following the *match-and-sanitize* strategy, we generate the mincut alphabet based on the vulnerability signature (line 11-12 of Algorithm 12). Table 4.3 shows the number of edges in the min-edge-cut for the vulnerability signature automata we computed, and the alphabet-cuts that correspond to these min-edge-cuts .

Our results show that our techniques are very effective. As we can see, the min-edge-cut results in a very small alphabet-cut, and the optimum solution in 1. We favored non-alphanumeric characters while generating the alphabet-cuts by increasing the weights of the alphanumeric characters during the min-cut algorithm (we assume that alphanumeric characters are more likely to represent normal user input and we prefer not to delete them unless necessary). This resulted in having non-alphanumeric characters in all the cuts but one. Notice that, existing sanitization operations such as `mysql_real_escape_string` that are in the analyzed benchmarks can add some additional characters to the alphabet-cut due to the conservative nature of our analysis that over-approximates the vulnerability signatures. For example, in 2 `'` and `"` are introduced by the PHP sanitization operation `mysql_real_escape_string`.

Finally, taking a close look at the vulnerability signature of (1) `MyEasyMarket-4.1`. The vulnerability signature actually accepts α^* `<`α^* `s`α^* `c`α^* `r`α^* `i`α^* `p`α^* `t`α^* with respect to the attack pattern Σ^* `<script`Σ^*. α is the set of characters, e.g., `!`, that are deleted in the program. An input such as `<!script` can bypass the filter that rejects Σ^* `<script`Σ^* and exploit the

vulnerability. This shows that simply filtering out the attack pattern can not prevent its exploits. On the other hand, the exploit can be prevented using our vulnerability signature instead.

4.1.5 Analyzing and Patching Open Source Applications:

We applied our analysis to three open source PHP web applications: (1) `Webchess 0.9.0` (a server for playing chess over the internet) (2) `EVE 1.0` (a tracker for players activity for an online game), and (3) `Faqforge 1.3.2` (a document management tool). The sizes of these applications are shown in 4.4. These applications are downloaded from *sourceforge* and are directly analyzed without any manual modification.

	Application	# of php files	total loc	# of sanitizers	
				XSS	SQLI
1	Webchess 0.9.0	23	3375	421	140
2	EVE 1.0	8	906	114	17
3	Faqforge 1.3.2	10	534	375	133

Table 4.4: The sizes of analyzed applications.

Table 4.5 and Table 4.6 summarize the results of our XSS and SQLI vulnerability analysis respectively and the performance for signature generation. Notice that we omited results for pre-image computation for multi-input sanitizers since we do not generate vulnerability signatures for vulnerabilities in such sanitizers. We discovered 55 XSS and 61 SQLI vulnerabilities in these applications. (single, two, three) indicates the number of detected vulnerabilities that have single input, two inputs and three inputs, respectively. For example, all detected vulnerabilities in Faqforge have single input (denoted as (20, 0, 0)). That is, all sanitizers extracted from this application are single-input sanitizers.

	# of Vul. (single, two, three)	Time (seconds) total	forward	backward	Memory (Kb) average
1	(24, 3, 0)	39.78	1.73	0.92	16850
1	(0, 0, 8)	160.7	6.80	–	125382
3	(20, 0, 0)	7.87	0.22	0.22	9948

Table 4.5: XSS vulnerability analysis results.

	# of Vul. (single, 2, 3, 4)	Time (seconds) total	forward	backward	Memory (Kb) average
1	(43, 3, 1, 2)	72.67	4.87	12.039	136790
2	(8, 3, 0, 0)	18.7	1.5	8.47	17280
3	(0, 0, 0, 0)	6.7	–	–	< 1

Table 4.6: SQLI vulnerability analysis results.

As shown in Table 4.5 and Table 4.6, the analysis cost seems affordable: the *total* time indicates the total time to analyze all php files in these applications from start to the end, which includes extraction time and policy-based repair time (vulnerability analysis and vulnerability signature generation for single-input sanitizers). It ranges from 7 seconds to 161 seconds. The *forward* time indicates the total time to detect vulnerabilities in all extracted sanitizers including post-image computation using forward analysis (Algorithm 1) and intersection with the attack pattern. The *backward* time indicates the total time to generate vulnerability signatures for all detected vulnerabilities in single-input sanitizers i.e., time to compute pre-image using backward analysis (Algorithm 2).

Mincut performance: The average time spent in generating the alphabet-cut from the vulnerability signature automata for for XSS (SQLI) was 0.05 (0) seconds per automaton for Faqforge and 0.06 (0.07) seconds per automaton for Webchess (we ignore EVE since it only contains multi-input sanitizers).

All of the generated alphabet-cuts contain only a single character per each input. For each XSS single track automata the cut is only the character < which is the optimum cut (consequently the optimum sanitization with respect to the attack pattern). The automatically generated sanitization (replace) statements from our analysis were almost the same as the ones that are manually written except that they delete the < character instead of replacing it with the HTML entity "`<`" as is typically done in manual sanitization. On the other hand, for each SQLI single track automata the cut is only the character = which is the optimum cut (consequently the optimum sanitization with respect to the attack pattern).

Match performance: We evaluated the overhead of running the generated match code to simulate one of the vulnerability signature automata from Webchess against a manually written PHP `preg_match` that performs the same task. Both `preg_match` and our `stranger_match` are written as C extensions to PHP and called from a PHP script on the same input. We evaluated the overhead of running this code on 10 sets of randomly generated strings each containing 1000 strings of the same length. The lengths started from 100 characters per string for the first set, adding 100 more characters for each new set and going up to 1000 characters per string for the last set. The results are shown in Figure 4.5. Clearly, automatically generated match does not cause an extra overhead compared to the manually written one. The time of matching a 1000 character string to the vulnerability signature automaton is less than 0.35 milliseconds.

How to use our analysis result: Our analysis produces two artifacts: a PHP extension that contains a number of patch functions F^P where each function contains a `stranger_match_*` code that simulate vulnerability signature DFA and a `preg_replace` to delete mincut alphabet.

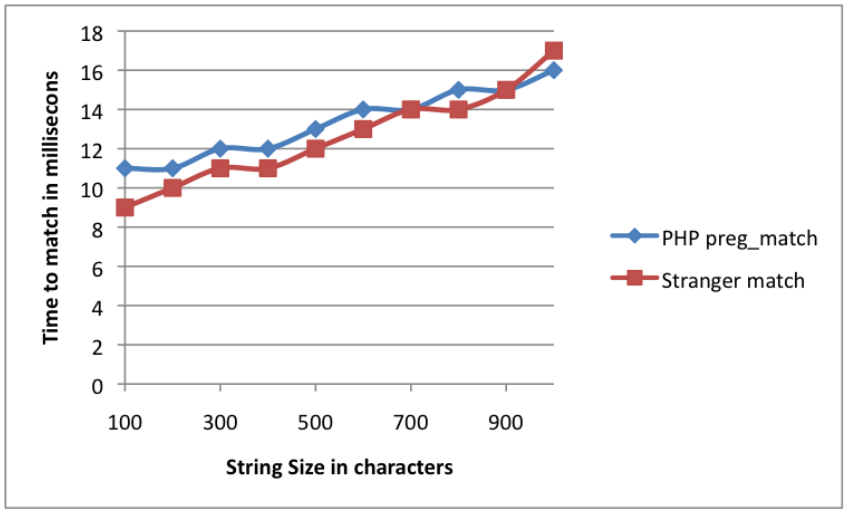

Figure 4.5: Input matching overhead using `stranger_match` to simulate vulnerability signature DFA.

We generate one patch function F^P for each extracted sanitizer. In PHP, user input from such places as `$_GET` and `$_POST` is always available at the first program point in the script. This means that if we want to sanitize the inputs, we need to do it at the first PHP line of the target script. Inserting calls to these patch functions can easily be automated as we have the file names for each of the input variables along with the variables' names from the parsing phase. Note that we are analyzing PHP scripts statically in a sound manner where we only deal with one script at a time along with all the files it includes

We used the result of our analysis to sanitize the three applications above by placing the automatically generated sanitization statements at the beginning of each vulnerable script. Then we ran our forward vulnerability analysis which reported zero vulnerabilities with regard to the attack pattern mentioned above.

```
1  sanitizer isValidEmail(x) {
2     x = trim(x);
3     if(!x matches
4         /^[a-z0-9!#$%&'*+/=?^_`{|}~-]+
5         (?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@
6         (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
7         [a-z0-9](?:[a-z0-9-]*[a-z0-9])$/))
8     {
8         reject;
10    }
11    return x;
12 }
```

Figure 4.6: An over-constrained validator that corresponds to Javascript function in Figure 1.7.

4.2 Verifying Client-Side Input Validation Against Minimum and Maximum Policies

In the previous section we showed how to automatically detect, given a security policy, if a server-side sanitizer function is under-constrained. An under-constrained sanitizer is a sanitizer that returns some bad output values that are not supposed to be in its output (i.e., its post-image). In this section we expand the policy-based bug detection to deal with over-constrained sanitizers [3]. An over-constrained sanitizer is a sanitizer that does not accept some good output values that are supposed to be in its output (i.e., its post-image). If this problem happens then it will affect the correctness of the web application even if it was on the client-side. For example, if a Javascript email address sanitizer is over-constrained, then it may not allow the user input (i.e., the email address) to reach the server even if it is correct. If this happens, then it will prevent the user from interacting with the web application in a correct way.

```
Email      →  /^[a-zA-Z0-9]+[.a-zA-Z0-9_\-]*@
              [.a-zA-Z0-9_\-]+\.[a-zA-Z]{2,6}$/
Date       →  /^(([0-9]{1,2})|[A-Za-z]{3})[\/\-]
              [0-9]{1,2}[\/\-][0-9]{2}([0-9]{2})?$/
Phone      →  /^(\(?[0-9]{3}\)?)?[\- ]?[0-9]{3}
              [\- ]?[0-9]{4}$/
Time       →  /^[0-9]{1,2}:[0-9]{2}([ap]m)?$/
Zip Code   →  /^[0-9]{5}([. ][0-9]{4})?$/
NotEmpty   →  /^.*[^ \n\t].*$/
```

Figure 4.7: Maximum input validation policies.

```
Email      →  /^[a-zA-Z0-9]+@[a-zA-Z]+\.[a-zA-Z]{3}$/
Date       →  /^[0-9]{1,2}\/[0-9]{1,2}\/[0-9]{4}$/
Phone      →  /^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$/
Time       →  /^[0-9]{2}:[0-9]{2}$/
Zip Code   →  /^[0-9]{5}$/
NotEmpty   →  /^.*[^ \n\t].*$/
```

Figure 4.8: Minimum input validation policies.

Consider the example IVSL validator in Figure 4.6 which corresponds to the Javascript function in Figure 1.7. As we have stated in the introduction, this validator is over-constrained since it does not return email addresses with capital letters. The regular expression in lines 4-7 does not allow strings with capital letters since it only allows `[a-z]` but not `[A-Z]`.

4.2.1 Minimum and Maximum Validation policies

To detect both under and over-constrained validators, we expand our policy-based bug detection to use two types of validation policies: *Max* and *Min*. The *Max* policy specifies the maximal set of strings that should be accepted while the *Min* policy specifies the minimal set of strings that should not be rejected. We use regular expressions for specification of the *Max* and *Min* policies. This is a natural choice since regular expressions are well-known and de-

velopers implementing input validation functions commonly use regular expressions in string manipulation functions. Figures 4.7 and 4.8 show a number of maximum and minimum input validation policies that we have used in our analysis. Each policy has two entries: the type of the input field that is checked against this policy and the specification of the policy as a regular expression. The syntax we use for specifying regular expressions is the syntax defined by IVSL. For some simpler input types we have the same maximum and minimum policies.

For example the *Email* policy in Figure 4.7 specifies the correct value for an email address. It is a more restrictive policy than RFC5322 which specifies valid email addresses. This is due to the fact that some major email providers such as Hotmail are much more restrictive in their email address policy than the previous standard. Although some of these policies look simple we were surprised to find that there are many validation functions that do not adhere to them. For example, four out of five validation functions that we found in JavaScript tutorials and textbooks that check for emptiness miss the fact that a form field with only spaces should be considered to be an empty field.

Notice that *Max* policy is the dual of the security policy in previous section. Both of these policies are used to check if a sanitizer is under-constrained but in the opposite way. On one hand, *Max* policy describes a white list of values while, on the other hand, the security policy describes a black list of values.

4.2.2 Min Max Policy Conformance Problem

Given two policies a Max policy P_{max} and a Min P_{min}, specified as regular expressions, and a validator/sanitizer F, we want to verify that F conforms to both policies i.e., the set of strings accepted by F is a subset of the *Max* policy and a superset of the *Min* policy. If F violates one or both of the policies then we consider it to be faulty. In other words, let us assume that the language of the *Max* policy that specifies the maximal valid set of inputs for the given field type is $L(P_{max})$ and the language of the *Min* policy that specifies the minimal valid set of inputs for the given field type is $L(P_{min})$. Then the *Min Max Policy Conformance Problem* is to check if $L(P_{min}) \subseteq \textsc{Post}(F, \Sigma^*) \subseteq L(P_{max})$.

4.2.3 Min Max Policy Conformance Algorithm

Algorithm 13 shows the Min Max Policy Conformance Algorithm. The algorithm takes as its input a validator F, two policies, a minimum policy P_{min} and a maximum policy P_{max} and the type of the function F, either a validator or a sanitizer. The algorithm uses the automata-based string analysis (from Chapter 3) to verify if F conforms to both policies i.e., $L(P_{min}) \subseteq \textsc{Post}(F, \Sigma^*) \subseteq L(P_{max})$. In the algorithm, each variable that has a name starting with M represents a DFA, each variable with a name starting with F represent a validator. The algorithm uses the DFA operations $\cap, \cup, \setminus, —, \mathcal{A}, \mathcal{L}$ as described in 3.2.4.

Since the string analysis is an undecidable problem in general, it is not possible to compute $\textsc{Post}(F, \Sigma^*)$ precisely. However, using our automata-based string analysis approach we can

Algorithm 13 MINMAXPOLICYCONFORMANCE($F, P_{min}, P_{max}, type$)

```
1:  M_min := A(L(P_min));
2:  M_max := A(L(P_max));
3:  M_Accept := A(POST+(F, Σ*));
4:  if (M_Accept ⊈ M_max) then
5:      if (type = validator) then
6:          s := GENERATESATISFYINGSTRING(M_Accept \ M_max);
7:          return s; //counter example
8:      else
9:          M_Accept_i := A(PRE+(F, M_Accept \ M_max));
10:         s := GENERATESATISFYINGSTRING(M_Accept_i);
11:         return s; //counter example
12:     end if
13: else
14:     REPORT("Conforms with Max Policy");
15: end if
16: M_Reject := A(POST+⊥(F, Σ*));
17: if (M_Reject ∩ M_min ≠ ∅) then
18:     if (type = validator) then
19:         s := GENERATESATISFYINGSTRING(M_Reject ∩ M_min);
20:         return s; //counter example
21:     else
22:         M_Reject_i := A(PRE+(F, M_Reject ∩ M_min));
23:         s := GENERATESATISFYINGSTRING(M_Reject_i);
24:         return s; //counter example
25:     end if
26: else
27:     REPORT("Conforms with Min Policy");
28: end if
```

compute an over-approximation $\text{POST}^+(F, \Sigma^*)$ such that $\text{POST}(F, \Sigma^*) \subseteq \text{POST}^+(F, \Sigma^*)$ (line 3). Note that, if $\text{POST}^+(F, \Sigma^*) \subseteq L(P_{max})$, then we can be sure that F conforms to the *Max* policy (lines 4,14). If $\text{POST}^+(F, \Sigma^*) \not\subseteq L(P_{max})$, on the other hand, we cannot definitely say that F violates the *Max* policy. Since $\text{POST}^+(F, \Sigma^*)$ is an over-approximation, we may have a false positive. In this case, in order to figure out if the validator/sanitizer is really faulty, we generate a string input value that would result in an output that is accepted by the function but not in the language of the maximum policy.

If we have a validator then since a validator does not modify its input, then we do not need to compute the pre-image to generate such input value. We just need to generate a string $s \in \text{POST}^+(F, \Sigma^*) \setminus L(P_{max})$ (line 6) and execute the original input validation function that

corresponds to the extracted validator F on that value. If the input validation function returns true for this input, then we are sure that the input validation function violates the *Max* policy and the generated string s serves as a counter-example demonstrating the policy violation.

If on the other hand we have a sanitizer, then we need to first compute its pre-image given the set of possible output strings by the sanitizer that we computed which are not in the language of the maximum policy i.e., $\text{PRE}^+(F,\text{POST}^+(F,\Sigma^*) \setminus M_{max}$. Then we generate a string s in the result pre-image to serve as a counter example (lines 9-11).

