UNIVERSITY OF CALIFORNIA
Santa Barbara

Automatic Verification of String Manipulating Programs

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

Doctor of Philosophy

in

Computer Science

by

Fang Yu

Committee in Charge:

Professor Tevfik Bultan, Chair

Professor Oscar H. Ibarra

Professor Richard Kemmerer

June 2010

The Dissertation of
Fang Yu is approved:

Professor Oscar H. Ibarra

Professor Richard Kemmerer

Professor Tevfik Bultan, Committee Chairperson

June 2010

Automatic Verification of String Manipulating Programs

by

Fang Yu

To my wife, my parents, and my lovely kids.

Acknowledgements

I am most grateful to my advisor Tevfik Bultan for his spiritual, technical and financial support that carried me through the doctoral program. Tevfik devotes considerable time and energy to his students. Without the countless discussions with him, this dissertation would simply not be written and completed. I wish to express my deep gratitude to Oscar H. Ibarra. His wisdom and knowledge have been a great source of inspiration to me. He taught me how to conduct rigorous research and his insightful comments on automata improved the content and presentation of this dissertation. I would like to thank Richard Kemmerer for his detailed corrections and suggestions for this dissertation. I would like to thank Marco Cova and Christopher Krugel for their valuable comments on web application security. I would also like to thank Amr El Abbadi, the department chair, for his encouragement, as well as Amanda Hoagland, Mandy Drasco, and the other department staff for their non-technical and technical support throughout my studies. Special thanks go to my colleague, Muath Alkhalaf, who worked closely with me to build and release the string analysis tool to the public.

I am greatly grateful for having chances to work on other verification topics with Chao Wang and Aarti Gupta at the NEC Laboratory America, Farn Wang and Sy-Yen Kuo at National Taiwan University, and Bow-Yaw Wang and Der-Tsai Lee at Academia Sinica. These experiences have strengthened and broadened my research in this field.

Finally, I wish to express my deepest gratitude to my wife, Wang-Ping Chen, and my parents, Tseng-Yi Yu and Mei-Yen Yu-Tseng, for all the love, help, and support that they have given to me. They have given me far beyond what I can thank.

Curriculum Vitæ

Fang Yu

Education

2000	Master of Business Administration in Information Management, National Taiwan University.
1998	Bachelor of Business Administration, National Taiwan University.

Experience

2006 – 2010	Graduate Student Researcher, University of California, Santa Barbara.
2007, 2008	Summer Intern, NEC Laboratories America, Princeton.
2005 – 2006	Teaching Assistant, University of California, Santa Barbara.
2001 – 2005	Research Assistant, Institute of Information Science, Academia Sinica, Taiwan.

Publications

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. “Stranger: An Automata-based String Analysis Tool for PHP.” In Proceedings of the *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010),* Paphos, Cyprus, March 2010. (Tool paper)

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. "Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses." In Proceedings of the *IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, Auckland, New Zealand, 2009. (Short paper)

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. "Symbolic String Verification: Combining String Analysis and Size Analysis." In Proceedings of the *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, York, UK, March 2009.

Fang Yu, Chao Wang, Aarti Gupta, and Tevfik Bultan. "Modular Verification of Web Services Using Efficient Symbolic Encoding and Summarization." In Proceedings of the 16th *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2008)*, Atlanta, GA, November 2008.

Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. "Symbolic String Verification: An Automata-based Approach." In Proceedings of the 15th *International SPIN Workshop on Model Checking of Software (SPIN 2008)*, Los Angeles, CA, August 2008.

Fang Yu, Tevfik Bultan, and Erik Peterson. “Automated Size Analysis for OCL.” In Proceedings of the *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007),* pp. 331-340, Dubrovnik, Croatia, September 2007.

Oscar H. Ibarra, Sara Woodworth, Fang Yu, and Andri Paun. “On Spiking Neural P Systems and Partially Blind Counter Machines.” *Natural Computing*, Vol 7 (1), pp. 3-19, Springer Netherlands, 1572-9796, March 2008.

Fang Yu and Bow-Yaw Wang. “SAT-based Model Checking for Region Automata.” *International Journal of Foundations of Computer Science*, Vol. 17, No. 4, pp. 775-796, August 2006.

Farn Wang, Geng-Dian Huang, and Fang Yu. “TCTL Inevitability Analysis of Dense-time Systems: from Theory to Engineering.” *IEEE Transactions on Software Engineering*, Vol. 32, No. 7, pp.510-526. July 2006.

Farn Wang, K. Schmidt, Fang Yu, Geng-Dian Huang, and Bow-Yaw Wang. “BDD-based Safety Analysis of Concurrent Software with Pointer Data Structures using Graph Automorphism Symmetry Reduction.”

IEEE Transactions on Software Engineering, Vol. 30, No. 6, pp. 403-417, 2004.

Farn Wang, Geng-Dian Huang, and Fang Yu. "Symbolic Simulation of Industrial Real-Time and Embedded Systems - Experiments with the Bluetooth baseband communication protocol." *Journal of Embedded Computing*, Vol. 1, pp. 39-56, 2004.

Oscar H. Ibarra, Sara Woodworth, Fang Yu, and Andri Paun. "On Spiking Neural P Systems and Partially Blind Counter Machines." In Proceedings of the *5th International Conference on Unconventional Computation (UC 2006)*, York, UK, September 2006.

Fang Yu, Chung-Hung Tsai, Yao-Wen Huang, Hung-Yaw Lin, Der-Tsai Lee, and Sy-Yen Kuo. "Efficient Exact Spare Allocation via Boolean Satisfiability." In Proceedings of the *20th IEEE International Symposium of Defect and Fault Tolerance in VLSI Systems (DFT 2005)*, pp. 361-370, Monterey, CA, October 2005.

Fang Yu and Bow-Yaw Wang. "Toward Unbounded Model Checking for Region Automata." In Proceedings of the *2nd International Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, LNCS 3299, pp. 20-33, Taipei, Taiwan, November 2004.

Fang Yu, Bow-Yaw Wang and Yaw-Wen Huang. "Bounded Model Checking for Region Automata." In Proceedings of the *Joint Conference on Formal Modeling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System (FORMATS-FTRTFT 2004),* LNCS 3253, pages 246-262, Grenoble, France, September 2004.

Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. "Verifying Web Applications Using Bounded Model Checking." In Proceedings of the *International Conference on Dependable Systems and Networks (DSN 2004),* pages 199-208, Florence, Italy, Jun 28-Jul 1, 2004.

Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. "Securing Web Application Code by Static Analysis and Runtime Protection." In Proceedings of the *13th International World Wide Web Conference (WWW 2004),* pages 40-52, New York, May 17-22, 2004.

Farn Wang, Geng-Dian Huang, and Fang Yu. "Numerical Coverage Estimation for the Symbolic Simulation of Real-Time Systems." In Proceedings of the *23rd IFIP International Conference on Formal Tech-*

niques for Networked and Distributed Systems (FORTE 2003), LNCS 2767, pages 160-176, October 2003.

Farn Wang, Geng-Dian Huang, and Fang Yu. "TCTL Inevitability Analysis of Dense-time Systems." In Proceedings of the *8th International Conference on Implementation and Application of Automata (CIAA 2003)*, LNCS 2759, pages 176-187, July 2003.

Farn Wang and Fang Yu. "OVL Assertion Checking of Embedded Software with Dense-Time Semantics." In Proceedings of the *9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, LNCS 2968, pages 254-278, February 2003.

Farn Wang, Geng-Dian Huang, and Fang Yu. "Symbolic Simulation of Real-Time Concurrent Systems." In Proceedings of the *9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, LNCS 2968, pages 595-617, February 2003.

Abstract

Automatic Verification of String Manipulating Programs

Fang Yu

In this dissertation, we investigate the *string verification problem:* Given a program that manipulates strings, we want to verify assertions about string variables. We formalize the string verification problem as reachability analysis of *string systems* and demonstrate that the string analysis problem is undecidable in general. We present sound automata-based symbolic string analysis techniques for automatic verification of string manipulating programs. String analysis is a static analysis technique that determines the values that a string expression can take during program execution at a given program point. This information can be used to detect security vulnerabilities and program errors, and to verify that program inputs are sanitized properly.

Most important Web application vulnerabilities, such as SQL Injection, Cross Site Scripting and Malicious File Execution, are due to inadequate manipulation of string variables. We use our automata-based string analysis techniques to detect and prevent such vulnerabilities in web applications. Our approach consists of three phases: Given an attack pattern, we first conduct a vulnerability analysis to identify if strings that match the attack pattern can reach security-sensitive functions. Next, we compute the vulnerability signature which characterizes all input strings that can exploit the

discovered vulnerability. Given the vulnerability signature, we then construct sanitization statements that 1) check if a given input matches the vulnerability signature and 2) modify the input in a minimal way so that the modified input does not match the vulnerability signature. Our approach is capable of generating relational vulnerability signatures (and corresponding sanitization statements) for vulnerabilities that are due to more than one input.

We extend our automata-based approach to analyze systems with both string and integer variables. We present a composite symbolic verification technique that combines string and size analyses with the goal of improving the precision of both. Our composite analysis automatically discovers the relationships among the integer variables and the lengths of the string variables. Finally, we present a relational string verification technique based on multi-track automata and abstraction. Our approach is capable of verifying properties that depend on relations among string variables.

We have developed a tool called STRANGER that implements our automata-based symbolic string analysis approach. STRANGER can be used to find and eliminate string-related security vulnerabilities in PHP applications.

Contents

Acknowledgements **v**

Curriculum Vitæ **vii**

Abstract **xiii**

List of Figures **xviii**

List of Tables **xx**

1 Introduction **1**
- 1.1 Automata-based String Analysis . . . 6
- 1.2 Symbolic Vulnerability Analysis . . . 12
- 1.3 Sanitization Synthesis . . . 18
- 1.4 Composite Analysis . . . 28
- 1.5 Relational String Analysis . . . 31
- 1.6 Summary of Contributions . . . 36

2 String Systems **39**
- 2.1 Decidability and Undecidability Results . . . 40

3 Automata-based String Analysis **48**
- 3.1 String Manipulation Operations . . . 49
- 3.2 String Operations on Automata . . . 52
- 3.3 Pre-image Computation . . . 64
  - 3.3.1 Concatenation . . . 64
  - 3.3.2 Replacement . . . 68
- 3.4 Widening Automata . . . 70

4 Symbolic Vulnerability Analysis **76**
4.1 Dependency Graph . . . 77
4.2 Vulnerability Analysis . . . 78
4.2.1 Forward Analysis . . . 80
4.2.2 Backward Analysis . . . 82
4.3 Inter-procedural Analysis . . . 84
4.3.1 Summarization . . . 85
4.3.2 Call Dependency Graph . . . 86
4.3.3 Generating Function Summaries . . . 89
4.3.4 Composing Function Summaries . . . 93
4.4 Experiments . . . 94
4.4.1 Forward Analysis . . . 94
4.4.2 Forward+Backward Analysis . . . 97

5 Sanitization Synthesis **102**
5.1 Sanitization Generation . . . 106
5.2 Relational Signatures . . . 113
5.3 Experiments . . . 118

6 Composite Analysis **123**
6.1 Length Automata Construction . . . 125
6.1.1 Length Constraints on String Automata . . . 126
6.1.2 From String Automata to Unary Length Automata . . . 128
6.1.3 From Unary Length Automata to Semilinear Set . . . 129
6.1.4 From Semilinear Set to Binary Length Automata . . . 130
6.2 Composite Verification . . . 135
6.2.1 Verification Framework . . . 139
6.2.2 Implementation . . . 145
6.3 Experiments . . . 148

7 Relational String Analysis **150**
7.1 Regular Approximation of Word Equations . . . 152
7.1.1 Aligned Multi-track DFAs . . . 153
7.1.2 Word Equations . . . 154
7.1.3 Construction of Multi-track DFAs for Word Equations . . . 157
7.2 Symbolic Reachability Analysis on Multi-track Automata . . . 166
7.2.1 Forward Fixpoint Computation . . . 166
7.2.2 Summarization . . . 169
7.2.3 Alignment . . . 172
7.3 String Abstractions . . . 176

7.3.1 Alphabet Abstraction . . . 177
7.3.2 Relation Abstraction . . . 178
7.3.3 Heuristics for Abstraction Selection . . . 180
7.3.4 Handling Complex String Operations . . . 183
7.4 Experiments . . . 183

8 Stranger Tool **190**
8.1 Tool Description . . . 192
8.1.1 PHP Parser and Taint Analyzer . . . 193
8.1.2 String Analyzer . . . 193
8.1.3 String Manipulation Library . . . 195
8.2 Experiments and Discussions . . . 196

9 Related Work **199**
9.1 String Analysis . . . 200
9.2 Size Analysis . . . 206
9.3 Composite Analysis . . . 208

10 Conclusion **211**

Bibliography **214**

List of Figures

1.1 The Percentages of String-related Web Application Vulnerabilities [19] 2
1.2 A Vulnerability Example . . . 8
1.3 A Sanitization Example . . . 9
1.4 A Small Example . . . 13
1.5 Results of Forward and Backward Analyses . . . 15
1.6 A Simple Example . . . 19
1.7 Patches for the example in Figure 1.6 . . . 21
1.8 A Vulnerability Signature . . . 22
1.9 Another Simple Example . . . 23
1.10 Dependency graph . . . 24
1.11 A Relational Vulnerability Signature . . . 26
1.12 Patch for the example from Figure 1.9 . . . 28
1.13 A Sanitization Example with a Length Condition . . . 29
1.14 A String Length Routine . . . 30
1.15 A Branch Example . . . 33
1.16 A Loop Example . . . 35

2.1 The Syntax of String Manipulating Programs . . . 40

3.1 Constructing $M_1^{'}$ from M_1 . . . 57
3.2 Constructing $M_2^{'}$ from M_2 and M_h . . . 58
3.3 Constructing $M^{''}$ from $M^{'}$. $M^{'}$ is the Intersection of ${M_1}'$ and $M_2^{'}$ . . . 60
3.4 $M_1^{''}$ is PROJECT$(M^{''}, k+2)$, M is PROJECT$(M_1^{''}, k+1)$ . . . 63
3.5 A Transducer M for $X = (ab)^{+}.Z$ . . . 65
3.6 A Transducer M for $X = Y.(ab)^{+}$ . . . 66
3.7 Widening Automata . . . 73
3.8 An Approximation Sequence . . . 74

4.1 A Simple Function 88
4.2 The Dependency Graphs: G_f and G_F 89
4.3 M_f: The Summary DFA 92

5.1 Input Matching Overhead 121

6.1 The Length Automata of $(baaab)^+$ 134
6.2 The Length Automata of $(baaab)^+ab$ 134
6.3 The Rewritten String Length Routine 138

8.1 The Architecture of STRANGER 192

List of Tables

4.1 The Forward Experimental Results of Stranger. 95
4.2 The Experimental Results of Saner. 98
4.3 The Basic Data of Dependency Graphs 98
4.4 Total Performance 99
4.5 String Function Performance 99
4.6 Attack and Vulnerability Signatures 101

5.1 Vulnerability Analysis Performance 119
5.2 Signature Generation Performance 120
5.3 Minimum Edge and Alphabet Cuts 121

6.1 Regular Languages and Their Length Sets 127
6.2 The Experimental Results of Composite Analysis. 149

7.1 Experimental Results against Basic and MFE Benchmarks Using Single-track Automata. 186
7.2 Experimental Results against Basic and MFE Benchmarks Using Multi-track Automata. 187
7.3 Experimental Results against Basic and MFE Benchmarks Using Multi-track Automata and Alphabet Abstraction. 187
7.4 Experimental Results against XSS Benchmarks. 189
7.5 Experimental Results against XSS Benchmarks Using Multi-track Automata and Alphabet Abstraction. 189

Chapter 1

Introduction

Web applications have become a crucial part of commerce, entertainment and social interaction. They are rapidly replacing desktop applications. And, in the near future, they are likely to play critical roles in national infrastructures such as healthcare, national security, and the power grid. There is a large stumbling-block to this ever increasing reliance on Web applications: Web applications are notorious for security vulnerabilities that can be exploited by malicious users. The global accessibility of Web applications makes this an extremely serious problem. In fact, in the Common Vulnerabilities and Exposures (CVE) list [19] (which documents computer security vulnerabilities and exposures) Web application vulnerabilities have occupied the first three positions in recent years.

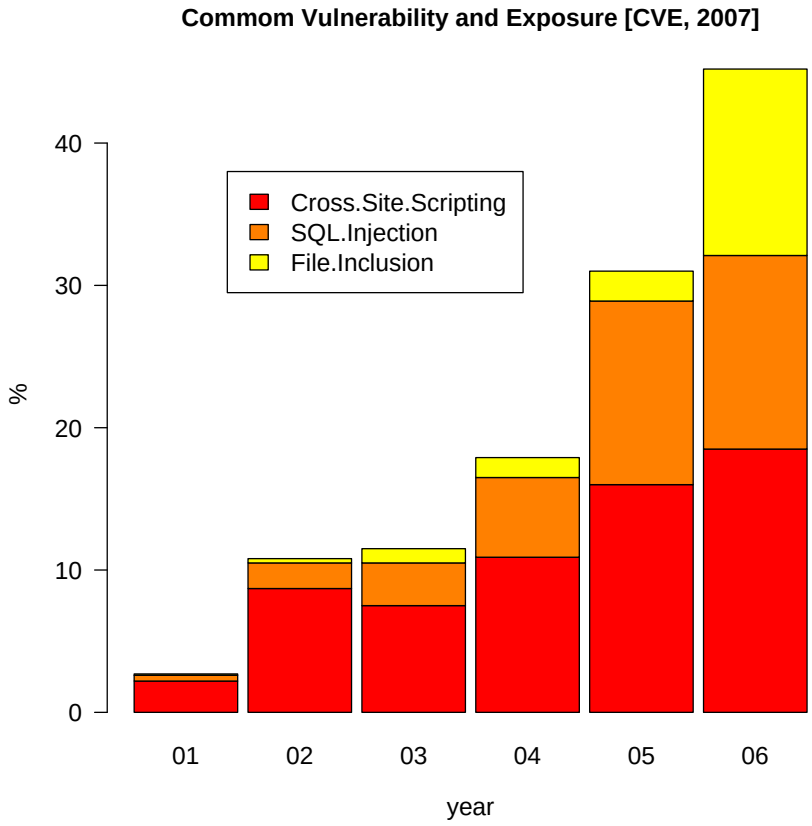


Figure 1.1: The Percentages of String-related Web Application Vulnerabilities [19]

Our motivation in this work is to eliminate string manipulation errors in Web applications. String variables play an essential role in Web applications. They are used for getting user inputs, querying databases, and constructing responses that are sent to users. In fact, the most important Web application vulnerabilities are due to inadequate manipulation of string variables. As shown in Figure 1.1, the percentage of string-related vulnerabilities over all reported vulnerabilities has increased from 3% in 2001 to 45% in 2006. According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious Web application vulnerabilities [41], the top three vulnerabilities reported in 2007 were: 1) Cross Site Scripting (XSS), 2) Injection Flaws (such as SQL injection) and 3) Malicious File Execution (MFE). After three years, the top two vulnerabilities are still Injection Flaws and XSS in OWASP's 2010 top ten list.

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. A SQL Injection vulnerability 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 a SQL command that the application executes. This permits the attacker to damage or get unauthorized access to data stored

in a database. MFE vulnerabilities occur when developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. All these vulnerabilities involve string manipulation operations and they occur due to inadequate sanitization and use of input strings provided by users.

Clearly, there is an urgent need for an automatic and sound approach to establishing correctness of string manipulation operations in Web applications. In this dissertation we present automata-based symbolic string analysis techniques that can be used to identify vulnerabilities related to string manipulation, generate characterization of user inputs that might exploit a discovered vulnerability, and generate sanitization statements to patch a vulnerability.

This dissertation consists of six parts: (1) String Systems, (2) Automata-based String Analysis, (3) Symbolic Vulnerability Analysis, (4) Sanitization Synthesis, (5) Composite String Analysis, and (6) Relational String Analysis. In the first part of the dissertation we formally define the string systems and present decidability and undecidability results about the verification of several classes of string systems. Next, we discuss how to use automata to represent values of string variables and how to model string functions on automata. Based on these automata constructions, we present symbolic forward and backward reachability analyses along with a novel summarization technique for interprocedural analysis.

Vulnerabilities related to string manipulation can be characterized as attack patterns, i.e., regular expressions that specify vulnerable values for sensitive operations (called sinks). Given an application and an attack pattern, we conduct a symbolic forward reachability analysis to identify if there are any input values that a user can provide to the application that could lead to a vulnerable value to be passed to a sensitive operation. Once a vulnerability is identified, the next important question is to identify what set of input values can exploit the given vulnerability. A vulnerability signature is a characterization of all such input values. We use a symbolic backward reachability analysis to generate the vulnerability signatures for the discovered vulnerabilities.

A vulnerability signature can be used to identify how to sanitize the user input to eliminate the discovered vulnerability, or it can be used to dynamically monitor the user input and reject or modify the values that can lead to an exploit. We use the vulnerability signatures to automatically generate sanitization statements for patching vulnerable Web applications.

We extend our approach to the verification of systems with both string and integer variables by combining size analysis with our string analysis. We use a forward fixpoint computation based on a composite symbolic representation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables. This composite analysis improves the pre-

cision of both string and size analyses and can be applied to other security problems such as buffer overflows.

Finally, we present a relational string analysis technique based on multi-track automata. Our approach is capable of verifying properties that depend on relations among string variables. We further propose alphabet and relation abstractions to adjust the precision and performance of our symbolic string analysis techniques.

In the following sections, we present the motivation for different parts of our work and illustrate our techniques on several examples.

1.1 Automata-based String Analysis

The string analysis technique we present utilizes forward and backward reachability computations that use deterministic finite automaton (DFA) as a symbolic representation. We use the symbolic DFA representation provided by the MONA DFA library [5], in which transition relations of the DFAs are represented as Multi-terminal Binary Decision Diagrams (MBDDs). 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 DFAs for all the variables, we compute the next state DFAs. We present algorithms for next state computation for string operations such as concatenation and language-based replacement. Particularly, we present an algorithm for the

language-based replacement operation that computes the DFA for REPLACE(M_1, M_2, M_3) where M_1, M_2, and M_3 are DFAs that accept the set of original strings, the set of match strings, and the set of replacement strings, respectively.

Our language-based replacement operation is essential for modeling various built-in functions that can be used to perform input validation in the PHP language, a widely used language to develop Web applications. These functions provide a general mechanism to scan a string for matches to a given pattern (expressed as a regular expression) and to replace the matched text with a replacement string. As an example of modeling these functions, consider the following statement:

```
$username = ereg_replace("<script *>", "", $_GET["username"]);
```

The expression `_GET["username"]` returns the string entered by the user, the `ereg_replace` call replaces all matches of the search pattern (`"<script *>"`) with the empty string (`""`), and the result is assigned to the variable `username`. This statement can be modeled by our language-based replacement operation, where M_1 accepts arbitrary strings, M_2 accepts the set of strings that start with `<script` followed by zero or more spaces and terminated by the character `>`, and M_3 accepts the empty string.

We believe that we are the first to extend the MONA automata package to analyze these complex string operations on real programs. In addition to computing the language-based replacement operation, another difficulty is implementing these string operations without using the standard constructions based on the ϵ-transitions, since

```
1:   foreach ($_POST as $name => $value) {
2:     if ($name != 'process' && $name != 'password2') {
3:       $count++;
4:       $result .= "'$name' = '$value'";
5:       if ($count <= $numofparts)
6:         $result .= ", ";
7:     }
8:   }
9:   $query = "UPDATE `pblguestbook_config` SET $result";
10:  mysql_query($query);
```

Figure 1.2: A Vulnerability Example

the MBDD-based automata representation used by MONA does not allow ϵ-transitions. We model non-determinism by extending the alphabet with extra bits and then project them away using the on-the-fly subset construction algorithm provided by MONA. We apply the projection one bit at a time, and after projecting each bit away, we use the MBDD-based automata minimization to reduce the size of the resulting automaton.

Consider the PHP program fragment in Figure 1.2 which demonstrates a vulnerability from a guestbook application called `PBLguestbook-1.32`. This program fragment traverses the input strings entered by the user (which are stored in the `_POST` array) in a loop (lines 1-8) and constructs a query string by accumulating them (by concatenating them to the `result` variable). This query is then sent to the back-end database (line 10).

This program has an SQL injection vulnerability. Input strings are concatenated in the loop at lines 1-8 to form the string used to query the application's database.

```
1:    foreach ($_POST as $name => $value) {
1.1:    $name = preg_replace("/[^a-zA-Z0-9]/", "", $name);
1.2:    $value = preg_replace("/'/", "", $value);
2:      if ($name != 'process' && $name != 'password2') {
3:        $count++;
4:        $result .= "'$name' = '$value'";
5:        if ($count <= $numofparts)
6:          $result .= ", ";
7:      }
8:    }
9:    $query = "UPDATE 'pblguestbook_config' SET $result";
10:   mysql_query($query);
```

Figure 1.3: A Sanitization Example

Since no sanitization is performed, an attacker can modify the query, for example, by injecting a parameter with value `'; DROP DATABASE #`. In this case, the SQL string sent to the database will be `UPDATE 'pblguestbook_config' SET 'name' = ''; DROP DATABASE #'`. Note that the ';' character separates distinct queries and the '#' character starts a comment. Therefore, if the database allows the execution of multiple queries, it will execute the legitimate query intended by the developer and the injected query that drops the entire database. The vulnerability can be fixed by adding a sanitization step on the input parameters before the query string is formed.

A properly sanitized version of this program fragment is shown in Figure 1.3. The sanitization is achieved in lines 1.1 and 1.2 by deleting potentially problematic characters in the variables `$name` and `$value`, hence preventing the presented SQL command injection attack.

We analyzed both the vulnerable and the sanitized versions of this program fragment using our string analysis tool. Our string analysis tool constructed a DFA that gives an over-approximation of the string values that the variable `query` can take at line 10. We wrote a regular expression characterizing strings that can be used for SQL command injection and converted it to a DFA. (Note that these types of attack DFAs can be constructed once and stored in a library. They do not have to be specified separately for each program that is being analyzed). Then, we checked if the intersection of the language recognized by the DFA for the `query` variable at line 10, and the DFA characterizing the SQL command injection attack is empty. When we applied our analysis to the vulnerable program fragment shown above, our string analysis tool reported that the intersection is not empty, i.e., the program fragment might be vulnerable. However, when we applied our analysis to the sanitized version, our tool reported that the intersection is empty, showing that the variables are properly sanitized.

It is worthwhile to note some of the challenges in analyzing the example given above. First, in order to prove that the variables are properly sanitized, we need to statically interpret the replacement function `preg_replace` with reasonable precision. Second, our fixpoint computation has to converge even though the above program fragment contains a loop. We are able to handle both of these challenges by 1) proposing and implementing a novel language-based replacement operation and 2) using an automata widening operator. Note that, for the sanitized program fragment,

the fixpoint computation without widening will not converge. Moreover, a naive over-approximation, that sets the values of the variables that are updated in a loop to all possible strings, will not be a tight enough approximation to verify the sanitized program fragment. To tackle this challenge, we use the automata widening technique proposed by Bartzis and Bultan [4] to compute an over-approximation of the least fixpoint. Briefly, we merge those states belonging to the same equivalence class identified by certain conditions. This widening operator was originally proposed for automata representation of arithmetic constraints but the intuition behind it is applicable to any symbolic fixpoint computation that uses automata.

We present automata construction and manipulation operations for the string analysis described above in Chapter 3. The main contributions for this part of our work are: (1) pre- and post- image computations of common string manipulation functions, including a novel language-based replacement automata construction, and (2) use of an automata based widening operation that accelerates fixpoint computations for string analysis. In the following section we discuss our vulnerability analysis technique and demonstrate how we perform symbolic reachability analysis using basic automata operations mentioned above to formally verify Web applications.

1.2 Symbolic Vulnerability Analysis

We use automata-based string analysis techniques that we mentioned above for vulnerability analysis and vulnerability signature generation. Our analysis takes an attack pattern specified as a regular expression and a PHP program as input and 1) identifies if there is any vulnerability based on the given attack pattern, 2) generates a DFA characterizing the set of all user inputs that may exploit the vulnerability.

As we have stated earlier, our string analysis framework uses a DFA to represent values that string expressions can take. At each program point, each string variable is associated with a DFA. To determine if a program has any vulnerabilities, we use a forward reachability analysis that computes an over-approximation of all possible values that string variables can take at each program point. Intersecting the results of the forward analysis with the attack pattern gives us the potential attack strings if the program is vulnerable.

The backward analysis computes an over-approximation of all possible inputs that can generate those attack strings. The result is a DFA for each user input that corresponds to the vulnerability signature. We will discuss how to use vulnerability signatures to generate effective sanitization routines in the next section. Here we focus on how to conduct forward and backward symbolic reachability analyses with summarization techniques.

```
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 1.4: A Small Example

Consider another simple PHP script shown in Figure 1.4. This script is a simplified version of code from a real Web application that contains a vulnerability. The script 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 pattern and the target. The goal is to find all the substrings of the target that match the match pattern and replace them with the replace pattern. In the replace command shown in line 4, the match pattern is the regular expression `[^A-Za-z0-9 .-@://]`, the replace pattern is the empty string (which corresponds to deleting all the substrings that match the match pattern), and the target is the variable `www`. After the sanitization step, the PHP program outputs the concatenation of the variable `l_otherinfo`, the string constant `": "`, and the variable `www`.

The `echo` statement in line 5 is a sink statement since it can contain a Cross Site Scripting (XSS) vulnerability. For example, a malicious user may provide an input that

contains the string constant `<script` and execute a command leading to a XSS attack. The goal of the replace statement in line 4 is to remove any special characters from the input to prevent such attacks.

Using string replace operations to sanitize user input is common practice in Web applications. However, this type of sanitization is error prone due to the complex syntax and semantics of regular expressions. In fact, the replace operation in line 4 in Figure 1.4 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., eliminating everything other than alpha-numeric 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. A 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.

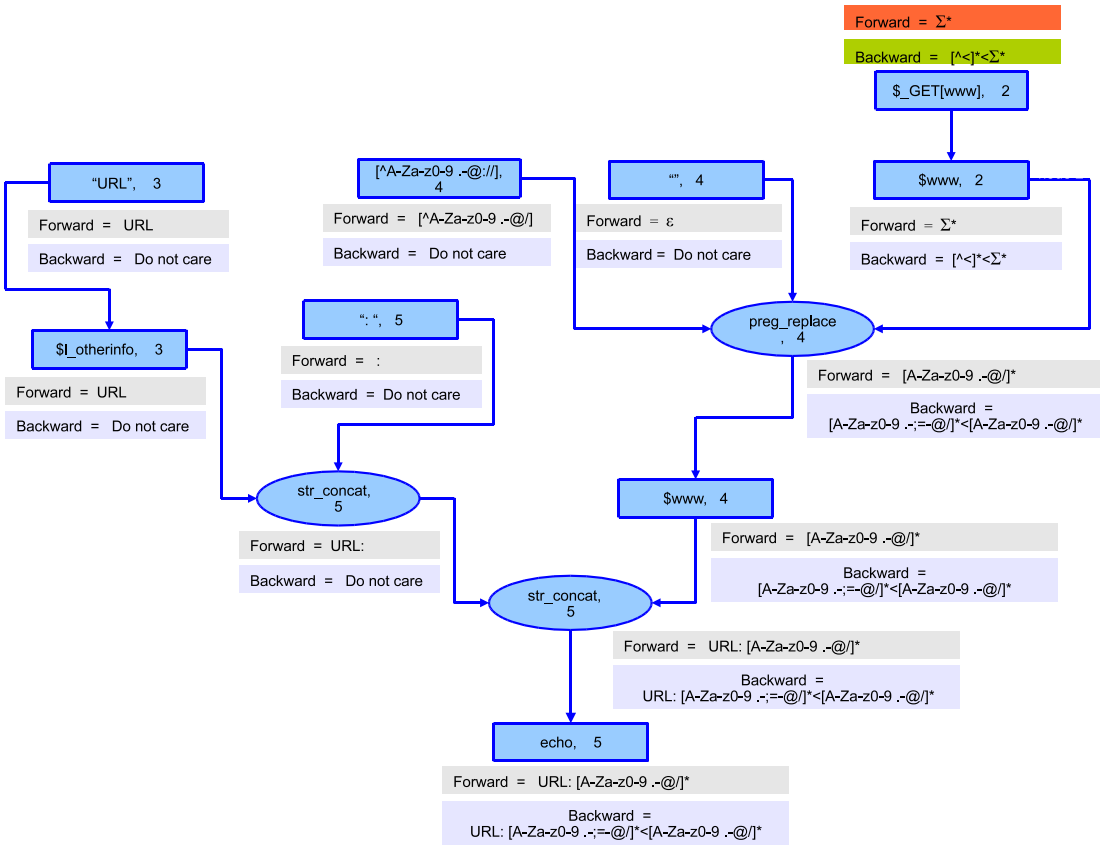


Figure 1.5: Results of Forward and Backward Analyses

Now, we will explain how our approach automatically detects this vulnerability. First, the attack pattern for the XSS attacks can be specified as Σ^* `<script` Σ^* (where Σ denotes any ASCII character), i.e., any string that contains the substring `<script` matches the attack pattern. If, during the program execution, a string that matches the attack pattern reaches a sink statement, then we say that the program is vulnerable.

For our small example, we simplify the attack pattern as $\Sigma^* < \Sigma^*$. Our analysis first generates the dependency graph for the input PHP program. Figure 1.5 shows the dependency graph for the PHP script in Figure 1.4 (the program segment that corresponds to a node and the corresponding line number are shown inside the node). Nodes 1 and 2 correspond to the assignment statement in line 2, nodes 3 and 4, correspond to the assignment statement in line 3, nodes 5, 6, 7 and 8 correspond to the replace statement in line 4, and nodes 9, 10, 11, and 12 correspond to the concatenation operations and the echo statement in line 5. Under each node we show the result of the forward and backward symbolic analyses as a regular expression.

During forward analysis we characterize all the user input as Σ^*, i.e., the user can provide any string as input. Then, using our automata-based forward symbolic reachability analysis, we compute all the possible values that each string expression in the program can take. For example, during forward analysis, node 2, that corresponds to the value of the string variable `www` after the execution of the assignment statement in line 2, is correctly identified as Σ^*. More interestingly, node 8, the value of the string variable `www` after the execution of the replace statement in line 4, is correctly identified as `[A-Za-z0-9 .-@:/]*` since any character that does not match the characters in the regular expression `[A-Za-z0-9 .-@://]` has been deleted.

Node 12 is the sink node. The result of the forward analysis identifies the value of the sink node as `URL:[A-Za-z0-9 .-@:/]*`. Next, we take the intersection of the

result of the forward analysis with the attack pattern to identify if the program contains a vulnerability. If the intersection is empty then the program is not vulnerable with respect to the given attack pattern. Since our analysis is sound, this means that there is no user input that can generate a string that matches the attack pattern at the sink node. However, in our example, the intersection of the attack pattern and the result of the forward analysis for the sink node is not empty and is characterized by the following regular expression: `URL:[A-Za-z0-9 .-;=-@:/]*<[A-Za-z0-9 .-@:/]*`.

The backward analysis starts from this intersection and traverses the dependency graph backwards to find out what input values can lead to string values at the sink node that fall into this intersection. Note that during backward analysis we do not need to compute any value for the nodes that are not on a path between an input node and a sink node. This means that during backward analysis we do not compute values for the nodes 3, 4, 5, 6, 9 and 10. The final result of the backward analysis is the result for the input node 1, which is characterized with the regular expression: `[^<]*<`Σ^*, i.e., any input string that contains the symbol < can lead to a string value at a sink node that matches the attack pattern.

This characterization of potentially harmful user inputs is called the *vulnerability signature* for a given attack pattern. It is an over-approximation of all possible inputs that can generate an attack string that matches the attack pattern. Based on the vulnerability signature our analysis computes for the program segment shown in Figure 1.4,

the programmer can eliminate the vulnerability either by fixing the erroneous replace statement in line 4 or by adding another replace statement that removes the < symbol from the input.

We present our vulnerability analysis in Chapter 4. The main contributions for this part of our work are: (1) a symbolic reachability analysis framework that combines forward and backward analyses, including a novel algorithm to generate vulnerability signatures, and (2) a summarization technique for interprocedural analysis. In the next section we discuss how to generate effective patches for vulnerable applications based on vulnerability signatures.

1.3 Sanitization Synthesis

We present techniques for automatically generating patches that eliminate string vulnerabilities in Web applications. We use two types of analysis: One based on string analysis with single-track automata that can be used to generate patches for vulnerabilities that depend on a single input, and another one based on string analysis with multi-track automata that can be used to generate patches for vulnerabilities that depend on multiple inputs. We present and implement two strategies for automatically generating patches based on vulnerability signatures: match-and-block and match-and-sanitize. We give an automata-theoretic characterization of the match-and-sanitize strategy and

```
1: <?php
2:    $name = $_GET["name"];
3:    $out = "NAME : " . $name;
4:    echo $out;
5: ?>
```

Figure 1.6: A Simple Example

prove that generating optimum modifications is an intractable problem. We instead give a heuristic approach based on a min-cut algorithm.

Once we compute a vulnerability signature using the techniques we discussed in the previous section, we automatically synthesize patches that eliminate the vulnerability. We use two strategies for patching:

- *Match-and-block:* We insert match statements in vulnerable Web applications and halt the execution when an input that matches the vulnerability signature is detected.

- *Match-and-sanitize:* We insert both match and replace statements in vulnerable Web applications. When an input that matches the vulnerability signature is detected, instead of halting the execution, the replace statement is executed. The replace statement deletes a small set of characters from the input such that the modified string no longer matches the vulnerability signature.

Consider the PHP script shown in Figure 1.6. This script starts with assigning the user input provided in the `_GET` array to the variable `name` in line 2. It concatenates

a constant string with variable `name` and assigns it to another variable `out` in line 3. Then it simply outputs the variable `out` using the `echo` statement in line 4.

The `echo` statement in line 4 is a sink statement since it can contain a Cross Site Scripting (XSS) vulnerability. For example, a malicious user may provide an input that contains the string constant `<script` and execute a command leading to a XSS attack. In order to prevent this vulnerability, it is necessary to *sanitize* the user inputs before using them in an `echo` statement. In the rest of this section we give an overview of how we use the results of our vulnerability analysis to generate the sanitization statements.

Let us again use the simplified attack pattern for XSS vulnerabilities characterized with the regular expression $\Sigma^* < \Sigma^*$. After the vulnerability analysis, we detect the segment is vulnerable and generate the vulnerability signature for the input `_GET["name"]` as a DFA that accepts the language $\Sigma^* < \Sigma^*$.

The vulnerability signature gives an over-approximation of all possible 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. In our *match-and-block* strategy we generate a patch that simply checks if the input string matches the vulnerability signature. If it does, it halts the execution without executing the rest of the script. The patch generated for the small example in Figure 1.6 based on the vulnerability signature $\Sigma^* < \Sigma^*$ and using the match-and-block strategy is shown in

```
1:    <?php
1.1:    if (preg_match(
          '/([=-\xfd]|[\x00-;])*<([\x00-\xfd])*/',$_GET["name"]))
1.2:      die("Invalid input");
2:      $name = $_GET["name"];
3:      $out = "NAME : " . $name;
4:      echo $out;
5:    ?>
```

(a) Patch 1 using match-and-block strategy

```
1:    <?php
1.1:    if (preg_match(
          '/([=-\xfd]|[\x00-;])*<([\x00-\xfd])*/',$_GET["name"]))
1.2:      $_GET["name"] =
              preg_replace('/</',"",$_GET["name"]);
2:      $name = $_GET["name"];
3:      $out = "NAME : " . $name;
4:      echo $out;
5:    ?>
```

(b) Patch 2 using match-and-sanitize strategy

Figure 1.7: Patches for the example in Figure 1.6

Figure 1.7(a). Note that the patched script will block any input string that contains the symbol <.

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 DFA of the vulnerability signature. Consider the DFA for the vulnerability signature $\Sigma^* < \Sigma^*$ shown in Figure 1.8 (we use $\Sigma - <$ to indicate any symbol other than $<$). 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

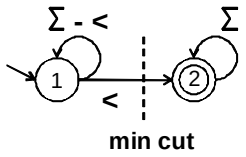


Figure 1.8: A Vulnerability Signature

accepted by the DFA. As we discuss in Section 5.1, this corresponds to finding a cut in the graph defined by the states and the transitions of the DFA, i.e., finding a set of edges such that when we remove them, there are no paths left in the graph from the initial state of the DFA to a final state. Note that each edge of the DFA is labeled with a symbol. After we find a cut, if we take the union of the symbols of the edges in the cut, we obtain a set of symbols such that any string accepted by the DFA must include at least one of the symbols in that set.

If we pick any cut, we may end up modifying the input more than necessary. For example, deleting all the characters from the input (which corresponds to including all the edges of the automata in the cut) will sanitize the input. However, deleting all the user input is not a very useful sanitization. What we want to do is to generate a patch that does not modify the input too much but guarantees that it will not lead to an attack string at the sink statement. Hence we use a min-cut algorithm to compute a cut that contains minimum number of edges. Then we generate a patch that deletes all the characters from the input that appear on the edges included in the cut set. For the DFA shown in Figure 1.8, the min-cut algorithm returns the single edge labeled with

```
1: <?php
2:    $title = $_GET["title"];
3:    $name = $_GET["name"];
4:    $out = "NAME : " . $title . $name;
5:    echo $out;
6: ?>
```

Figure 1.9: Another Simple Example

the symbol $<$. So we know that deleting the symbol $<$ from the input is sufficient for preventing attacks and we generate a patch that deletes all the $<$ symbols from the input as shown in Figure 1.7(b). Note that, unlike the patch shown in Figure 1.7(a), the patch generated based on the match-and-sanitize strategy continues to execute the script after the sanitization.

Relational Vulnerability Signature Generation: Consider the simple example shown in Figure 1.9. This example is similar to the one shown in Figure 1.6 with one significant difference: there are two input variables that both contribute to the string expression used at the sink statement at line 5.

Assume that we use our single-track automata based analysis described above to analyze this script. The set of attack strings generated for the sink statement at line 5 will again be: `NAME :` $\Sigma^* < \Sigma^*$. However, the result of the backward analysis will be different. The crucial step is the pre-condition computation for the statement in line 4. The input to this pre-condition computation will be a DFA that accepts the attack strings characterized by the regular expression given above. The result of the precondition

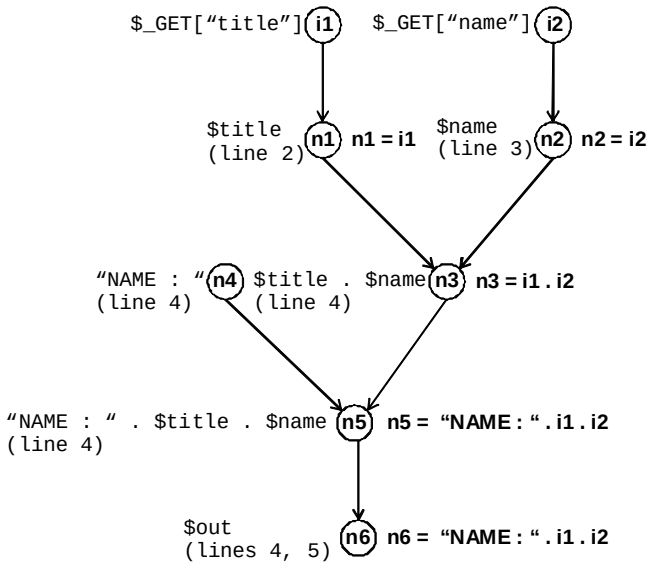


Figure 1.10: Dependency graph

computation will generate two DFAs, one for the variable `name` and one for the variable `title`, and these DFAs will characterize all possible values these two variables can take just before the execution of statement in line 4 that can lead to generation of an attack string at the sink statement in line 5. When we do this pre-condition computation we get two DFAs that accept the same language Σ^*, i.e., any value of either variable can lead to an attack string. Although this is a sound approximation it fails to capture the information that *at least one of these variables should contain the character* $<$. Note that this condition cannot be expressed as a constraint on an individual variable, it identifies a *relation* between the two string variables.

We developed a string analysis technique based on multi-track automata (MDFA) that computes relational vulnerability signatures. Our relational analysis uses one

multi-track automaton for each program point to capture the relationship between the input values and possible values of string expressions in the program. We use a forward analysis that operates on the dependency graph. We show the dependency graph for the example from Figure 1.9 in Figure 1.10. We write the string expression in the program that corresponds to each node in the dependency graph to the left side of the node and also give the line number. Our analysis starts from the input nodes and traverses the dependency graph while generating one MDFA for each internal node of the dependency graph. Each MDFA has one track for each input variable and one track for the string expression that corresponds to that node, and represents the relation between them. In Figure 1.10 we show a string constraint on the right side of each internal node. That string constraint characterizes the set of strings accepted by the MDFA for that node. For example, for node $n3$, the string constraint is $n3 = i1.i2$ which indicates that the string expression that corresponds to node $n3$ is equal to the concatenation of input $i1$ and input $i2$.

When the analysis reaches a sink node, we intersect the track that corresponds to the string expression for the sink node (in our example this would be the track that corresponds to node $n6$) with the attack pattern DFA (by extending the attack pattern DFA to an MDFA by adding extra tracks that accept all strings). After the intersection, we project away the track for the sink node, leaving only the tracks for the input nodes. The resulting MDFA represents the relational vulnerability signature. For our example,

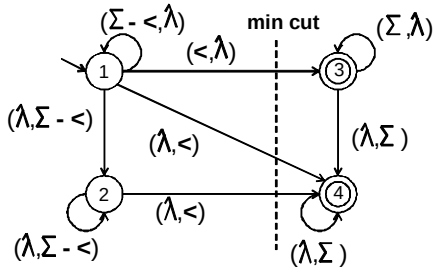


Figure 1.11: A Relational Vulnerability Signature

the vulnerability signature MDFA is shown in Figure 1.11 (where each transition is marked with two symbols, one for each track, and if a track is marked with the symbol λ then that means that no symbol from that track is consumed when that transition is taken). Note that this automaton accepts tuples of strings, where either the first string in the tuple or the second string in the tuple contains at least one $<$ symbol.

Once we compute the MDFA for the vulnerability signature, we need to generate the match and replace statements. For the match statement in the single-track case, one option is to convert the standard DFA representation to a regular expression and then use the PHP `preg_match` function. Although this is not very efficient (as we discuss in Chapter 5), the patches shown in Figure 1.7 use this option. However, if we try to generate two regular expressions, one for each input, from the automaton shown in Figure 1.11, we again get Σ^* for both inputs, so all inputs match. This could be OK if we use the match-and-sanitize strategy since, although all the input strings will be considered potentially vulnerable, only a small set of symbols that relate to the

vulnerability will be replaced. For example, the patch generated using this approach for the example in Figure 1.9 is shown in Figure 1.12.

However, if we use the match-and-block approach using the regular expression Σ^* will block all the inputs which is not acceptable. We will discuss how to generate more precise match statements from MDFA based vulnerability signatures in Chapter 5.

In order to generate the sanitization statements from relational vulnerability signatures, we find a min-cut in the vulnerability signature MDFA as we did for the single-track case. Then, for each track, we take the union of the symbols on that track for all the edges in the min-cut. In order to sanitize the input we need to remove the symbols for each track from the input that corresponds to that track. For example, based on the min-cut shown in Figure 1.11, we need to delete the symbol $<$ both from the inputs `_GET["name"]` and `_GET["title"]`. The automatically generated replace statements for this example are shown in Figure 1.12.

The main contributions for this part of our work include (1) an automated sanitization generation technique based on DFA-based vulnerability signatures, (2) an algorithm for sanitization generation that sanitizes the input by modifying it in a minimal way, and (3) novel relational vulnerability signature generation and sanitization synthesis techniques based on MDFAs.

```
1: <?php
1.1:    if (preg_match('/([\x00-\xfd])*/', $_GET["title"])
        and preg_match('/([\x00-\xfd])*/', $_GET["name"])) {
1.2:      $_GET["title"] =
               preg_replace('/</',"",$_GET["title"]);
1.3:      $_GET["name"] =
               preg_replace('/</',"",$_GET["name"]); }
2:      $title = $_GET["title"];
3:      $name = $_GET["name"];
4:      $out = "NAME : " . $title . $name;
5:      echo $out;
6: ?>
```

Figure 1.12: Patch for the example from Figure 1.9

1.4 Composite Analysis

We extend our automata-based static analysis to systems having both unbounded string and integer variables. We present a composite symbolic verification technique that combines string [1, 15, 52, 58] and size [20, 22, 46] analyses with the goal of improving the precision of both. We use a forward fixpoint computation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables.

Below, we present two motivating examples to demonstrate the advantages of the proposed composite string and size analysis technique. Consider the PHP program segment shown in Figure 1.13, which secures an identified vulnerable point [58]. This vulnerability appeared at line 218 in `trans.php`, distributed with `MyEasyMarket-4.1`.

```
1: <?php
2:       $www = $_GET["www"];
3:       $l_otherinfo = "URL";
4:       $www = ereg_replace("[^A-Za-z0-9 .\-@://]","",$www);
5:       if(strlen($www)<$limit)
6:         echo "<td>" . $l_otherinfo . ": " . $www . "</td>";
7:  ?>
```

Figure 1.13: A Sanitization Example with a Length Condition

Without proper sanitization and protection (lines 4 and 5) of the user-controlled variable `$www`, an attacker can inject the string `<script src=http://evil.com/attack.js>` and perform a XSS attack at line 6. The above code prevents such attacks by: (1) removing abnormal characters from `$www` at line 4, and (2) limiting the length of `$www` at line 5. Our analysis shows that this code segment is free from the specified attacks (with respect to the attack pattern) by showing that at line 6 (1) the length of the string `$www` is less than the allowed limit, and (2) under that limit the string variable `$www` cannot contain a value that matches the attack pattern. Note that if one performs solely size analysis, without knowing the contents of `$www`, the length of `$www` can not be determined precisely after line 3. On the other hand, if one performs solely string analysis, the branch condition at line 4 must be ignored. Both of these approximations may lead to false alarms.

Now, consider a standard `strlen` routine in C (Figure 1.14) that returns the length of a given string by traversing each character until hitting the end character, i.e., `'\0'`.