We cannot use the over-approximation $\text{POST}^+(F,\Sigma^*)$ to check conformance to the *Min* policy since $L(P_{min}) \subseteq \text{POST}^+(F,\Sigma^*)$ does not imply that $L(P_{min}) \subseteq \text{POST}^+(F,\Sigma^*)$. In order to check conformance to the *Min* policy we need an under-approximation of $\text{POST}^+(F,\Sigma^*)$. However, since our string analysis is a sound analysis technique, it can only generate over-approximations. We solve this problem by using our string analysis to compute an over-approximation of the set of values that reach the negative sinks (i.e., `reject` statements) in F. Let us call this set $\text{POST}_\perp(F,\Sigma^*)$. Using our string analysis we compute $\text{POST}^+_\perp(F,\Sigma^*)$ such that $\text{POST}_\perp(F,\Sigma^*) \subseteq \text{POST}^+_\perp(F,\Sigma^*)$ (line 16). Then, we check if $\text{POST}^+_\perp(F,\Sigma^*) \cap L(P_{min}) = \emptyset$ (line 17). If the intersection of $\text{POST}^+_\perp(F,\Sigma^*)$ and $L(P_{min})$ is empty, then we can be sure that the validator F does not reject (at the negative sinks) any value that is allowed by the *Min* policy, i.e., it conforms to the *Min* policy (line 27). If, on the other hand, $\text{POST}^+_\perp(F,\Sigma^*) \cap L(P_{min}) \neq \emptyset$, we cannot be sure that F violates the *Min* policy since it can be a false positive. In this case we generate a string $s \in \text{POST}^+_\perp(F,\Sigma^*) \cap L(P_{min})$ if we have a validator (line 19)

or a string $s \in \text{PRE}^+(F,\text{POST}^+_\perp(F, \Sigma^*) \cap M_{min}$ if we have a sanitizer for the same reason stated for the maximum policy (lines 22, 23).

Finally, we execute corresponding input validation function on input s. If the input validation function returns false for this input, then we can be sure that F violates the *Min* policy and s is a counter-example demonstrating this policy violation.

4.2.4 Empirical Evaluation

The approach presented in this paper can be used both as a forward engineering approach (as an analysis used during the application development) or as a reverse engineering approach (by automatically extracting and analyzing input validation functions after deployment). We evaluated the forward engineering scenario on input validation functions collected from tutorials and books for teaching JavaScript. We evaluated the reverse engineering scenario on several real-world applications by extracting and analyzing their input validation functions. In our experiments we used a MacBook Pro with a 2.53 GHz core 2 due processor and 4 GB of memory.

4.2.5 Verifying Stand-Alone Input Validation Functions

In this section we show the results of verifying 23 JavaScript validation functions collected from a JavaScript book [72] and several JavaScript input validation tutorials on the internet. The book on JavaScript and the tutorials represent a set of validation functions that should have been written very carefully given that these are examples used for teaching JavaScript

	FuncName	Source	Type	Time (seconds)	Memory (MB)	Result	
						Max Policy	Min Policy
1	validateEmail	Book - HeadFirst JavaScript	Email	0.640	9.6	X	√
2	validateEmail	Tut - http://www.webcheatsheet.com/	Email	0.150	24.3	X	√
3	emailValidator	Tut - http://www.tizag.com/	Email	0.140	44.7	X	√
4	checkEmail	Tut - http://developer.apple.com/	Email	0.560	8.3	X	√
5	isEmailAddress	Tut - http://www.devshed.com/	Email	0.140	19.9	X	X
6	isAlphabet	Tut - http://www.tizag.com/	Alphabet	0.014	24.1	√	√
7	isAlphabetic	Tut - http://www.devshed.com/	Alphabet	0.014	25.0	√	√
8	isAlphanumeric	Tut - http://www.tizag.com/	AlphaNum	0.018	26.1	√	√
9	isAlphaNumeric	Tut - http://www.devshed.com/	AlphaNum	0.009	27.2	√	√
10	isNumeric	Tut - http://www.tizag.com/	Numeric	0.013	28.0	√	√
11	isNumber	Tut - http://www.devshed.com/	Numeric	0.011	28.7	√	√
12	validateDate	Tut - http://www.the-art-of-web.com/	Date	0.041	32.0	√	√
13	validateDate	Book - HeadFirst JavaScript	Date	0.507	5.2	√	X
14	validatePhone	Tut - http://www.webcheatsheet.com/	Phone	0.075	15.6	√	√
15	checkPhone	Tut - http://developer.apple.com/	Phone	0.058	27.4	√	√
16	validatePhone	Book - HeadFirst JavaScript	Phone	0.076	37.5	X	X
17	validateTime	Tut - http://www.the-art-of-web.com/	Time	0.031	43.1	√	√
18	validateZipCode	Book - HeadFirst JavaScript	ZipCode	0.040	48.1	√	√
19	validateEmpty	Tut - http://www.webcheatsheet.com/	NotEmpty	0.448	0.8	X	√
20	notEmpty	Tut - http://www.tizag.com/	NotEmpty	0.013	1.8	X	√
21	isEmpty	Tut - http://developer.apple.com/	NotEmpty	0.017	2.9	X	√
22	isEmpty	Tut - http://www.devshed.com/	NotEmpty	0.014	4.2	√	√
23	validateNonEmpty	Book - HeadFirst JavaScript	NotEmpty	0.021	5.9	X	√

Table 4.7: Results of our analysis on input validation functions collected from JavaScript books and tutorials.

These functions also include a wide variety of string operations and predicates that one expects to see in a JavaScript application and hence form a good benchmark for evaluating the effectiveness of our string analysis techniques. The functions we analyzed cover all the policies mentioned in Section 4.2.1. Five of the validation functions were changed slightly to conform to our model of input validation and sanitization functions, so that the modified function do not return empty/nonempty error messages in case of acceptance/rejection of input.

Results. The total time it took for analyzing the 23 functions was 3.05 seconds during which 488 lines of code have been analyzed. Table 4.2.5 shows the individual results for each validation function using our analysis. The first column is the validation function name. The second one is the source for this function. The third column shows the type of data that is validated by this validation function which is also the type of policy used to verify the function itself. Columns four and five show the performance of our string analysis in terms of time and memory. Last column shows the result for verification against both the maximum and the minimum policies. Failing the maximum policy means that the function accepts and returns some values that are invalid according to our policy. For example, the first function does not satisfy the maximum policy which means that it returns email addresses that are considered to be invalid by our policy. On the other hand failing the minimum policy means that the function rejects some values (at the sink) that are correct according to our policy. For example, function number 13 does not satisfy the minimum policy which means that it rejects some time inputs that we consider to be correct.

Discussion. Among the 23 functions we have analyzed, 10 of them violated a maximum policy while 3 of them violated a minimum policy. We tested these results using the counter-examples generated by our analysis and did not find any false positives due to over-approximation. It is interesting to see that all the functions that validate the email addresses failed to comply with our maximum email policy (which is the most complicated policy we used). It is even more interesting to see that four of the five functions that validate non emptiness failed to comply with our maximum non emptiness policy although this check seems very simple.

The most subtle error in email validation functions is the one where the developer forgot to escape the dash character inside a character class. This results in accepting email addresses with invalid characters such as `[`. Another problem that we have found is the usage of a black list to block invalid characters in an email address rather than a white list where only valid characters are accepted. All of these black lists miss at least one invalid character. We think that the white list approach that we used in our policy is much simpler and less error prone.

The phone validation function number 16 failed to comply with our minimum policy because it rejects a phone number that has two parentheses around its area code. This is the most common format to write a phone number in US and, hence, in our minimum policy we specify that this should be accepted as valid input.

The most surprising result is the errors in non emptiness checks. The reason behind the four failures is that these four functions accept and return a field value that only consists of white space characters while such value should be considered empty. Such a validation error

will likely cause an unnecessary interaction with the web server, and if the same validation check is also erroneous at the server side it can lead to fatal errors in the application.

Our results demonstrate that 1) writing input validation checks in JavaScript is an error-prone task and even the sample validation functions shown in tutorials and books on JavaScript contain errors. 2) Using the string analysis techniques we presented, we can efficiently check the conformance of a JavaScript input validation function to a given input validation policy.

4.2.6 Verifying Input Validation in Deployed Web Applications

We applied our verification technique to a number of real world websites that use JavaScript to validate their HTML form inputs. For each of these websites we pick an HTML form, fill it out, and submit it. During submission we automatically extract the validation function for one target field (as described in 2.5) and analyze this function statically (Algorithm 13) to see if it confirms to our predetermined minimum and maximum policies for the type of that field. We applied our technique on fields with common input format such as email, phone number, etc. Our analysis can also be applied to other fields that need specific policies chosen by the organization running the website such as username and password fields.

	Source	Type	Code Size (LOC)		Execution Trace Size (LOC)		Extraction Time (s)		Analysis Time (s)		Result	
			Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Max	Min
1	www.google.com	Email	407	407	103	138	17.82	13.42	0.57	0.47	X	√
2	www.bloomberg.com	Email	2138	2133	776	769	14.85	12.84	0.48	0.44	X	√
3	www.bloomberg.com	PhoneNumber	2138	2138	808	808	18.81	16.92	0.45	0.44	X	√
4	www.stc.com.sa	Email	124	102	93	62	67.88	54.23	0.65	0.50	X	X
5	www.koora.com	Email	35	35	10	11	10.50	7.67	0.60	0.45	X	√
6	www.multiply.com	Email	171	171	230	271	8.74	9.22	0.50	0.47	X	√
7	www.acm.com	Email	1644	1644	867	871	10.19	11.41	0.61	0.43	X	X
8	www.netflix.com	Email	518	518	2411	2411	24.27	24.27	-	-	X	√
9	www.bjunkie.org	Email	104	104	20	21	12.16	12.01	0.50	0.52	X	√
10	www.pcmag.com	Email	514	514	31808	31808	32.13	32.13	-	-	X	√
11	www.pcmag.com	ZipCode	570	570	31850	31850	33.48	31.81	0.33	0.43	X	√
12	www.apple.com	Email	1925	1925	1783	1783	14.15	13.95	0.51	0.42	X	√
13	www.acm.com	ZipCode	1666	1644	1052	1047	12.70	10.42	0.49	0.66	X	√

Table 4.8: Results of our analysis on deployed websites.

Results. Table 4.8 shows the results of applying our analysis to a number of websites. Each row represents the results for extracting and verifying the validation code for a single input field in a form in the given website. Since this is done twice for a valid and invalid input we show two sub-columns for each. The first column is name of website where we got the form from. The second column shows the type of data that is validated by this validation function which is also the type of policy used to verify the function itself. Column three shows the size (in lines of code) of the form submission handling code including the validation code from which we extracted the validation function. Column four shows the number of lines of code that has been executed as part of submitting the form. Column five shows the time it takes to dynamically extract the validation function while column six shows the time to statically analyze the extracted function. In two cases there was no validation code in the application and the extracted validation function was empty. Hence, there was no string analysis done for these two cases and the corresponding column is left empty. Last column shows the result for verification against both the maximum and the minimum policies where an X means a policy violation.

Policy Violations. We have found a policy violation in each of the websites that we tested. Some of these policy violations are a result of subtle bugs in the validation code, some of them are a result of writing light weight validation code or even not writing any, and some of them are due to minor differences between our policies and websites' policies.

Two subtle bugs that we have found are in 7 and 4. In 7, a condition in the validation code was supposed to reject any email that ends with "`@csta.acm.org`". This was written

as "`if(ckEmailVal.match("@csta.acm.org")){..}`" The programmer forgot to escape the dot as a special character in the regular expression so when JavaScript converts this string into a regular expression, it will interpret dot as any character. Our analyzer output "**`A@cstaAacm.org`**" as an example for an email that should not be rejected.

In 4 (a website for a large telecom company) the developers claim that they follow the RFC standard for email addresses. We found that they disallowed capital characters from emails with no obvious reason and our analyzer gave the following example that should not be rejected "**`A@A.AAA`**".

Some of the other websites have lightweight validation code that will accept incorrect input. For example, 2 only checks for presence of `@` and `.` in an email and our analyzer gave "**`.@n`**" as an invalid email that is accepted. Number 1 accepts any email that does not have ␣ (whitespace), `'` or `"` in it. Our analyzer gave "**`0x1f@0x1f.0x1f`**" as an invalid email that is accepted. This latter example was randomly generated and happened to be not printable but for this case there are counter examples with printable characters. Finally, two websites 8 and 10, had no validation code at all and our slicer returned an empty validation function, meaning that all input values are accepted. In this corner case there is no need to run the string analysis and we only report a maximum policy violation.

Chapter 5

Differential Bug Detection and Repair

Effectiveness of policy-based bug detection and repair that we presented in previous chapter depends on the correctness and precision of the written policies in characterizing good and bad string values. It is often possible, for instance, to encode well-known attacks into security policies (in the form of attack patterns) and write down policies for common input fields such as email address and zip code. In other cases, however, the checks to be performed on the inputs are specific to the functionality of the web application, and the input validation may be tightly coupled with and dependent on the application logic. Because they are specific to individual applications, there are no pre-specified policies that can be used to assess these types of input checks. In these cases, to make sure that the input validation is adequate, it would be necessary to specify a different policy for each different application, which is a tedious and error-prone task.

In this chapter, we present a differential analysis and repair approach [2, 4] for analyzing and repairing validation and sanitization functions in web applications. This new approach eliminates the need to write manual specifications by exploiting redundancy in input validation and sanitization code.

Web application developers often introduce redundant input validation and sanitization code in the client and server-side code of a web application. The checks done on the client-side improve the responsiveness of the application by preventing unnecessary communication with the server and reduce the server load at the same time. However, since a malicious user can by-pass the client-side checks, it is necessary to re-validate and re-sanitize at the server-side. Moreover, many applications repeat the checks for different types of fields in different parts of the application which can be exploited to obtain multiple instances of the validation and sanitization code with the same intended functionality. Finally, across different applications, one can easily find multiple instances of validation and sanitization code used to check standard formats (such as email) or to protect against same class of vulnerabilities (such as SQL injection and XSS). Using the differential analysis and repair techniques presented in this chapter, we exploit these redundancies within and application and across applications, to automatically detect and repair differences between input validation and sanitization functions by comparing them against each other.

Motivating Example. Let us take a look at this example taken from a real web application called JGOSSIP (http://sourceforge.net/ projects/jgossipforum/), a message board written using Java technology. Figures 5.1 and 5.2 show two snippets of client- and server-side valida-

```
1  <html>
2  ...
3  <script>
4  function validateEmail(form) {
5    var emailStr = form["email"].value;
6    if(emailStr.length == 0) {
7      return true;
8    }
9    var r1 = new RegExp("( )|(@.*@)|(@\\.)");
10   var r2 = new RegExp("^[\\w]+@([\\w]+\\.[\\w]{2,4})$");
11   if(!r1.test(emailStr) && r2.test(emailStr)) {
12     return true;
13   }
14   return false;
15 }
16 </script>
17 ...
18 <form name="subscribeForm" action="/Unsubscribe"
19     onsubmit="return validateEmail(this);">
20   Email: <input type="text" name="email" size="64" />
21   <input type="submit" value="Unsubscribe" />
22 </form>
23 ...
24 </html>
```

Figure 5.1: JavaScript and HTML code snippets for client-side validation.

```
1  public class Validator {
2    public boolean validateEmail(Object bean, Field f, ..) {
3      String val = ValidatorUtils.getValueAsString(bean, f);
4      Perl5Util u = new Perl5Util();
5      if (!(val == null || val.trim().length == 0)) {
6        if ((!u.match("/( )|(@.*@)|(@\\.)/", val))
7          && u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/",
8                     val)) {
9          return true;
10       } else {
11         return false;
12       }
13     }
14     return true;
15   }
16 ...
17 }
```

Figure 5.2: Java server-side validation code snippet.

tion code, respectively, from this application (we slightly simplified the code to make it more readable and self-contained)[1]. The user fills the client-side form, shown on lines 18–22 of

[1]We present the original functions rather than the IVSL extracted sanitizers to show an example of an actual difference between two validation functions written in different languages in a web application

Figure 5.1, by providing an email address to the HTML input element with name "email" and by clicking on the "Submit" button. When this button is clicked, the browser invokes the JavaScript function `validateEmail`, which is assigned to the `onsubmit` event of the form. This function first fetches the email address supplied by the user from the corresponding form field. It then checks if this address has zero length and, if so, accepts the empty address on line 6. Otherwise, on lines 9 and 10, the function creates two regular expressions. The first one specifies three patterns that the email address should not match: a single space character, a string with the `@` symbol on both ends, and the string "`@.`". The second one specifies a pattern that the email address should match: start with a set of alphanumeric characters, followed by symbol `@`, further followed by another set of alphanumeric characters, and finally terminated by a dot followed by two to four additional alphanumeric characters. If the email address does not match the first regular expression and matches the second one, this function returns true, indicating acceptance of the email address (line 12), and the form data is sent to the server. Otherwise, the function rejects the email address by returning false on line 14. This results in an alert message to inform the user that the email provided is invalid.

When the form data is received by the server, it is first passed to the server-side validation function. For the specific form in this example, the validation function used is method `validateEmail` from class `Validator`, which is shown in Figure 5.2. This method calls a routine on line 3 to extract the value contained in the email field from the form object (`bean`) and stores it in variable `val`. It then uses library `Perl5Util` to perform the regular expression match operations, which allows for using the same Perl style regular expression syntax

used in the client. First, the method checks whether the email string is `null` or has zero length after applying the `trim` function, on line 5. If so, it accepts the string. Otherwise, it checks the address using the same regular expressions used on the client side. As shown on lines 6–12, the address is accepted if it satisfies these regular expression checks, and it is used for further processing on the server side (e.g., it may be sent as a query string to the database); otherwise, it is rejected on the server side, and the user is taken back to the form.

As shown in this example, the regular expression checks are similar on both ends, which emphasizes that validations on both ends should allow or reject the same set of inputs. Otherwise, there would be mismatches that may create problems for the application. As we stated in the Introduction, if the server side is less strict than the client side, this would be considered a vulnerability (even when such a vulnerability is not exploitable) since it violates a common security policy that server-side checks should not be weaker than the client-side checks: a malicious user could bypass the client-side checks and submit to the server an address that does not comply with the required format, which may result in an attack. For example, an attacker could inject SQL code in the email that may result in an SQL injection attack [42]. In general, server-side checks that are less strict than the client-side checks could lead to two types of undesirable behaviors: (1) the server side allows some wrong or malicious data to enter the system, leading to failures or attacks; (2) the client side rejects legitimate values that should be accepted, resulting in the user being unable to access some of the functionality provided by the web application.

In our example, the client-side validation code shown in Figure 5.1 rejects a sequence of one of more white space characters (e.g., " "), for which the condition on line 6 evaluates to false and the regular expression check on line 11 fails, thereby resulting in the function returning false. However, for the same input, the second condition on line 5 of the server-side validation method (Figure 5.2) evaluates to false, due to the `trim` function call, and the string is therefore accepted by the server. This would lead to white spaces being accepted as email addresses by the server, which might in turn lead to failures (e.g., the web application might try to send an email to the user, which would fail due to an invalid email address) or attacks, such as a denial-of-service attack.

To solve the problem in this example we present a differential analysis and repair technique that consists of four main steps:

1. Extracting and mapping input validation and sanitization functions at the client and/or server sides.

2. Identifying and reporting inconsistencies in corresponding input validation and sanitization functions.

3. Automatically repairing inconsistent input validation and sanitization functions against each other.

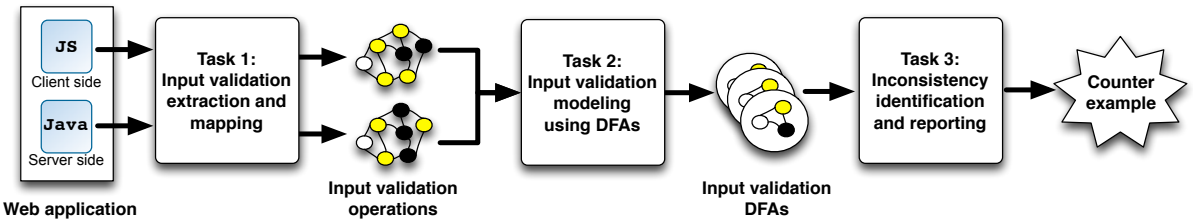

Figure 5.3: High-level view of differential analysis technique.

5.1 Discovering Client- and Server-Side Input Validation and Sanitization Inconsistencies

In this section, we present a new differential string analysis based technique to discover inconsistencies between the client- and the server-side code in web applications [4]. The technique (1) extracts and maps input validation functions at the client and server sides, (2) models input validation functions as deterministic finite automata (DFAs) using string analysis techniques from chapter 3, and (3) identifies and reports inconsistencies in corresponding input validation functions. Figure 5.3 provides a high-level view of the three steps of our technique.

5.1.1 Mapping Input Validation and Sanitization Functions at the Client- and the Server-Side

As far as languages are concerned, we target web applications that use Java on the server-side and JavaScript on the client-side, as these are commonly used languages for the development of web applications. As for technologies, we focus on web applications built using

any Java Enterprise Edition (EE) web framework, such as Struts (`http://struts.apache.org/`), and Spring MVC (`http://www.springsource.org/`). We focused on these technologies because they are the most popular application development frameworks for Java EE web applications. Based on our experience, our work could be extended to handle additional frameworks with relatively low effort.

We start the extraction and mapping process by identifying entry points of the web application, that is, points where user input is read. At the client side, such points correspond to input fields in web forms. Modern Java EE web application frameworks let developers specify in a configuration file[2] the input fields of a web application, together with the JavaScript validation functions to be applied to each field. By leveraging this information, we can identify (1) the complete set of validated inputs on the client side, and (2) the corresponding set of JavaScript functions that are used for validating such inputs.

The identification of the input validation code on the server side is analogous to that of the client side, with the difference that validation is performed using a different language—Java instead of JavaScript—and that parameters are read through calls to input functions. This second difference is almost irrelevant because, analogously to the client side, web application frameworks also allow developers to specify server side inputs and corresponding validation functions. Also for the server side, therefore, an analysis of the web application's configuration file can provide us with (1) the complete set of validated inputs for the server side and (2) the set of Java functions that are used for validating each of these inputs.

[2]Web Deployment Descriptor file, `web.xml`, as specified in the Java EE specification [25]

It is worth noting that web applications could also perform input validation checks directly in the code, without explicitly specifying inputs and corresponding validating functions in a configuration file. In the next section we introduce a different mapping technique for PHP in which we analyze all dynamically generated web pages and corresponding web forms to find and map input validation functions to each other.

5.1.2 Inconsistency Identification Problem

Given a client-side validator/sanitizer F_c and a server-side validator/sanitizer F_s for an HTML input field i, the *Inconsistency Identification Problem* is the problem of finding if F_c and F_s return different output values for the same set of inputs. Formally, two functions F_c and F_s are inconsistent if $\text{POST}(F_s, \Sigma^*) \neq \text{POST}(F_c, \Sigma^*)$.

5.1.3 Inconsistency Identification Algorithm

Algorithm 14 shows the Inconsistency Identification Algorithm. The algorithm takes as its input a client-side single-input validator/sanitizer F_c and a server-side single-input validator/sanitizer F_s both working on the same HTML input field i and the type of the functions if they are validators of sanitizers. The algorithm uses the automata based string analysis described in chapter 3. In the algorithm, each variable that has a name starting with M represents a DFA, each variable with a name starting with F represent a validator. The algorithm uses the DFA operations $\cap, \cup, \setminus, —, \mathcal{A}, \mathcal{L}$ as described in 3.2.4. The algorithm starts by computing

Algorithm 14 INCONSISTENCYIDENTIFICATION($F_c, F_s, type$)

```
1:  M_c := A(POST+(F_c, Σ*));
2:  M_s := A(POST+(F_s, Σ*));
3:  M_{s-c} := M_s \ M_c;
4:  if (L(M_{s-c}) ≠ ∅) then
5:      if (type = validators) then
6:          w := GENERATESATISFYINGSTRING(M_{s-c});
7:          if (EXECUTE(F_c(w)) = reject && EXECUTE(F_s(w)) = accept) then
8:              REPORT("Bug in Server-Side Validator F_s");
9:              return w;  //counter example
10:         end if
11:     else
12:         M_{s_i} := A(PRE+(F_s, M_{s-c}));
13:         w := GENERATESATISFYINGSTRING(M_{s_i});
14:         if (EXECUTE(F_c(w)) = reject && EXECUTE(F_s(w)) = accept) then
15:             REPORT("Bug in Server-Side Validator F_s");
16:             return w;  //counter example
17:         end if
18:     end if
19: end if
20: M_{c-s} := M_c \ M_s;
21: if (L(M_{c-s}) ≠ ∅) then
22:     if (type = validators) then
23:         w := GENERATESATISFYINGSTRING(M_{c-s});
24:         if (EXECUTE(F_s(w)) = reject && EXECUTE(F_c(w)) = accept) then
25:             REPORT("Bug in Client-Side Validator F_c");
26:             return w;  //counter example
27:         end if
28:     else
29:         M_{c_i} := A(PRE+(F_c, M_{c-s}));
30:         w := GENERATESATISFYINGSTRING(M_{c_i});
31:         if (EXECUTE(F_s(w)) = reject && EXECUTE(F_c(w)) = accept) then
32:             REPORT("Bug in Client-Side Validator F_c");
33:             return w;  //counter example
34:         end if
35:     end if
36: end if
```

two DFAs: $M_c(i)$ (client side) and $M_s(i)$ (server side), where $M_c(i) = \mathcal{A}(\text{POST}^+(F_c, \Sigma^*))$ and $M_s(i) = \mathcal{A}(\text{POST}^+(F_s, \Sigma^*))$ (lines 1,2).

Using $M_c(i)$ and $M_s(i)$, we construct two new automata:

- $M_{s-c}(i)$ where $M_{s-c}(i) = M_s(i) \setminus M_c(i)$ (line 3), and
- $M_{c-s}(i)$ where $M_{c-s}(i) = M_c(i) \setminus M_s(i)$ (line 20).

We call $M_{s-c}(i)$ and $M_{c-s}(i)$ *difference signatures,* where:

- $\mathcal{L}(M_{s-c}(i))$ contains strings that are accepted and returned by the server side but rejected by the client side, and
- $\mathcal{L}(M_{c-s}(i))$ contains strings that are accepted and returned by the client side but rejected by the server side.

Let us now consider various scenarios for the difference signatures. If $\mathcal{L}(M_{s-c}(i)) = \mathcal{L}(M_{c-s}(i)) = \emptyset$, this means that our analysis could not identify any difference between the client- and server-side validation functions, so we have no errors to report. Note that, due to over-approximation in our analysis, this does not mean that the client and server-side validation functions are proved to be equivalent. It just means that our analysis could not identify an error.

If $\mathcal{L}(M_{s-c}(i)) \neq \emptyset$, there might be an error in the server-side validation function (line 4). A server-side input validation function should not accept a string value that is rejected by the client-side input validation function—as we discussed earlier, this would be a security vulnerability that should be reported to the developer. Due to over-approximation in our analysis, however, our result could be a false positive. To prevent generating false alarms, we validate the error as follows.

We try to find an example input string that would result in the difference between the client and server-side. Since a validator does not modify its input, then we do not need to compute the preimage of the difference. Instead, we generate a string $s \in \mathcal{L}(M_{s-c}(i))$ and execute both the client and server-side input validation functions by providing s as the input value for the input field i. If client-side function rejects the string, and server-side function accepts it,

then we are guaranteed that there is a problem with the application and report the string s as a counter-example to the developer. If we cannot find such a string s, then we do not report an error (lines 5-11).

If we have sanitizers then we need to do pre-image computation to get the set of input values that resulted in the difference. Then we generate the example from this set i.e., generate $s \in \text{PRE}^{+}(F_s, M_{s-c})$ (lines 12-17).

We also check if $\mathcal{L}(M_{c-s}(i)) \neq \emptyset$ and, if so, we again generate a string to demonstrate the inconsistency between the client and server-side validation functions. Note that client-side validation functions accepting a value that the server rejects may not be as severe a problem as their counterpart (lines 20-35). It is nevertheless valuable to report this kind of inconsistencies because fixing them can improve the performance and response time of the web application and prevent client-side vulnerabilities [81].

5.1.4 Empirical Evaluation

To assess the usefulness of our approach, we implemented our approach in a prototype tool called VIEWPOINTS to perform an empirical evaluation on a set of real-world web applications. In our study, we investigated the following two research questions:

RQ1: Can VIEWPOINTS identify inconsistencies in client- and server-side input validation functions (or establish equivalence between them otherwise)?

Name	*URL*
JGOSSIP	http://sourceforge.net/projects/jgossipforum/
VEHICLE	http://code.google.com/p/vehiclemanage/
MEODIST	http://code.google.com/p/meodist/
MYALUMNI	http://code.google.com/p/myalumni/
CONSUMER	http://code.google.com/p/consumerbasedenforcement
TUDU	http://www.julien-dubois.com/tudu-lists
JCRBIB	http://code.google.com/p/jcrbib/

Table 5.1: Web applications used in our empirical evaluation.

RQ2: Is VIEWPOINTS efficient enough to analyze real-world web applications within acceptable time and memory usage limits?

In the rest of this section, we first describe the details of the experimentation performed for investigating RQ1 and RQ2, and then discuss our results.

Experimental Subjects

For our experiments, we selected seven real-world web applications from two open source code repositories: Sourceforge (`http://sourceforge.net`) and Google Code (`http://code.google.com`). Because our current implementation can handle web applications written using Java EE frameworks, we searched the two repositories for web applications with these characteristics. In addition, we discarded projects with a small user base or with a low activity level, so as to privilege web applications that were more likely to be widely used and well maintained.

Table 5.1.4 shows the list of web applications that we used for our experimentation and the URL from which they were obtained. The first four applications in the list are written us-

ing the Struts framework (`http://struts.apache.org/`): JGOSSIP is a messaging board application; VEHICLE is an application to manage vehicles owned by a company; MEODIST is an application for managing information about a club members; and MYALUMNI is a social network application for school alumni. The last three applications are written using the Spring MVC framework (`http://www.springsource.org/`): CONSUMER is a customer relationship management application; TUDU is an on-line application for managing todo lists; and JCRBIB is a virtual library application that supports user collaboration. Based on their descriptions, these web applications cover a wide spectrum of application domains. Moreover, because of the way they were selected, most of these applications are popular and widely used in practice. JGOSSIP, for instance, has been downloaded almost 30,000 times from its Sourceforge page.

Experimental Procedure and Results

For conducing our experiments, we used a Ubuntu Linux machine with an Intel Core Duo 2.4Ghz processor and 2GB of RAM running Java 1.6. To collect data to answer RQ1 and RQ2, we ran VIEWPOINTS on the web applications considered. For each web application, VIEWPOINTS first analyzed the application's configuration to identify its inputs and corresponding client- and server-side validation functions. It then built client- and server-side validation functions for each input.

Table 5.3 shows relevant data for this part of the analysis. The first column in the table lists the application name, followed by the number of forms extracted (Frm) and the total number

Subject	*Client-Side DFA*							*Server-Side DFA*						
		min		max		avg			min		max		avg	
	Avg size (mb)	S	B	S	B	S	B	Avg size (mb)	S	B	S	B	S	B
JGOSSIP	6.03	4	10	35	706	6	39	6.05	4	24	35	706	6	41
VEHICLE	4.83	4	24	7	41	5	26	4.84	4	24	7	41	5	26
MEODIST	5.67	5	25	5	25	5	25	5.67	5	25	5	25	5	25
MYALUMNI	3.17	4	10	4	10	4	10	3.16	3	24	5	25	5	25
CONSUMER	5.34	4	10	17	132	5	25	5.34	4	24	17	132	7	41
TUDU	6.12	4	10	4	10	4	10	6.12	3	24	23	264	8	68
JCRBIB	5.37	4	10	4	10	4	10	5.38	5	25	5	25	5	25

Table 5.2: Relevant data on the input validation modeling step of the technique.

of inputs across all forms ($Inputs$). Column VI_C (resp., VI_S) lists the number of inputs for which a client-side (resp., server-side) validation function is specified in the configuration. Similarly, column ET_C (resp., ET_S) lists the time taken, in seconds, to extract the summary validation functions for these inputs on the client side (resp., server side). For example, web application CONSUMER contains 3 forms, for a total of 21 inputs. Of these inputs, 14 are validated on the client side, whereas all of 21 of them are validated on the server side. It took 68.4 and 1.1 seconds to extract the validation functions on the client side and server side, respectively. Note that the time required to compute the client-side validation functions is much higher than the time to extract server-side validation functions. This difference is due to the additional time required to perform dynamic slicing on the client side, which in turn requires VIEWPOINTS to load and run JavaScript functions in the browser.