```
unsinged int strlen(char *s){
1:    char *ptr = s;
2:    unsigned int cnt =0;
3:    while(*ptr != '\0'){
4:        ++ptr;
5:        ++cnt;
6:    }
7:    return cnt;
}
```

Figure 1.14: A String Length Routine

This kind of standard string routine is widely used in legacy C systems, e.g., Apache, Samba, Sendmail, and WuFTP.

Let `*s`.*length* denote the size of the string pointed to by `s`. An essential property of this routine is that at line 7, `cnt = *s`.*length*, which can be used as the summary of this routine and significantly alleviates size analysis overhead [20, 53], however, none of the size analysis tools prove this property before using it. Our composite analysis is capable of proving this property. We first construct an assertion (arithmetic) automaton that accepts the values that satisfy `cnt = *s`.*length*. We then conduct our composite analysis by computing the forward fixpoint with widening. Upon reaching the fixpoint, at line 7, (1) the arithmetic automaton actually catches the relation that `*s`.*length* $=$ `*ptr`.*length* $+$ `cnt`, and (2) the string automaton of `*ptr` only accepts $\{\epsilon\}$. We prove the property by showing that the intersection of the language of (1) and the length of the language of (2) is included in the language of the assertion automaton.

In sum, we extend our automata-based symbolic analysis technique to systems having both string variables and integer variables. This extension is designed for statically analyzing string and integer variables in programs. We present a simple imperative language and perform the composite analysis on this language. In addition to the string operations supported in [58] and the arithmetic operations supported in [3, 4] , this language supports two new string operations: prefix and suffix.

The contributions of this part consist of: (1) a novel algorithm to construct length automata, (2) a composite analysis that combines string analysis and size analysis, (3) a prototype tool that integrates previous techniques, as well as new features that include length automata construction, string operations (prefix, suffix) and boundary operations (min and max).

1.5 Relational String Analysis

We present a new relational string analysis technique based on multi-track automata and abstraction. Our approach is capable of verifying properties that depend on relations among string variables. We present and implement a forward symbolic reachability analysis technique that computes an over-approximation of the reachable states of a string system using widening and summarization. We use multi-track deterministic finite automata (MDFAs) as a symbolic representation to encode the set of possi-

ble values that string variables can take at a given program point. Unlike prior string analysis techniques, the analysis is *relational*, i.e., it is able to keep track of the relationships among the string variables, improving the precision of the string analysis and enabling verification of invariants such as $X_1 = X_2$ where X_1 and X_2 are string variables. We describe the precise construction of MDFAs for linear word equations, such as $c_1 X_1 c_2 = c_1' X_2 c_2'$ and show that non-linear word equations (such as $X_1 = X_2 X_3$) cannot be characterized precisely as a MDFA. We propose a regular approximation for non-linear equations and show how these constructions can be used to compute the post-condition of branch conditions and assignment statements that involve concatenation. We use summarization for inter-procedural analysis by generating a multi-track automaton (transducer) characterizing the relationship between the input parameters and the return values of each procedure. To be able to use procedure summaries during our reachability analysis we *align* multi-track automata so that normalized automata are closed under intersection.

To improve the efficiency of our approach, we propose two string abstraction techniques: alphabet and relation abstractions. In alphabet abstraction, we identify a set of characters that we are interested in and use a special symbol to represent the rest of the characters. In relation abstraction, we identify the variables that are related and encode them as a single multi-track automata. For those that are not related, we use multiple single-track automata to encode their values, where relations among them are

```
1: input X1;
2: input X2;
3: if (X1 = X2) goto 6;
4: X1:=X2.c;
5: goto 7;
6: X1:=X1.c;
7: assert (X1 = X2.c);
```

Figure 1.15: A Branch Example

abstracted away. We define an abstraction lattice that combines these abstractions under one framework and show that earlier results on string analysis can be mapped to several points in this abstraction lattice.

Consider an example shown in Figure 1.15. Existing automata-based string analysis techniques are not able to prove the assertion at the end of this program segment since they use single-track automata. Consider a symbolic analysis technique that uses one automaton for each variable at each program point to represent the set of values that the variables can take at that program point. Using this symbolic representation we can do a forward fixpoint computation to compute the reachable state space of the program. For example, the automaton for variable X_1 at the beginning of statement 2, call it $M_{X_1,2}$, will recognize the set $L(M_{X_1,2}) = \Sigma^*$ to indicate that the input can be any string. Similarly, the automaton for variable X_2 at the beginning of statement 3, call it $M_{X_2,3}$, will recognize the set $L(M_{X_2,3}) = \Sigma^*$. The question is how to handle the branch condition in statement 3. If we are using single track automata, all we can do at the beginning of statement 6 is the following: $L(M_{X_1,6}) = L(M_{X_2,6}) =$

$L(M_{X_1,3}) \cap L(M_{X_2,3})$. The situation with the else branch is even worse. All we can do at line 4 is to set $L(M_{X_1,4}) = L(M_{X_1,3})$ and $L(M_{X_2,4}) = L(M_{X_2,3})$. Both branches will result in $L(M_{X_1,7}) = \Sigma^*.c$ and $L(M_{X_2,7}) = \Sigma^*$, which is clearly not strong enough to prove the assertion.

Using the presented techniques, we can verify the assertion in the above program. In our approach, we use a single multi-track automaton for each program point, where each track of the automaton corresponds to one string variable. For the above example, the multi-track automaton at the beginning of statement 3 will accept any pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$. However, the multi-track automaton at the beginning of statement 6 will only accept pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2$. Let $[X/X']$ denote replacing X' with X. We compute the post-condition $(\exists X_1.(X_1 = X_2) \wedge (X_1' = X_1.c))[X_1/X_1']$, and generate the multi-track automaton that only accepts pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2.c$. Similarly, the multi-track automaton at the beginning of statement 4 will only accept pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 \neq X_2$, and after the assignment, we will generate the multi-track automaton that only accepts pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2.c$. Hence, we are able to prove the assertion in statement 7.

Consider another simple example shown in Figure 1.16. There are several challenges in proving that the assertion at line 5 holds. First, this program contains an infinite loop and does not terminate. If we try to compute the reachable configurations