After extracting client- and server-side validation functions for each input, VIEWPOINTS constructed the corresponding DFAs, as described in Chapter 3. Table 5.2 shows details about this part of the analysis. For each application, and both for the client side and the server side, the table shows: the average size of the DFAs in megabytes, followed by the minimum, maximum, and average number of states (column S) and BDD nodes (column B). (The number of BDD

Subject	*Frm*	*Inputs*	VI_C	ET_C(s)	VI_S	ET_S(s)
JGOSSIP	25	83	74	329.80	83	4.38
VEHICLE	17	41	41	155.48	41	2.04
MEODIST	18	62	62	192.20	62	1.93
MYALUMNI	46	141	0	0.00	141	4.28
CONSUMER	3	21	14	68.40	21	1.10
TUDU	3	11	0	0.00	11	0.78
JCRBIB	21	45	0	0.00	45	1.51

Table 5.3: Relevant data on the input validation extraction step of the technique.

nodes represents the size of the symbolic representation of the DFA's transition relation.) As an example, the application TUDU has DFAs with an average of 4 states and 10 BDD nodes for the client side, whereas it has DFAs with an average of 8 states and 68 BDD nodes for the server side. Note that, when client-side validation is absent for an input, the DFA for that input accepts the language Σ^*. Hence, TUDU has a client-side DFA even though it has no client-side validation code (see Tables 5.2 and 5.3).

Finally, VIEWPOINTS compared client- and server-side DFAs to identify possible inconsistencies among them. The results of this comparison for our subjects is shown in Table 5.7. For each application, the table reports the time it took VIEWPOINTS to perform differential string analysis, in milliseconds, and the number of inputs with identified (and confirmed) inconsistencies. Specifically, column M_{C-S} shows the number of inputs for which the client side accepts strings that would be rejected by the server side, whereas column M_{S-C} shows the opposite. For JGOSSIP, for instance, the differential string analysis took around 3 seconds and identified nine client-side inconsistencies and two server-side inconsistencies.

Subject	*Time* (ms)	M_{C-S}	M_{S-C}
JGOSSIP	3220	9	2
VEHICLE	1486	0	0
MEODIST	1745	0	0
MYALUMNI	2853	141	0
CONSUMER	1019	7	0
TUDU	595	11	0
JCRBIB	1168	45	0

Table 5.4: Data on the inconsistency identification step of the approach and overall results.

Discussion

As the results in Table 5.7 show, VIEWPOINTS was able to find both types of inconsistencies: client checks that are more strict than server checks and vice versa. We manually checked all the results and 1) verified that all identified inconsistencies correspond to actual inconsistencies (i.e., our tool did not generate any false positives), and 2) confirmed that there are no inconsistencies other than those found by our automated analysis (i.e., our tool did not generate any false negatives). For JGOSSIP, in particular, VIEWPOINTS found two instances of the inconsistency that we presented in our motivating example. As we discussed before, such inconsistencies represent actual vulnerabilities in the code that a malicious user may be able to exploit. For the remaining applications, four out of six contain input validation inconsistencies on the client side. A special case is that of MYALUMNI, which has 141 inputs that are inconsistently validated at the client side and server side. For this application, the developers provided no validation whatsoever on the client side, and thus all the 141 inputs that are checked on the server side are inconsistently validated.

Although these results are preliminary, and further experimentation would be needed to confirm them, they provide strong supporting evidence for answering RQ1: ViewPoint is in-

deed able, at least for the cases considered, to identify inconsistencies in client- and server-side input validation functions.

RQ2 relates to the efficiency and practicality of the analysis. From Table 5.3, we can see that the extraction phase of VIEWPOINTS took between 0.78 and 4.38 seconds for the server-side validation functions. Although for the client-side functions the numbers are higher, due to the more expensive analysis performed on the client side (see 5.1.4), the maximum total time needed to analyze one of the web applications considered is less than six minutes.

Table 5.2 illustrates the space cost of VIEWPOINTS. As the table shows, the space needed to store the DFAs is negligible, as it is less than seven megabytes in all cases. Finally, Table 5.7 shows the time needed to perform the comparison of two DFAs. Also in this case, the time it takes VIEWPOINTS for the comparison is in the order of a few seconds and, thus, negligible. We can therefore provide a positive answer to RQ2 as well.

Overall, these results provide preliminary, yet clear evidence that our differential bug finding approach can be both practical and useful.

5.2 Semantic Differential Repair For Input Validation and Sanitization

After finding the differences between two input validation and sanitization functions, the next step is to repair the functions against each other to remove the difference. In this section we present a novel semantic differential repair algorithm [2] that exploits redundancies in in-

```
function reference_function($x){
  if (strlen($x) > 4)
    exit();
  else {
    $x = preg_replace('/</', '', $x);
    if ($x == '')
      exit();
    else
      return $x;
  }
}


function target_function($y){
  $y = preg_replace('/"/', '\"', $y);
  return $y;
}
```

Figure 5.4: A small, but illustrative example, showing a target function to be repaired based on a reference function.

put validation and sanitization within and application and across applications to automatically repair input validation and sanitization functions by comparing them against each other.

In this section we give an overview of our automated differential repair technique that strengthens the validation and sanitization functionality of a given *target* function based on a given *reference* function. Consider the example functions shown in Figure 5.4[3]. The reference function starts with a validation check that blocks any string that is longer than 4 characters. This is followed by a sanitization operation which replaces the character < with ϵ (i.e., deletes <). Finally, the result of the sanitization operation goes through another validation check that blocks the empty string. The target function in Figure 5.4 does not do any validation. It only sanitizes the input string by replacing the character " with the string "\"" (i.e., it escapes the double quote characters).

[3]We kept the original functions written in PHP language to help the reader when comparing the original functions with the generated patches

```
function validation_patch($x){
  if (preg_match('/<*|[^<].{4,}|<[^<].{3,}|<<[^<].{2,}|
                  <<<[^<].+/',$x))
    exit();
  else
    return $x;
}

function length_patch($x){
  if (preg_match(
    '/"".{1,2}|".{1,2}"|.{1,2}""|"[^"]{3,3}|[^"]{3,3}"/',$x))
    exit();
  else
    return $x;
}

function sanitization_patch($x){
  $x = preg_replace('/</', "", $x);
  return $x;
}

function repaired_function($x){
  return target_function(
           sanitization_patch(
             length_patch(
               validation_patch($x)
             )
           )
         );
}
```

Figure 5.5: The repaired function that is generated by our differential repair algorithm for the target function shown in Figure 5.4.

The goal of our differential repair technique is to strengthen the validation and sanitization operations in the target function as much as the reference function. More precisely, the goal is to make sure that the repaired target function does not return a string that is not returned by the reference function or the original target function. Before explaining how our differential repair technique works on these two functions, we would like to discuss two potential repair techniques that may seem to be the natural choice in this case and explain why they do not work. This would help the reader to understand the motivation behind the choices we made when we invented our differential repair algorithm.

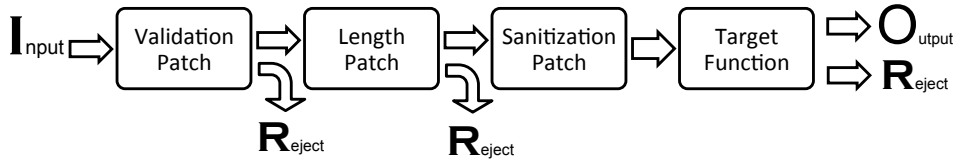

Figure 5.6: The repaired function is the composition of three automatically generated patches and the original target function.

Repair by Composition. One question we may ask is: why not simply run two sanitizers one after another? Due to lack of idempotency in some string sanitization operations, one can not blindly compose two given sanitizers to get a stronger one without first computing the difference between them. For example, composing a reference and a target sanitizers that have some differences but at the same time share the following sanitization operation `preg_replace('/"/', '\"', $x)`—which escapes the `"` with a `\`—is problematic. Since this operation is not idempotent, the composition would result in double escaping i.e., "`ab"c`" would become "`ab\\"c`" instead of "`ab\"c`". Furthermore, we repair sanitizers that are extracted from different programming languages (and different applications in some cases). The original two pieces of code where we extracted the two sanitizers from are written in different languages with different semantics and contain other logic related to the functionality of the application where they were extracted from. This shows the importance of the 1) extraction phase in removing code unrelated to validation and sanitization and 2) using our language agnostic string analysis framework in which we handle the semantical differences by reducing them to differences between regular languages.

Repair using Multi-Track DFAs (i.e., Transducers). The second option is to use a transducer to model the input-output relation for each of the two santizers then find the difference between the two transducers. This would help to generate a patch that captures the difference between the two sanitizers in the way each input is mapped to its corresponding output . At the same time, this patching technique avoid the idempotency problem and gives a unified framework to deal with semantical differences between languages. Unfortunately, the class of deterministic finite transducers (i.e., regular relations) is not closed under difference operation [37]. Further more, computing the post-image of a transducer over `replace` operation with enough precision is not possible [110]. Finally, since client-side code is always composed with (i.e., run before) server-side code, we can only repair the validation difference between the two (see 5.2.3 for more details) and in this case we do not need a transducer.

Since these two options do not work, we describe a new diffirential repair technique that avoids the aforementioned issues. The goal of the technique is to make sure that the post-image of the repaired function does not contain any string that is not in the post-image of the reference function and the original target function.

The post-image for the reference function in Figure 5.4 is the language of all strings that are shorter than 5 characters and not empty and do not contain the character <, while the post-image for the target function is the language of all strings that do not contain the character `"` unless it is preceded by the character `\`. For example, the string "`foo`" is an element in the reference function's post-image while the string "`foo<`" is not since it contains the < character. Also, the strings "`foo`" and "`foo\"bar`" are elements in the target function's post-image while

the string "`foo"bar`" is not since it contains the character " without being preceded by the character \.

Our differential repair algorithm works in three phases, where each phase generates a patch-function with a specific purpose: (1) a *validation patch*, (2) a *length patch,* and (3) a *sanitization patch.* The final repair is obtained by applying the composition of all three patch-functions together as shown in Figure 5.6.

Validation patch The purpose of this phase is to generate a patch that makes sure that the repaired function rejects all the inputs that are rejected by the reference function. Figure 5.5 shows the validation-patch produced in this phase of the repair algorithm for our running example. The validation patch blocks all input strings that are either empty, consist of one or more < characters or longer than 4 characters. For example, the strings "", "<", "<<<" and "`<html>`" will be blocked by the validation patch. On the other hand, the strings "`fo`" and "`<a>`" will not be blocked.

The validation patch blocks the inputs that generate a string that is in the post-image of the target function but not in the post-image of the reference function. Note that our algorithm is able detect that some input strings are blocked by the reference function only after being sanitized such as the string "<<<" (which is first converted to empty string by deletion of < and then blocked by the reference function). So, for this case, to make sure that the string "<< <" is not in the post-image of the repaired function, the validation patch blocks it.

Length patch The purpose of this phase is to make sure that (1) the maximum length of the strings that are in the the post-image of the repaired function is not bigger than the maximum length of the strings that are in the post-image of the reference function and (2) the minimum length of the strings that are in the post-image of the repaired function is not smaller than the minimum length of the strings that are in the post-image of the reference function.

For the reference function in our example, the minimum length is 1, since it blocks the empty string, and the maximum length is 4. On the other hand, for the target function, after the validation patch is applied, the minimum length is 1 since it also blocks the empty string, but the maximum length is not 4 but 8. The reason is that the sanitization in the target function escapes the `"` character so that an input string of length 4 like "`""""`" (which passes the validation patch) is escaped to produce the string "`\"\"\"\"`" at the sink, which is of length 8.

This example shows that due to the sanitization operation in the target function, we get a length difference in the post-image languages even though the validation patch has already blocked all strings longer than 4. To address this issue we generate a length patch that blocks any input string that results in a string longer than 4 characters at the target sink even if the input string itself is shorter than 4 characters. For example, the length patch blocks the string "`"a"`" although it has 3 characters only since it will result in the string "`\"a\"`" of length 5 at the sink which is longer than 4 characters. On the other hand, the string "`foo`" will not be blocked by the length patch since it will reach the sink as it is, 3 characters long.

Figure 5.5 shows the length patch-function for our example. Note that the function assumes that the validation patch function is applied before it so it only blocks things not blocked by the validation patch function. In section 5.2.3 we explain how to automatically generate the length patch-function.

Sanitization patch The purpose of this final phase is to take care of the differences that are due to sanitization operations. Our goal is to make sure that the post-image of the repaired function is a subset of the post-image of the reference function.

In our example, there is one sanitization operation in the reference function in which the character < is deleted. Even after application of the validation and length patches, this behavior would not be fully replicated at the repaired target function. Although the validation patch will prevent some strings such as "<<<" from reaching the sink at the repaired function, there are still other strings, such as "a<b" for example, that will still be in the post-image of the repaired function but not in the post-image of the reference function, since the character < gets deleted. The goal of the sanitization patch is to remedy such situations, and make sure that the sanitization operations in the target function are as strong as the sanitization operations in the reference function.

Unlike the previous two phases, the sanitization patch does not block the input strings that are found in the difference between the post-images of the target and reference functions. Instead we use the *min-cut algorithm* (4.1.3) to generate a sanitization code that will delete (or escape) certain characters in the input strings such that the difference between the two

post-images is removed. Using this min-cut algorithm, our differential repair algorithm will generate the sanitization patch-function shown in Figure 5.5. This function does not block input strings that contain the character <, but rather, deletes this character from these input strings and returns the corresponding string without that character. This repair simulates the same sanitization behavior of the reference function in the new repaired function. In section 5.2.3 we explain the min-cut algorithm and how to automatically generate the sanitization patch.

Given the final sanitization phase, one might think that the first two phases are redundant. However, without the first two phases, the repair generated by our approach can become too conservative by rejecting all input strings or by deleting all characters from the input string. Dividing the repair generation to three separate phases enables us to generate a combined repair that is not overly conservative.

The final result of the differential repair algorithm for our running example is shown in Figure 5.5. The *repaired function*, is obtained by composing the three patch-functions, in the order in which they were introduced here, with the original target function.

We extract one sanitizer function per input field which characterizes all the validation and sanitization operations that are used for that particular field. Validation and sanitization operations involve use of regular expressions and validation operations such as string *match*, *substring*, and sanitization operations such as string *replace, trim, addslashes, htmlspecialchars,* etc. In the next section we discuss how to extract sanitizer functions from a web application and when to map two functions to each other.

5.2.1 Extracting and Mapping Sanitizers From the Client- and/or the Server-side

Before we run our differential repair algorithm, we need to extract sanitizer functions from the client and/or server side and map them to each other to generate target, reference sanitizer pairs. We built a crawler using HTMLUnit [34] to find input fields and corresponding sinks in a PHP web application. When the crawler hits a web page with an HTML form, it fills it out automatically using a set of pre specified profiles and submits it. Then, for each HTML input field, JavaScript validation and sanitization code is dynamically extracted as described in 2.5, resulting with one sanitizer function per each input field. On the server-side, we also collect the execution traces to figure out the inputs and the sinks (where the inputs flow into) during crawling. We use that information later on to map server-side sanitizer functions to client-side sanitizer functions. Then, we use the front end of Pixy [55] to statically extract the corresponding sanitizer functions from the server-side as described in 2.6.

Client-client sanitizers are mapped to each other based on the type of data they operate on (i.e., email address, phone number, ... etc). Server-server mapping is done within the same application and across different applications based on field names that are extracted from the PHP `$_POST` and `$_GET` arrays.

5.2.2 Differential Repair Problem

Given a target sanitizer function F_T and a reference sanitizer function F_R, the goal of differential repair is to generate a new sanitizer function F^P, called a patch, such that when F_T is patched by composing it with F^P, the resulting repaired function returns a string only if F_R and F_T can both return that string. I.e., we want to make sure that a string is not in the post-image of the repaired function if it is not in the post-image of F_T or F_R.

In order to formalize this, let us define the difference between the post-images of two sanitizer functions F_1 and F_2 as follows:

$$\textsc{Diff}(F_1, F_2) = \{x \mid \exists y \in \Sigma^* : F_1(y) = x \wedge (\forall z \in \Sigma^* : F_2(z) \neq x)\}$$

which is the set of strings that are in the post-image of F_1 but not in the post-image of F_2. Given this definition (along with the definition of sanitizer's composition from section 2.3), the differential repair problem is to automatically construct a patch F^P such that $\textsc{Diff}(F_T \circ F^P, F_R) = \emptyset$, which means when we compose F_T with F^P we want to make sure that the result, $F_T \circ F^P$, is at least as strict as F_R, i.e., its post-image is a subset of the post-image of F_R. We call this new composed function the *differential repair* F_{DR}, where $F_{DR} = F_T \circ F^P$.

Note that, due to the way we are constructing the differential repair, by composing the target function F_T with the automatically generated patch F^P, we guarantee that the repaired function F_{DR} is at least as strict as F_T, i.e., its post-image is also a subset of the post-image of F_T.

5.2.3 Differential Repair Algorithm

Algorithm 15 DIFFERENTIALREPAIR(F_T, F_R)

```
1:  M_1 := A(PRE^+_⊥(F_R));
2:  M_2 := A(PRE^+_⊥(F_T));
3:  if (L(M_1 \ M_2) ≠ ∅) then
4:      M^V := M_1 \ M_2;
5:      F^V := GENERATEBLOCKINGSIMULATOR(M^V);
6:  else
7:      F^V := IDENTITYFUNCTION; M^V := A(∅);
8:  end if
9:  M_1 := A(POST^+(F_R, Σ*));
10: M_2 := A(POST^+(F_T, L(M^V)));
11: M_d = M_2 \ M_1;
12: if (L(M_d) ≠ ∅) then
13:     if (len_min(M_2) < len_min(M_1) ∨ len_max(M_2) > len_max(M_1)) then
14:         M_3 := RESTRICTLENGTH(M_2, M_1);
15:         M^L := A(PRE^+(F_T, L(M_2 \ M_3)));
16:         F^L := GENERATEBLOCKINGSIMULATOR(M^L);
17:         M_2 := M_3;
18:     else
19:         F^L := IDENTITYFUNCTION;
20:     end if
21:     M_d := M_2 \ M_1;
22:     if (L(M_d) ≠ ∅) then
23:         M_mc := A(PRE^+(F_T, L(M_d)));
24:         Σ_mc := MINCUT(M_mc);
25:         F^S := GENERATESANITIZER(Σ_mc, M_1);
26:     else
27:         F^S := IDENTITYFUNCTION;
28:     end if
29: else
30:     F^S := F^L := IDENTITYFUNCTION;
31: end if
32: F_DR := F_T ∘ F^S ∘ F^L ∘ F^V;
33: return F_DR;
```

Given a target sanitizer F_T and a reference sanitizer F_R, our differential repair algorithm consists of three phases that produce three patches: (1) The *validation patch generation* phase produces F^V, (2) the *length patch generation* phase produces F^L, and (3) the *sanitization patch generation* phase produces F^S. The result of our differential repair algorithm is a patch that is the composition of these three individual patches: $F^S \circ F^L \circ F^V$ and the repair we generate is the composition of this patch with the target function, i.e., $F_{DR} = F_T \circ F^S \circ F^L \circ F^V$.

Our differential repair algorithm is shown in Algorithm 15. The algorithm takes a target sanitizer F_T and a reference sanitizer F_R as input and generates sanitizer F_{DR} as output which corresponds to differential repair of F_T with respect to F_R. Our differential repair algorithms is based on automata based string analysis from chapter 3, and computes post or pre-images of given sanitizers as DFA as we described before. In Algorithm 15, each variable that has a name starting with M represents a DFA, each variable with a name starting with F represent a sanitizer. The algorithm uses the DFA operations $\cap, \cup, \setminus, —, \mathcal{A}, \mathcal{L}$ as described in 3.2.4. In the remaining part of this section we discuss the three phases of the Algorithm 15.

Phase I: Validation Patch Generation

Our goal is to generate a validation patch F^V such that:

$$\forall x \in \Sigma^* : F_R(x) = \bot \Rightarrow F_T \circ F^V(x) = \bot,$$

i.e., the validation patch F^V guarantees that $F_T \circ F^V$ does not accept inputs that F_R rejects. In order to compute the validation patch, we first need to identify the set of strings that are rejected by F_T and F_R i.e., their negative pre-images.

As we said before, it is not possible to compute the pre or post-image of a sanitizer precisely due to undecidability of string analysis problem. We use automata-based backward symbolic string analysis techniques discussed in section 3.2.7 to compute an over approximation of the negative pre-image, $\text{PRE}^+_\bot(F)$, where $\text{PRE}^+_\bot(F) \supseteq \text{PRE}_\bot(F)$. This means that, we may conclude that certain strings are rejected by F when they are not. On the other hand, since we

are computing an over-approximation, any string that is rejected by F is guaranteed to be in $\text{PRE}^+_\perp(F)$. Since we are using automata-based symbolic string analysis, the result of the negative pre-image computation is an automaton that accepts the language $\text{PRE}^+_\perp(F)$, and we denote this automaton as $\mathcal{A}(\text{PRE}^+_\perp(F))$.

In lines 1 and 2 of Algorithm 15 we construct two automata M_1 and M_2, that accept an over-approximation of the negative pre-images of F_T and F_R, respectively, where $\mathcal{L}(M_1) = \text{PRE}^+_\perp(F_R)$ and $\mathcal{L}(M_2) = \text{PRE}^+_\perp(F_T)$. The next step (line 3) checks if the reference function F_R rejects more input values than the target function F_T by computing the difference between negative pre-images of M_1 and M_2. If the difference is empty then F^V is assigned the identity function (line 7) which is a sanitizer function that returns the input as it is without blocking any value (i.e., it is a no-op). If the difference is not empty, the target function must be patched to reject the values rejected by the reference function. To achieve this we automatically generate a patch that rejects only the strings that are rejected by F_R but not F_T.

Note that the validation patch we generate is not sound due to over-approximation of the negative pre-image of the target function F_T. The set of strings that are in $\text{PRE}^+_\perp(F_R) \cap (\text{PRE}^+_\perp(F_T) \backslash \text{PRE}_\perp(F_T))$ will not be blocked by the patch we generate, whereas they should be blocked in order to reach our precise goal. We can make the validation patch sound by blocking all the strings in $\text{PRE}^+_\perp(F_R)$ without computing the set difference with $\text{PRE}^+_\perp(F_T)$, but, that would result in generation of a validation patch in many cases even when it is not necessary. Our experiments indicate that the imprecision in our pre-image computation is not a

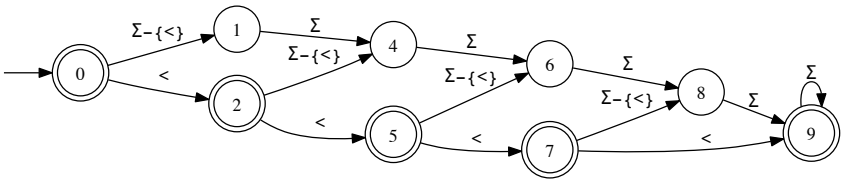

Figure 5.7: The validation patch automaton M^V for the example in Figure 5.4. The validation patch F^V blocks the strings accepted by this automaton.

problem in practice since for all the examples we manually checked we observe that $\text{PRE}^+_\perp(F_R) \cap (\text{PRE}^+_\perp(F_T) \backslash \text{PRE}_\perp(F_T)) = \emptyset$.

Figure 5.7 shows the validation patch automaton M^V that is automatically generated for the example shown in Figure 5.4 where Σ represents the ASCII characters. To save space we collapsed all transitions between any two states s_i and s_j into one transition t_{ij}. We annotate this transition with a set of characters $\Sigma_C \subseteq \Sigma$ such that if a character c is in Σ_C then there is a transition on c between s_i and s_j. The sink state along with transitions into and out of it are omitted.

Since our analysis represents the set of strings at each program point using DFA, we generate the patch repair function F^V based on the DFA that is computed by our analysis. The validation patch code that is generated with GENERATEBLOCKINGSIMULATOR filters the inputs by simulating the resulting automaton M^V in Figure 5.7 to determine if the input string is accepted by M^V. If the input string is accepted by the automaton M^V, then F^V will return $\perp$ to block the input, otherwise it will return the input string without modification.

Phase II: Length Patch Generation

The goal of length patch generation is to generate a patch F^L such that:

$$\forall x \in \Sigma^* :$$
$$((\exists y, z \in \Sigma^* : |F_R(y)| \leq |F_T \circ F^V(x)| \leq |F_R(z)|) \Rightarrow F^L(x) = x) \wedge$$
$$(\neg(\exists y, z \in \Sigma^* : |F_R(y)| \leq |F_T \circ F^V(x)| \leq |F_R(z)|) \Rightarrow F^L(x) = \perp)$$

i.e., given the target function F_T composed with the validation patch F^V, F^L rejects any input string that will cause the output of $F_T \circ F^V$ to contain a string of length longer or shorter than all the strings in the output of the reference function F_R.

The validation patch makes sure that any input string rejected by the reference sanitizer is also rejected by the repaired target sanitizer. However, this does not mean that the set of strings that are returned by the repaired target sanitizer and the reference sanitizer are the same after the validation patch since they may be using different sanitization operations. The length patch is the first step in establishing that the repaired target sanitizer does not return any string that is not returned by the reference sanitizer. The length patch makes sure that the length of any string returned by the repaired target function is not larger or smaller than all the strings returned by the reference sanitizer.

The lines 9-20 in Algorithm 15 construct the length patch. The lines 9 and 10 compute an over-approximation of the post-images i.e., the automata that accept the set of strings that are returned by the reference sanitizer and the target sanitizer that is composed with the validation

patch. The lines 11 and 12 in Algorithm 15 check if there are any strings that are returned by the target sanitizer composed with the validation patch that are not returned by the reference sanitizer by checking if $\textsc{Post}^+(F_T, \mathcal{L}(\overline{M^V}))\setminus \textsc{Post}^+(F_R, \Sigma^*) = \emptyset$. If the difference is empty, then we consider $F_T \circ F^V$ to be as strict as F_R and the analysis concludes by assigning IDENTITYFUNCTION (i.e., no-op) to length and sanitization patches F^L and F^S (line 30).

Note that, due to over-approximation in our analysis, it is not guaranteed that $F_T \circ F^V$ is as strict as F_R even if the difference is empty. However, again manual inspection of our experiments indicate that our approximate analysis always finds the differences if they exist since the precision of our post-image computation is quite good in practice.

If a difference is found, then we check if the difference corresponds to a length difference in line 13. Let us first define len_{max} and len_{min} for an automaton. Given an automaton M, $\mathit{len}_{max}(M) = \infty$ if M accepts an infinite set, and $\mathit{len}_{max}(M)$ is the length of the longest string accepted by M otherwise. We can check if $\mathit{len}_{max}(M) = \infty$ by checking if there are cycles in M on any path from the starting state to an accepting state. If there is at least one cycle, then $\mathit{len}_{max}(M) = \infty$. If there are no cycles, then $\mathit{len}_{max}(M)$ is finite, and we use a depth first search to compute the length of the longest string accepted by M. On the other hand, given an automaton M, $\mathit{len}_{min}(M)$ is the length of the shortest string accepted by M. If the start state is an accepting state then $\mathit{len}_{min}(M) = 0$. Otherwise, $\mathit{len}_{min}(M)$ is computed by finding the length of the shortest path from the start state to an accepting state.

If a length difference is found, then we restrict the length of the set of strings accepted by F_T to remove the length difference using the following operation in line 14:

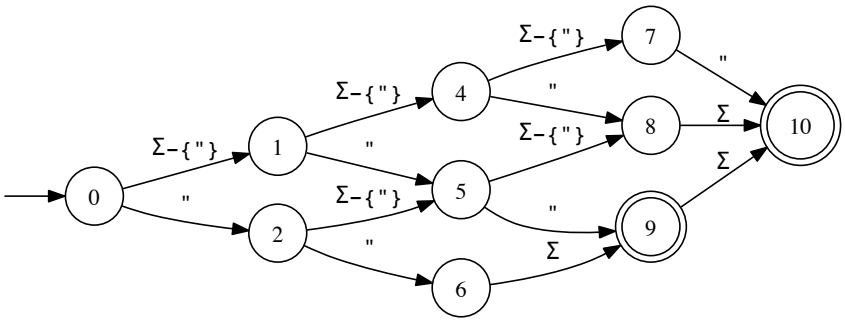

Figure 5.8: The length patch automaton M^L for the example in Figure 5.4. The length patch F^L rejects the strings accepted by this automaton.

$$\textsc{RestrictLen}(M_2, M_1) \equiv M_2 \cap \bigcup_{i=\mathit{len}_{min}(M_1)}^{\mathit{len}_{max}(M_1)} \Sigma^i$$

After the length restriction, in line 15, we use the pre-image computation (see sections 3.1.2 and 3.2.6) to compute an over-approximation of the set of input strings that cause the length discrepancy i.e., $\textsc{Pre}^+(F, L) \supseteq \textsc{Pre}(F, L)$. This over-approximation may result in blocking input strings that do not contribute to the length discrepancy.

In line 16 we generate the length patch F^L that blocks the strings that are accepted by the automaton M^L in Figure 5.8 and returns the strings rejected by M^L without any change. Figure 5.8 shows the length patch automaton M^L that our algorithm computes for the example shown in Figure 5.4.

Phase III: Sanitization Patch Generation

The third and final phase in the repair algorithm is the sanitization patch generation, which results in a patch function F^S such that:

$$\begin{aligned} \forall x \in \Sigma^* \quad : \quad & (\forall y \in \Sigma^* : F_R(y) \neq x) \Rightarrow \\ & (\forall z \in \Sigma^* : F_T \circ F^S \circ F^L \circ F^V(z) \neq x) \end{aligned}$$

which means that, after adding the sanitization patch F^S to the previously generated patches, we want the differential repair $F_{DR} = F_T \circ F^S \circ F^L \circ F^V$ to be as strict as F_R in terms of the set of strings it returns.

The lines 21-28 in Algorithm 15 generate the sanitization patch. First, in the lines 21, 22, we check if there is a difference between what F_T returns (after validation and sanitization patches are applied) and what F_R returns assuming any input. If no difference is found, then we consider $F_T \circ F^L \circ F^V$ to be as strict as F_R and the analysis concludes by assigning IDENTITYFUNCTION to F^S (line 27). This indicates that there is no sanitization patch. Note that, as we discussed before, due to over-approximation our repair algorithm can miss differences, however we have not observed this in our experiments.

If a difference is found, then, in the line 23, we compute an over-approximation of the set of input strings that result in such a difference. The set $\mathcal{L}(M_{mc})$ represents an over-approximation of the set of all input strings that are the cause of the difference between the set of strings returned by F_R and $F_T \circ F^L \circ F^V$. We call M_{mc} the *mincut* automaton and in the line 24 we use this mincut automaton to generate a mincut alphabet using the MinCut algorithm in

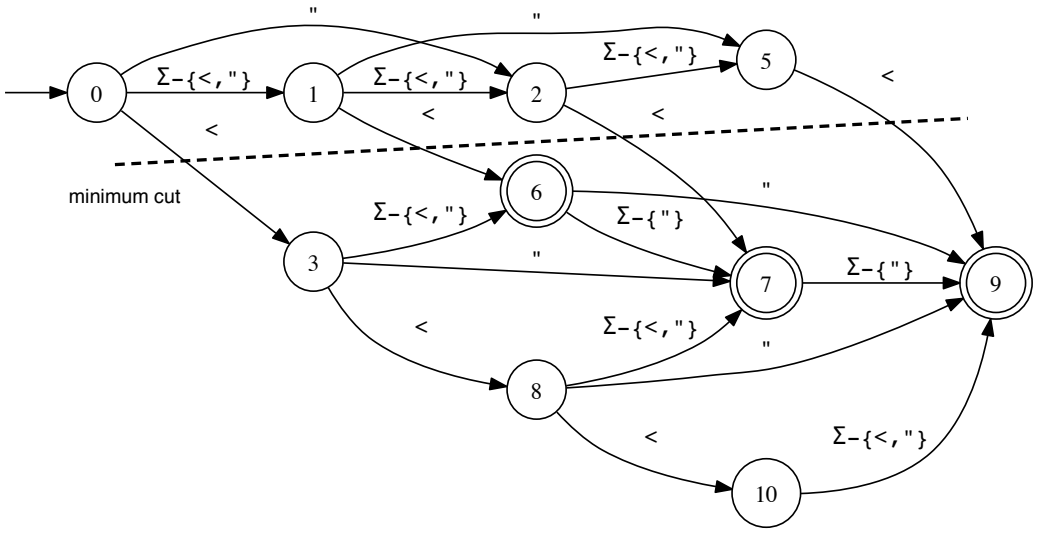

Figure 5.9: The mincut automaton M_{mc} for the example in Figure 5.4. The dotted line shows the mincut edges with the corresponding mincut alphabet $\{<\}$.

section 4.1.3, such that if the symbols in the mincut alphabet are removed from the input strings, then the difference between the post-images of F_R and $F_T \circ F^L \circ F^V$ disappear. Figure 5.9 shows the mincut automaton M_{mc} for our running example in Figure 5.4 along with the mincut edges which correspond to the mincut alphabet $\Sigma_{mc} = \{<\}$.