```
1: X1 := a;
2: X2 := a;
3: X1 := X1.b;
4: X2 := X2.b;
5: assert (X1=X2);
6: goto 3;
```

Figure 1.16: A Loop Example

of this program by iteratively adding configurations that can be reached after a single step of execution, our analysis will never terminate. However, there exists a fixpoint characterizing the reachable configurations at each program point. We incorporate a widening operator to accelerate our symbolic reachability computation and compute an over-approximation of the fixpoint that characterizes the reachable configurations. Second, the assertion is an implicit property, i.e., there is no assignment, such as $X_1 := X_2$, or branch condition, such as $X_1 = X_2$, that implies that this assertion holds. Finally, the assertion specifies the equality among two string variables. Analysis techniques that encode reachable states using multiple single-track DFAs will raise a false alarm, since, individually, X_1 can be abb and X_2 can be ab at program point 5, but they cannot take these values at the same time. It is not possible to express this constraint using single-track automata.

For this example, our multi-track, automata-based string analysis technique terminates in three iterations and computes the precise result. The multi-track automaton that characterizes the values of string variables X_1 and X_2 at program point 5, call it

M_5, recognizes the language: $L(M_5) = (a, a)(b, b)^+$. Since $L(M_5) \subseteq L(X_1 = X_2)$, we conclude that the assertion holds. Although in this case the result of our analysis is precise, it is not guaranteed to be precise in general. However, it is guaranteed to be an over-approximation of the reachable configurations. Hence, our analysis is sound and if we conclude that an assertion holds, the assertion is guaranteed to hold for every program execution.

Furthermore, instead of using the full alphabet, e.g., full ASCII encoding, using our abstraction technique, the above example can be verified efficiently with an abstract alphabet $\{a, b, \star\}$, where $\star$ represents all characters other than a or b.

In Chapter 7 we present our contributions on relational string analysis which include (1) a sound symbolic analysis technique for string verification that over-approximates the reachable states of a given string system using multi-track automata and summarization and (2) alphabet and relation abstractions to adjust the precision and performance of our symbolic string analysis technique. We evaluate the presented techniques with respect to several string analysis benchmarks extracted from real Web applications.

1.6 Summary of Contributions

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

1. We formally characterize the string verification problem as the reachability analysis of string systems and show decidability/undecidability results for several string analysis problems.

2. We develop a sound symbolic analysis technique for string verification that over-approximates the reachable states of a given string system using automata.

3. We propose the first composite approach that combines string analysis with size analysis and show how the precision of both analyses can be improved by using length automata.

4. We develop the first relational string analysis technique using multi-track automata which is capable of proving properties that depend on the relations among string variables.

5. We propose alphabet and relation abstractions that can be used to adjust the precision and performance of our symbolic string analysis techniques.

6. We propose a novel algorithm for language-based replacement to model string replace functions that are commonly used to modify user inputs in Web applications.

7. We adopt a symbolic automata encoding and leverage its efficient manipulations.

8. We incorporate an automata-based widening operator to accelerate the fixpoint computations and develop a novel summarization technique for inter-procedural analysis.

9. We combine symbolic forward and backward reachability analyses to generate vulnerability signatures that characterize all malicious inputs that can exploit vulnerable Web applications.

10. We present techniques for automatically synthesizing sanitization statements from vulnerability signatures.

11. We implement STRANGER, the first public automata-based string analysis tool. STRANGER can automatically detect XSS, SQL Injection, and MFE vulnerabilities in PHP Web applications, as well as generate the vulnerability signatures for each detected vulnerability.

Chapter 2

String Systems

We start with defining string systems. Figure 2.1 presents the basic syntax for string systems. We only consider string variables and hence variable declarations need not specify a type. All statements are labeled. We only consider one string operation (concatenation) at this point (which is enough to prove some undecidability results). Function calls use call-by-value parameter passing. We allow goto statements to be non-deterministic (if a goto statement has multiple target labels, then one of them is chosen non-deterministically). If a string system contains a non-deterministic goto statement it is called a non-deterministic string system, otherwise, it is called a deterministic string system.

prog ::= *decl** *func**
decl ::= `decl` *id*$^+$;
func ::= *id* (*id**) `begin` *decl** *lstmt*$^+$ `end`
lstmt ::= *l*:*stmt*
stmt ::= *seqstmt*
| `if` *exp* `then goto` *l*;
| `goto` *L*; where *L* is a set of labels
| `input` *id*;
| `output` *exp*;
| `assert` *exp*;
seqstmt::=*id* := *sexp*;
| *id* := `call` *id* (*sexp**);
exp ::= *bexp* | *exp* $\wedge$ *exp* | $\neg$ *exp*
bexp ::= *atom* = *sexp*
sexp ::= *sexp*.*atom* | *atom*
atom ::= *id* | *c*, where *c* is a string constant

Figure 2.1: The Syntax of String Manipulating Programs

2.1 Decidability and Undecidability Results

Before discussing our symbolic string analysis technique we prove that string analysis is an undecidable problem and, therefore, any sound string analysis technique has to use conservative approximations in order to guarantee convergence.

Let $S(X_1, X_2, \ldots, X_n)$ denote a string system with string variables X_1, X_2, ..., X_n and a finite set of labeled instructions. There are several attributes we can use to classify string systems. For example, as mentioned above, a string system can be deterministic or non-deterministic. We can also classify a string system based on the alphabet used by the string variables, such as a string system with a unary alphabet or a string system with a binary alphabet, etc. Additionally, we can restrict the number of variables in the

string systems, such as a string system with only 2 variables ($S(X_1, X_2)$) or 3 variables ($S(X_1, X_2, X_3)$), etc. Finally, we can restrict the set of string expressions that can be used in the assignment and conditional branch instructions.

In order to identify different classes of string systems we will use the following notation. We will use the letters D and N to denote deterministic and non-deterministic string systems, respectively. We will use the letters B and U to denote if the alphabet used by the string variables is the binary alphabet $\{a, b\}$ or the unary alphabet $\{a\}$, respectively. We will use K to denote an alphabet of arbitrary size. For example, $DUS(X_1, X_2, X_3)$ denotes a deterministic string system with three variables and the unary alphabet whereas $NBS(X_1, X_2)$ denotes a nondeterministic string system with two variables and the binary alphabet. We will denote the set of assignment instructions allowed in a string system as a superscript and the set of expressions involved in conditional branch instructions as subscript. Hence, $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3,X_2=X_3}$ denotes a deterministic string system with three variables X_1, X_2, and X_3, and the unary alphabet $\{a\}$ where the assignment instructions are of the form $X_1 := X_1a$, $X_2 := X_2a$, or $X_3 := X_3a$ (i.e., we only allow concatenation of one symbol to a string variable in each assignment instruction) and the conditional branch instructions can only be of the form: **if** $X_3 = X_1$ **goto** L or **if** $X_3 = X_2$ **goto** L (i.e., we only allow equality checks and do not allow comparison of X_1 and X_2.)

The *halting problem* for string systems is the problem of deciding, given a string system S, where initially the string variables are initialized to the null string, ϵ, whether S will halt on some execution. More generally, the *reachability problem for string systems* (which need not halt) is the problem of deciding, given a string system S and a configuration C (i.e., the instruction label and the values of the variables), whether at some point during a computation, C will be reached. Note that we define the halting and the reachability conditions using existential quantification over the execution paths, i.e., the halting and the reachability conditions hold if there exists an execution path that halts or reaches the target configuration, respectively. Hence, if the halting problem is undecidable, then the reachability problem is undecidable.

Theorem 1 *The halting problem for* $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3,X_2=X_3}$ *is undecidable.*

Proof 1 *It is well-known that the halting problem for two-counter machines, where initially both counters are 0, is undecidable [39]. During the execution of a counter machine, at each step, a counter can be incremented by 1, decremented by 1, and tested for zero. The counters can only assume nonnegative values.*

We will show that a two-counter machine M *can be simulated with a string system* $S(X_1, X_2, X_3)$ *in* $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3,X_2=X_3}$. *The states of* M *can be represented as labels in the string system* S. *The states where the counter-machine* M *halts will be represented with the halt instruction in string system* S. *We will use the lengths of the strings* X_1, X_2 *and* X_3 *to simulate the values of the counters* C_1 *and* C_2. *The*

value of C_1 will be simulated by $|X_1| - |X_3|$, and the value of C_2 will be simulated by $|X_2| - |X_3|$.

The counter machine M starts from the initial configuration $(q_0, 0, 0)$ where q_0 denotes the initial state and the two integer values represent the initial values of counters C_1 and C_2, respectively. The initial configuration of the string system S will be $(q_0, \epsilon, \epsilon, \epsilon)$ where q_0 is the label of the first instruction, and the strings $\epsilon, \epsilon, \epsilon$ are the initial values of the string variables X_1, X_2 and X_3, respectively. The instructions of the counter-machine C will be simulated as follows (where each statement is followed by a goto statement that transitions to the next state or instruction):

Counter machine	*String system*
inc C_1	$X_1 := X_1 a$
inc C_2	$X_2 := X_2 a$
dec C_1	$X_2 := X_2 a; X_3 := X_3 a$
dec C_2	$X_1 := X_1 a; X_3 := X_3 a$
if $(C_1 = 0)$	**if** $(X_1 = X_3)$
if $(C_2 = 0)$	**if** $(X_2 = X_3)$

Note that although this transformation will allow the simulated counter values to possibly take negative values, this can be fixed by adding a conditional branch instruction before each decrement that checks that the simulated counter value is not zero

before the instructions simulating the decrement instruction is executed. The string system S constructed from M based on these rules will simulate M. Hence, halting problem is undecidable for the string systems in $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3,X_2=X_3}$.

In fact, Theorem 1 can be strengthened: There is a *fixed* string system $S(X_1, X_2, X_3)$ in $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3,X_2=X_3}$ such that it is undecidable to determine, given an arbitrary nonnegative integer d, whether $S(X_1, X_2, X_3)$ will halt when X_1 is initially set to string a^d and X_2 and X_3 are initially set to ϵ. This follows from the fact that there exists a fixed universal 2-counter machine M that can simulate a universal single-tape deterministic Turing machine. Given a description of a Turing machine TM as input, M halts if and only if TM halts on blank tape. Since it is undecidable to determine if a Turing machine halts on blank tape, it is undecidable to determine if M will halt on some input. Since, we can construct a fixed string system $S(X_1, X_2, X_3)$ simulating M, as in Theorem 1, it is undecidable to determine if $S(X_1, X_2, X_3)$ will halt starting from some initial configuration.

Next, we show that the three variables in Theorem 1 are necessary in the sense that when there are only two variables, reachability is decidable. This result does not hold when the system is nondetereministic, as we shall see in Theorem 3.

Consider the class of deterministic 2-variable string systems where the constants are over an alphabet with arbitrary cardinality, and we are allowed to use conditional branch instructions of the form: **if** $X_1 = X_2$ **goto** L. (Note that because the alphabet is

not necessarily unary, this **if** statement is not equivalent to **if** $|X_1| = |X_2|$ **goto** L as in the case of the unary alphabet.) Assignment statements are of the form: $X_i := X_i a$ or $X_i := aX_i$, where a is a single symbol. And, there is a halt instruction, which we may assume occurs at the end of the program.

Theorem 2 *The halting problem for* $DKS(X_1, X_2)^{X_i:=X_ia, X_i:=aX_i}_{X_1=X_2}$ *is decidable.*

Proof 2 *Let* S *be a string system in* $DKS(X_1, X_2)^{X_i:=X_ia, X_i:=aX_i}_{X_1=X_2}$ *and* k *be its length (i.e., number of instructions), including the assignments, and the conditional and unconditional branch statements.*

Label the instructions of S *by* $1, ..., k$*. We can think of each assignment,* $i: \ A$ *as equivalent to the instruction,* $i: \ A;$ **goto** $i+1$*. Hence, every instruction except the halt instruction and the* **if** *statements has a* **goto***.*

By an "execution of a positive **if** *statement", we mean that when the* **if** *statement is executed,* $X_1 = X_2$*.*

During the computation of S*, if it is not in an infinite loop, then the interval (i.e., number of steps) between the executions of any two consecutive positive* **if** *statements is at most* k*. The reason for this is that during the interval,* S *executes only* **goto***'s and assignment statements with* **goto***'s (note that a non-positive* **if** *statement leads directly to the instruction following the* **if***). Hence, the number of steps would be at most* k*, since there are at most* k **goto***'s and assignments with* **goto***'s.*

Now, an execution of a positive **if** *statement leads to a* **goto** *label, and there are at most k different labels. It follows that if S is not in an infinite loop, it cannot run more than $k.k = k^2$ steps.*

The above theorem can be generalized to show the decidability of reachability for multi-variable string systems as long as in a conditional branch statement we allow equality check between only two specific variables, i.e., no other variables can be compared for equality.

In contrast to Theorem 2, we can show that the halting problem is undecidable for nondeterministic 2-variable string systems with constants over the alphabet $\{a, b\}$, by a reduction from the Post Correspondence Problem (PCP) which is undecidable.

Theorem 3 *The halting problem for $NBS(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ is undecidable.*

Proof 3 *Given an instance (C, D) of PCP, where $C = (c_1, ..., c_n)$ and $D = (d_1, ..., d_n)$, define constant strings $\{c_1, ..., c_n, d_1, ..., d_n\}$, where c_i, d_i are non-null strings over alphabet $\{a, b\}$, we construct a string system S in $NBS(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ as follows:*

0: goto 1 or 2 or ... or n

1: $X_1 := X_1c_1$ and $X_2 := X_2d_1$; goto 0 or n+1

2: $X_1 := X_1c_2$ and $X_2 := X_2d_2$; goto 0 or n+1

...

n: $X_1 := X_1c_n$ and $X_2 := X_2d_n$; goto 0 or n+1

n+1: if $X_1 = X_2$ *goto n+2 else go to 1*

n+2: halt

Clearly, there is a computation that will reach the halt instruction if and only if the PCP instance (C, D) *has a solution. The theorem follows.*

Theorem 2 demonstrates that there are non-trivial string analysis problems that are decidable. Theorems 1 and 3, on the other hand, show that the string analysis problem can be undecidable even when we restrict a deterministic string system to three variables or a non-deterministic string system to two variables. Since the general string analysis problem is undecidable, it is necessary to develop conservative approximation techniques for verification of string systems. In this dissertation, we present several symbolic verification techniques that conservatively approximate the reachable states of a string system. We can also analyze (extended) string systems that have complex string operations, e.g., replacement, prefix and suffix. We discuss these operations in the next Chapter.

Chapter 3

Automata-based String Analysis

Unsanitized string variables are a common cause of security vulnerabilities in Web applications. In typical interactive Web applications, user-provided input strings are often used to query back-end databases. If the user input is not properly checked and filtered (i.e., sanitized), the input strings that contain hidden destructive commands can be sent to back-end databases and cause damage. Using the string analysis techniques proposed in this work, it is possible to automatically verify that a string variable is properly sanitized at a program point, showing that such attacks are not possible.

We present a string analysis technique that computes an over approximation of possible values that a string expression can take at a given program point. We use a deterministic finite automaton (DFA) to represent the set of values string expressions can take. At each program point, each string variable is associated with a DFA. The lan-

guage accepted by the DFA corresponds to the values that the corresponding string variable can take at that program point.

3.1 String Manipulation Operations

Most of the string manipulation operations performed in real-world applications can be reduced to the following four operations:

- *assignment*: assigns the current string value of a variable to another variable (the assignment operator in PHP is "=");
- *concatenation*: concatenates two string variables and/or constants (the concatenation operation in PHP is "`.`");
- *replacement*: replaces the parts of a string that match the given pattern with the given replacement string (there are several string replacement functions in PHP such as `htmlspecialchars`, `tolower`, `toupper`, `str_replace`, `trim`, `preg_replace` and `ereg_replace`, and they can all be converted to this form).
- *restriction*: restricts the value of a string variable based on a branch condition.

Automata Operations: In order to implement the automata-based string analysis, we implement the following operations:

- CONSTRUCT(regexp e): Returns a DFA M, $L(M) = \{w \mid w \in L(e)\}$.
- CLOSURE(DFA M_1): Returns a DFA M, $L(M) = \{w_1 w_2 \dots w_k \mid k > 0, \forall i, 1 \leq i \leq k, w_i \in L(M_1)\}$.
- CONCAT(DFA M_1, DFA M_2): Returns a DFA M, $L(M) = \{w_1 w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.
- REPLACE(DFA M_1, DFA M_2, DFA M_3): Returns a DFA M, $L(M) = \{w_1 c_1 w_2 c_2 \dots w_k c_k w_{k+1} \mid k > 0, w_1 x_1 w_2 x_2 \dots w_k x_k w_{k+1} \in L(M_1), \forall_i,\ x_i \in L(M_2),\ w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.
- UNION(DFA M_1, DFA M_2) : Returns a DFA M, $L(M) = L(M_1) \cup L(M2)$.
- INTERSECT(DFA M_1, DFA M_2): Returns a DFA M, $L(M) = L(M_1) \cap L(M_2)$.
- WIDENING(DFA M_1, DFA M_2): Returns a DFA M, $L(M) \supseteq L(M_1) \cup L(M_2)$.
- EQUCHECK(DFA M_1, DFA M_2): Checks whether $L(M_1) = L(M_2)$.
- EMPCHECK(DFA M): Checks whether $L(M) = \emptyset$.
- EMPTY(): Returns a DFA which does not accept any string.
- UNIVERSAL(): Returns a DFA which accepts all the strings.

Symbolic Automata Representation: We use the DFA library of MONA [5] to implement the string operations listed above. In MONA, transition relations of DFA are symbolically represented using Multi-terminal Binary Decision Diagrams (MBDDs). A MBDD is a BDD with multiple roots and multiple leaves. In MONA's DFA representation, each state of the DFA is a root and points to a BDD node, and each leaf value is a state of the DFA. Given the current state and a symbol $a \in B^k$, where B^k is alphabet of bit vectors of length k, one can find the next state by following the BDD nodes according to the bit vector of a from the BDD node pointed by the current state. We use a 7-bit vector, i.e., B^7, as our alphabet representing the binary value of ASCII symbols, e.g., for the ASCII symbol 'a', the ASCII code is 97 which is represented as '1100001' in our encoding.

The MONA DFA library provides efficient implementations of standard automata operations. These operations include product, project and determinize, and minimize [5]. The product operation takes the Cartesian product of the states of the two input automata. We use the product operation to implement the intersection and union operations. The project and determinize operation, denoted as $\textsc{Project}(M, i)$, where $1 \leq i \leq k$, converts a DFA M recognizing a language L over the alphabet B^k, to a DFA M' recognizing a language L' over the alphabet 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 al-

phabet. The process consists of removing the i^{th} track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction.

3.2 String Operations on Automata

In this section, we describe how to implement the closure, concatenate and replace operations. Since we use MBDD representation for DFA, we are not able to introduce ϵ-transitions. Instead, to avoid the non-determinism introduced by these operations, we extend the alphabet by adding extra bits, and then use projection to map the resulting DFA to the original alphabet.

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$ where Q is a finite set of states, q_0 is the initial state, $\Sigma \subseteq B^k$ is the alphabet, where each symbol is encoded as a k-bit string. $F : Q \rightarrow \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \rightarrow Q$ is the transition relation. A state q of M is a *sink* state if $\forall\alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a $sink$ state. In the constructions below, we also ignore the transitions that lead to a sink state.

Given $\alpha \in B^k$, we use $\alpha 0$ or $\alpha 1 \in B^{k+1}$ to denote the bit string that is α appended with '0' or '1'. For instance, if α is '110011' then $\alpha 0$ is '1100110'.

Closure: The DFA M is a closure-DFA of the DFA M_1, if $L(M) = \{\ w_1 w_2 \ldots w_k \mid \exists k > 0, \forall 1 \leq i \leq k, w_i \in L(M_1)\}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, its closure M can be constructed by first constructing an intermediate DFA $M' = \langle Q_1, q_{10}, \Sigma', \delta', F_1 \rangle$ as:

- $\Sigma' = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta'(q, \alpha 0) = q'$, if $\delta_1(q, \alpha) = q'$.
- $\forall q \in Q_1, \delta'(q, \alpha 1) = q'$, if $F_1(q) = +$ and $\delta_1(q_{10}, \alpha) = q'$.

Then, $M = \textsc{Project}(M', k+1)$ is the closure of M_1.

Since M_1 is a DFA, the project operation requires the subset construction only when there exists $q \in Q_1$, $F_1(q) = +$, and $\exists \alpha, q', q'', \alpha \in \Sigma, q', q'' \in Q_1, q' \neq q'', \delta_1(q, \alpha) = q', \delta_1(q_{10}, \alpha) = q''$.

Concatenation: The DFA M is a concatenation-DFA of the DFA M_1 and M_2, if $L(M) = \{w_1 w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$ and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, the concatenation-DFA M can be constructed as follows. Without loss of generality, we assume that $Q_1 \cap Q_2$ is empty. We first construct an intermediate DFA $M' = \langle Q', q_{10}, \Sigma', \delta', F' \rangle$, where

- $Q' = Q_1 \cup Q_2$

- $\Sigma^{'} = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta^{'}(q, \alpha 0) = q'$, if $\delta_1(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta^{'}(q, \alpha 0) = q'$, if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_1, \delta^{'}(q, \alpha 1) = q'$, if $F_1(q) = +$ and $\exists q' \in Q_2, \delta_2(q_{20}, \alpha) = q'$
- $\forall q \in Q_1, F^{'}(q) = +$, if $F_1(q) = +$ and $F_2(q_{20}) = +$; $F^{'}(q) = -$, o.w.
- $\forall q \in Q_2, F^{'}(q) = F_2(q)$.

Then, $M = \text{PROJECT}(M^{'}, k+1)$. Again, since both M_1 and M_2 are DFA, the subset construction happens only when there exists $q \in Q_1, F_1(q) = +$ such that $\exists \alpha, q', q^{''}, \alpha \in \Sigma, q' \in Q_1, q^{''} \in Q_2, \delta_1(q, \alpha) = q', \delta_2(q_{20}, \alpha) = q^{''}$.

Replacement: A DFA M is a replaced-DFA of a DFA tuple (M_1, M_2, M_3), if and only if $L(M) = \{w \mid k > 0, w_1 x_1 w_2 \dots w_k x_k w_{k+1} \in L(M_1), w = w_1 c_1 w_2 ... w_k c_k w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k+1, w_i \notin \{w_1' x' w_2' \mid x' \in L(M_2), w_1', w_2' \in \Sigma^*\}\}$.

This definition requires that all occurrences of matching sub-strings in a word are replaced. The intuition of the implementation of this language-based replacement is that we first insert marks into automata, then identify matching sub-strings by intersection of automata, and finally construct the final automaton by replacing these matching sub-strings.

We consider a new alphabet $\bar{\Sigma} = \{\bar{\alpha} | \alpha \in \Sigma\}$, and let $\bar{x}$ denote a new string in which we add bar to each character in x. Assume that M_1, M_2, M_3 have the same alphabet Σ, where $\sharp_1, \sharp_2 \notin \Sigma$, and $\forall \alpha \in \Sigma, \bar{\alpha} \notin \Sigma$. We define $M_1^{'}$, $M_2{'}$ and M as follows, and claim that M accepts the same language as the replaced-DFA of the tuple (M_1, M_2, M_3).

- $M_1^{'}$, where $L(M_1^{'}) = \{w' \mid k > 0, w = w_1 x_1 w_2 \dots w_k x_k w_{k+1} \in L(M_1), w' = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}\}$.

- $M_2^{'}$, where $L(M_2^{'}) = \{w' \mid k > 0, w' = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), \forall 1 \leq i \leq k+1, w_i \in L(M_h)\}$, where $L(M_h)$ is the set of strings which do not contain any substring in $L(M_2)$. The language $L(M_h)$ is defined as the complement set of $\{w_1 x w_2 \mid x \in L(M_2), w_1, w_2 \in \Sigma^*\}$.

- M, where $L(M) = \{w \mid k > 0, w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1} \in L(M_1^{'}) \cap L(M_2^{'}), w = w_1 c_1 w_2 \dots w_k c_k w_{k+1}, \forall 1 \leq i \leq k, c_i \in L(M_3)\}$.

To distinguish the original and bar alphabets, we append an extra bit to α so that α is $\alpha 0$ and $\bar{\alpha}$ is $\alpha 1$. Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, and $M_3 = \langle Q_3, q_{30}, \Sigma, \delta_3, F_3 \rangle$, the process to construct a replaced-DFA M can be decoupled into the following steps:

1. Construct $M_1{'}$ from M_1,

2. Construct $M_2^{'}$ from M_2,

3. Generate $M^{'}$ as the intersection of $M_1^{'}$ and $M_2^{'}$,

4. Construct $M^{''}$ from $M^{'}$ where the strings that appear between $\sharp_1$ and $\sharp_2$ are replaced by words in $L(M_3)$, and

5. Generate M from $M^{''}$ by projection.

We formally describe the implementation of these steps below. As a running example, we use $L(M_1) = \{baab\}$, $L(M_2) = a^+$ (M_2 accepts the language $\{a, aa, aaa, \ldots\}$) and $L(M_3) = \{c\}$ or $L(M_3) = \{\epsilon\}$. Let $|M|$ denote the number of states of M. An upper bound for each intermediate automaton before projection and minimization is also described.

Step 1: $M_1^{'} = \langle Q_1^{'}, q_{10}, \Sigma^{'}, \delta_1^{'}, F_1^{'} \rangle$ is constructed from M_1, where

- $Q_1^{'} = Q_1 \cup Q_{1'}$, $Q_{1'}$ is the duplicate of Q_1. For all $q \in Q_1$, there is a one to one mapping $q' \in Q_{1'}$.

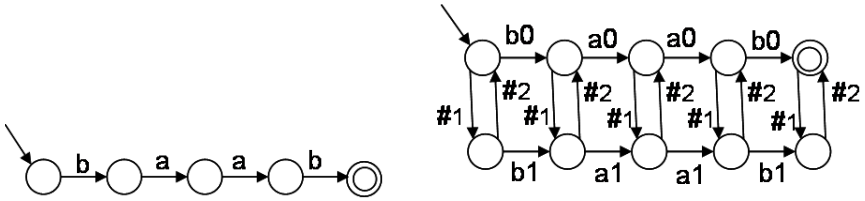
- $\Sigma^{'} = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\} \cup \{\sharp_1, \sharp_2\}$

- $\delta_1^{'}(q_1, \alpha 0) = q_2$ and $\delta_1^{'}(q_{1'}, \alpha 1) = q_{2'}$, if $\delta_1(q_1, \alpha) = q_2$

- $\forall q_1 \in Q_1, \delta_1^{'}(q_1, \sharp_1) = q_{1'}$ and $\delta_1^{'}(q_{1'}, \sharp_2) = q_1$

- $\forall q \in Q_1, F_1^{'}(q) = F_1(q)$ and $\forall q \in Q_{1'}, F_1^{'}(q) = 0$.

An example for constructing $M_1^{'}$ from M_1, where $L(M_1) = \{baab\}$, is given in Fig 3.1. $|M_1^{'}|$ is bounded by $2|M_1|$.



(a) $M_1 : \{baab\}$ (b) $M_1^{'}$

Figure 3.1: Constructing $M_1^{'}$ from M_1

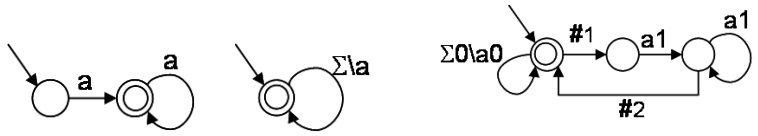
Step 2: To construct $M_2^{'}$, we first construct M_h which accepts the complement set of $\{w_1 x w_2 \mid w_1, w_2 \in \Sigma^*, x \in L(M_2)\}$. For instance, as shown in Fig 3.2(b), for $L(M_2) = a^+$, M_h is the DFA that accepts $(\Sigma \setminus \{a\})^*$. Let M_* be the DFA accepting Σ^*. M_h can be constructed by NEGATE(CONCAT(CONCAT(M_*, M_2), M_*)). We obtain the DFA in Fig 3.2(b) by applying this construction with minimization.

Assume $M_h = \langle Q_h, q_{h0}, \Sigma, \delta_h, F_h \rangle$, and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$. $M_2^{'} = \langle Q_2^{'}, q_{h0}, \Sigma^{'}, \delta_2^{'}, F_2^{'} \rangle$ can then be constructed as:

- $Q_2^{'} = Q_h \cup Q_2$
- $\Sigma^{'} = \{\alpha 0 \mid \forall \alpha \in \Sigma\} \cup \{\alpha 1 \mid \forall \alpha \in \Sigma\} \cup \{\sharp_1, \sharp_2\}$
- $\forall q, q' \in Q_h, \delta_2^{'}(q, \alpha 0) = q'$, if $\delta_h(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta_2^{'}(q, \alpha 1) = q'$, if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_h, \delta_2^{'}(q, \sharp_1) = q_{20}$ if $F_h(q) = +$
- $\forall q \in Q_2, \delta_2^{'}(q, \sharp_2) = q_{h0}$ if $F_2(q) = +$

- $\forall q \in Q_h, F_2^{'}(q) = F_h(q)$ and $\forall q \in Q_2, F_2^{'}(q) = -$.

The corresponding $M_2^{'}$ for our example is shown in Fig 3.2(c). $|M_2^{'}|$ is bounded by $|M_h| + |M_2|$, where $|M_h|$ is bounded by $|M_2| + 2$.



(a) M_2 (b) M_h (c) $M_2^{'}$

Figure 3.2: Constructing $M_2^{'}$ from M_2 and M_h

Step 3: $M^{'} = \langle Q^{'}, q_0^{'}, \Sigma^{'}, \delta^{'}, F^{'} \rangle$ is generated as the intersection of $M_1^{'}$ and $M_2^{'}$ based on production. The example $M^{'}$ is shown in Fig 3.3 (a). $|M^{'}|$ is bounded by $|M_1^{'}| \times |M_2^{'}|$.

Step 4: Before we construct $M^{''}$ from $M^{'}$, we first introduce a function $reach$: $Q^{'} \rightarrow 2^{Q^{'}}$, which maps a state to all its $\sharp$-reachable states in $M^{'}$. We say q' is $\sharp$-reachable from q if there exists a sequence $q, q_1, \ldots, q_n, q'$ so that (1) $n \geq 1$, (2) $\delta^{'}(q, \sharp_1) = q_1$, (3) $\delta^{'}(q_n, \sharp_2) = q'$, and (4) $\forall 0 < i < n, \delta^{'}(q_i, x) = q_{i+1}$, where $x \in \{\alpha 1 \mid \forall \alpha \in \Sigma\}$. For instance, in Fig 3.3 (a), one can find that $reach(i) = \{j, k\}$ and $reach(j) = \{k\}$. Intuitively, one can think that each pair (q, q'), where $q' \in reach(q)$, identifies a word in $L(M_2)$.

Our goal is, for each $q' \in reach(q)$, inserting paths between q and q' that recognize all words in $L(M_3)$. If there exist $q', q^{''} \in reach(q)$ and $q' \neq q^{''}$, this insertion

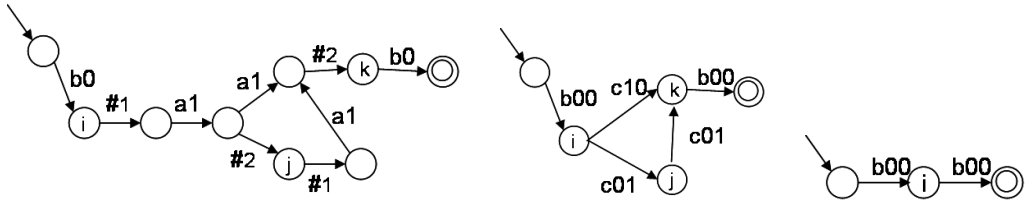
will cause nondeterminism. To tackle this problem, as we did in the construction of closure and concatenation, we add extra bits to the alphabet and later project them away. Assume n is the maximum size of $reach(q)$ for all $q \in Q^{'}$. We need at most $\lceil log(n+1) \rceil$ bits to be added to the alphabet so that the construction can result in a DFA. Let $P = \{q \mid q \in Q^{'}, reach(q) > 0\}$. Let $m = \lceil log(n+1) \rceil$, where n is the maximum size of $reach(q)$ for all $q \in P$. Let m_q be an m-bit string. For $\alpha \in B^k$, $\alpha m_q \in B^{k+m}$ is a string in which m_q is appended to α. Let m_0 be an m-bit string of 0s. We assume $\forall q, m_q \neq m_0$, and for any $q \in P$, $m_q^{'} \neq m_q^{''}$ if $q^{'}, q^{''} \in reach(q)$.

The construction of $M^{''}$ depends on $L(M_3)$. We consider the following three cases: (1) M_3 only accepts single characters, i.e., $L(M_3) \subseteq \Sigma$, (2) M_3 only accepts words with more than one character, i.e., $L(M_3) \subseteq \Sigma^+ \setminus \Sigma$, (3) M_3 only accepts the empty string, i.e., $L(M_3) = \{\epsilon\}$.

Case 1: $\forall w \in L(M_3), |w| = 1$. $M^{''} = \langle Q^{'}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{'} \rangle$ is constructed as:

- $\Sigma^{''} \subseteq B^{k+m}$
- $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q^{'}$, if $\delta^{'}(q, \alpha 0) = q^{'}$
- $\forall q \in P, \forall q^{'} \in reach(p), \forall \alpha \in L(M_3), \delta^{''}(q, \alpha m_{q^{'}}) = q^{'}$.

In Fig 3.3(a), $P = \{i, j\}$, $reach(i) = \{j, k\}$ and $reach(j) = k$. Let $L(M_3) = \{c\}$. $M^{''}$ of our example is shown in Fig 3.3(b). Each symbol is appended with two extra bits, e.g., $\delta(i, c01) = j$ and $\delta(i, c10) = k$. $|M^{''}|$ is bounded by $|M^{'}|$.



(a) $M^{'}$ (b) $M^{''}$: case 1 (c) $M^{''}$: case 3

Figure 3.3: Constructing $M^{''}$ from $M^{'}$. $M^{'}$ is the Intersection of $M_1{'}$ and $M_2^{'}$

Case 2: $\forall w \in L(M_3), |w| \geq 2$. For each $p \in P$, we construct a copy of M_3 as $M_p = \langle Q_p, q_{p0}, \Sigma, \delta_p, F_p \rangle$. $M^{''}$ is constructed by inserting M_p between p and $reach(p)$.

$M^{''} = \langle Q^{''}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{''} \rangle$, where

- $Q^{''} = Q^{'} \bigcup_{p \in P} Q_p$
- $\Sigma^{''} \subseteq B^{k+m}$
- $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha 0) = q'$
- $\forall p \in P, \forall q \in Q_p, \delta^{''}(q, \alpha m_0) = q'$, if $\delta_p(q, \alpha) = q'$.
- $\forall p \in P, \delta^{''}(p, \alpha m_q) = q$, if $\delta_p(q_{p0}, \alpha) = q$.
- $\forall p \in P, \forall q \in reach(p), \delta^{''}(q', \alpha m_0) = q$, if $\delta_p(q', \alpha) = q^{''}$ and $F_p(q^{''}) = +$.
- $\forall q \in Q^{'}, F^{''}(q) = F^{'}(q)$
- $\forall p \in P, q \in Q_p, F^{''}(q) = -$.

In this case, $|M^{''}|$ is bounded by $|M^{'}| + |M^{'}| \times |M^{'}| \times |M_3|$.

Case 3: $\forall w \in L(M_3), |w| = 0$. We consider this case as *deletion*. Before we start the construction, it is worth to know that for deletion, one may change the argument M_2 to N, where $L(N) = L(M_2)+$ (Kleene plus closure) , and get the same result. We specify this property as follows.

Property 1 Let M=REPLACE(M_1, M_2, M_3), and $M^{'}$=REPLACE(M_1, N, M_3), where $L(N) = L(M_2)+$. $L(M) = L(M^{'})$ if $L(M_3) = \{\epsilon\}$.

The correctness comes from the fact that, by construction, if there exists $w \in L(N)$, then there exists $k > 0$, $w = w_1 w_2 \dots w_k$, where $\forall 1 \leq i \leq k, w_i \in L(M_2)$. Since w or any w_i will be deleted after the replacement, using N instead of M_2 yields the same result.

Note that the $\sharp$-reachable states of M' using N is actually the set of reachable closure of the $\sharp$-reachable states of M' using M_2. This facilitates our construction by taking all deleted pairs into account in one step. In the following construction, without loss of the generality, we assume that the matching strings are accepted by N. N can be constructed from the original M_2 by our closure operation.

$M^{''}$ can then be constructed as $\langle Q^{'}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{''} \rangle$, where

- $\Sigma^{''} \subseteq B^{k+m}$
- $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha 0) = q'$
- $\forall p \in P, \forall q \in reach(p), \delta^{''}(p, \alpha m_{q'}) = q'$, if $\delta^{'}(q, \alpha 0) = q'$.

- $\forall p \in P, F^{''}(p) = +$, if $\exists q \in reach(p), F^{'}(q) = +$.
- $F^{''}(q) = F^{'}(q)$, o.w.

Let $L(M_3) = \{\epsilon\}$. The result of $M^{''}$ is shown in Fig 3.3(c). Note that if $M_2 = \{a\}$, we would get the same result. $|M^{''}|$ is bounded by $|M^{'}|$.

Finally, consider M_3 as a general DFA. REPLACE(M_1, M_2, M_3) can be constructed as the union of the results of the following three operations:

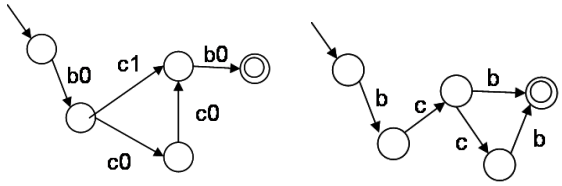
- REPLACE(M_1, M_2, M_{3_1}), where $L(M_{3_1}) = L(M_3) \cap \Sigma$
- REPLACE(M_1, M_2, M_{3_2}), where $L(M_{3_2}) = L(M_3) \cap \ \Sigma^+ \setminus \Sigma$
- REPLACE(M_1, M_2, M_{3_3}), where $L(M_{3_3}) = L(M_3) \cap \{\epsilon\}$

Our replacement operation is defined in a general case in terms of M_3. For all replacement statements in PHP programs, such as `str_replace`, `preg_replace`, and `ereg_replace`, $L(M_3)$ is a constant string. In our implementation, we determine which type of construction to apply based on the length of this string.

Step 5: Finally, we get M over Σ by iteratively projecting away the extra bits. The subset construction is only applied when needed.

The final DFA M =REPLACE(M_1, M_2, M_3), where $L(M_1) = \{baab\}$, $L(M_2) = a^+$, and $L(M_3) = \{c\}$, is shown in Fig 3.4. M accepts $\{bcb, bccb\}$.

Lemma 4 shows that a potential exponential blow-up of the number of states of the final DFA is inevitable in a replacement operation.



(a) $M_1^{''}$ (b) M

Figure 3.4: $M_1^{''}$ is PROJECT$(M^{''}, k+2)$, M is PROJECT$(M_1^{''}, k+1)$

Lemma 4 *For every $n \geq 1$, there exists a DFA M with O(n) states accepting a language $L \subseteq \{0,1\}^*\#\{0,1\}^*$ such that any DFA accepting the language L' obtained from L by replacing $\#$ with 1 requires O(2^n) states.*

Proof 4 *Let $n \geq 1$. Let $L = \{x\#y \mid x, y \in \{0,1\}^*, |y| = n-1\}$. Clearly, L can be accepted by a DFA M with O(n) states. Now $L' = \{x1y \mid x, y \in \{0,1\}^*, |y| = n-1\}$. Below we show that any DFA accepting L' requires O(2^n) states. Assume a DFA A accepting L'. Let w be any binary string of length n, i.e., $|w| = n$. Let $s(w)$ denote the state that A enters after processing w. The proof is based on the fact that for any w and w' s.t. $|w| = |w'| = n$ and $w \neq w'$, $s(w) \neq s(w')$. Since there are 2^n distinct strings of length n, there are 2^n distinct $s(w)'s$. Hence, A has at least 2^n states.*

In PHP programs, replacement operations such as `ereg_replace` can use different replacement semantics such as *longest match* or *first match*. Our replacement operation provides an over approximation of such more restricted replace semantics. For the example above, in the longest match semantics, M only accepts bcb, in which the longest match aa is replaced by c. In the first match semantics, M only accepts $bccb$, in

which two matches a and a are replaced with c. Both of these are included in the result obtained by our replacement operation. This over approximation works well for our benchmarks, and does not raise false alarms. Indeed, we have observed that most statements we encountered yield the same result in the first and longest match semantics, e.g.,`ereg_replace("<script *>","",$_GET["username"]);`, which are precisely modelled by our language-based replacement operation.

3.3 Pre-image Computation

In this section, we discuss how to compute the pre-images of string manipulation operations. We introduce three automata operations: PRECONCATPREFIX(M_x, M_z), PRECONCATSUFFIX(M_x, M_y), and PREREPLACE(M_x, M_m, M_r). These automata operations are used to perform our backward analysis in Chapter 4.

3.3.1 Concatenation

To compute the pre-image of *concatenation*, we first introduce *concatenation transducer* that specifies the relation among the values of the output (the concatenation-DFA of M_1 and M_2) and the two inputs: prefix (M_1) and suffix (M_2) of the concatenation operation.

A concatenation transducer is a DFA over the alphabet that consists of 3 tracks. The 3-track alphabet is defined as $\Sigma^3 = \Sigma \times (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\})$, where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ ($1 \leq i \leq 3$) to denote the i^{th} track of $w \in \Sigma^3$. All tracks are aligned. $w[1] \in \Sigma^*$, $w[2] \in \Sigma^*\lambda^*$ is left justified, and $w[3] \in \lambda^*\Sigma^*$ is right justified. We use $w'[1], w'[2] \in \Sigma^*$ to denote the λ-free prefix of $w[1]$ and the λ-free suffix of $w[2]$. We say w is accepted by a concatenation transducer M if $w[1] = w'[2].w'[3]$. Note that a concatenation transducer binds the values of different tracks character by character and hence is able to identify the prefix and suffix relations precisely.

Below we show two examples of concatenation transducers. Let α indicate any character in Σ. In Figure 3.5, the third track of M can be used to identify all suffixes of X that follow any string in $(ab)^+$. In Figure 3.6, the second track of M can be used to identify all prefixes of X that are followed by any string in $(ab)^+$.

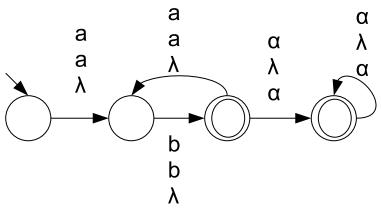


Figure 3.5: A Transducer M for $X = (ab)^+.Z$

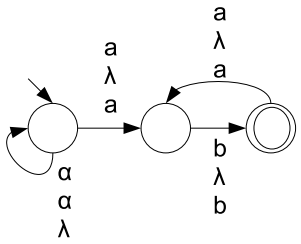


Figure 3.6: A Transducer M for $X = Y.(ab)^+$

In the following, we describe how to construct these transducers in general, how to remove λ, and how to compute the pre-images of a concatenation operation using concatenation transducers.

Prefix: We first consider how to compute the pre-image of the prefix, i.e., Y in $X := YZ$, given regular sets characterizing possible values of the output node X and the suffix node Z. Let $M_x = \langle Q_x, \Sigma, \delta_x, q_x 0, F_x \rangle$, $M_z = \langle Q_z, \Sigma, \delta_z, q_{z0}, F_z \rangle$ accept values of X and Z respectively. PRECONCATPREFIX(M_x, M_z) returns M_y.

- Extend M_x to a 3-track DFA M', so that M' accepts $\{w \mid w[1] \in L(M_x)\}$.
- Construct the concatenation transducer M that accepts $\{w \mid w[1] = w'[2].w'[3], w'[3] \in L(M_z)\}$. $M = \langle Q, \Sigma^3, \delta, q_0, F \rangle$, where:
  - $Q = \{q_0\} \cup Q_z$,
  - $\forall a \in \Sigma, \delta(q_0, (a, a, \lambda)) = q_0$,
  - $\forall a \in \Sigma, \delta(q_0, (a, \lambda, a)) = q'$ if $\delta_z(q_{z0}, a) = q'$.
  - $\forall q, q' \in Q_z, \forall a \in \Sigma, \delta(q, (a, \lambda, a)) = q'$ if $\delta_z(q, a) = q'$.

  - $F = \{q_0\} \cup F_z$ if $q_{z0} \in F_z$. $F = F_z$, otherwise.

- Intersect M' with M. The result accepts $\{w \mid w[1] = w'[2].w'[3], w[1] \in L(M_x), w'[3] \in L(M_z)\}$. We then project away the first and the third tracks. Let the result be $M'_y = \langle Q_y, \Sigma \cup \{\lambda\}, \delta, q'_{y0}, F'_y \rangle$.

- Remove λ tails if any. We construct $M_y = \langle Q_y, \Sigma, \delta_y, q_{y0}, F_y \rangle$ as below.

  - $\forall q, q' \in Q_y, \forall a \in \Sigma, \delta_y(q, a) = q'$ if $\delta'_y(q, a) = q'$.

  - $F_y = F'_y \cup F_\lambda$, where $F_\lambda = \{q \mid \exists q' \neq sink, \delta'_y(q, \lambda) = q'\}$.

Suffix: We next consider how to compute the pre-image of the suffix, i.e., Z in $X := YZ$, given regular sets characterizing possible values of X and the prefix node Y. Again, let $M_x = \langle Q_x, \Sigma, \delta_x, q_x 0, F_x \rangle$, $M_y = \langle Q_y, \Sigma, \delta_y, q_{y0}, F_y \rangle$ accept values of X and Y respectively. preConcatSuffix(M_x, M_y) returns M_z.

- Extend M_x to a 3-track DFA M', so that M' accepts $\{w \mid w[1] \in L(M_x)\}$.

- Construct the concatenation transducer M that accepts $\{w \mid w[1] = w'[2].w'[3], w'[2] \in L(M_y)\}$. $M = \langle Q, \Sigma^3, \delta, q_{y0}, F \rangle$, where:

  - $Q = Q_y \cup \{q_f\}$

  - $\forall q, q' \in Q_y, \forall a \in \Sigma, \delta(q, (a, a, \lambda)) = q'$ if $\delta_y(q, a) = q'$.

  - $\forall q \in F_y, \forall a \in \Sigma, \delta(q, (a, \lambda, a)) = q_f$.

  - $\forall a \in \Sigma, \delta(q_f, (a, \lambda, a)) = q_f$.
  - $F = \{q_f\} \cup F_y$.

- Intersect M' with M. The result accepts $\{w \mid w[1] = w'[2].w'[3], w[1] \in L(M_x), w'[2] \in L(M_y)\}$. We then project away the first and the second tracks. Let the result be $M'_z = \langle Q'_z, \Sigma \cup \{\lambda\}, \delta'_z, q'_{z0}, F'_z \rangle$.

- Remove λ heads if any. This final step can be done by constructing $M_z = \langle Q_z, \Sigma, \delta_z, q_{z0}, F_z \rangle$ as below.

  - $Q_z = q_0 \cup Q'_z$.
  - $\forall q \in Q'_z, \forall a \in \Sigma, \delta_z(q, a) = q'$ if there exists $q' \in Q'_z, \delta'_z(q, a) = q'$.
  - $\forall q \in Q'_z, \forall a \in \Sigma, \delta_z(q_0, a) = q'$ if there exists $q', q'' \in Q'_z$, $\delta'_z(q'', \lambda) = q$ and $\delta'_z(q, a) = q'$.
  - $F_z = \{q_0\} \cup F'_z$, if $\exists q \in F'_z$ and there exists $q', q'' \in Q'_z$, so that $\delta'_z(q'', \lambda) = q$ and $\delta'_z(q, a) = q'$. $F_z = F'_z$, otherwise.

3.3.2 Replacement

Recall that a `replace` operation has three inputs: target, match, and replacement. We only consider the pre-image of the target given regular sets characterizing possible values of the output, the match, and the replacement. Precisely, let M_x = REPLACE(M_t,

M_m, M_r). We are interested in computing M_t, given $M_x, M_m,$ and M_r. An intuitive solution of PREREPLACE(M_x, M_m, M_r) is REPLACE(M_x, M_r, M_m). However, since not all matches of M_r that appear in M_x are due to the replacement operation, this may break the soundness of our approach. Consider a simple example. M_t, M_m and M_r accept $\{aab\}$, $\{b\}$, and $\{a\}$, respectively. M_x = REPLACE(M_t, M_m, M_r) accepts $\{aaa\}$. M'_t = REPLACE(M_x, M_r, M_m) accepts $\{bbb\}$. Since $\{bbb\}$ does not include $\{aab\}$, this intuitive approach is not sound. Instead, we conservatively model PREREPLACE(M_x, M_m, M_r) as REPLACE(M_x, M_r, $M_m \cup M_r$). The result is an over approximation of the pre-image of the target node. For the simple example, M'_t = REPLACE(M_x, M_r, $M_m \cup M_r$) accepts $(a|b)(a|b)(a|b)$, which includes all $L(M_t)$ such that REPLACE(M_t, M_m, M_r) accepts $\{aaa\}$.

Deletion REPLACE(M_t, M_m, M_r) performs deletion if M_r accepts the empty string. That is, it will delete all the matches in $L(M_t)$. In this case, to compute the pre-image of the target, we would not be able to find a match of M_r (an empty string in this case) to replace with M_m. In this case, REPLACE(M_x, M_r, $M_m \cup M_r$) will return M_x. To deal with deletion, we conservatively generate a DFA M_{insert} that accepts $L(M_m)$ to be repeated many times between any character of $L(M_x)$. Formally speaking, M_{insert} accepts $\{w_0c_0w_1c_1 \dots w_nc_nw_{n+1} \mid c_0c_1 \dots c_n \in L(M_x), \forall_i, w_i \in L^*(M_m)\}$, where $L^*(M_m)$ denotes the closure of $L(M_m)$. To construct $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, the basic idea is inserting M_m to each state of M_x. $|Q|$ is bounded by $|Q_m| \times |Q_x|$. Depend-

ing on M_m, we consider two cases to insert M_m. First, let $\sigma_m = L(M_m) \cap \Sigma$ be the set of accepted single characters. If $\sigma_m \neq \emptyset$, we insert a self loop for each $a \in \sigma_m$ for all $q \in Q_x$, i.e., $\forall q \in Q_x, a \in \sigma_m, \delta(q, a) = q$. Second, let $M'_m = \langle Q'_m, \Sigma, \delta'_m, q'_{m0}, F'_m \rangle$ accept $L(M_m) \backslash \sigma_m$. If $L(M'_m) \neq \emptyset$ (i.e., M_m accepts some words that are not single character), we insert M'_m for all $q \in Q_x$, which can be done by setting (1) $\delta(q, a) = q'$ if there exists $q' \in Q'_m, \delta'_m(q'_{m0}, a) = q'$, and (2) $\delta(q', a) = q$ if there exists $q', q'' \in Q'_m, \delta'_m(q', a) = q''$ and $q'' \in F'_m$.

In sum, PREREPLACE(M_x, M_m, M_r) returns:

- REPLACE(M_x, M_r, $M_m \cup M_r$) if M_r accepts non empty strings, and
- M_{insert} if M_r accepts an empty string.

3.4 Widening Automata

In this section, we describe a widening operator on automata that we use to accelerate the fixpoint computations in Chapter 4. This automata operation was originally proposed for arithmetic automata by Bartzis and Bultan [4].

Given two finite automata $M = \langle Q, q_0, \Sigma, \delta, F \rangle$ and $M' = \langle Q', q'_0, \Sigma, \delta', F' \rangle$, we first define the binary relation $\equiv_\nabla$ on $Q \cup Q'$ as follows. Given $q \in Q$ and $q' \in Q'$, we

say that $q \equiv_\nabla q'$ and $q' \equiv_\nabla q$ if and only if

$$\forall w \in \Sigma^*.\ F(\delta^*(q, w)) = + \Leftrightarrow\ F(\delta'^*(q', w)) = +. \tag{3.1}$$

$$\text{or} \quad q, q' \neq sink \wedge \exists w \in \Sigma^*.\ \delta^*(q_0, w) = q \wedge \delta'^*(q'_0, w) = q', \tag{3.2}$$

where $\delta^*(q, w)$ is defined as the state that M reaches after consuming w starting from state q. In other words, condition 3.1 states that $q \equiv_\nabla q'$ if $\forall w \in \Sigma^*$, w is accepted by M from q then w is accepted by M' from q', and vice versa. Condition 3.2 states that $q \equiv_\nabla q'$ if $\exists w \in \sigma$, M reaches state q and M' reaches state q' after consuming w from its initial state. For $q_1 \in Q$ and $q_2 \in Q$ we say that $q_1 \equiv_\nabla q_2$ if and only if

$$\exists q' \in Q'.\ q_1 \equiv_\nabla q' \wedge q_2 \equiv_\nabla q' \ \ \vee \ \ \exists q \in Q.\ q_1 \equiv_\nabla q \wedge q_2 \equiv_\nabla q \tag{3.3}$$

Similarly we can define $q'_1 \equiv_\nabla q'_2$ for $q'_1 \in Q'$ and $q'_2 \in Q'$.

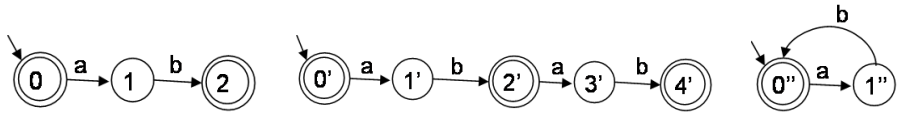
It can be seen that $\equiv_\nabla$ is an equivalence relation. Let C be the set of equivalence classes of $\equiv_\nabla$. We define $M \nabla M' = \langle Q'', q_0'', \Sigma, \delta'', F'' \rangle$ by:

$$
\begin{aligned}
Q'' &= C \\
q_0'' &= c \text{ s.t. } q_0 \in c \wedge q_0' \in c \\
\delta''(c_i, \sigma) &= c_j \text{ s.t. } (\forall q \in c_i \cap Q.\ \delta(q,\sigma) \in c_j \vee \delta(q,\sigma) = sink) \wedge \\
&\qquad (\forall q' \in c_i \cap Q'.\ \delta'(q',\sigma) \in c_j \vee \delta'(q',\sigma) = sink) \\
F''(c) &= + \text{ s.t. } \exists q \in F \cup F'.\ q \in c.\ \ F''(c) = - \text{ o.w.}
\end{aligned}
$$

In other words, the set of states of $M \nabla M'$ is the set C of equivalence classes of $\equiv_\nabla$. Transitions are defined from the transitions of M and M'. The initial state is the class containing the initial states q_0 and q_0'. The set of final states is the set of classes that contain some of the final states in F and F'. It can be shown that, given two automata M and M', $L(M) \cup L(M') \subseteq L(M \nabla M')$ [4].

In Fig 3.7, we give an example for the widening operation. $L(M) = \{\epsilon, ab\}$ and $L(M') = \{\epsilon, ab, abab\}$. The set of equivalence classes is $C = \{q_0'', q_1''\}$, where $q_0'' = \{q_0, q_0', q_2, q_2', q_4'\}$ and $q_1'' = \{q_1, q_1', q_3'\}$. $L(M \nabla M') = (ab)^*$.

As shown in our symbolic reachability analysis, we use this widening operator iteratively to compute an over-approximation of the least fixpoint that corresponds to the reachable values of string expressions. To simplify the discussion, let us assume a



(a) M (b) M' (c) $M \nabla M'$

Figure 3.7: Widening Automata

program with a single string variable represented with one automaton M. Let M_i represent the automaton computed at the i^{th} iteration and let I denote the initial value of the string variable. The fixpoint computation will compute a sequence M_0, M_1, ..., M_i, ..., where $M_0 = I$ and $M_i = M_{i-1} \cup post(M_{i-1})$ where the post-condition for different statements is computed accordingly (We will discuss in Chapter 4). We reach the least fixpoint M_j if at some iteration, $M_j = M_{j-1}$. Since we are dealing with an infinite state system, the computation may not converge. In the following, we use M_∞ to denote the least fixpoint.

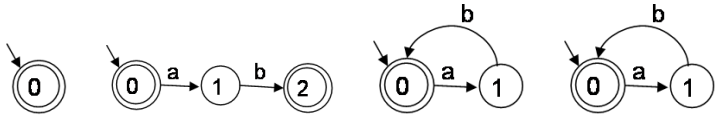
Given the widening operator, we actually compute a sequence M'_0, M'_1, ..., M'_i, ..., that over-approximates the fixpoint computation where M'_i is defined as: $M'_0 = M_0$, and for $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}))$. Let M'_∞ denote the least fixpoint of this approximate sequence. Then we have the following result [4]:

Definition 5 *$M_1 = \langle Q_1, q_{01}, \Sigma, \delta_1, F_1 \rangle$ is simulated by $M_2 = \langle Q_2, q_{02}, \Sigma, \delta_2, F_2 \rangle$ iff there exists a total function $f : Q_1 \setminus \{sink\} \rightarrow Q_2$ such that $\delta_1(q, \sigma) = sink$ or $f(\delta_1(q, \sigma)) = \delta_2(f(k), \sigma)$ for all $q \in Q_1 \setminus \{sink\}$ and $\sigma \in \Sigma$. Furthermore, $f(q_{01}) = q_{02}$ and for all $q \in F_1$, $f(q) \in F_2$.*

Definition 6 *$M = \langle Q, q0, \Sigma, \delta, F \rangle$ is state-disjoint iff there is no state $q \in Q$ such that there exist $\alpha \in \Sigma$ and $q', q'' \in Q$, $q' \neq q''$, and $\delta(q', \alpha) = q$ and $\delta(q'', \alpha) = q$.*

Theorem 7 *If (1) M_∞ exists, (2) M_∞ is a state-disjoint automaton, and (3) M_0 is simulated by M_∞, then (1) M'_∞ exists and (2) $M'_\infty = M_\infty$.*

Consider a simple example where we start from an empty string and simply concatenate a substring ab at each iteration. The exact sequence M_0, M_1, ..., M_i, ... will never converge to the least fixpoint, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$. However, M_∞ exists and $L(M_\infty) = (ab)^*$. In addition, M_∞ is a state-disjoint automaton, and M_0 is simulated by M_∞. Based on Theorem 7, these conditions imply that once the computation of the approximate sequence reaches the fixpoint, the fixpoint is equal to M_∞ and the analysis is precise. Computation of the approximate sequence is shown in Fig 3.8. $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}, R))$, where $post(M)$ returns an automaton that accepts $\{wab \mid w \in L(M)\}$. In this case, we reach the fixpoint at the 3^{rd} iteration and $M'_\infty = M_\infty = M'_3$.



(a) M'_0 (b) M'_1 (c) M'_2 (d) M'_3

Figure 3.8: An Approximation Sequence

A more general case that we commonly encounter in real programs is that we start from a set of initial strings (accepted by M_{init}), and concatenate an arbitrary but fixed

set of strings (accepted by M_{tail}) at each iteration. Based on Theorem 7 one can conclude that if the DFA M that accepts $L(M_{init})L(M_{tail})^*$ is state-disjoint, then our analysis via widening will reach the precise least fixpoint when it terminates.

Chapter 4

Symbolic Vulnerability Analysis

In this chapter, we describe how to perform symbolic reachability analyses on dependency graphs. We are interested in answering two questions:

1. Can a node have an attack string as its value (with respect to a given attack pattern)?

2. If it can, what values of its predecessors can generate the attack string?

If a node represents a sensitive sink in the program, the positive answer of the first question identifies a vulnerable point of the program. In this case, the answer of the second question characterizes the values of its predecessors to exploit this vulnerability. If a predecessor represents a user input, these values identify a vulnerability signature; i.e., a characterization of all possible inputs that can be used to exploit this vulnerability.

We first define the dependency graphs that specify how inputs flow to sensitive functions. We then describe how to perform forward and backward symbolic string analyses on dependency graphs. These analyses are based on automata constructions and operations in the previous chapter. Last, we propose a new summarization technique to tackle the interprocedural analysis.

4.1 Dependency Graph

A dependency graph specifies the data flow in the 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_j depends on the value of n_i. Each node $n \in N$ can be (1) a `normal` node including `input`, `constant`, `variable`, or (2) an `operation` node including `concat` and `replace`. An `input` node identifies the data from untrusted parties, e.g., an input from web forms. A `constant` node is associated with a constant value. Both nodes have no predecessors. A `concat` node n has two predecessors 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. A `replace` node has three predecessors labeled as the target node ($n.t$), the match node ($n.m$), and the replacement node ($n.r$). It performs the following operations for each value of $n.t$: (1) identifies

all the matches (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. We define the following: 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. If n is a `concat` node, $Pred(n) = \{n.p, n.s\}$. If n is a `replace` node, $Pred(n) = \{n.t, n.m, n.r\}$. For a dependency graph G, we also define $Root(G) = \{n \mid Pred(n) = \emptyset\}$ and $Leaf(G) = \{n \mid Succ(n) = \emptyset\}$.

4.2 Vulnerability Analysis

Our vulnerability analysis takes the following inputs: a dependency graph (denoted as G), a set of sink nodes (denoted as $Sink$), and an attack pattern (denoted as $Attk$). An attack pattern can be either taken from an attack pattern specification library or written by the web application developer. $Sink$ denotes the nodes that are associated with sensitive functions that might lead to vulnerabilities. $Attk$ is a regular expression represented as a DFA that accepts the set of attack strings .

Our vulnerability analysis is shown in Algorithm 1. The analysis consists of two phases. In the first phase, we perform a forward symbolic reachability analysis from root nodes to compute all possible values that each node can take (by calling forward analysis at line 3). We use this information to collect vulnerable program points, as well

Algorithm 1 VULANALYSIS($G, Sink, Attk$)