Then, in the line 25, we generate the sanitization patch F^S to either delete or escape the set of symbols in the mincut alphabet. Finally, in the lines 32 and 33, we construct and return the differential repair function F_{DR} as the composition of the target function F_T with the three patch functions generated during the three phases of the repair algorithm.

Code Generation Heuristics.

Once we compute an alphabet-cut Σ_{mc}, we generate the sanitization patch F^S with a `replace` statement that deletes the symbols in Σ_{mc} from the input, making sure that the resulting string does not match M_{mc}. Although the function F^S is a sound repair that will guarantee

that $POST^+(F_T \circ F^S \circ F^L \circ F^V, \Sigma^*) \subseteq POST^+(F_R, \Sigma^*)$, we apply two heuristics to generate more accurate repair functions.

The first heuristic is the *escape* heuristic. We look at the automaton M_1 generated in line 9 of the Algorithm 15 (which represents all the string values returned by F_R), and check if all the characters in the mincut alphabet Σ_{mc} are always preceded by the same single character e. If that is the case, we we call the character e the escape character. Formally speaking, given DFA $M_1 = \langle Q_1, q_0, \Sigma, \delta_1, F_R \rangle$, we check that $\forall q \in Q_1, \forall c \in \Sigma_{mc} : \delta_1(q, c) \neq sink \Rightarrow (\forall q' \in Q_1 : \delta_1(q', c') = q \Rightarrow c' = e)$. If this is the case, then the sanitization patch F^S we generate uses the `replace` operation to escape all the characters in the mincut alphabet Σ_{mc} (instead of deleting them) by prepending them with the escape character e.

The second heuristic we use is the *trim* heuristic. Here, if we get a mincut Σ_{mc} which contains space characters, we first check if M_1 accepts any string that contains a space character (which can be determined by checking if transitions on space characters always go to the sink). If not, then we generate a patch that deletes the space characters as in our basic mincut based patch generation algorithm. If M_1 does accept a string that contains a space character, then we check if it is the case that the strings accepted by M_1 do not start or end with space characters. Formally speaking, given DFA $M_1 = \langle Q_1, q_0, \Sigma, \delta_1, F_R \rangle$, we check that for all space character s $\delta_1(q_0, s) = sink$ and $\forall q \in Q_1 : \delta_1(q, c) \in F_R \Rightarrow c \neq s)$. If so, then we generate patch F^S which uses the `trim` function to delete the space characters from the beginning and end of each input string.

5.2.4 Empirical Evaluation

We have implemented our differential repair algorithm in a tool called SEMREP (see 6.3). In order to evaluate our repair algorithm we experimented on five open-source PHP applications 1) PHPNews v1.3.0 (news publishing software), 2) UseBB v1.0.16 (forum software), 3) Snipe Gallery v3.1.5 (image management system), 4) MyBloggie v2.1.6 (weblog system), 5) Schoolmate v1.5.4 (school administration software) along with a number of JavaScript sanitizer-function benchmarks from various websites [3]. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 12.04.

Results. Table 5.5 shows the total number of target-reference sanitizer pairs we analyzed and the number of patches generated at each phase of the algorithm for four categories: Client-server (C-S) where the server (target) is patched against the client (reference), server-client (S-C) where the client (target) is patched against the server (reference), server-server (S-S) and client-client (C-C). Note that for the server-server and client-client cases we analyze each pair twice by considering each sanitizer function as target once and as reference once. The most commonly generated patches are validation patches which indicates that inconsistencies in validation policies are common. There are also a significant number of sanitization patches generated, except that the client-server pairs generated no sanitization patches. We checked the client-server pairs manually and confirmed that this is an accurate result (i.e., there are no cases where the sanitization at the client-side is stronger than the sanitization at the server-side). Among 49 server-server validation patches, 48 of them are generated for the pairs that are from different applications. We found 14 server-server sanitization patches within the same

application, which indicates inconsistent sanitization policy within the application. For server-server and client-client, there are no length patches since the validation patches in these cases do not involve length restrictions.

Table 5.6 shows the details of sanitization patches and results of the mincut algorithm. As we can see our mincut heuristics were able to identify 41 trim operations and 10 escape operations. This identification is very helpful since applying sanitization patches that escape certain characters is not idempotent which is critically important to be avoided for server-client pairs. In the client-client sanitization patches there was an interesting case in which the mincut was Σ. The reason was that the languages of the post-image DFAs were disjoint which means that the two functions return completely different sets of strings (in this case the discrepancy was due to the presence/absence of the dash symbol in phone number fields).

	#pairs	**#valid.**	**#length**	**#sanit.**
C-S	122	61	11	0
S-C	122	53	2	30
S-S	206	49	0	33
C-C	19	34	0	5

Table 5.5: Number of patches generated.

	mincut avg size	**mincut max size**	**#trim**	**#escape**	**#delete**
S-C	4	10	15	10	20
S-S	3	5	23	0	20
C-C	7	15	3	0	2

Table 5.6: Sanitization patch results.

Table 5.7 shows the memory and time performance of our repair algorithm. Rows I, II, and III corresponds to validation, length, and sanitization patch generation phases, respectively. Memory performance is represented as number of BDD nodes that is needed to represent a

DFA where the size of each BDD node is 16 bytes. In general our algorithm is efficient both in terms of time and memory. The average total time for the algorithm is 1.35 seconds and the maximum time is 186.22 seconds. During our experiments we had to abort the algorithm in 36 among 469 pairs (%7.6) due to MONA's limit on BDD size. The main reason for exceeding the BDD size limit is length constraints with large numbers. For example, in one of the cases in the experiments, validation patch restricts the length of the language of the input strings to 255. Then, sanitization function htmlspecialchars in the target increases the maximum length of the post-image to 1275. Since we use finite automata to represent sets of strings, the automaton has to count the length of the strings with its states. For this reason, we can see from Table 5.7 that the second phase of the algorithm which deals with the length constraints has the highest time and memory consumption.

repair phase	DFA size (#bddnodes)		peak DFA size (#bddnodes)		time (seconds)	
	avg	max	avg	max	avg	max
I	997	32,650	484	33,041	0.14	4.37
II	129,606	347,619	245,367	4,911,410	9.39	168.00
III	2,602	11,951	4,822	588,127	0.17	14.00

Table 5.7: Time and memory performance of analysis.

Chapter 6

Tools

We present in this chapter a String Analysis Library called LIBSTRANGER [103] along with two new tools called STRANGER [103] and SEMREP.

6.1 LIBSTRANGER

LIBSTRANGER [103] is a string manipulation library that handles all core string and automata operations described in chapter 3 such as general language replacement, concatenation, intersection, union, widen and special replace operations. During the string forward and backward analysis carried out by STRANGER and SEMREP, all string and automata manipulation operations that are needed to compute the post and pre-images of string operations are available in LIBSTRANGER. LIBSTRANGER uses MONA [16] library developed by Klarlund et al. to provide the symbolic representation of automata using MBDDs. The core

of LIBSTRANGER is implemented in C language as a shared library `libstranger.so` to get a faster computation and a tight control on memory. A C++/Java class called *Stranger-Automaton* is available to encapsulate the low level algorithms and data structures and provide a much elegent interface to the library. We used JNA (Java Native Access) to bridge the C language and Java code. LIBSTRANGER along with its source code and manual is available `https://github.com/vlab-cs-ucsb/LibStranger`.

6.2 STRANGER

We developed a tool called STRANGER [103] (STRing AutomatoN GEneratoR) that implements our approach in 4.1 to check the correctness of string validation and sanitization functions in Web applications with respect to known attacks.

STRANGER is implemented in Java and uses PIXY—which is developed by Jovanovic et al. [54]—as a front end and our string manipulation library LIBSTRANGER (see 6.1) along with MONA [16] for automata manipulation. STRANGER takes a PHP program and a set of attack patterns as input and automatically analyzes it and outputs the possible XSS or SQL Injection vulnerabilities (characterized as attack patterns) in the program. For each input that leads to a vulnerability, it also outputs the vulnerability signature, i.e., an automaton (in a dot format) that characterizes all possible string values for this input which may exploit the vulnerability along with the patch generated using the mincut algorithm. STRANGER and several benchmarks are available at `http://www.cs.ucsb.edu/~vlab/stranger`.

6.2.1 Architecture

The architecture of STRANGER is shown in Figure 6.1. The tool consists mainly of the two modules: (1) the vulnerability analysis module that uses PIXY to parse PHP code and perform the taint analysis and then performs the vulnerability analysis and repair and (2) the string analysis module that implements the post- and pre-image computation and uses LIBSTRANGER and MONA for automata manipulation operations.

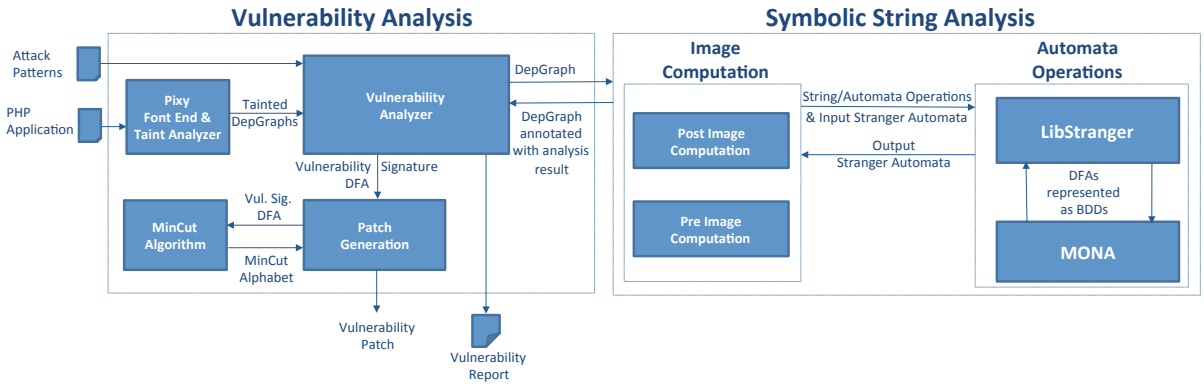

Figure 6.1: The architecture of STRANGER.

The first step in the vulnerability analysis is done by PIXY [54], a taint analysis tool for detecting web application vulnerabilities. PIXY parses the PHP program and constructs the control flow graph (CFG). PHP programs do not have a single entry point as in some other languages such as C and Java, so we process each script by itself along with all files included by that script. The CFG is passed to the taint analyzer in which alias and dependency analyses are performed to generate dependency graphs (see 2.6). If no tainted data flow to the sink,

taint analysis reports the dependency graph to be secure; otherwise, the dependency graph is considered to be tainted and passed to the vulnerability analyzer for more inspection.

The vulnerability analyzer implements our policy-based repair algorithm (Algorithm 12). It uses the symbolic string analyzer which implements our post- and pre-image computation from chapter 3 (that is modified slightly to work with dependency graphs). The dependency graphs are pre-processed to optimize the image computation. First, a new acyclic dependency graph is built where all the nodes in a cycle (identifying cyclic dependency relations) are replaced by a single strongly connected component (SCC) node. The vulnerability analysis is conducted on the acyclic graph so that the nodes that are not in a cycle are processed only once. In the forward analysis, we propagate the post images to nodes in topological order, initializing input nodes to DFAs accepting arbitrary strings. Upon termination, we intersect the language of the DFA of the sink node with the attack pattern. If the intersection is empty, we conclude that the sink is not vulnerable with respect to the attack pattern. Otherwise, we perform the backward analysis and propagate the pre images to nodes in the reverse topological order, initializing the sink node to a DFA that accepts the intersection of the result of the forward analysis and the attack pattern. Upon termination, the vulnerability signatures are the results of the backward analysis for each input node. For both analyses, when we hit an SCC node, we switch to a work queue fixpoint computation on nodes that are part of the SCC represented by the SCC node. During the fixpoint computation we apply automata widening on reachable states to accelerate the convergence of the fixpoint computation. We added the ability to choose when to apply the widening operator. This option enables computation of the precise fixpoint in cases where

the fixpoint computations converges after a certain number of iterations without widening. We also incorporate a coarse widening operator that guarantees the convergence to avoid potential infinite iterations of the fixpoint computation.

6.3 SEMREP

SEMREP is a new tool to analyze and repair validation and sanitization functions in web applications. SEMREP implements the new language-agnostic automated semantic differential repair algorithm from 5.2 to analyze and repair validation and sanitization functions in web applications. Most of SEMREP is implemented in C++. MinCut algorithm and patch generator is implemented in Java. SEMREP uses our LIBSTRANGER library (see 6.1) along with MONA library for automata manipulation operations. Source code for latest version along with the manual are available online from `https://github.com/vlab-cs-ucsb/SemRep`.

6.3.1 Architecture

SemRep consists of two modules: (1) the differential repair module which implements the differential repair algorithm in 5.2 and (2) the symbolic string analysis module which computes the pre and post-images of a sanitizer. Figure 6.2 shows the architecture of the tool.

The tool takes as input the dependency graphs (see 2.6) of two sanitizer functions. After parsing the dependency graphs, difference computation component will send each graph to negative pre-image computation component. In general, image computation components use

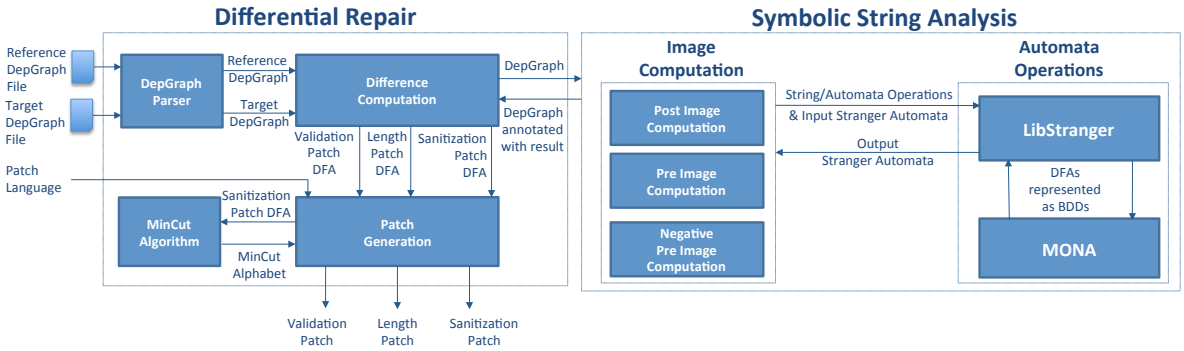

Figure 6.2: The architecture of SEMREP.

forward and backward string analyses to compute post and pre-images of the function represented by the dependency graph similar to the way done in STRANGER. Each node in the dependency graph will be annotated with a DFA (stored in an object of type StrangerAutomaton) that accepts all possible string values that may reach this node going forward or backward. (Negative) Pre-image component annotates the `input` node with the (negative) pre-image DFA while post-image component annotates the `return` node with the post-image DFA.

After negative pre-image computation component finishes, the difference computation component will then use the two DFAs associated with the `input` nodes to calculate the validation patch. Next, it will annotate the `input` node in the target dependency graph with the validation patch DFA (if a validation patch is needed) and send the two dependency graphs to the post-image computation component. Then, it will check to find out if the there is a length difference between the validation-patched target and the reference by checking the length difference between post-image DFAs that are associated with the `return` nodes. If a difference is found, it will (1) restrict the length of the DFA at the target `return` node by the length of the DFA

at the reference `return` node and (2) annotate the `return` node at the target with the new length-restricted DFA indicating that the language of this DFA represents the preferred output. It then sends the target dependency graph to pre-image computation component to compute the pre-image for this preferred output which represents the length patch DFA.

Then, if after the length restriction there is still a difference between the DFAs at the sinks (`return` nodes), the difference computation component will annotate the `return` node of the target dependency graph with the DFA that represents this difference (which we call sanitization difference DFA), indicating a non-preferred output, and sends that dependency graph to the pre-image computation component. The pre-image computation component will annotate the `input` node with the sanitization patch DFA. Then, the difference computation component will send the sanitization patch DFA along with the validation and length patch DFAs to the patch generation component.

The patch generation component will do the following: (1) Generate the code for the validation and length patches in the preferred programming language provided in the patch language input. These patches are functions that simulate the validation and length-patch DFAs. (2) Send the sanitization patch DFA to the mincut algorithm and uses the returned mincut alphabet to generate the code for the sanitization patch.

Chapter 7

Related Work

7.1 Bug Detection in Web Applications

There has been lots of work on bug detection in web applications. We first present work done on the client-side then move on to the server-side.