```
1:  Init(POST, PRE);
2:  set Vul := {};
3:  FWDANALYSIS(G, POST);
4:  for each n ∈ Sink do
5:      tmp: = POST[n] ∩ Attk;
6:      if L(tmp) ≠ ∅ then
7:          Vul := Vul ∪ {n};
8:          PRE[n] := tmp;
9:      end if
10: end for
11: if Vul ≠ ∅ then
12:     BWDANALYSIS(G, POST, PRE, Vul);
13:     for each input n do
14:         Report the vulnerability signature PRE[n];
15:     end for
16:     return "Vulnerable";
17: else
18:     return "Secure";
19: end if
```

as the reachable attack strings of those vulnerable program points (at line 4-10). If the program is vulnerable; i.e., there exists some vulnerable program points, we proceed to the second phase (by calling backward analysis at line 12). In the second phase, we perform a backward symbolic reachability analysis from the vulnerable program points to compute all possible values of their predecessors that will result in attack strings at these vulnerable program points.

Our analysis is an automata-based analysis. The set of string values is approximated as a regular language and represented symbolically as a DFA that accepts the language. To associate each node with its automata, we create two automata vectors *POST* and *PRE*. The size of both is bounded by $|N|$. *POST*$[n]$ is the DFA accepting all possible

values that node n can take. ***PRE***$[n]$ is the DFA accepting all possible values that node n can take to exploit the vulnerability. Initially, all these automata accept nothing; i.e., their language is empty. $Vul \subseteq Sink$ is the set of vulnerable program points and initially is set to an empty set.

At line 3, we first compute *POST* by calling the forward analysis. At line 4, for each node $n \in Sink$, we generate a DFA tmp by intersecting the attack pattern and the possible values of n. If $L(tmp)$ is not empty, we identify that n is a vulnerable program point and add it to Vul at line 7. In fact, tmp accepts the set of reachable attack strings at node n that can be used to exploit the vulnerability. Hence, we assign tmp to ***PRE***$[n]$ at line 8. If Vul is not empty, we compute *PRE* by calling our backward analysis at line 12. (We will discuss the backward analysis later.) Note that for $n \in Vul$, ***PRE***$[n]$ has been assigned. We report vulnerability signatures for each `input` node based on *PRE* at line 13-15. If Vul is an empty set, we report that the program is secure with respect to the attack pattern.

4.2.1 Forward Analysis

The forward symbolic reachability analysis is based on a standard work queue algorithm (Algorithm 2). We iteratively update the automata vector *POST* until a fixpoint is reached. At line 6, CONSTRUCT(n) returns a DFA that: (1) accepts arbitrary strings if n is an `input` node, (2) accepts an empty string if n is a `variable` node, or (3) accepts

Algorithm 2 FWDANALYSIS(G, *POST*)

```
1:  queue WQ := NULL;
2:  WQ.enqueue(Root(G));
3:  while WQ ≠ NULL do
4:      n := WQ.dequeue();
5:      if n ∈ Root(G) then
6:          tmp := CONSTRUCT(n);
7:      else if n is concat then
8:          tmp : = CONCAT(POST[n.p], POST[n.s]);
9:      else if n is replace then
10:         tmp : = REPLACE(POST[n.t], POST[n.m], POST[n.r]);
11:     else
12:         tmp : = ⋃_{n'∈Pred(n)} POST[n'];
13:     end if
14:     tmp := (tmp ∪ POST[n])∇POST[n];
15:     if tmp ⊈ POST[n] then
16:         POST[n] := tmp;
17:         WQ.enqueue(Succ(n));
18:     end if
19: end while
```

the constant value if n is a `constant` node. At line 8 and line 10, we incorporate two automata-based string manipulation operations [58]:

- CONCAT(DFA M_1, DFA M_2) returns a DFA M that accepts $\{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.

- REPLACE(DFA M_1, DFA M_2, DFA M_3) returns a DFA M that accepts $\{w_1c_1w_2c_2 \ldots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \ldots w_kx_kw_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.

At line 14, we incorporate the automata widening operator ∇ to accelerate the fixpoint computation [4]. Upon termination, *POST*$[n]$ records the DFA whose language includes

all possible values that n can take. This information is then passed to our backward analysis.

4.2.2 Backward Analysis

Backward analysis uses the results of the forward analysis. Particularly, we are interested in computing all possible values of each node n that can exploit the identified vulnerability. We need the pre-image computations on string manipulating functions for backward analysis. We use the following automata-based operations defined in Chapter 3 for pre-image computation.

- 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 $M = \text{REPLACE}(M_1, M_2, M_3)$.

The backward analysis is shown in Algorithm 3. For $n \in Vul$, ***PRE***$[n]$ is set to the intersection of ***POST***$[n]$ and $Attk$ before the backward analysis starts. The predecessors of $n \in Vul$ are the starting points of the backward analysis. Similar to the forward analysis, the computation is based on a standard work queue algorithm. We

Algorithm 3 BWDANALYSIS($G, POST, PRE, Vul$)

```
1: queue WQ = NULL;
2: for each n ∈ Vul do
3:    WQ.enqueue(Pred(n));
4: end for
5: while WQ ≠ NULL do
6:    n := WQ.dequeue();
7:    tmp' := NULL;
8:    for each n' ∈ Succ(n) do
9:       if n' is concat then
10:         if n is n'.l then
11:            tmp := preConcatPrefix(PRE[n'], POST[n'.r]);
12:         else
13:            tmp := preConcatSuffix(PRE[n'], POST[n'.l]);
14:         end if
15:      else if n' is replace then
16:         if n is n'.t then
17:            tmp := preReplace(PRE[n'], POST[n'.m], POST[n'.r]);
18:         end if
19:      else
20:         tmp := PRE[n'];
21:      end if
22:      tmp' := tmp' ∪ tmp;
23:   end for
24:   tmp' := tmp' ∩ POST[n];
25:   tmp' := (tmp' ∪ PRE[n])∇PRE[n];
26:   tmp' := tmp' ∩ POST[n];
27:   if tmp' ⊈ PRE[n] then
28:      PRE[n] := tmp';
29:      WQ.enqueue(Pred(n));
30:   end if
31: end while
```

first put the predecessors of $n \in Vul$ into the work queue as shown at line 2-4. We iteratively update the *P*RE array (by adding pre-images) until we reach a fixpoint. If the successor of n is an operation node, the pre-image (tmp) of n is computed by calling the defined automata-based functions. (line 11, 13, 17). Otherwise, the pre-image of n is directly derived from the successor of n (line 20). Note that *POST*$[n]$ records all possible values that n can take. We use this information during the pre-image computation by restricting the arguments of operations such as replace. We union the pre-images of n as tmp' at line 22. Since we are interested only in reachable values of n (i.e., *PRE*$[n] \subseteq$ *POST*$[n]$ by definition), we intersect tmp' with *POST*$[n]$ at line 24. Similar to the forward analysis, we widen the result at line 25 to accelerate the fixpoint computation. At line 26, we intersect tmp' with *POST*$[n]$ again to remove unreachable values (that might have been introduced due to widening) at node n. If tmp' accepts more values than *PRE*$[n]$, we update *PRE*$[n]$ at line 28 and add the predecessors of n to the working queue at line 29. Upon termination, *PRE*$[n]$ records the DFA that accepts all possible values of n that may exploit the identified vulnerability.

4.3 Inter-procedural Analysis

In this section, we propose a conservative summarization technique to proceed the inter-procedural analysis. During the forward fixpoint computation if we encounter

a call to a function that has not been summarized, we go to an internal phase of the analysis, which is summarization. Each function is summarized when needed, and once a function is summarized, the summary DFA is used to compute the return values at the call sites without going through the body of the function. During the summarization phase, (recursive) functions are summarized as unaligned multi-track DFAs that specify the relations among their inputs and return values. We first build (cyclic) dependency graphs to specify how the inputs flow to the return values among functions. Each node in the dependency graph is associated with an unaligned multi-track DFA that traces the relation among inputs and the value of that node. We iteratively compute post images of reachable relations and join the results until we reach a fixpoint. Upon termination, the summary is the union of the unaligned DFAs associated with the return nodes.

4.3.1 Summarization

In this section, we discuss how to compute function summaries. We assume parameter-passing with call-by-value semantics and we are able to handle recursion. Each function f is summarized as a multi-track DFA, denoted as M_f, that captures the relation among its input variables and return values. Return values of a function are represented with an auxiliary output track. Given a function f with n parameters, M_f is an $(n+1)$-track DFA, where n tracks represent the n input parameters and one track X_o is the output track representing the return values. We also use a special symbol $\lambda \notin \Sigma$ for padding.

The alphabet of an n-track DFA is a subset of $\Sigma \cup \{\lambda\} \times \ldots \times \Sigma \cup \{\lambda\}$ (n times). Once M_f has been computed, it is not necessary to reanalyze the body of f. Instead, one can intersect the values of input parameters with M_f to obtain the return values. Our approach consists of two steps: (1) Build a call dependency graph and (2) Generate its summary accordingly.

4.3.2 Call Dependency Graph

In addition to the dependency graph that we defined in Section 4.1, we add *call* nodes to specify function calls in a *call* dependency graph. Formally speaking, a call 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_j depends on the value of n_i. Each node $n \in N$ can be

- a `normal` node including `return`, `input`, `constant`, `variable`,
- an `operation` node including `concat` and `call`.

Similarly, a `return` node is a sink node (no successors) that corresponds to a return statement. An `input` node corresponds to a parameter of the function f, labeled as $f.p_i$, where i indicates the i^{th} parameter. A `constant` node is associated with a constant value. Both `input` and `constant` nodes have no predecessors. A `concat` node n has two predecessors 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. A `call` node is associated with a function $callee$. If $callee$ has m parameters, there are m predecessors of a `call` node as its arguments (labeled as $n.a_1, \ldots, n.a_m$).

Given a function f, the call dependency graph G_f specifies how the inputs flow to the return values in f. Assume that we want to compute the summary of a given function $main$. Let F denote the set of related functions that include $main$ and its $callees$ (including nested function calls). Our first step is generating the dependency graph for each $f \in F$, which is done by a bottom-up dependency analysis starting from the return statements.

Let the call dependency graph of f be $G_f = \langle N_f, E_f \rangle$. To simplify the description, we use $Input(G_f)$ to denote the set of its `input` nodes, $Call(G_f)$ to denote the set of its `call` nodes, and $Return(G_f)$ to denote the set of its `return` nodes. For each function f (callee), we use $Caller(f)$ to denote the set of `call` nodes that are associated with f.

Our second step is generating a composed dependency graph G_F from $\{G_f \mid f \in F\}$. $G_F = \langle N_F, E_F \rangle$ is constructed as follows:

- $N_F = \cup_{f \in F} N_f$.
- $E_F = E_n \cup E_i \cup E_r$, where
  - $E_n = \{(n, n') \mid f \in F, (n, n') \in E_f, n' \notin Call(G_f)\}$.

```
f(X)
begin
1: goto 2, 3;
2: X: = call f(X.a);
3: return X;
end
```

Figure 4.1: A Simple Function

- $E_i = \{(n.a_i, callee.p_i) \mid f \in F, n \in Call(G_f)\}$. $callee.p_i$ is the `input` node that identifies the i_{th} parameter of the function $callee$ associated with n.
- $E_r = \{(n, n') \mid f \in F, n \in Return(G_f), n' \in Caller(f)\}$.

Briefly, G_F connects the set of G_f by (1) redirecting the predecessors of `call` nodes to the `input` nodes of their callees, and (2) adding edges that direct `return` nodes of callees to the `call` nodes of their callers. For $n \in N_F$, $Succ(n) = \{n' \mid (n, n') \in E_F\}$ is the set of successors of n and $Pred(n) = \{n' \mid (n', n) \in E_F\}$ is the set of predecessors of n. We also define $Input(G_F) = \{n \mid Pred(n) = \emptyset\}$. Note that after composition, a `return` node may have successors and an `input` node may have predecessors.

Consider a simple example given in Figure 4.1. Function f has one parameter X, which non-deterministically returns its input (goto 3) or makes a self call (goto 2) by concatenation its input and the constant a. Let $F = \{f\}$. G_f and G_F are shown in Figure 4.2.

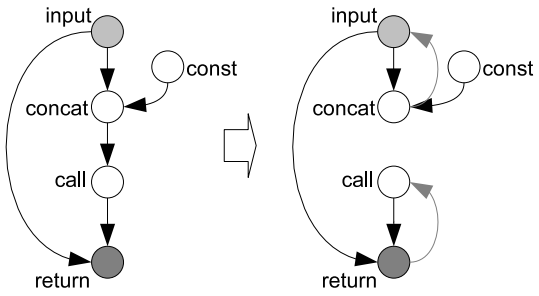


Figure 4.2: The Dependency Graphs: G_f and G_F

4.3.3 Generating Function Summaries

In this section, we describe how to compute a summary on G_F, given two sets of nodes In and Out. If we aim to summarize function f ($f \in F$), $In \subseteq N_f$ is the set of its `input` nodes and $Out \subseteq N_f$ is the set of its `return` nodes in G_f. Each $n \in In$ recognizes one input variable, denoted as X_n, and the summary of $\langle G_F, In, Out\rangle$ is an unaligned $(|In|+1)$-track DFA. The first $|In|$ tracks are labeled as X_n for each $n \in In$. The extra track, labeled as X_o, is used to record the output values.

The algorithm to generate the summary is shown in Algorithm 4. We use a DFA vector S to record the reachable summary at each node. We initialize S at line 1. Initially, for each $n \in In$, $S[n]$ is a 2-track (associated with X_n and X_o) DFA that accepts the identity relation on X_n and X_o. For each $n \in Input(G_F)\backslash In$, $S[n]$ is a 1-track (associated with X_o) DFA that accepts Σ^* if n is a `variable` node, or a constant value if n is a `constant` node. For the rest ($n \notin In$), $S[n]$ accepts an empty set. Similar to Algorithm 2, the algorithm is a standard work queue algorithm incorporating

the automata widening operator. We iteratively update the summary at each node until reaching a fixpoint.

Algorithm 4 GENERATESUMMARY(G_F, In, Out)