Client-Side In the past ten years, there has been lots of work on static type checking of Javascript programs [6, 7, 41, 52, 53, 58]. Historically, type checking has been one of the most widely used static verification techniques. However, it was not extensively applied towards detection of the class of bugs and vulnerabilities that are caused by string manipulation. Static control [40], information [21] and taint [39] flow analyses have been used for Javascript programs to detect security vulnerabilities. GATEKEEPER [38] uses static analysis to verify the enforcement of security policies written in Datalog on JavaScript widgets. These policies are different than ours and they are not related to input validation.

In general, although, pure static analyses for Javascript can be used not just to detect bugs but also to proof certain program correctness properties, they still suffer from enough loss of precision that renders them to be practically useless. The reason for that is that Javascript is very dynamic language and the dynamic features of the language are used heavily [78]. This was the main reason for us to use a hybrid analysis for bug detection in Javascript where the dynamic part can help with precision while the static part helps with vastly expanding the state space that we are searching. Nevertheless, we would like to investigate if we can use our string analysis as a preliminary analysis to improve precision and lower the rate of reporting spurious bugs in these static analyses (especially [39][1]) while still scaling to large applications. The main reason that led us to believe that our string analysis has a potential to improve the result is that objects and arrays in Javascript are maps from strings to strings. However, there is still one important feature that we need to add to our string analysis which index sensitivity. We believe that this is achievable if we only consider the cases where all possible values for an index are a finite set of integers.

In addition to static analysis, dynamic analysis techniques have been used in [28, 60, 80, 81] to extract and/or analyze Javascript code. FLAX [81] uses dynamic analysis techniques to discover client side validation vulnerabilities. The authors use dynamic taint analysis to extract validation code related to a certain sink and then use random fuzzing to test this sink. In our techniques we use a similar approach to extract the validation function but then we statically analyze the extracted code to see if it violates the given policies which allows us to

[1] Unfortunately, this tool is not publicly available at this time

expand the state space under investigation. In [80] authors developed a symbolic execution framework for JavaScript. At the core of their framework there is a string constraint solver called KUDZU that is built on top of the bounded string solver HAMPI [59]. In general and compared to (dynamic) symbolic execution, we use—in the extraction phase (chapter 2)—a set of profiles for generating input values that we use when extracting client-side code. We can mix the two approaches by 1) starting our dynamic extraction with input values taken from these profiles and then 2) compute pre-images of extracted functions using a single backward pass that is slightly different from the negative backward analysis and finally 3) generate a satisfying example for the complement of the computed DFA from 2. Compared to the static analysis phase (chapter 3), we do infinite state symbolic model checking where we do not bound (i.e., unroll) loops as is the case with symbolic execution. We discuss in the next section the differences between our techniques for handling string manipulation operations compared to the techniques in [59, 80] .

Server-Side There has been many static vulnerability detection techniques that have been developed for PHP and Java web applications. Many of these techniques such as [19, 36, 70, 87, 96,97] are based exclusively on static string analysis and we will discuss them more in the next section. Pixy [55] uses different static analysis techniques to build dependency graphs that represent the data flow from sources to sinks in a PHP web application. Then it uses taint analysis to detect if there are vulnerabilities in web applications. We built our tools STRANGER [88] and SEMREP [2] on top of Pixy and used our string analysis to improve precision as shown in [2–4, 102, 104].

Xie and Aiken [101] addressed the problem of statically detecting SQL injection vulnerabilities in PHP scripts by means of a three-tier architecture. In this architecture, information is computed bottom-up for the intra-block, intra-procedural, and inter-procedural scope. As a result, their analysis is flow-sensitive and inter-procedural. They use traditional data flow analysis to determine whether unchecked user inputs can reach security-sensitive functions (so-called sinks) without being properly checked. However, they do not calculate any information about the possible strings that a variable might hold. Thus, they can neither detect all types of vulnerabilities (such as subtle SQL injection bugs) nor determine whether sanitization routines work properly. RIPS [26] used the same technique with extensive modeling for PHP builtin functions and libraries and extended their application to other types of vulnerabilities such XSS and MFE. Phantm [62] runs the PHP web application to solve include statements and configuration values then uses the dynamically collected data to improve precision of static analysis. In our case we consider all include files and possible configuration values for a specific version of PHP. Apollo [8] uses dynamic symbolic execution to generate test cases for the web application. It applies some techniques to minimize the conditions on the inputs that cause a failure to provide better error reporting. Wassermann et al. [98] uses dynamic symbolic execution along with grammar based string analysis to generate test inputs for PHP web applications. Saner [9] mixes string-based static and dynamic techniques to discover vulnerabilities. Compared to symbolic execution techniques, our analysis for PHP programs is completely sound with respect to the policies since we do not bound the loops or the length of the strings. We are

able, as we have shown in chapter 4, not just to find bugs and generate counter examples but also to automatically fix these bugs in a sound way.

7.2 String Analysis

String analysis refers to static and dynamic techniques used to reason about string expressions in a program. This kind of analysis is crucial for verification tools that need to verify programs with string expressions. There are three types of string analysis techniques. *Static string analysis* in which the input programs are statically analyzed to compute an over approximation of string expressions' values throughout the program. *Hybrid string analysis* in which dynamic analysis and monitoring is used along with static string analysis. *String constraint solving* in which a decision procedure is developed to solve constraints that involve string variables and the results are used later as test input cases or counter examples.

7.2.1 Static String Analysis

Static string analysis is the process of computing an over approximation of string expressions' values throughout a program statically without running it.

Context Free Grammar Based Analysis. JSA [19] was the first static string analyzer and it was designed for Java programs. Given a Java program, JSA first constructs a directed flow graph for every specified hotspot, that captures the flow of strings and string operations while abstracting everything else away. Nodes represent string constants, variables, expressions,

and operations while edges represent possible data flow. Hotspots represent program points where the user of the tool is interested in string values that may reach them. This graph is then transformed into a context free grammar (CFG) such that each nonterminal represents a node in the graph and each terminal represents a constant string. The grammar has the property that for a node n in the flow graph, the associated nonterminal A_n has $L(A_n)$ (the set of strings that can be derived from nonterminal A_n) that contains all possible string values the string variable or expression represented by this node may have. This grammar is then over-approximated (using Mohri-Nederhof algorithm) into a finite state automaton (FSA) A such that $L(A)$ (language accepted by the automaton A) represents an over approximation of string values that may reach the hotspot. Kirkegaard et al. applied JSA to statically analyze the XML transformations in Java programs [61] by using DTD schemas as types and modeling the effect of XML transformation operations. Gould et al. [36] use this grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications. Christodorescu et al. [20] present an implementation of the grammar-based string analysis technique for executable programs for the x86 architecture.

Minamide [70] extended previous CFG-based approach by providing support for string-based replacement operations in PHP applications. He approximates the whole HTML output of a PHP program at once instead of one hotspot at a time. Instead of converting the resulting CFG into an FSA and then giving the results represented as an FSA, he stops at the CFG creation phase and directly uses the resulting CFG in two ways. First, he checks XSS attacks by intersecting the resulting CFG language with a regular expression language that represents

the dangerous output that contains an XSS attack. Second, he checks for HTML output well-formedness by checking if the CFG language is a subset of HTML CFG language. Since this problem is undecidable, he bounds the depth of tag-nesting in HTML CFG language which converts it into a regular language. If the subset check succeeds then program's output is valid otherwise the bound is increased and a test is done again. To model PHP string operations such as *str_replace*, finite state transducers (FST) are used where an FST transforms a CFG language based on a string operation by computing its post image under that operation.

Wassermann and Su [96] used the string analyzer developed by Minamide to check for SQL injection vulnerabilities in PHP applications. A CFG language that approximates strings that may reach an SQL hotspot is computed. Nonterminals in this CFG are annotated with taint values from taint analysis. Then for each tainted nonterminal, two checks are applied. First, they check if it is in a literal string syntactic position in an SQL query. If so, then they check if it intersects with a regular language that represents strings with odd number of unescaped quotes. If so then it is considered to be vulnerable. In [97] Wassermann and Su used similar approach to check for XSS vulnerabilities.

Compared to CFG-based string analysis, we can state the following advantages for our automata-based string analysis:

1. The widening operation used in the CFG-based analyses to deal with loops gives a very coarse approximation in the presence of string replacement operations. According to P. Cousot [24] "The definition of the widening operation must be a balance between

compelling the convergence of the global analysis of the program and discovering as much information as possible about programs."

2. In the case of differential analysis, if CFGs are used to approximate the post-images of sanitizers, then we can not detect the difference since CFGs are not closed under difference [50].

3. To our knowledge, CFGs has been used to compute pre-images of sanitizers. Pre-image computation is essential to the repair algorithms that we presented.

The main disadvantage of automata-based string analysis is that, CFG-based string analysis can be used to detect certain classes of vulnerabilities that affect structured output such as SQL queries and HTML documents. It does so by detecting the possibility of changing the syntax tree of the auto-generated structured output using user input. This is done without the need for policies and assuming no replace (sanitization) operations inside loops.

Automata Based Analysis. Hooimeijer et al. propose a symbolic automata representation [46] and use finite state transducers [92, 95] to analyze behaviors of sanitization operations. Their tool Bek is able to identify whether a target string is a valid output of a sanitization routine. Later on, Loris D'Antoni et al. [27, 30, 31, 91] extended this line of research to string encoding and decoding operations. All these tools limit their analysis to single-input sanitizers. Unlike this transducer-based approach, we use an MBDD-based symbolic automata representation where we use an efficient automata construction for the string replace operations on top of MBDDs as shown in [2, 105, 106] and chapter 3. The construction prevents the potential

explosion in the size of the automata due to multiple conjunctions of transducers. Nevertheless, Bek can be combined with our analysis to increase the confidence in the result. We assume that builtin string sanitization operation written as part of PHP, Java and Javascript builtin libraries do what they claim to do in their specifications. Tools like Bek can be used to guarantee this by providing a language for writing and verifying such string sanitization operations.

Rex [93, 94] combine SMT solvers (using Z3 [68]) with symbolic automata and show its effectiveness to encode and manipulate strings having large alphabets such as Unicode. The presented approach here is also capable of encoding large alphabets by increasing BDD variables in MBDDs. In fact, Yu et al. [107] show that by adjusting BDD variables for various encodings one can adjust the precision and performance of string analysis.

A multitrack automata based string backward analysis has been proposed in [104] to compute a relational vulnerability signature for multi-input sanitizers. This backward analysis has been used to automatically patch vulnerabilities in PHP web applications. The multitrack automata has been used to model relations between string inputs and output [109] [110].

Language-based replacement has been discussed in computational linguistics [35, 57, 71, 90]. These algorithms are based on the composition of finite state transducers. By composing specific transducers, constraints like longest match and first match can be precisely modeled. However, each composition may result in a quadratic size of non-deterministic automaton, and is more likely to blow-up compared to algorithms that we used here. The transducer-based replacement function [71] has been implemented in Finite State Automata utilities (FSA) [89], where automata are stored and manipulated using an explicit representation. We use a symbolic

DFA representation based on MBDDs. This symbolic encoding enables us to perform complex automata operations, such as closure, concatenation, replace, and widening, efficiently using the MBDDs.

Choi et al. [18] investigates a widening method to analyze strings. The widening operator is defined on strings and the widening of a set of strings is achieved by applying the widening operator pairwise to each string pair. The widening operator we use is defined on automata, and was originally proposed for arithmetic constraints [11]. The intuition behind this widening operator is applicable to any symbolic fixpoint computation that uses automata. In [11] it is proved that for a restricted class of systems the widening operator computes the precise fixpoint and we extend this result to our analysis. Moreover, in our experiments, the over-approximation computed by this widening operator works well for proving the properties we were interested in.

Index Sensitive Analysis. Tateishi et al. [87] propose a path- and index- sensitive string analysis based on Monadic Second-Order Logic (M2L) [44]. They statically encode string operations that are used in java sanitization code into M2L and then check if a string generated by the sanitization code satisfies a pre-specified constraint using an M2L constraint solver such as MONA [16]. Compared to our automata-based approach, they do not handle loops.

7.2.2 Hybrid String Analysis

In AMNESIA [43] SQL Injection attacks are fought by first applying static string analysis to approximate the syntactic structure of an SQL query at a hotspot in a program. Then dynamic

monitoring is used to enforce this structure when executing the program. The key insights behind their technique is that information needed to predict the possible structure of SQL queries in a web application is contained within the application's code. So an SQLI attack, by injecting additional SQL statements into a query, would violate that structure. AMNESIA uses static string analysis from JSA [19] to analyze the application code and automatically build a model for the legitimate queries. This analysis is applied per each hotspot in the application in which an SQL query, stored in a string variable, is sent to the database for execution. The model used is a non-deterministic finite automaton (NFA). The alphabet of the NFA is SQL keywords and operators, delimiters and place holders for input string values. After that, at runtime, all dynamically generated queries are monitored and checked for compliance with previously statically generated model (NFA). Queries that violate the model are classified as illegal, prevented from executing on the database and reported to the application developers and administrators.

Balzarotti et al. [9] combine both dynamic and static techniques in their tool Saner to verify PHP programs. They support language-based replacement by incorporating FSA [89] and if a sanitizer is found to be vulnerable, then a dynamic analysis is performed to check using a pre-defined set of dangerous test cases if sanitization operations could miss any of these test cases. They only support bounded computation for loops and approximate variables updated in a loop as arbitrary strings once the computation does not converge within a fixed bound. We incorporate the widening operator in [11] to tackle this problem and obtain a tighter approximation that enables us to verify a larger set of programs.

7.2.3 String Constraint Solvers

Constraint solving received a lot of attention recently since it is used by all symbolic execution tools which have become very popular. String constraints solvers are used specifically to deal with constraints that involve string variables. Nikolaj Bjorner et al. [15] studied the decidability problem of string constraints extracted from path conditions for programs using .NET string library. The .NET string library is modeled using a first order language called *the string library language*. They proved that the satisfiability of the string library language where length of string variables is fixed and replace operation is removed is decidable. If the length is not fixed but replace is still removed then the problem is open. If replace is introduced then the satisfiability problem is undecidable for constraints with multiple variables.

Automata-based Solvers. Shannon et al. [83] propose symbolic execution to perform string analysis on Java programs. In their approach, automata are used to trace path constraints and encode the values of string variables. Fu et al. [32] use symbolic execution to find SQLI vulnerabilities in .NET applications. They use automata to represent string constraints and support string-based replacement (as opposed to language-based replacement which we support in our analysis). Wassermann et al. [98] use finite state transducers (FST) to model constraints in PHP web applications for test input generation. Their approach is based on concolic execution [82], where results of a concrete execution are used to collect constraints on program execution. These constraints are then used to generate new test cases. Hooimeijer and Weimer [48, 49] present an automata-based decision procedure for solving equations over regular language variables using partial state space construction. Since they use a single track automata encoding,

the techniques in their paper can only provide an approximation for solving equations over string variables. One potential solution is using multitrack automata to model relations among string variables [109] [110]. Rex [93, 94] uses symbolic automata to solve string constraints involving regular expressions.

Our automata operations that we presented here can be used for string constraints solving. One advantage over previous solvers is that we can solve constraints involving replace operations.

Bounded Solvers. Nikolaj Bjorner et al. [15] designed a string constraint solver that solves string constraint in two phases. First, a string constraint is abstracted into an integer constraint by replacing each string variable with an unquantified integer variable. After solving the new integer constraint with an SMT solver, results are used to fix (i.e., bound) the lengths of the strings. Then the original string constraint is solved after fixing the length of strings variables in it.

HAMPI [59] allows string constraints to be specified as a membership in a context free language or a regular language. Then a higher bound on string variables' lengths is specified which converts the constraint into a constraint on a finite (i.e., regular) language. The ability to specify constraints with context free languages is only a convenience feature which makes it much easier to specify constraints on variables that hold a context free language values such as SQL queries. Given an input constraint, it is normalized into a *core string constraint* where each constraint is of the form $x = R$ or $x \neq R$ where R is a regular expression. A simple algorithm is provided to convert a bounded CFG into a regular expression. Then core

constraints are translated into a quantifier-free logic of bit-vectors constraints which are passed into a special constraint solver called STP. If there is a solution, HAMPI decodes the output bit-vectors into a string solution.

A string constraint solver called Kaluza [80] was built on top of the HAMPI. It uses the same approach of bounding the lengths of the execution paths (by bounding loops) and using a bounded string solver. Kaluza is used by KUDZU—a symbolic execution framework for javascript—to solve string constraints in javascript and generate new input that is used to explore more execution paths.

On one hand, bounded solvers are able to handle a larger set of string operations and predicates compared to automata-based string solvers. However, since they bound the length of strings they may miss some solutions that we can catch especially given how well our automata-based algorithms scale well with length (see end of chapter 3) The ability to handle unbounded length in addition to unbounded loops allows us to prove, for example, that a validation function conforms to a given policy while they can not.