```
1: INIT(S, Input(G_F), In);
2: queue WQ := NULL;
3: for n ∈ In ∪ Input(G_F) do
4:    WQ.enqueue(Succ(n));
5: end for
6: while WQ ≠ NULL do
7:    n := WQ.dequeue();
8:    if n is concat then
9:       tmp : = CONCATSUMMARY(S[n.p], S[n.s]);
10:   else
11:      tmp : = ⋃_{n'∈Pred(n)} S[n'];
12:   end if
13:   tmp := (tmp ∪ S[n])∇S[n];
14:   if tmp ⊈ S[n] then
15:      S[n] := tmp;
16:      WQ.enqueue(Succ(n));
17:   end if
18: end while
19: return ⋃_{n∈Out} S[n];
```

Below we only consider one string operation: concatenate. We can extend our approach to other string operations, e.g., replacement, by using transducers [38]. Note that summaries may have tracks that are associated with different variables. Below we discuss how to compute CONCATSUMMARY(S_1, S_2) where S_1 represents the summary at the prefix node and S_2 represents the summary at the suffix node. Let $S_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ be a multi-track DFA whose tracks are associated with the set of input variables χ_1 and X_o where $\Sigma_1 = (\Sigma \cup \lambda)^{|\chi_1|} \times \Sigma$. Let $S_2 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2 \rangle$ be a multi-track DFA whose tracks are associated with the set of input variables χ_2 and X_o

where $\Sigma_2 = (\Sigma \cup \lambda)^{|\chi_2|} \times \Sigma$. We first extend S_1 and S_2 to the DFAs that have common tracks, so that both are associated with $\chi_1 \cup \chi_2$ and X_o.

The extension of S_1, denoted as S_1^λ, is $\langle Q_1, \Sigma_1^\lambda, \delta_1^\lambda, I_1, F_1 \rangle$, where

- $\Sigma_1^\lambda = (\Sigma \cup \lambda)^{|\chi_1|} \times \lambda^{|\chi_2 - \chi_1|} \times \Sigma$, and
- $\delta_1^\lambda(q, \alpha) = q'$ if $\delta_1(q, \beta) = q'$ and $\alpha[X] = \beta[X]$ if $X \in \chi_1 \cup X_o$, and $\alpha[X] = \lambda$, otherwise.

The extension of S_2, denoted as S_2^λ, is $\langle Q_2, \Sigma_2^\lambda, \delta_2^\lambda, I_2, F_2 \rangle$, where

- $\Sigma_2^\lambda = \lambda^{|\chi_1|} \times (\Sigma \cup \lambda)^{|\chi_2 - \chi_1|} \times \Sigma$, and
- $\delta_2^\lambda(q, \alpha) = q'$ if $\delta_2(q, \beta) = q'$ and $\alpha[X] = \lambda$ if $X \in \chi_1$, and $\alpha[X] = \beta[X]$, otherwise.

Intuitively, we extend S_1 (prefix) by allowing only λ in the added tracks, while we extend S_2 (suffix) by allowing only λ in both the added tracks and the common tracks that are also associated with S_1. CONCATSUMMARY(S_1, S_2) returns the $(|\chi_1 \cup \chi_2| + 1)$-track DFA that accepts the concatenation of S_1^λ and S_2^λ.

To deal with the union or widening operator on S_1 and S_2 that are associated with different variables, we extend both tracks to $\chi_1 \cup \chi_2$ and X_o by allowing arbitrary symbols in the added tracks (i.e., the value of an unspecified track is not restricted). We then perform union or widening on these extension DFAs. Finally, the summary of $\langle G_F, In, Out \rangle$ is the union of the DFAs that are associated with nodes in Out.

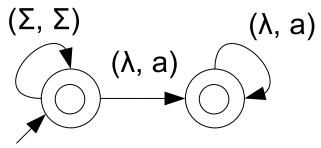


Figure 4.3: M_f: The Summary DFA

In sum, to summarize a specific function f, we first find the set of related functions F. The summary of f, denoted as M_f, is the result of GENERATESUMMARY(G_F, In, Out), where $In = \{n \mid n \in Input(G_f),$ where n is not a `constant` node$\}$, and $Out = \{n \mid n \in Return(G_f)\}$. The alphabet of M_f is $(\Sigma \cup \lambda)^{|In|} \times \Sigma$. Let $w[i]$ be the value of the i^{th} track of w. For any $w \in L(M_f)$, we have the following:

- $1 \leq i \leq |In|$, $w[i] \in \lambda^*\Sigma^*\lambda^*$, and
- $w[|In| + 1] \in \Sigma^*$.

Consider the previous simple example. The generated summary is shown in Figure 4.3. M_f is a 2-track DFA, where the first track is associated with its parameter X_{p_1}, and the second track is associated with X_o representing the return values. The edge (Σ, Σ) represents a set of identity edges; i.e., if $\delta(q, (\Sigma, \Sigma)) = q'$ then $\forall a \in \Sigma, \delta(q, (a, a)) = q'$. The summary DFA M_f precisely captures the relation $X_o = X_{p_1}.a^*$ between the input variable and the return values.

4.3.4 Composing Function Summaries

In this subsection, we describe how to use a function summary to compute the pre-image and the post-image of a function call. Note that the summary DFAs are multi-track DFAs. To bridge the gap between single-track DFAs (accepting the values of one variable) and multi-track DFAs (accepting the relations among many variables), we implemented two mapping functions between a single-track automaton and a multi-track automaton. Extract(M, i) takes a multi-track DFA M and returns a single-track DFA that accepts the values of the i^{th} track; i.e., $L(M') = \{w[i] \mid w \in L(M)\}$. Extend$(M, i, n)$ takes a single-track DFA M, an index i, and returns an n-track DFA that accepts $\{w \mid \hat{w}[i] \in L(M), \forall 1 \leq k \leq n, w[k] \in \lambda^*\Sigma^*\lambda^*\}$.

To compute the post-image of a function call, we (1) compute the single-track DFA that accepts the values of each argument, (2) extend the single-track automata to the multi-track automata accordingly, (3) intersect the multi-track automata with the summary automaton, (4) extract the output-track from the result of the intersection.

Consider our simple example and a function call $f(X)$. Let M_x be the DFA accepting the values of X. POST($f(X)$) returns Extract$(M', 2)$, where M' is $M_f \cap$ Extend$(M_x, 1, 2)$. Assuming $L(M_x) = \{b\}$, POST($f(X)$) returns M' such that $L(M') = ba^*$. In this case, using the summary to compute the post image is as precise as traversing the function body and iteratively adding the post image until hitting the fixpoint.

4.4 Experiments

To evaluate our approach, we first performed forward analysis and compared the results with another string analysis tool: Saner, developed by Balzarotti et al. [1] (we discuss this tool in related work). We then performed forward and backward analyses and reported the vulnerability signatures that we generated.

4.4.1 Forward Analysis

We experimented with our string analysis tool on a number of test cases extracted from a set of real-world, open source applications: `MyEasyMarket-4.1` (a shopping cart program), `PBLguestbook-1.32` (a guestbook application), `Aphpkb-0.71` (a knowledge base management system), `BloggIT-1.0` (a blog engine), and `proManager-0.72` (a project management system). We believe that these programs are representative of how web applications use regular expression based replacement functions to modify their input (in particular, in a security context, to perform input sanitization), and, thus, are good test cases for our technique. These vulnerable functions were identified and sanitized by Balzarotti et al. in [1, 2].

Table 4.1 shows the results of applying our string analysis tool to these programs. The first column of Table 4.1 identifies the application, the function that was analyzed and the line number for the vulnerable operation. A1: MyEasyMarket-4.1, trans.php

App.	Version	Result	Final DFA state(bdd)	Peak DFA state(bdd)	Time user+sys(sec)	Mem (kb)
A1	o	y	17(133)	17(148)	0.010+0.002	444
	m	n	17(132)	17(147)	0.009+0.001	451
A2	o	y	42(329)	42(376)	0.019+0.001	490
	m(5)	n	77(649)	77(725)	0.065+0.060	1532
	m(20)	n	182(1609)	182(1790)	1.082+0.015	12047
	m(100)	n	742(6729)	742(7470)	101.2+0.305	395921
	m(widen)	n	49(329)	42(376)	0.016+0.002	626
A3	o	y	842(6749)	842(7589)	2.57+0.061	13310
	m	n	774(6192)	740(6674)	1.221+0.007	8184
A4	o	y	27(219)	289(2637)	0.045+0.003	2436
	m	n	18(157)	1324(15435)	0.177+0.009	11388
A5	o	y	79(633)	79(710)	0.499+0.002	3569
	o	y	126(999)	126(1123)		
	o	y	138(1095)	138(1231)		
	m	n	79(637)	93(1026)	0.391+0.006	5820
	m	n	115(919)	127(1140)		
	m	n	127(1015)	220(2000)		
A6	o	y	387(3166)	2697(29907)	1.771+0.042	13900
	m	n	423(3470)	2697(29907)	2.091+0.051	19353

Table 4.1: The Forward Experimental Results of Stranger.

(218). A2: PBLguestbook-1.32, pblguestbook.php (1210). A3: PBLguestbook-1.32, pblguestbook.php (182). A4: Aphpkb-0.71, saa.php (87). A5: BloggIT 1.0, admin.php (23,25,27). A6: proManager-0.72, message.php (91). The other information about the table is listed as below: Version: o-original, m-modified. Result: y-the intersection of attack strings is not empty (vulnerable), n-the intersection of attack strings is empty (secure). Final DFA is the minimized DFA at the checked program point, and Peak DFA is the largest DFA observed during the fixpoint iteration. ”state” denotes the number of states of a DFA. ”bdd” denotes the number of bdd nodes that are used to symbolically encode the transitions of a DFA.

For each test case we analyzed the original version of the program (that contained the vulnerability) and a modified version which was modified with the intention of fixing the vulnerability. Our analysis is quite efficient and takes less than three seconds for all benchmarks. Since our string analysis tool is sound, it identifies the existing vulnerabilities correctly in each case. However, since our conservative approximations can lead to false positives, the fact that our tool identifies a possible vulnerability does not mean that it is guaranteed to be a vulnerability.

The impressive part of our results is that for all the modified program segments our approach is able to prove that the sanitization is correct. This indicates that the approximations we use work quite well in real-world applications.

We also experimented with Saner [1] to check these benchmarks. The results are shown in table 4.2. ”n” denotes the number of warnings raised by Saner; for ”type”, ”xss” denotes cross site scripting vulnerability, ”sql” denotes SQL injection vulnerability, and ”reg” denotes regular expression error. Compared to Table 4.1, our tool performs slightly better than Saner in terms of time. It is interesting to note that there are some conflicts on the verification results. Saner performs bounded verification and approximates the value of out of bound computation as arbitrary strings. This rough approximation raises a false alarm while checking the sanitized version of `PBLguestbook-1.32`(1210). While checking `BloggIT-1.0`, Saner, in the default configuration, assumes that data from the database are sanitized; while we assume that these data may be tainted and model them the same as data from users. Saner raises an error for the sanitization routine in `PBLguestbook-1.32`(182) since it does not support the syntax of the replace operator used in that routine.

4.4.2 Forward+Backward Analysis

We selected four vulnerable web applications: (1) `MyEasyMarket-4.1` (a shopping cart program), (2) `PBLguestbook-1.32` (a guestbook application), (3) `BloggIT-1.0` (a blog engine), and (4) `proManager-0.72` (a project management system) to generate their vulnerability signatures. In Table 4.3, we show some basic data about these dependency graphs: #sinks indicates the number of sensitive sinks,

Application	Version	n(type)	Time(sec)
A1	o	1(xss)	1.173
	m	0	1.139
A2	o	1(sql)	1.264
	m	1(sql)	1.665
A3	o	1(reg)	4.618
	m	1(reg)	4.331
A4	o	1(xss)	1.220
	m	0	1.622
A5	o	0	0.558
	m	0	0.559
A6	o	1(xss)	6.980
	m	0	7.201

Table 4.2: The Experimental Results of Saner.

#inputs indicates the number of `input` nodes. Since the application is identified as vulnerable by taint analysis, both values are at least one. #literals is the sum of the length of constant strings that are used in the graph. Note that these dependency graphs are built for sensitive sinks where unrelated parts have been shrunk. Hence, their sizes are much smaller than the original programs.

	n (type)	#nodes	#edges	#sinks	#inputs	#literals
1	1(xss)	21	20	1	1	51
2	1(sql)	41	44	1	2	99
3	1(xss)	32	31	1	1	142
4	3(xss)	119	117	3	3	450

Table 4.3: The Basic Data of Dependency Graphs

In our experiments, we used an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We use 8 bits to encode each character in

ASCII. The performance of our vulnerability analysis is shown in Table 4.4. The backward analysis dominates the execution time from 77% to 96%. Taking a closer look, Table 4.5 shows the frequency and execution time of each of the string manipulating functions. PRECONCAT (including prefix and suffix) consumes a large portion, particularly for (4) `proManager-0.72` that has a large size of constant literals involved. One reason is generating concatenation transducers during the computation. Note that the transducer has 3-tracks and uses 24 bits to encode its alphabet. On the other hand, our computation does not suffer exponential blow-up as expected for explicit DFA representation. This shows the advantage of using symbolic DFA representation (provided by the MONA DFA library), in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs).

	Total time(s)	Fwd time(s)	Bwd time(s)	Mem(kb)
1	0.569	0.093	0.474	2700
2	3.449	0.124	3.317	5728
3	1.087	0.248	0.836	18890
4	16.931	0.462	16.374	116097

Table 4.4: Total Performance

	CONCAT	REPLACE	PRECONCAT	PREREPLACE
	#operations/time(s)			
1	6/0.015	1/0.004	2/0.411	1/0.004
2	19/0.082	1/0.004	11/3.166	1/0.0
3	22/0.038	4/0.112	2/0.081	4/0.54
4	14/0.014	12/0.058	26/11.892	24/3.458

Table 4.5: String Function Performance

Finally, Table 4.6 shows the data about the DFAs that Stranger generated. Reachable Attack is the DFA that accepts all possible attack strings at the sink node. Vulnerability Signature is the DFA that accepts all possible malicious inputs that can exploit the vulnerability. We closely look at the vulnerability signature of (1) `MyEasyMarket-4.1`. The 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.

It is also worth noting that both vulnerability signatures of (2) `PBLguestbook-1.32` accept arbitrary strings. By manually tracing the program, we find that both inputs are concatenated to an SQL query string without proper sanitization. Since an input can be any string, the pre-image of one input is the prefix of Σ^* `OR '1'='1'`Σ^* that is equal to Σ^*, while the pre-image of another input is the suffix of Σ^* `OR '1'='1'`Σ^* that is also equal to Σ^*. This case shows a limitation in our approach. Since we do not model the relations among inputs, we can not specify the condition that one of the inputs must contain `OR '1'='1'`. In the next Chapter, we will propose a novel algorithm to generate *relational vulnerability signatures* to tackle this issue. We will also describe how

to generate effective patches for vulnerable web applications from these vulnerability signatures.

	Reachable Attack (Sink)		Vulnerability Signature (Input)	
	#states	#bdd nodes	#states	#bdd nodes
1	24	225	10	222
2	66	593	2	9
			2	9
3	29	267	92	983
4	131	1221	57	634
	136	1234	174	1854
	147	1333	174	1854

Table 4.6: Attack and Vulnerability Signatures

Chapter 5

Sanitization Synthesis

We use the presented automata-based static string analysis techniques to automatically generate sanitization statements for patching vulnerable Web applications. Given the vulnerability signatures, we construct sanitization statements that 1) check if a given input matches the vulnerability signature and 2) modify the input in a minimal way so that the modified input does not match the vulnerability signature. Our approach is capable of generating *relational* vulnerability signatures (and corresponding sanitization statements) for vulnerabilities that are due to more than one input.

Our approach works as follows. We start with a set of attack patterns (regular expressions) that characterize possible attacks (either taken from an attack pattern specification library or written by the web application developer). Given an attack pattern, our string analysis approach works in three phases:

Phase 1: Vulnerability Analysis: First, we perform the presented symbolic forward analysis on single-track automata to determine if the web application is vulnerable to attacks characterized by the given attack pattern and generate a characterization of the potential attack strings if the application is vulnerable.

Phase 2: Vulnerability Signature Generation: We then project these attack strings to user inputs by computing an over-approximation of all possible inputs that can generate those attack strings. This characterization of potentially harmful user inputs is called the *vulnerability signature* for a given attack pattern. We use two different vulnerability signature generation techniques: (1) for a vulnerability that is caused by a single user input, we apply the presented backward analysis on single-track automata to generate the corresponding vulnerability signature; (2) for a vulnerability that is caused by multiple user inputs, we propose a novel forward analysis on multi-track automata to generate the relational vulnerability signature.

Phase 3: Sanitization Generation: Once we have the vulnerability signature, we automatically synthesize patches that eliminate the vulnerability. We use two strategies for patching:

- *Match-and-block:* We insert match statements to vulnerable web applications and halt the execution when an input that matches a vulnerability signature is detected.

- *Match-and-sanitize:* We insert both match and replace statements to vulnerable web applications. When an input that matches a vulnerability signature is detected, instead of halting the execution, the replace statement is executed. The replace statement deletes a small set of characters from the input such that the modified string no longer matches the vulnerability signature.

We use two different techniques for vulnerability signature generation. In the first one, we adopt the vulnerability analysis presented in Chapter 4, where we start with the DFA that represents the intersection of the the forward symbolic reachability analysis at the sink and the attack pattern. Then we use a backward symbolic reachability analysis to compute an over-approximation of all possible inputs that can generate those attack strings. The result is a DFA that characterizes the dangerous user inputs, i.e., the vulnerability signature.

However, this approach is not effective for vulnerabilities that are due to more than one input. For example, if an attack string is generated by concatenating two input strings, it may not be possible to prevent the attack by blocking only one of the inputs. Since our automata-based vulnerability signature generation technique is sound, in such cases the generated vulnerability signature will include all possible input strings, meaning that any string coming from one input can lead to an attack if it is concatenated with a suitably constructed string coming from another input. Using such a vulnerability signature for automated patch generation would mean blocking or erasing all the user

input, which would make the web application unusable. However, if we do an analysis that keeps track of the relationships among different string variables, then we may be able to block only the combinations of input strings that lead to an attack string.

We use multi-track deterministic finite automata (MDFA) to implement a relational vulnerability signature generation algorithm. A multi-track automaton has multiple tracks and reads one symbol for each track in each transition; i.e., a multi-track automaton recognizes *tuples of strings* rather than a single string. We use a forward symbolic reachability analysis using MDFA to compute if any possible input values can lead to an attack string at a sink. During the forward analysis, each generated MDFA has one track for each input variable and represents the relation between the inputs and the program variable at that program point. Intersecting the MDFA at a sink with the attack pattern and projecting the resulting MDFA to the input tracks gives us the vulnerability signature. The vulnerability signature MDFA accepts all combinations of inputs that can exploit the vulnerability.

Once we generate the vulnerability signature we generate match and replace statements based on the vulnerability signature. The match statement basically simulates the vulnerability signature automaton and reports a match if the input string is accepted by the automaton. In the match-and-block strategy this is all we need, and we halt the execution if there is a match. In the match-and-sanitize strategy, however, we also need to generate a replace statement that will modify the input so that it does not match the

vulnerability signature. Since inputs that match the vulnerability signature may come from normal, non-malicious users (who, for example, may have accidentally typed a suspicious character), it would be preferable to change the input in a minimal way. We present an automata theoretic characterization of this *minimality* and show that solving it precisely is intractable. We show that we can generate a replace statement that is close to optimal in practice by adopting a polynomial-time min-cut algorithm.

5.1 Sanitization Generation

In this section we describe how we generate sanitization statements given a vulnerability signature that is characterized either as a standard single-track automaton (DFA) or a multi-track automaton (MDFA). We discuss the details of vulnerability signature generation in later sections.

In order to implement the match-and-block and match-and-sanitize strategies we need to generate code for the *match* and *replace* statements.

Match Generation: There are two ways of doing matching: 1) *Regular-expression-based matching:* Generate a regular expression from the vulnerability signature automaton and then use the PHP function `preg_match` to check if the input matches the generated regular expression, or 2) *Automata-simulation-based matching:* Generate code that, given an input string, simulates the vulnerability signature automaton to

determine if the input string is accepted by the vulnerability signature automaton, i.e., if the input string matches the vulnerability signature.

We first tried the regular-expression-based matching approach. However, this approach ends up being very inefficient due to the implementation of `preg_match` in PHP. The alphabet of the vulnerability signature automata consists of the 256 ASCII characters and the vulnerability signature automata can have a large number of states if there are a lot of complex string manipulation operations in the code. In one of the examples we analyzed the vulnerability signature automaton consists of 811 states. The size of the regular expression generated from the vulnerability signature automaton can be exponential in the number of states of the automaton [29]. Hence, we may end up with very large regular expressions. Moreover, the `preg_match` function in PHP does not only check if a given input matches the given regular expression but it also computes all the substrings that match the parenthesized subexpressions of the given regular expression. Since the DFA to regular expression conversion algorithm can generate a lot of parenthesized subexpressions, this means that the `preg_match` function will do a lot of unnecessary extra work during the match, resulting in an inefficient match implementation.

In order to do efficient matching we use the DFA simulation algorithm which has linear time complexity [29]. Given the vulnerability signature DFA, we generate a function that takes a string as input, simulates the DFA, and returns true if the DFA accepts

the string or false otherwise. We insert the match function instead of the `preg_match` statements shown in the patches in Figures 1.7 and 1.12.

Relational vulnerability signatures are characterized as multi-track automata (MDFA). Given a vulnerability signature MDFA, one approach could be to generate one DFA for each input (by erasing the other tracks using homomorphism) and then use the DFA simulation algorithm on each track. However, as we mentioned in Section 1.3, a relational vulnerability signature characterizes a relation among multiple inputs and may not be characterizable as constraints on individual tracks. For the vulnerability signature MDFA shown in Figure 1.11 projecting to each track leads to matching all inputs.

To address this problem, we can try to generate a DFA from the MDFA that recognizes the concatenation of the inputs, i.e., given an MDFA that recognizes tuples of strings (x, y) we can try to generate a DFA that recognizes the strings in the form $x.y$. Unfortunately, this type of MDFA to DFA conversion cannot be done precisely since the language recognized by MDFA may not be regular when it is written as concatenation of its tracks. For example, it is easy to construct a two-track MDFA that accepts the set of tuples $\{(x, y)|x = y\}$. If we write the same constraint in single-track form by concatenating the strings for two tracks we get the set $\{x.y|x = y\}$. This set is not regular, and, hence, cannot be represented as a DFA. However, for any given bound on the length of the strings, we can generate a DFA that is precise for the strings within

that bound but may accept more strings than the corresponding MDFA for strings that are longer than that bound.

The most precise solution for match generation from vulnerability signature MDFA is to generate code that simulates the MDFA directly and this is the option we use. The MDFA simulation algorithm is similar to the DFA simulation algorithm, it just keeps a separate pointer for each input string to keep track of how much of each track is processed at any given time and advances the state of the MDFA based on the tuples of input symbols and the transition relation of the MDFA. The simulation time for MDFA is linear in the total length of the input strings.

Replace Generation: For the match-and-sanitize strategy, our automated sanitization generation algorithm takes the vulnerability signature automaton as input, and it generates a replace statement that modifies a given input string in such a way that the modified string is not accepted by the vulnerability signature automaton (meaning that the modified string cannot cause an attack). We modify the input strings by just deleting a set of characters using the `preg_replace` function (our approach can be extended so that escape characters can be inserted in front of a set of characters rather than deleting them). 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.

First, we will formalize this problem in automata-theoretic terms. Let $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ denote a DFA where Q is the set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. $L(M)$ denotes the language accepted by M. We say $S \subseteq \Sigma$ is an *alphabet-cut* of M, if $L(M) \cap L_{\bar{S}} = \emptyset$, 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, $|S_{min}| \leq |S|$. For the example automaton in Figure 1.8 the min-alphabet-cut is $\{<\}$.

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 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. For the example automaton in Figure 1.8 the min-edge-cut is $\{(1, <, 2)\}$, which also corresponds to the min-alphabet-cut.

Note that, if the vulnerability signature DFA 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 will assume that the DFA for the vulnerability signature does not accept the empty string (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).

Theorem: *The min-alphabet-cut problem is NP-hard.*

We prove this by a reduction from the vertex cover problem. A vertex cover of a graph $G = (V, E)$ is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a minimum vertex cover is known to be NP-complete. Vertex cover problem can be reduced to the *min-alphabet-cut* problem as follows. Given $G = (V, E)$ we build an automaton $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ with the set of states $Q = E \cup \{q_0, q_F\}$, the initial state q_0, set of final states $F = \{q_F\}$, alphabet $\Sigma = V$, and the transition relation δ defined as follows: $e = (v, v') \in E \Rightarrow (q_0, v, e) \in \delta \wedge (e, v', q_F) \in \delta$. The *min-alphabet-cut* for the automaton M is the minimum vertex cover for the graph G.

Since the min-alphabet-cut problem is intractable, 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. In fact, there is a very good candidate for a heuristic solution. 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 [17] 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 an 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}. In our experiments this heuristic has been very effective and we typically obtained alphabet-cuts with only one to three symbols.

Once we compute an alphabet-cut S using our heuristic, we generate a `preg_replace` statement that deletes the symbols in S from the input, making sure that the resulting string does not match the vulnerability signature.

The definition of the min-alphabet-cut problem is slightly different for multi-track automata. Given an n-track DFA M over $(\Sigma \cup \lambda)^n$, we say an n-tuple $S = (S_1, \ldots S_n)$, where $S_i \subseteq \Sigma$, is an alphabet-cut of M, if $L(M) \cap L_{\bar{S}} = \emptyset$, where $L_{\bar{S}} = (((\Sigma \setminus S_1) \cup \lambda) \times \ldots ((\Sigma \setminus S_n) \cup \lambda)))^*$ is the set of all strings whose i^{th} track does not contain any character in S_i. Let $|S| = |S_1| + \ldots + |S_n|$. The *min-alphabet-cut* problem for a MDFA M is finding the alphabet cut S_{min} of M, such that for any alphabet cut S of M, $|S_{min}| \leq |S|$.

Since min-alphabet-cut is intractable for single-track DFA, it is also intractable for multi-track DFA. We use min-edge-cut also as an approximation for min-alphabet-cut for MDFA. When we find a min-edge-cut, we compute the corresponding multi-track alphabet-cut by computing a set of symbols for each track by collecting the set of

distinct symbols (other than λ) on each track on the edges in the min-edge-cut. The resulting alphabet cut is an n-tuple $S = (S_1, \ldots, S_n)$, where each S_i is the set of symbols for track i, i.e., input i. For the example automaton in Figure 1.11, the min-edge-cut is $\{(1, (<, \lambda), 3), (1, (\lambda, <), 4), (2, (\lambda, <), 4)\}$, which also corresponds to the min-alphabet-cut $(\{<\}, \{<\})$.

Once we compute the alphabet-cuts, we generate one `preg_replace` statement for each input variable i, that deletes every symbol in S_i from the input i making sure that the resulting input strings do not match the vulnerability signature.

5.2 Relational Signatures

In this section, we discuss how to generate *relational vulnerability signatures* where the vulnerability signature involves multiple input variables with respect to one sink. Formally speaking, a relational vulnerability signature M of n inputs is a MDFA over the n-track alphabet Σ^n, defined as $(\Sigma \times \{\lambda\}) \times \ldots \times (\Sigma \times \{\lambda\})$ (n times), where $\lambda \notin \Sigma$ is the special symbol for padding. We further restrict M, so that all tracks are aligned and for any $w \in L(M)$, $w[i] \in \lambda^*\Sigma^*\lambda^*$ ($1 \leq i \leq n$). Let $w'[i]$ denote the longest λ-free substring of $w[i]$.

Given a dependency graph G, a set of input nodes $\mathbf{In}$, a sink node $sink$, and an attack pattern $Attk$, we aim to generate a relational vulnerability signature M that sat-

isfies the following conditions: (1) M is a $|\mathbf{In}|$-track MDFA. Each track is associated with an input variable X_n, $n \in \mathbf{In}$. (2) For any word w ($w[i] \in \lambda^*\Sigma^*\lambda^*$), we have $w \in L(M)$ if the following condition holds: if we set $w'[i]$ as the initial value of the input node i and propagate the values of the nodes along with G accordingly, the value of the node $sink$ matches the pattern $Attk$. That is, w identifies the malicious inputs whose combination may exploit the vulnerability.

The algorithm to generate a relational vulnerability signature is shown in Algorithm 5. We perform *forward* fixpoint computation on the dependency graph where `replace` nodes are ignored. Our relational vulnerability signature algorithm is not capable of handling replace statements. However, since we run the vulnerability signature generation after a vulnerability is detected, we argue that it is reasonable to ignore the sanitization statements in the code (which is the typical use for the replace statements). After we generate the relational vulnerability signature, the existing sanitization statements can be commented out and replaced with the automatically generated sanitization statements. However, in cases where the replace statements are used to manipulate input for purposes other than sanitization, our relational vulnerability signature technique will not be sound. In such cases, we can still use the single-track vulnerability generation algorithm described earlier to obtain a sound result.

Similar to the other analyses we presented, we use a standard work queue algorithm incorporating the automata widening operator. Each node is associated with a signature,

a i+1-track MDFA where the first i tracks are associated with some input variables, e.g., $X_n, n \in \mathbf{In}$, and the last track (output track) is associated with X_o used to represent the values of the current node. More specifically, i ($0 \leq i \leq |\mathbf{In}|$) is the number of the input variables whose values have been used to construct the values of the current node. The signature of node n specifies the relations among the values of the input variables and the values of the current node.

Algorithm 5 RELSIGGEN($G, \mathbf{In}, sink, Attk$)

```
1: INIT(S, G, In);
2: queue WQ := NULL;
3: for n ∈ In ∪ Root(G) do
4:    WQ.enqueue(Succ(n));
5: end for
6: while WQ ≠ NULL do
7:    n := WQ.dequeue();
8:    if n is concat then
9:       tmp : = CONCATSIGNATURE(S[n.p], S[n.s]);
10:   else
11:      tmp : = ⋃_{n'∈Pred(n)} S[n'];
12:   end if
13:   tmp := (tmp ∪ S[n])∇S[n];
14:   if tmp ⊈ S[n] then
15:      S[n] := tmp;
16:      WQ.enqueue(Succ(n));
17:   end if
18: end while
19: M:= S[sink] ∩ M_Attk;
20: Project the output track away from M;
21: return M;
```

In Algorithm 5, we use a MDFA vector S to record the updated signature at each node. $S[n]$ is the signature associated with node n. Initially, for each input node $n \in \mathbf{In}$, $S[n]$ is a 2-track DFA (associated with X_n and X_o) that accepts the identity relation on

X_n and X_o, i.e., the value of the current node is equal to the value of the input variable X_n. For a node $n \in Root(G) \setminus \mathbf{In}$, $S[n]$ is a single-track DFA (associated with X_o) that either accepts Σ^* if n is a `variable` node, or accepts a constant value if n is a `constant` node; i.e., the current value of the node is an arbitrary string or a constant. In both cases, it is not related to any input variable. For the rest, i.e., $n \notin Root(G)$, $S[n]$ accepts an empty set.

After we initialize S at line 1, we perform our fixpoint computation using the work queue algorithm. Between lines 6 and 18, we iteratively update the signature at each node until the queue is empty (reaching a fixpoint). To deal with the union or widening operator on S_1 and S_2 that may be associated with the different sets of input variables, say $\mathbf{X_1}$ and $\mathbf{X_2}$, we extend both tracks to $\mathbf{X}_1 \cup \mathbf{X}_2$ and X_o by padding λs in the added tracks. We then apply standard union or widening to these extended MDFA.

Below we describe how to concatenate two signatures: CONCATSIGNATURE(S_1, S_2), where S_1 is the signature of the prefix node and S_2 is the signature of the suffix node. Let $S_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ be a MDFA whose tracks are associated with the set of input variables $\mathbf{X}_1$ and X_o where $\Sigma_1 = (\Sigma \cup \lambda)^{|\mathbf{X}_1|} \times \Sigma$. Let $S_2 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2 \rangle$ be a MDFA whose tracks are associated with the set of input variables $\mathbf{X}_2$ and X_o where $\Sigma_2 = (\Sigma \cup \lambda)^{|\mathbf{X}_2|} \times \Sigma$. We first extend S_1 and S_2 to two MDFA that are associated with $\mathbf{X}_1 \cup \mathbf{X}_2$ and X_o. We extend S_1 (prefix) by adding λ in the added tracks, while we extend S_2 (suffix) by adding λ in both the added tracks and the common tracks that

are also associated with S_1. Formally speaking, the extension of S_1, denoted as S_1^λ, is $\langle Q_1, \Sigma_1^\lambda, \delta_1^\lambda, I_1, F_1 \rangle$, where

- $\Sigma_1^\lambda = (\Sigma \cup \lambda)^{|\mathbf{X}_1|} \times \lambda^{|\mathbf{X}_2 \setminus \mathbf{X}_1|} \times \Sigma$, and
- $\delta_1^\lambda(q, \alpha) = q'$ if $\delta_1(q, \beta) = q'$ and $\alpha[X] = \beta[X]$ if $X \in \mathbf{X}_1 \cup X_o$, and $\alpha[X] = \lambda$, otherwise.

The extension of S_2, denoted as S_2^λ, is $\langle Q_2, \Sigma_2^\lambda, \delta_2^\lambda, I_2, F_2 \rangle$, where

- $\Sigma_2^\lambda = \lambda^{|\mathbf{X}_1|} \times (\Sigma \cup \lambda)^{|\mathbf{X}_2 \setminus \mathbf{X}_1|} \times \Sigma$, and
- $\delta_2^\lambda(q, \alpha) = q'$ if $\delta_2(q, \beta) = q'$ and $\alpha[X] = \lambda$ if $X \in \mathbf{X}_1$, and $\alpha[X] = \beta[X]$, otherwise.

CONCATSIGNATURE(S_1, S_2) returns the $(|\mathbf{X}_1 \cup \mathbf{X}_2| + 1)$-track DFA that accepts the concatenation of S_1^λ and S_2^λ.

The intuition of this implementation is to keep the values of the tracks that are associated with input variables unchanged (except padding λs in the front or end) and concatenate the values of the output tracks without inserting any λ. Since, during the computation λs are only attached to tracks that are associated with input variables, for the values of the current node, i.e., the values of the output track, we have $w[X_o] \in \Sigma^*$ for any w accepted by a signature at each node. This property enables us to avoid interference by λs while taking intersection on the output track.

After reaching a fixpoint, at line 19, we intersect the signature of $sink$ with the attack pattern on the output track. Let M_{Attk} accepts $\{w \mid w[X_o] \in Attk\}$. This is done by the standard intersection of $S[sink]$ and M_{Attk}. After the intersection, the output track identifies the reachable attack strings, and the input tracks identify all the malicious inputs whose combination can yield an attack string. At line 20, we project away the output track from M, and return the result at line 21 as the relational vulnerability signature of $\langle G, \mathbf{In}, sink, Attk \rangle$. Note that M may have λs as prefix and suffix which can be removed by the same approach presented for single-track DFA.

5.3 Experiments

We evaluated our approach on five vulnerabilities from three open source web applications (1) `MyEasyMarket-4.1` (a shopping cart program), (2) `BloggIT-1.0` (a blog engine), and (3) `proManager-0.72` (a project management system). We used the following XSS attack pattern $\Sigma^* < SCRIPT\Sigma^*$. The dependency graphs of these benchmarks are built for sensitive sinks where unrelated parts have been removed using slicing. Hence, their sizes (approximately 20-30 nodes) are much smaller than the original programs.

In our experiments, we used an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We use 8 bits to encode each character in ASCII.

Vulnerability Analysis: The performance of our vulnerability analysis is shown in Table 5.1. We also show 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 accepts all reachable attack strings at the sink node. For all five benchmarks, $L(M)$ is not an empty set and we conclude that all benchmarks are vulnerable. (”y” indicates that the benchmark is vulnerable and #inputs indicates the number of `input` nodes.)

	Time(s)	Mem(kb)	Result	#states / #bdds	#inputs
1	0.08	2599	y	23/219	1
2	0.53	13633	y	48/495	1
3	0.12	1955	y	125/1200	2
4	0.12	4022	y	133/1222	1
5	0.12	3387	y	125/1200	1

Table 5.1: Vulnerability Analysis Performance

Signature Generation: Table 5.2 summarizes the performance of vulnerability signature generation. ”S” indicates that we used backward analysis to generate single track vulnerability signature, while ”R” indicates that we generated relational vulnerability signature via forward analysis. The last column shows the size of the vulnerability signature DFA or MDFA. Since only benchmark 3 contains two inputs, we only use relational vulnerability analysis for this benchmark. 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. Since the benchmark is vulnerable, i.e., the existing sanitization routine is not good enough to eliminate the specified attacks, one may consider commenting out these replacement operations, which would improve the performance of our backward analysis.

	Signature type	Time(s)	Mem(kb)	#states /#bdds
1	S	0.46	2963	9/199
2	S	41.03	1859767	811/8389
3	S	2.35	5673	20/302, 20/302
3	R	0.66	6428	113/1682
4	S	2.33	32035	91/1127
5	S	5.02	14958	20/302

Table 5.2: Signature Generation Performance

Sanitization Synthesis: We use the vulnerability signature automata to automatically generate sanitization code. For matching, we generate code that simulates the vulnerability signature automaton. 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 character 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 5.1. The overhead of matching a 1000 character string to the vulnerability signature automaton is less than 0.35 milliseconds.

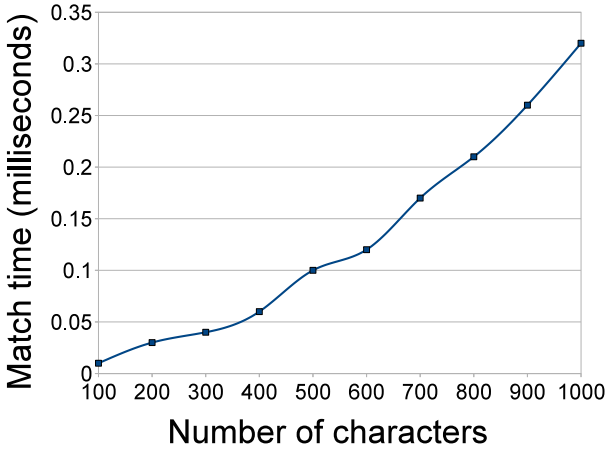


Figure 5.1: Input Matching Overhead

For the replace statements, we simply use the PHP `preg_replace` that is based on the minimum edge cut of the vulnerability signature automaton. Table 5.3 shows the number of edges in the min-edge-cut for the vulnerability signature automata we computed earlier, and the alphabet-cuts that correspond to these min-edge-cuts.

Signature	1	2	3S	3R	4	5
#edges	1	8	4	3	4	4
alphabet-cut	$\{<\}$	$\{S,',"\}$	Σ, Σ	$\{<\}$, $\{S\}$	$\{<,',"\}$	$\{<,',"\}$

Table 5.3: Minimum Edge and Alphabet Cuts

Table 5.3 shows that the min-edge-cut results in a very small alphabet-cut. Especially for the first benchmark, it is clear that we got the optimum solution since we have a single symbol in the cut. The result for benchmark 3 using single-track DFA is Σ for both inputs (i.e., we need to delete all characters for both inputs). This is due to the fact that our analysis that uses single track automata can not keep the relation between

the input and output and any of the two inputs can contribute the attack to the sink. In this case the vulnerability signature is Σ^*. But when we use the relational vulnerability signature for this example, the min-edge-cut for the multi-track automaton has 3 edges corresponding to character '<' for input 1 and 'S' for input 2 (i.e., we only need to delete '<' for input 1 and delete 'S' for input 2).

Chapter 6

Composite Analysis

We present a composite symbolic verification technique [12] that combines string [1, 15, 52, 58] and size [20, 22, 46] analyses with the goal of improving the precision of both. We use a forward fixpoint computation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables.

Similar to prior size analysis techniques [20, 22, 46] we associate each string variable with an auxiliary integer variable that represents its length. At each program point, we symbolically compute all possible values of all integer variables (including the auxiliary variables), as well as all possible values of all string variables. The reachable values of all integer variables are over-approximated as a Presburger arithmetic (linear arithmetic) formula and symbolically encoded as *arithmetic automata* [3, 51]. Simi-

lar to some prior string analysis techniques [1, 58], the values that string variables can take are over-approximated as regular languages and symbolically encoded as *string automata*. Our composite analysis is as a forward fixpoint computation with widening on these arithmetic and string automata.

There are two challenges we need to overcome to connect the information contained in the string automata and the arithmetic automata (hence, improving the precision of both) during our composite analysis: 1) Given a string automaton, we need to derive the arithmetic automaton that accepts the length of the language accepted by the string automaton, and 2) Given an arithmetic automaton, we need to restrict a string automaton so that the length of the language is accepted by the arithmetic automaton.

To tackle the first challenge, we present techniques for constructing a *length automata* for a given regular language. It is known that the length of the language accepted by a DFA forms a semilinear set. Given an arbitrary DFA, we are able to construct DFAs that accept either unary or binary representation of the length of its accepted words. The unary automaton can be used to identify the coefficients of the semilinear set, while the binary automaton can be composed with other arithmetic automata on integer variables to enforce or check length constraints.

To tackle the second challenge, we identify the boundary of the lengths of string variables from the arithmetic automaton. Precisely, we compute the lower and upper bound of the values of the string lengths accepted by the arithmetic automaton. We

prove that, given a one-track arithmetic automaton, the lower bound forms a shortest path to an accepting state while the upper bound (if it exists) forms the longest loop-free path. Both can be computed in linear complexity to the size of the arithmetic automaton. We can restrict the target string automaton by intersecting the string automaton that accepts arbitrary strings within this boundary.

This chapter is organized as follows. We present the length automata construction in Section 6.1. We present our composite analysis technique that integrates string and arithmetic analyses in Section 6.2. We present our experiments with our prototype tool in verifying small C routines, buffer-overflow benchmarks and PHP web applications in Section 6.3.

6.1 Length Automata Construction

Given a string automaton M, we want to construct a DFA M_b (over a binary alphabet) such that $L(M_b)$ is the set of binary representations of the lengths of the words accepted by M. We tackle this problem in two steps. We first construct a DFA M_u (over a unary alphabet) such that $L(M_u)$ is the set of unary representations of the lengths of the words accepted by M. It is known that this set is a semilinear set. We identify the formula that represents the semilinear set from M_u. We then construct M_b from the

formula, such that $w \in L(M_b)$ if and only if the binary value of w satisfies the formula (i.e., the unary representation of the binary value of w is in $L(M_u)$).

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where Q is a finite set of states, q_0 is the initial state, Σ is a finite set of symbols. $F : Q \to \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \to Q$ is the transition function. The cardinality of a finite set A is denoted as $\sharp A$. The set of arbitrary words over a finite alphabet Σ is denoted as Σ^*. The length of a word $w \in \Sigma^*$ is denoted as $|w|$. A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a $sink$ state. In the constructions below, we also ignore the transitions that lead to a sink state.

A string automaton M is a DFA that consists of a tuple of $\langle Q, q_0, B^k, \delta, F \rangle$. M accepts a set of words, where each symbol is encoded as a k-bit string.

6.1.1 Length Constraints on String Automata

We are interested in characterizing lengths of the accepted words. We characterize these lengths as a set of natural numbers by a *length constraint*. Formally speaking, the length constraint of a given string automaton M is a formula f over a variable x, such that $f[c/x]$ evaluates to true if and only if there exists a word w, such that $w \in L(M)$ and $c = |w|$.

Regular language	Length set
$baaab$	$\{5\}$
$(baaab)^+$	$\{5+5k \mid k \geq 0\}$
$(baaab)^+ab$	$\{7+5k \mid k \geq 0\}$
$((baaab)^+ab)^+$	$\{7, 12, 14, 17, 19, 21, 22, 24, 25, 26, 27, 28\} \cup \{29+k \mid k \geq 0\}$
$(abb)^+$	$\{3+3k \mid k \geq 0\}$
$(abb)^+ \mid (baaab)^+ab$	$\{3+15k \mid k \geq 0\} \cup \{6+15k \mid k \geq 0\} \cup \{7+15k \mid k \geq 0\} \cup$ $\{9+15k \mid k \geq 0\} \cup \{12+15k \mid k \geq 0\} \cup \{15+15k \mid k \geq 0\} \cup$ $\{17+15k \mid k \geq 0\}$

Table 6.1: Regular Languages and Their Length Sets

Property 1: For any DFA M, $\{|w| \mid w \in L(M)\}$ forms a semilinear set.

Property 2: For any DFA M, f_M is in the form that $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = a_j + b_j \times k$, where a_j, b_j and c_i are constants. f_M can be written as $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, such that c_i, r_j, C, R are constants, and $\forall i, c_i < C$, and $\forall j, r_j < R$. We say that a semilinear set in this form is *well-formed*.

In the following, we give the algorithm to construct the automata that accept unary or binary representation of the length of the language accepted by a given string automata. This construction shows that the length constraint of a DFA is a well formed semilinear set, and hence gives a constructive proof of Property 1 and Property 2.

Before delving into the construction details, in Table 6.1, we give some examples of a regular language and the set of lengths of its words. It can be seen that identifying the length set of an arbitrary regular language is not trivial, particularly for those having nested closure. The results in Table 6.1 are obtained automatically by implementing our construction.

6.1.2 From String Automata to Unary Length Automata

It is known that the unary representation of the values of a semilinear set can be uniquely identified by a unary automaton. In the following, we first show how to construct an automaton M_u (over a unary alphabet) from a given string automaton M, such that $L(M_u)$ is the set of unary representations of $\{|w| \mid w \in L(M)\}$. We say M_u is the unary length automaton of M.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F\rangle$, a naive construction of the unary length automaton is $M_u = \langle Q, q_0, B^1, \delta', F\rangle$, where $\delta'(q, 1) = q'$ if $\exists \alpha, \delta(q, \alpha) = q'$. However, M_u constructed this way will be an NFA. The MBDD representations that we use cannot encode NFAs. Instead, we use a construction which combines the projection and determinization steps as follows.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F\rangle$, we first construct an intermediate automaton $M' = \langle Q, q_0, B^{k+1}, \delta', F\rangle$, where

- $\forall q, q' \in Q$, and both are not sink states, $\delta'(q, \alpha 1) = q'$, if $\delta(q, \alpha) = q'$.

M' is a DFA that accepts the same words as M except that each symbol in the word is appended with '1'. M_u can then be constructed from M' by projecting the first k bits away. This projection is done by iterative determinization and minimization. During determinization, the subset construction is applied on the fly.

6.1.3 From Unary Length Automata to Semilinear Set

Here we describe how to identify the well formed formula of a semilinear set from a unary automaton.

Property 3: A finite deterministic unary automaton $M = \langle Q, q_0, B^0, \delta, F \rangle$ can be in two forms: a linear list of states that starts from the initial state with finite length $\sharp Q$, or a linear list of states that starts from the initial state with finite length, C, and ends in a cycle with finite length, R, where $C + R = \sharp Q$ (i.e., a lasso).

Given a deterministic unary automaton, Q can be labeled such that

- $\sharp Q = n + 1$.
- $\forall 0 \leq i < n, \delta(q_i, 1) = q_{i+1}$.

Cycle Case: If $\exists 0 \leq m < n, \delta(q_n, 1) = q_m$, the well-formed formula of a unary automaton is $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, where

- $C = m$, $R = n - m$.
- $\forall i$, $\exists q_t, t < m, F(q_t) = +$, $c_i = t$.
- $\forall j$, $\exists q_t, t \geq m, F(q_t) = +$, $r_j = t - m$.

No Cycle Case: Otherwise, the well-formed formula of a unary automaton is $\bigvee_i x = c_i$, where $\forall i$, $\exists q_t, t \leq n, F(q_t) = +$, $c_i = t$.

6.1.4 From Semilinear Set to Binary Length Automata

We propose a novel construction to derive a DFA M such that $L(M)$ is equal to the set of binary representations (from the least significant bit) of a well-formed semilinear set. We say M is a binary length automaton of the string automaton, the length of whose accepted words forms the semilinear set.

Assume that we are given a well-formed semilinear set $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$. Let N be $max(C, R)$. A DFA M that accepts the binary representation of the given semilinear set can be constructed as a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$, where:

- We assume that there exists a sink state $q_{sink} \in Q$, s.t., $F(q_{sink}) = -$, $\delta(q_{sink}, 0) = q_{sink}$ and $\delta(q_{sink}, 1) = q_{sink}$, and all transitions that are ignored in this construction are going to q_{sink}.

- Other than the sink state, each state $q \in Q$ is a tuple (t, v, b), where $t \in \{\texttt{val}, \texttt{rem}_t, \texttt{rem}_f\}$, $v \in \{0, \ldots, N\}$, and $b \in \{\bot\} \cup \{1, \ldots, N\}$. $q.t$ is the type of state q, which indicates the meaning of the value of $q.v$ and $q.b$. While $q.t = \texttt{val}$, $q.v$ is equal to the value of the binary word accepted from the initial state to the current state, and $q.b$ is equal to the binary value of the previous bit in the word. We assume $2 \bot = 1$. While $q.t = \texttt{rem}_t$ or $\texttt{rem}_f$, $q.v$ is equal to the remainder of which the dividend is the value of the binary word accepted from the initial state to the current state and the divisor is R; $q.b$ is the remainder of which the dividend is

the binary value of the previous bit in the accepted word and the divisor is R. $q.t = \texttt{rem}_t$ indicates the value of the binary word accepted from the initial state to the current state is greater or equal to C; $q.t = \texttt{rem}_f$ indicates the value is less than C.

- q_0 is $(\texttt{val}, 0, \perp)$.
- $\Sigma = \{0, 1\}$ (i.e., B^1).
- $\delta(q, 1) = q'$ if and only if one of the following condition holds:
  - $q.t = \texttt{val}$, $q.v + 2q.b \geq C$, $q'.t = \texttt{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
  - $q.t = \texttt{val}$, $q.v + 2q.b < C$, $q'.t = \texttt{val}$, $q'.v = q.v + 2q.b$, $q'.b = 2q.b$.
  - $q.t = \texttt{rem}_t$, $q'.t = \texttt{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
  - $q.t = \texttt{rem}_f$, $q'.t = \texttt{rem}_t$, $q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
- $\delta(q, 0) = q'$ if and only if one of the following condition holds:
  - $q.t = \texttt{val}$, $q.v + 2q.b \geq C$, $q'.t = \texttt{rem}_f$, $q'.v = q.v \mod R$, $q'.b = (2q.b) \mod R$.

  - $q.t = \texttt{val}$, $q.v + 2q.b < C$, $q'.t = \texttt{val}$, $q'.v = q.v$, $q'.b = 2q.b$.
  - $q.t = \texttt{rem}_t$, $q'.t = \texttt{rem}_t$, $q'.v = q.v$, $q'.b = (2q.b) \mod R$.
  - $q.t = \texttt{rem}_f$, $q'.t = \texttt{rem}_f$, $q'.v = q.v$, $q'.b = (2q.b) \mod R$.

- $F(q) = +$, for all $q \in \{q \mid q.t = \texttt{val}, \exists i, q.v = c_i\} \cup \{q \mid q.t = \texttt{rem}_t, \exists j, q.v = (C + r_j) \mod R\}$; $F(q) = -$, o.w.

By definition, $\sharp Q$ is $O(N^2)$. Precisely, in our construction, the number of states that $q.t = \texttt{val}$ is bounded by C. The number of states that $q.t = \texttt{rem}_t$ is bounded by R^2 and the number of states that $q.t = \texttt{rem}_f$ is bounded by $C \times R$. On the other hand, we have observed that after minimization, $\sharp Q$ is often reduced to N.

An Incremental Algorithm: Below we give an incremental algorithm to construct a Binary Length Automaton (BLA) M. The construction is achieved by calling the procedure CONSTRUCT_BLA . The input is given as a well-formed semilinear formula, $\bigvee_{0 \le i \le n} x = c_i \vee \bigvee_{0 \le j \le m} \exists k.x = \mathrm{C} + \mathrm{r_j} + \mathrm{R} \times \mathrm{k}$. At line 3, we first build Q^b, the set of binary states that will be reached by calling the procedure ADD_BSTATE. A binary state is actually the value of the tuple (t, v, b) as described in the previous section. Each binary state is further associated with an index, a true branch and a false branch, which are used to construct the state graph. Briefly, ADD_BSTATE is a recursive function which incrementally adds the reached binary state if it has never been explored. Initially, the binary state is $(\texttt{val}, 0, \perp)$. Note that ADD_BSTATE is guaranteed to ter-

minate since the number of binary states are bounded. Upon termination, all reached binary states will have been added to Q^b. For each binary state in Q^b, as line 4 to 9, we iteratively generate a state q and set its transition relation and accepting status, which are used to construct the final automaton at line 10.

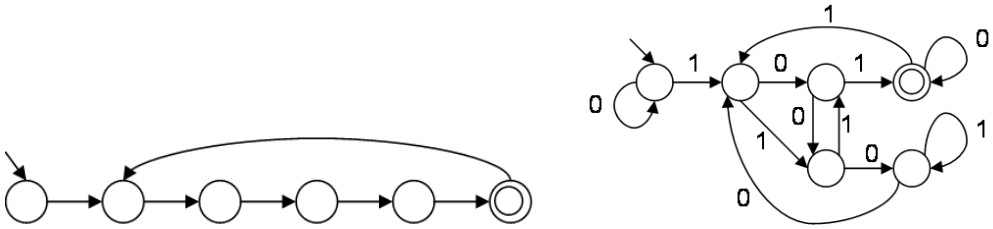
Algorithm 6 ADD_BSTATE(Q, C, R, t, v, b)

1: **if** $\exists q = (t, v, b) \in Q$ **then**
2: **return** $q.index$;
3: **else**
4: Create $q = (t, v, b)$;
5: $q.index = \sharp Q$;
6: $q.true = -1$;
7: $q.false = -1$;
8: Add q to Q;
9: **if** $t == \texttt{val} \wedge (v + 2 \times b \geq$C) **then**
10: $q.true$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
11: $q.false$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
12: **else if** $t == \texttt{val} \wedge (v + 2 \times b < C)$ **then**
13: $q.true$ =ADD_BSTATE(Q, C, R, val, $v + 2 \times b$, $2 \times b$);
14: $q.false$ =ADD_BSTATE(Q, C, R, val, v, $2 \times b$);
15: **else if** $t == \texttt{rem}_t$ **then**
16: $q.true$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
17: $q.false$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_t$, $v\%$R, $(2 \times b)\%$R);
18: **else if** $t == \texttt{rem}_f$ **then**
19: $q.true$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
20: $q.false$ =ADD_BSTATE(Q, C, R, $\texttt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
21: **end if**
22: **return** $q.index$;
23: **end if**

We have implemented the above algorithms using the MONA DFA package. Minimal unary and binary length automata for a regular language are shown Figure 6.1 and Figure 6.2. It is interesting to note that in both cases, the minimal unary length

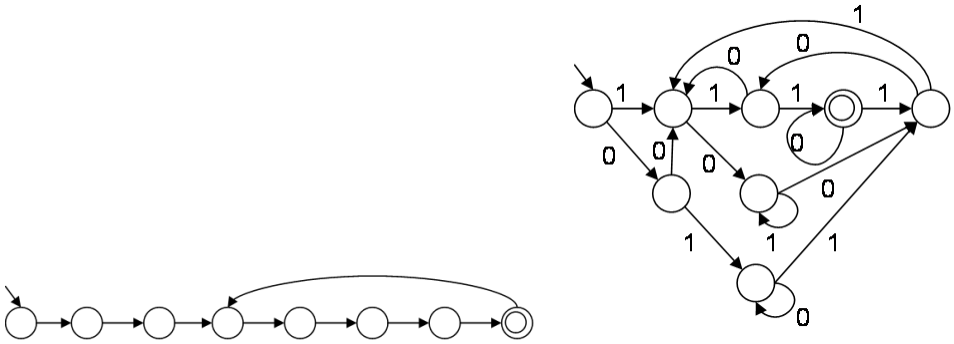
Algorithm 7 CONSTRUCT_BLA(C, R, $\mathcal{C} = \{c_1, c_2, \ldots c_n\}$, $\mathcal{R} = \{r_1, r_2, \ldots r_m\}$)

1: $Q^b = \emptyset$;
2: $Q = \emptyset$;
3: $init$ =ADD_BSTATE(Q^b, C, R, `val`, 0,$\perp$);
4: **for** each $q^b \in Q^b$ **do**
5: Add $q = q_{q.index}$ to Q;
6: $\delta(q, 1) = (q^b.true \neq -1?q_{q^b.true} : q_{sink})$;
7: $\delta(q, 0) = (q^b.false \neq -1?q_{q^b.false} : q_{sink})$;
8: $F(q) = ((q^b.t == 0 \wedge \exists c \in \mathcal{C}.q^b.v == c) \vee (q^b.t == 1 \wedge \exists r \in \mathcal{R}.q^b.v == (r+\text{C})\%\text{R}) :' +'?'-')$;
9: **end for**
10: Construct $M = \langle Q \cup \{q_{sink}\}, q_{init}, B^1, \delta, F\rangle$;



(a) Unary (b) Binary

Figure 6.1: The Length Automata of $(baaab)^+$



(a) Unary (b) Binary

Figure 6.2: The Length Automata of $(baaab)^+ab$

automaton and binary length automaton have the same number of states. In Figure 6.1, both automata accept the set $\{5+5k|k \geq 0\}$. Consider the number 1865735, whose binary encoding is 11100011110000000111. One can test that the bit string from the least significant is accepted by the binary length automaton shown in 6.1(b). In Figure 6.2, both automata accept the set $\{7+5k|k \geq 0\}$. Consider the number 1087, whose binary encoding is 10000111111. The bit string from the least significant is also accepted by the binary length automaton shown in 6.2(b).

We have presented the algorithms to construct length automaton from an arbitrary string automaton. The construction of both unary and binary length automaton has been implemented using the MONA DFA package. We are able to identify the precise semilinear set for arbitrary regular language, and construct both unary and binary length automata which accept the unary and binary representations of the semilinear set respectively. In the following section, we present a *composite analysis* that integrates length automata with string and arithmetic analyses.

6.2 Composite Verification

We first introduce a simple imperative language (the syntax is similar to the one used in [53]) as our target language. This language consists of a set of labeled statements $l : stat$. Labels correspond to instruction addresses. We use s to denote a string

variable, i to denote an integer variable, and c to denote a constant. Each $s \in S$ is associated with one auxiliary integer variable, denoted as $s.length$. Let S denote the set of string variables and I denote the set of integer variables, and I_L denote the set of auxiliary variables. A statement can be one of the following:

- A termination statement `halt` or `abort`.
- A string assignment statement $s := strexp$, where $strexp$ is a string expression that can be one of the following:
  - `input`(i) which returns an arbitrary string value up to the length equal to the value of i.
  - a string variable $s \in S$.
  - a regular expression $regexp$ over S.
  - `prefix`(s, i) which returns the prefix of s up to the first c characters where c is equal to the value of i.
  - `suffix`(s, i) which returns the the suffix of s starting from the c^{th} character, where c is equal to the value of i.
  - `concat`(s_1, s_2) that returns the concatenation of the value of s_1 and the value of s_2.

  - `replace`(s_1, s_2, s_3) that returns the result of the following actions: (1) scan the value of s_1 and find the substrings that match to the value of s_2, and (2) replace the matched substrings with the value of s_3.

- An integer assignment statement $i := intexp$, where $intexp$ is an integer expression in the form $\sum_t c_t * i_t$ that returns a value of the linear function $\sum_t c_t * i_t$, where each variable $i_t \in I \cup I_L$.

- A conditional statement `if` $(bexp)$ `goto` l', where $bexp$ is a binary expression (defined below). l' is a program label which indicates the label of the next statement when $bexp$ evaluates to true.

- An assertion statement `assert`$(\bigwedge bexp)$. An assertion holds if $\bigwedge bexp$ evaluates to true. A program is correct if all assertions hold on all executions.

A $bexp$ is either a string or an integer formula defined as follows:

- A string formula can be in two forms: (1) $s \in regexp$, or (2) $s[c_1, c_2] \in regexp$, which specifies that the value of s or the value of the substring (from the c_1^{th} to c_2^{th} character) of s is within a constant regular language. Note that here *regexp* is restricted to a constant set of string values. $s \notin regexp$ is an abbreviation of $s \in regexp'$, where $regexp'$ is the complement set of $regexp$. $s = c$ is an abbreviation of $s \in \{c\}$ and $s \neq c$ is an abbreviation of $s \notin \{c\}$, where c is a constant string.