7.2.4 Relationship to Model Checking, Abstract Interpretation and Pre- and Post-Condition Computation

In chapter 3, we presented our string analysis for IVSL programs using the classical data flow analysis [1, 56]. In the following discussion, we present our string analysis from other perspectives.

Model Checking

State Space. From a model checking perspective, our string analyzer is an infinite state symbolic model checker [17, 67] for string manipulating programs (such as IVSL programs). A state S for an IVSL program P with n variables $v_1, v_2, \ldots, v_n$ consists of the program counter pc along with a tuple of string values $\langle w_1, w_2, \ldots, w_n \rangle$ where each element w_i in the tuple represents current string value for the corresponding IVSL variable v_i in state S. The initial state for an IVSL program S_0 has the value of pc as 0, which represents the first line in an IVSL program, along with the tuple of values $\langle \epsilon, \epsilon, \ldots, \epsilon \rangle$ (i.e., empty strings). Starting from the initial set of reachable states (which contains only the initial state S_0 of the program), in each step of the model checking algorithm, we compute, given the current set of reachable states, the next set of reachable states using the transfer functions in chapter 3 as post-image computers. At each step of the model checking algorithm, a set of reachable states is represented as a tuple of sets of strings where each set represents all reachable values for a string variable at that step.

Symbolic Representation. To represent the set of reachable states, we use two levels of symbolic representation, DFAs which symbolically represent regular sets of strings and MBDDs which symbolically represent the transition relation of a DFA (see 3.2.1). A DFA provides a finite symbolic representation of the possibly infinite set of reachable values for a string variable. Since the set that represents all reachable values for a string variable is not guaranteed to be regular, our representation is an over-approximation of this set (i.e., at each step of the model checking algorithm, we over-approximate the set of reachable states).

Model Checking Algorithm Termination. To guarantee the termination of the model checking algorithm we need to guarantee that the model checking algorithm is going to find the least fixed point for the set of reachable states. However, the problem is that the lattice of possible reachable states (cartesian product of the lattices for each variable which is discussed in 3.2.4) has an infinite height. To deal with this problem (i.e., guarantee termination) we use the concept of "widening" from abstract interpretation [23, 24] to approximate the least fix-point for the set of reachable states by computing the least upper bound of the fix-point (using the automata widening operator from [11]).

Safety Properties. The policies that we use in policy-based verification are similar to the safety properties used in model checking. We use regular expression to specify these policies instead of temporal logic since we do not need to express time. We can formulate a safety property Φ—based on the security policy—such that it specifies that the output of an IVSL program should not be an element in the language of the security policy. Using CTL logic, we check for $AG(\Phi)$.

Pre- and Post-Condition Computation

Here we present our analysis as a computation of pre- and post-conditions in Floyd-Hoare Logic [45]. Let us start with the following observation, given a variable x, both a DFA and a unary (i.e., single-variable) predicate on x (i.e., a condition or a constraint with x as the only free variable that appear in it) are finite structures that represent or encode the (possibly infinite) set of values that the variable x can take. Assuming any input, the post-image of

a sanitizer (see 3.1.1) is equivalent to the post-condition. Given a subset of the co-domain of a sanitizer function, its pre-image (see 3.1.2) is equivalent to computing the weakest pre-condition for a given post-condition. A DFA M_1 is weaker that a DFA M_2 if $M_1 \supseteq M_2$ (i.e., we check if a DFA represents the weakest pre-condition by replacing the logical implication operator $\Rightarrow$ with the automaton operator $\supseteq$). In the policy-based bug detection, for example, we start the pre-image computation with the intersection of the policy (or its negation) with the post-image (or its negation) as the post-condition. Then, we propagate this post-condition backwards by conjuncting it (intersecting it) with branch conditions and using our pre-image computation algorithms for string functions (i.e., the backward transfer functions) as Dijkstra predicate transformers [29] for DFAs. Finally, the widening operator is used to find the loop invariant I.

7.3 Differential Analysis and Repair

Differential Analysis. Differential analysis techniques [63–65, 77] typically stop after finding differences between different pieces of code without trying to repair it. In [77] differential symbolic execution is used to find differences between original and refactored code by summarizing procedures into symbolic constraints and then comparing different summaries using an SMT solver. SYMDIFF [63] computes the difference between two different functions in a language agnostic way by reducing both functions to Boggie [10] intermediate language then finds semantic differences using the Z3 SMT solver [68]. There are several specialized dif-

ferential analysis techniques that focus on web applications. In NoTamper [13] the authors analyze client-side script code using dynamic symbolic execution to generate test cases that are subsequently used as inputs to the server side of the application. Since the approach relies on dynamic (black-box) testing, it can suffer from limited code coverage. In a recent follow up paper [14], the same authors propose WAPTEC, which uses symbolic execution of the server code to guide the test case generation process and expand coverage. In addition to finding semantic differences, our work also generates a fix for such difference. MiTV [86] uses dynamic symbolic execution engine Pex [69] to test the correctness of user input validation functions for .NET web applications. These functions are first classified according to the type of input they validate. Then each validation function is tested by comparing it to a subset of the functions under the same class. In our differential analysis, we provided more precise mapping techniques since we map two functions—on the client to the server-side— that have high potential of being similar.

Differential Program Repair. Differential program repair [5, 74, 79, 84, 85, 99, 100] became an active topic recently. Weimer et al. [99, 100] repair bugs detected through manually written test suites by using genetic programming. The abstract syntax tree (AST) of the program is randomly mutated multiple time by deleting, swapping and/or copying subtrees related to the execution path taken by the test suite. Mutation is done until a mutated version passes the original test suite. Compared to our differential repair approach, this approach needs test suites and can introduce new errors into the program since it does not consider the semantics. Son et al. [84, 85] find access control problems in PHP by comparing a possibly buggy AST

with one that is considered to be correct and then patch the difference by inserting statements from the latter into the former. Unlike these syntax based approaches, our differential repair algorithm uses a semantic approach which enables us to generate precise repairs in multiple languages. In [74, 79] test suites are used to find bugs then symbolic execution is used to find constraints on variables that result in such bugs. Using the solution to the negation of these constraints, a patch is synthesized for the program such that it passes all test suites. Unlike this approach, our approach does not require a test suite and it can handle unbounded loops using fixpoint computation. Livshits et al. [66] automatically place a set of sanitizers in a sanitizer free program based on a user defined policy and a flow graph. The sanitizers are placed in the flow graph such that they satisfy the specified policy and at the same time avoid idempotency problems. In our case we take into account the previous sanitization code and the way we generate the repair allows us to place it at the beginning of the code that is under repair without requiring a placement policy. Placing the repair before the original code, instead of changing the code, allows us to avoid interference with the original sanitization code that may have side-effects.

Chapter 8

Conclusions

Input validation and sanitization is the cornerstone in the correctness and security of web applications. Since most of web applications input comes from HTML input fields, inadequate manipulation of string variables causes many bugs and serious vulnerabilities in web applications. In this dissertation, we presented a number of techniques for automatic detection and repair of input validation and sanitization bugs in web applications.

We separated the verification problem into three stages: 1) in the first stage we showed how to extract input validation and sanitization functions from web applications; 2) in the second stage we showed how to analyze the extracted functions using automata-based symbolic string analysis; 3) in the third phase we gave a number of bug detection and repair techniques that utilize the analysis results from the second stage.

In the extraction phase, we gave a formal specification for different types of input validation and sanitization functions. Then we presented a new domain specific language called Input

Validation and Sanitization Language (IVSL) that can express these different types of input validation and sanitization functions. We showed how IVSL was designed to be capable of capturing input validation and sanitization semantics in a number of popular web programming languages such as PHP, Javascript and Java. Then we showed how to extract input validation and sanitization functions statically and dynamically from Javascript, PHP and Java into IVSL programs.

Given an extracted input validation and sanitization function, we presented an automata-based symbolic string analysis framework that can be used to over-approximate the 1) output of the function assuming any input, 2) input to the function that may result in a given output, and 3) input to the function that will be rejected. We designed a number of algorithms to model predicates that are used for input validation operations along with a number of new specialized language-based replacement algorithms that can be used to model sanitization operations in an efficient and precise way.

Using this analysis framework, we presented two novel techniques to detect and repair bugs and vulnerabilities in web applications. The first technique is a policy-based technique that verifies the input validation and sanitization function against a given policy to see if either 1) the function is under-constrained and accepts bad inputs, or 2) the function is over-constrained and rejects good inputs. Utilizing security policies, we applied this technique to detect and repair vulnerabilities on the server-side of the web applications. Furthermore, utilizing min and max policies for common input fields such as email address, we applied this technique to detect bugs in client-side input validation code.

Since policy-based verification is dependent on manually written policies, we presented a novel differential analysis and repair technique that avoids this limitation. Given two input validation and sanitization functions, the technique is capable of 1) finding the semantical difference between the two functions 2) if a difference is found, then the technique is capable of automatically repairing this difference.

We built two new verification tools, STRANGER and SEMREP based on a new and more efficient and precise version of the automata manipulation library LIBSTRANGER that is extended with our new specialized language-based replacement algorithms. Using these tools, we demonstrated the effectiveness of our bug detection and repair techniques on several benchmarks, as well as some large-scale applications written in PHP, Java and Javascript.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[2] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic Differential Repair for Input Validation and Sanitization. In *To appear in ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.

[3] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, ICSE 2012, pages 947–957, Piscataway, NJ, USA, 2012. IEEE Press.

[4] M. Alkhalaf, S. R. Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel. ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'12)*, 2012.

[5] J. Andersen and J. L. Lawall. Generic patch inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 337–346, Washington, DC, USA, 2008. IEEE Computer Society.

[6] C. Anderson and P. Giannini. Type checking for javascript. *Electronic Notes in Theoretical Computer Science*, 138(2):37–58, 2005.

[7] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005-Object-Oriented Programming*, pages 428–452. Springer, 2005.

[8] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *Software Engineering, IEEE Transactions on*, 36(4):474–494, 2010.

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the Symposium on Security and Privacy*, 2008.

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, pages 321–333, 2004.

[12] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, LNCS 1260*. Springer Verlag, 1997.

[13] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

[14] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 575–586, New York, NY, USA, 2011. ACM.

[15] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] BRICS. The MONA project. `http://www.brics.dk/mona/`.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 10< sup> 20</sup> states and beyond. *Information and computation*, 98(2):142–170, 1992.

[18] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *APLAS*, pages 374–388, 2006.

[19] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[20] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. ACM Press, sep 2005.

[21] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM Sigplan Notices*, volume 44, pages 50–62. ACM, 2009.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[23] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[24] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[25] N. Coward and Y. Yoshida. Java Servlet Specification Version 2.4. Technical report, nov 2003.

[26] J. Dahse and T. Holz. Simulation of built-in php features for precise static code analysis. In *In Proceedings of Network and Distributed System Security (NDSS'14) Symposium*, 2014.

[27] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 541–554. ACM, 2014.

[28] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 382–391. IEEE, 2009.

[29] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[30] L. DâĂŹAntoni and M. Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification*, pages 624–639. Springer, 2013.

[31] L. DâĂŹAntoni and M. Veanes. Static analysis of string encoders and decoders. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, pages 209–228. Springer, 2013.

[32] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC*, pages 87–96, 2007.

[33] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, Apr. 1997.

[34] Gargoyle Software. HtmlUnit: headless browser for testing web applications. `http://htmlunit.sourceforge.net/`.

[35] D. Gerdemann and G. van Noord. Transducers from rewrite rules with backreferences. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, 1999.

[36] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of 26th International Conference on Software Engineering (ICSE)*, 2004.

[37] T. V. Griffiths. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *Journal of the ACM (JACM)*, 15(3):409–413, 1968.

[38] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[39] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[40] A. Guha, S. Krishnamurthi, and T. Jim. Static analysis for ajax intrusion detection. In *In International World Wide Web Conference*. Citeseer, 2009.

[41] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *ECOOP 2010–Object-Oriented Programming*, pages 126–150. Springer, 2010.

[42] W. Halfond, J.Viegas, and A.Orso. A classification of sql injection attacks and countermeasures. In *In Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[43] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.

[44] J. G. Henriksen, O. J. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. B. Sandholm. Mona: Monadic second-order logic in practice. In *IN PRACTICE, IN TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS, FIRST INTERNATIONAL WORKSHOP, TACAS '95, LNCS 1019*. Springer-Verlag, 1995.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[46] 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*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[47] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCIA'11)*, VMCAI'11, 2011.

[48] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, 2009.

[49] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, 2010.

[50] J. E. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation*. Pearson Addison Wesley, third edition, 2007.

[51] IBM. The IBM X-Force Trend and Risk Report. `http://www-935.ibm.com/services/us/iss/xforce/trendreports/`, 2011.

[52] S. H. Jensen, M. Madsen, and A. Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[53] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Static Analysis*, pages 238–255. Springer, 2009.

[54] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P*, pages 258–263, 2006.

[55] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 258–263, 2006.

[56] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[57] L. Karttunen. The replace operator. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 16–23, 1995.

[58] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *Proceedings of the 9th symposium on Dynamic languages*, pages 17–26. ACM, 2013.

[59] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[60] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *Programming Languages and Systems*, pages 326–341. Springer, 2008.

[61] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Transactions on Software Engineering*, 30(3), March 2004.

[62] E. Kneuss, P. Suter, and V. Kuncak. Phantm: Php analyzer for type mismatch. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 373–374, New York, NY, USA, 2010. ACM.

[63] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[64] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 345–355, New York, NY, USA, 2013. ACM.

[65] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.

[66] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

[67] K. L. McMillan. *Symbolic model checking*. Springer, 1993.

[68] Microsoft Inc. Z3 SMT Solver. `http://z3.codeplex.com`.

[69] Microsoft Research. Pex. `http://research.microsoft.com/en-us/projects/pex/`.

[70] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441, 2005.

[71] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 231–238. Association for Computational Linguistics, 1996.

[72] M. Morrison. *Head First JavaScript*. O'Reilly Media, 2007.

[73] Mozilla Foundation. Rhino: Javascript for Java. `http://www.mozilla.org/rhino/`.

[74] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[75] O. W. A. S. P. (OWASP). Top ten project. `http://www.owasp.org/`, May 2007.

[76] O. W. A. S. P. (OWASP). Top ten project. `http://www.owasp.org/`, May 2010.

[77] S. J. Person. *Differential Symbolic Execution*. PhD thesis, Lincoln, NB, USA, 2009. AAI3365729.

[78] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[79] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[80] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proc. of the 31st IEEE Symposium on Security and Privacy (Oakland 2010)*, volume 0, pages 513–528, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[81] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[82] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pages 263–272, 2005.

[83] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[84] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

[85] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[86] K. Taneja, N. Li, M. R. Marri, T. Xie, and N. Tillmann. Mitv: multiple-implementation testing of user-input validators for web applications. In *ASE*, pages 131–134, 2010.

[87] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 166–176, New York, NY, USA, 2011. ACM.

[88] UCSB VLab. Stranger: Semantic Differential Repair tool. `http://cs.ucsb.edu/~vlab/stranger/`.

[89] G. van Noord. FSA utilities toolbox. `http://odur.let.rug.nl/~vannoord/Fsa/`.

[90] G. van Noord and D. Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Proc. of the 4th International Workshop on Implementing Automata (WIA)*, pages 122–139. Springer-Verlag, July 1999.

[91] M. Veanes. Symbolic string transformations with regular lookahead and rollback. In *Proceedings of the 9th Ershov Informatics Conference (PSI'14)*. Springer, 2014.

[92] M. Veanes and N. Bjørner. Symbolic automata: The toolkit. In *TACAS*, pages 472–477, 2012.

[93] M. Veanes, N. Bjørner, and L. De Moura. Symbolic automata constraint solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 640–654. Springer, 2010.

[94] M. Veanes, P. De Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 498–507. IEEE, 2010.

[95] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: algorithms and applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 137–150, New York, NY, USA, 2012. ACM.

[96] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[97] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[98] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 249–260, 2008.

[99] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, May 2010.

[100] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[101] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

[102] F. Yu, M. Alkhalaf, and T. Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009.

[103] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, 2010.

[104] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 251–260, New York, NY, USA, 2011. ACM.

[105] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[106] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *15th International SPIN Workshop on Model Checking Software (SPIN 2008)*, pages 306–324, 2008.

[107] F. Yu, T. Bultan, and B. Hardekopf. String abstractions for string verification. In *SPIN*, pages 20–37, 2011.

[108] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 322–336, 2009.

[109] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *CIAA*, pages 290–299, 2010.

[110] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. *Int. J. Found. Comput. Sci.*, 22(8):1909–1924, 2011.