```
strlen(s1){
1: cnt := 0;
2: s2:=s1;
3: if(s2='\0') goto 7;
4: s2:=suffix(s2, 1);
5: cnt := cnt +1;
6: if(s2 != '\0') goto 4;
7: assert(s1.length = cnt);
8: halt;
}
```

Figure 6.3: The Rewritten String Length Routine

- An integer formula can be in the form: $\sum_t c_t * i_t \sim c$, where $i_t \in I \cup I_L$ and $\sim \in \{=, <, \leq, \geq, >\}$.

We assume that for each $l : stmt$, $l + 1$ is a valid label if $stmt$ is not a termination statement. For each conditional statement `if` $(bexp)$ `goto` l', l' is a valid label.

Modeling a C Example: To analyze normal C programs, one can consider each dereference of a pointer, e.g., $*p$, as a string variable. A sequence value from the address pointed by the pointer is a string value of the string variable. The pointer arithmetic operation, e.g., $p_1 := p_2 + i$, can be considered as a string suffix statement that assigns the suffix of the dereference of p_2 to the dereference of p_1.

The string length routine shown in Figure 1.14 can be rewritten using this simple language as shown in Figure 6.3.

6.2.1 Verification Framework

Assume that $S = \{s_1, \ldots, s_m\}$ and $I = \{i_1, \ldots, i_n\}$ denote the set of string and integer variables in our target program, respectively. In our analysis, each string variable s_k, $1 \leq k \leq m$, is associated with an auxiliary integer variable i_{n+k} as its length $s_k.length$. Hence, we also have the set of auxiliary integer variables $I_L = \{i_{n+1}, \ldots i_{n+m}\}$. A state for each program label consists of a string-automata vector $\vec{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ and an $n+m$-track arithmetic automaton a.

Each string variable s_k is associated with the string automaton α_k in $\vec{\alpha}$, which accepts an over approximation of the set of all possible values that s_k can take at the corresponding program label. Each track of the arithmetic automaton a is a binary encoding starting from the least significant bit of the value of an integer variable (the first n tracks) or the value of the length of a string variable (the last m tracks).

A word accepted by the arithmetic automaton corresponds to a valid valuation for the integer variables and the lengths of string variables at the corresponding program point during the execution of the program. The arithmetic automaton accepts an over approximation of the set of possible words at the corresponding program label. Each word w is an assignment of the integer variables and the lengths of the string variables; and each track of w is actually the value that $i \in I \cup I_L$ can take at the corresponding program label. We use $w[k]$ to denote the k^{th} track of the word w. For $1 \leq k \leq n$, $w[k]$ is the value of the integer variable i_k. For $n+1 \leq k \leq n+m$, $w[k]$ is the length of the

string variable s_k. We say a string w is the value of a string variable s_k if $w \in L(\alpha_k)$, and $\exists w' \in L(a)$ such that $w'[k]$ is equal to the binary encoding of $|w|$ starting from the least significant bit.

Forward Fixpoint Computation: Our analysis is based on a standard forward fixpoint computation on $\vec{\alpha}$ and a for all program labels. For simplicity, we use $\nu[l]$ to denote $\vec{\alpha}[l]$ and $a[l]$, where $\vec{\alpha}[l]$ is the string-automaton vector and $a[l]$ is the arithmetic automaton at the program label l. The algorithm is a standard work-queue algorithm as shown in table 11.

For sequential operations (string/integer assignments), we are continuously computing the post image of $\nu[l]$ against $l : stmt$, and join the result to $\nu[l+1]$ where $l+1$ is the label of the next statement. For branch statement l : `if`$(bexp)$ `goto` l', if the intersection of the language of $\nu[l]$ and $bexp$ is not an empty set, we add the result to $\nu[l']$. If the intersection of the language of $\nu[l]$ and the complement set of $bexp$ is not an empty set, we add the result to $\nu[l+1]$. For checking statement l : `assert`(ϕ), if the language of $\nu[l]$ is not included in ϕ, we raise an alarm.

Upon joining the results, we check whether a fixpoint of that program point is reached. If it is not, we update ν at that program point and push its labeled statement into the queue. Since we target infinite state systems, the fixpoint computation may not terminate. We incorporate an automata widening operator, denoted as ∇_A,

Algorithm 8 COMPOSITEANALYSIS(l_0)

```
1:  Init(ν);
2:  queue WQ;
3:  WQ.enqueue(l_0 : stmt_0);
4:  while WQ ≠ NULL do
5:     e := WQ.dequeue(); Let e be l : stmt;
6:     if stmt is sequential operation then
7:        tmp := post(ν[l], stmt);
8:        tmp := (tmp ∪ ν[l + 1])∇ν[l + 1];
9:        if tmp ⊈ ν[l + 1] then
10:          ν[l + 1] := tmp;
11:          WQ.enqueue(l + 1);
12:       end if
13:    end if
14:    if stmt is if bexp goto l' then
15:       if CheckIntersection(ν[l], bexp) then
16:          tmp := ν[l] ∧ bexp;
17:          tmp := (tmp ∪ ν[l'])∇ν[l'];
18:          if tmp ⊈ ν[l'] then
19:             ν[l'] := tmp;
20:             WQ.enqueue(l');
21:          end if
22:       end if
23:       if CheckIntersection(ν[l], ¬bexp) then
24:          tmp := ν[l] ∧ ¬bexp;
25:          tmp := (tmp ∪ ν[l + 1])∇ν[l + 1];
26:          if tmp ⊈ ν[l + 1] then
27:             ν[l + 1] := tmp;
28:             WQ.enqueue(l + 1);
29:          end if
30:       end if
31:    end if
32:    if stmt is assert(φ) then
33:       if ¬ CheckInclusion(ν[l], φ) then
34:          Assertion violated!
35:       end if
36:    end if
37: end while
```

proposed by Bartzis and Bultan in [4] to accelerate the fixed point computation. $\nu\nabla\nu'$ is implemented as $\alpha_1\nabla_A\alpha'_1, \ldots, \alpha_m\nabla_A\alpha'_m$ [58] and $a\nabla_A a'$ [4].

Finally, we detail how to compute post and restrict computations (i.e., post$(\nu, stmt)$ and $\nu \wedge bexp$) in the following paragraphs.

Basic Operations: Before we detail the algorithms of post and restrict computations, we first define some notations and basic operations to simplify our presentation. We use a to denote the arithmetic automaton, and a_k to denote the one-track arithmetic automaton that accepts the values of the k^{th} track of the arithmetic automaton a. We use α to denote a string automaton and $\vec{\alpha}$ to denote a vector of string automata. α_k is the k^{th} string automaton of $\vec{\alpha}$. `bla`(α) returns the binary length automaton of the string automaton α. The binary length automaton can be considered as an one-track arithmetic automaton. We use α^c, where c is an integer constant, to denote the string automaton which accepts arbitrary words having length equal to c. That is $L(\alpha^c) = \{w \mid w \in \Sigma^*, |w| = c\}$. This notation is also extended to a range $[c_1, c_2]$, where c_1, c_2 are integer constants. We say that $\alpha^{[c_1,c_2]}$ is the string automaton that accepts $\{w \mid w \in \Sigma^*, c_1 \leq |w| \leq c_2\}$.

- Extraction: $a \lfloor_k$, returns an one-track arithmetic automaton a_k so that $w \in L(a_k)$ if $\exists w' \in L(a)$ and $w'[k] = w$. a_k is constructed by projecting away all tracks except the k^{th} track of the arithmetic automaton a.

- Projection: $a \upharpoonright_k$, returns a new arithmetic automaton a' which accepts $\{w | w' \in L(a), \forall 1 \leq t \leq m+n, t \neq k, w'[t] = w[t]\}$. a' is constructed by projecting away the track k of the arithmetic automaton a.

- Composition: $a \circ \alpha_k$, returns a new arithmetic automaton a' so that $L(a') = \{w \mid w \in L(a), w[k] \in L(\texttt{bla}(\alpha_k))\}$. a' is constructed by intersecting a with an arithmetic automaton that the track k is accepted by the binary length automaton of the string automaton α_k, and other tracks are unrestricted. This composition restricts $L(a)$ to a smaller set where the length of s_k (the value of the track k) is accepted by the binary length automaton of α_k.

- Boundary: $\texttt{min}(a_k)$ returns the lower bound of the set of integer values whose binary encodings from the least significant bit are accepted by the one-track automaton a_k. $\texttt{max}(a_k)$ returns the upper bound.

Post Images: Recall that there are m string variables and n integer variables. Given $stmt$ and the state ν that consists of $\vec{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ and the arithmetic automaton a, we want to compute $\vec{\alpha'} = \langle \alpha'_1, \ldots, \alpha'_m \rangle$ and a' as the result of the post image against $stmt$. We assume that the automata that are not specified remain the same. Let $stmt$ be one of the following:

- $s_k := \texttt{input}(i_p)$. $\alpha'_k := \alpha^{[c_1,c_2]}$, where $c_1 = \texttt{min}(a_p)$ and $c_2 = \texttt{max}(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k} := i_p)$.

- $s_{k_1} := s_{k_2}$. $\alpha'_{k_1} := \alpha_{k_2}$. $a' :=$ CONSTRUCT$(a, i_{n+k_1} := i_{n+k_2})$.
- $s_k := regex$. $\alpha'_k :=$ CONSTRUCT$(regexp)$. $a' := a \restriction_{n+k} \circ \alpha'_k$.
- $s_{k_1} :=$ `prefix`(s_{k_2}, i_p). $\alpha'_{k_1} :=$ PREFIX$(\alpha_{k_2}, [c_1, c_2])$, where $c_1 =$ `min`(a_p) and $c_2 =$ `max`(a_p). $a' :=$ CONSTRUCT$(a, i_{n+k_1} := i_p) \wedge$CONSTRUCT$(i_{n+k_2} - i_p \geq 0)$.
- $s_{k_1} :=$ `suffix`(s_{k_2}, i_p). $\alpha'_{k_1} :=$ SUFFIX$(\alpha_{k_2}, [c_1, c_2])$, where $c_1 =$ `min`(a_p) and $c_2 =$ `max`(a_p). $a' :=$ CONSTRUCT$(a, i_{n+k_1} := i_p) \wedge$CONSTRUCT$(i_{n+k_2} - i_p \geq 0)$.
- $s_k :=$ `strcat`(s_{k_1}, s_{k_2}). $\alpha'_k :=$ CONCAT$(\alpha_{k_1}, \alpha_{k_2})$. $a' :=$ CONSTRUCT$(a, i_{n+k} := i_{n+k_1} + i_{n+k_2})$.
- $s_k :=$ `replace`$(s_{k_1}, s_{k_2}, s_{k_3})$. $\alpha'_k :=$ REPLACE$(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$. $a' := a \restriction_{n+k} \wedge a_{tmp}$, where a_{tmp} accepts $\{w \mid w[k] \in L($`bla`$(\alpha'_k))\}$.
- $i_p := intexp$. $a' :=$ CONSTRUCT$(a, i_p := intexp)$.

Restriction: Here we describe the result of $\nu \wedge bexp$, where ν is the state consists of $\vec{\alpha}$ and a. Let $bexp$ be one of the following:

- $s_k \in regexp$. $\alpha'_k = \alpha_k \wedge$ CONSTRUCT$(regexp)$. $a' = a \circ \alpha'_k$.
- $s_k[c_1, c_2] \in regexp$. $\alpha'_k = \alpha_k \wedge \alpha_{tmp}$, where α_{tmp} is constructed by CONCAT(CONCAT$(\alpha^{[c_1,c_2]},$ CONSTRUCT$(regexp)), \alpha^*)$. $a' = a \circ \alpha'_k$.

- $\sum_t c_t * i_t \sim c.\ \forall t > n.\alpha'_t = \alpha_t \wedge \alpha^{[c_1,c_2]}$, where $c_1 = \texttt{min}(a' \lfloor_t)$ and $c_2 = \texttt{max}(a' \lfloor_t)$. $a' = a \wedge \text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$.

6.2.2 Implementation

Automaton Construction: Here we describe how to construct the corresponding arithmetic and string automata used in our composite analysis. The constructions of arithmetic automata including $\text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$ and $\text{CONSTRUCT}(a, i := \sum_t c_t * i_t)$ are detailed in [3]. The latter returns an arithmetic automaton which accepts the result of the post image computation on a against the integer assignment $i := \sum_t c_t * i_t + c$. This construction is implemented by quantifier elimination and variable renaming; i.e., $(\exists i, \Phi(a) \wedge i' = \sum_t c_t * i_t)[I'/I]$. For some special cases, the time complexity of this construction is linear to the size of a [3]. The constructions of string automata including $\text{CONSTRUCT}(\mathit{regexp})$, $\text{CONCAT}(\alpha_{k_1}, \alpha_{k_2})$, and $\text{REPLACE}(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$ have been detailed in [58]. We describe the implementation of $\text{PREFIX}(\alpha, [c_1, c_2])$ and the implementation of $\text{SUFFIX}(\alpha, [c_1, c_2])$ below.

Prefix: Formally speaking, α' is a prefix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid w \in \Sigma^{[c_1,c_2]}, \exists w', ww' \in L(\alpha)\}$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F\rangle$ and $[c_1, c_2]$, we first construct $\alpha' = \langle Q, q_0, \Sigma, \delta, F'\rangle$, where $\forall q \in Q, F'(q) =' +'$. α' accepts the prefix of $L(\alpha)$. The next step is restricting its length to the range $[c_1, c_2]$.

PREFIX$(\alpha, [c_1, c_2])$ returns the the result of the intersection of α' and $\alpha^{[c_1,c_2]}$, which is exactly the prefix-DFA of α regarding to the range $[c_1, c_2]$.

Suffix: Formally speaking, α' is a suffix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid \exists w' \in \Sigma^{[c_1,c_2]}, w'w \in L(\alpha)\}$. We first introduce the function REACH$(\alpha, [c_1, c_2])$. REACH$(\alpha, [c_1, c_2])$ returns the set of all $[c_1, c_2]$-reachable states. We say a state is $[c_1, c_2]$-reachable if it is reachable from the initial state by k steps and $c_1 \leq k \leq c_2$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F\rangle$ and $[c_1, c_2]$, we first compute $R =$ REACH$(\alpha, [c_1, c_2])$ via a breadth-first search. We then construct the following finite automaton $\alpha' = \langle Q', q_0', \Sigma, \delta', F'\rangle$, where

- $Q' = Q \cup \{q_0'\}$
- $\forall q, q' \in Q, \delta'(q, \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $\forall q \in R, q' \in Q, \delta'(q_0', \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $F'(q_0) =' +'$, if $\exists q \in R, F(q) =' +'$.
- $\forall q \in Q, F'(q) = F(q)$.

Note that α' constructed by the above construction may be a nondeterministic finite automaton. We add auxiliary bits to resolve nondeterminism as proposed in [58]. $\text{SUFFIX}(\alpha, [c_1, c_2])$ returns the result of the minimization and determinization of α'.

Boundary: Below we describe how to identify the boundary of a one-track arithmetic automaton, which accepts the binary encodings of a set of integer values from the least significant bit.

Property 4: For an one-track minimized DFA $a = \langle Q, q_0, B^1, \delta, F\rangle$: $\forall q, q' \in Q$, if $\delta(q, 0) = q'$, then $F(q) = F(q')$.

Property 4 states that transitions labelled by 0 cannot change accepting status, which holds due to the fact that by definition, the arithmetic automaton accepts a word and any number of 0 in its higher significant bits. It follows that for any accepted integer value (except 0), the word from the least significant bit up to the most non-zero significant bit of its binary encoding forms a unique path (ended by 1) from the initial state to an accepting state. Furthermore, an accepted non-zero minimal integer value forms the shortest path from the initial state to an accepting state. On the other hand, if there exists an accepted non-zero maximal integer value, the maximal value forms the longest loop-free path from the initial state to an accepting state. Note that if there exists an accepted path containing a loop, a accepts an infinite set and the maximal value does not exist. In this case, we use *inf* to denote the maximal value.

For `min`(a) and `max`(a), we have implemented two functions MIN(a) and MAX(a). Let m_s be the length of the shortest path that ends with 1 and m_l be the length of the longest loop-free path that ends with 1. Both m_s and m_l can be determined by a breadth first search up to $\sharp Q$ steps. In our implementation, we first check whether a accepts any non-zero integer value. If this is the case, MIN(a) returns 2^{m_s-1}, which is a lower bound for the shortest path. If there exists a path containing a loop, MAX(a) returns *inf*. Otherwise MAX(a) returns $2^{m_l+1} - 1$, which is an upper bound for the longest path. Note that our implementation is a conservative approximation. These bounds can be tightened by tracing the values along paths.

6.3 Experiments

We experimented with our composite analysis tool on a number of test cases extracted from C string library, buffer overflow benchmarks [36] and web vulnerability benchmarks [58]. These test cases are rather small but involve pointer arithmetic, string content constraints, length constraints, loops, and replacement operations. We manually convert them to our simple imperative language.

For `int strlen(char *`s`)`, we verify the invariant that the return value is equal to the length of the input string. For `char *strrchr(char *`s`, int` c`)`, we verify whether the language accepted by the return string is included in $\{cx \mid x \in \Sigma^*\} \cup \{\epsilon\}$

Test case (bad/ok)	Result	Time (s)	Memory (kb)
int strlen(char *s)	T	0.037	522
char *strrchr(char *s, int c)	T	0.011	360
gxine (CVE-2007-0406)	F/T	0.014/0.018	216/252
samba (CVE-2007-0453)	F/T	0.015/0.021	218/252
MyEasyMarket-4.1 (trans.php:218)	F/T	0.032/0.041	704/712
PBLguestbook-1.32 (pblguestbook.php:1210)	F/T	0.021/0.022	496/662
BloggIT 1.0 (admin.php:27)	F/T	0.719/0.721	5857/7067

Table 6.2: The Experimental Results of Composite Analysis.

upon reaching the fixpoint. For buffer overflow benchmarks, we check whether the identified memory may overflow its buffer upon reaching the fixpoint for both buggy (*bad*) and modified (*ok*) cases. For web vulnerability benchmarks, we check whether the identified sensitive function may take any attack string as its input before (*bad*) and after (*ok*) inserting limit constraints and sanitization routines. If it does not, the sensitive function is SQL attack free with respect to the attack pattern Σ^*`<script`Σ^*. Limit constraints are written as new statements that limit the length of string variables using a `$limit` variable. The experimental results are shown in Table 6.2, where "T" indicates buffer overflow free or SQL attack free. The results show that our composite analysis works well in these test cases in terms of both accuracy and performance. As a final remark, for web vulnerability benchmarks, one may restrict limit constraints, e.g., set `$limit` less than 7, to prevent the specified attacks without adding/modifying sanitization routines. In this case, pure string analysis [58] will raise false alarms.

Chapter 7

Relational String Analysis

Verification of string manipulation operations is a crucial problem in computer security. In this part, we present a new relational string analysis technique based on multi-track automata and abstraction. Our approach is capable of verifying properties that depend on relations among string variables. This enables us to prove that vulnerabilities that result from improper string manipulation do not exist in a given program.

In Chapter 2, we have formalized the string verification problem as reachability analysis of *string systems* and demonstrated that the string analysis problem is undecidable even for two binary variables or three unary variables with comparisons. In this chapter, we use multi-track deterministic finite automata (DFAs) as a symbolic representation to encode the set of possible values that string variables can take at a given program point. We apply the forward symbolic reachability analysis technique

that computes an over-approximation of the reachable states of a string system using widening and summarization. Unlike prior string analysis techniques, our analysis is *relational*; i.e., it is able to keep track of the relationships among the string variables, improving the precision of the string analysis and enabling verification of invariants such as $X_1 = X_2$ where X_1 and X_2 are string variables. We describe the precise construction of multi-track DFAs for linear word equations, such as $c_1 X_1 c_2 = c_1' X_2 c_2'$ and show that non-linear word equations (such as $X_1 = X_2 X_3$) cannot be characterized precisely as a multi-track DFA (Section 7.1). We propose a regular approximation for non-linear equations and show how these constructions can be used to compute the post-condition of branch conditions and assignment statements that involve concatenation. We use summarization for inter-procedural analysis (presented in Chapter 4) by generating a multi-track automaton (transducer) characterizing the relationship between the input parameters and the return values of each procedure. To be able to use procedure summaries during our reachability analysis we *align* multi-track automata so that normalized automata are closed under intersection.

To improve the efficiency of our approach, we propose two string abstraction techniques: alphabet and relation abstractions (Section 7.3). In alphabet abstraction, we identify a set of characters that we are interested in and use a special symbol to represent the rest of the characters. In relation abstraction, we identify the variables that are related and encode them as a single multi-track automata. For those that are not

related, we use multiple single-track automata to encode their values, where relations among them are abstracted away. We define an abstraction lattice that combines these abstractions under one framework and show that earlier results on string analysis can be mapped to several points in this abstraction lattice. We also extend our symbolic analysis technique by presenting algorithms for computing the post condition of complex string manipulation operations such as replacement. We implemented these algorithms using the MONA automata package [27] and analyzed several PHP programs demonstrating the effectiveness of our string analysis techniques.

7.1 Regular Approximation of Word Equations

To analyze string systems, we approximate configurations over string variables as a regular language accepted by a multi-track deterministic finite automaton (DFA). Our analysis is based on the facts that: (1) The transitions and the configurations of a string system can be symbolically represented using word equations with existential quantification, (2) Word equations can be represented/approximated using multi-track DFAs, which are closed under intersection, complement, projection, and (3) the operations required during reachability analysis (such as equivalence checking) can be computed on DFAs.

Before we discuss how to perform symbolic reachability analysis on string systems, we introduce the multi-track DFAs and word equations in this section. We characterize word equations that can be expressed using multi-track DFAs, as well as detail the construction of these multi-track DFAs. Using these constructions, in the next section, we show how to perform symbolic reachability analysis on string systems.

7.1.1 Aligned Multi-track DFAs

A multi-track DFA is a DFA over an alphabet that consists of many tracks. An n-track alphabet is defined as $\Sigma^n = (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\}) \times \ldots \times (\Sigma \cup \{\lambda\})$ (n times), where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ ($1 \leq i \leq n$) to denote the i^{th} track of $w \in \Sigma^n$. An *aligned* multi-track DFA is a multi-track DFA where all tracks are left justified (i.e., λ's are right justified). That is, if w is accepted by an aligned n-track DFA M, then for $1 \leq i \leq n$, $w[i] \in \Sigma^*\lambda^*$. We say $L(M)$ is an n-track language. We also use $\hat{w}[i] \in \Sigma^*$ to denote the longest λ-free prefix of $w[i]$. For the following descriptions, a multi-track DFA is an aligned multi-track DFA unless we explicitly state otherwise.

Multi-track DFAs are closed under intersection, disjunction, complementation, and homomorphism. Precisely: Given two n-track DFAs M_1, M_2, there exists an n-track DFA M that accepts $L(M_1) \cup L(M_2)$, or accepts $L(M_1) \cap L(M_2)$. Given an n-track DFA M_1, there exists an n-track DFA M that accepts the complement set of $L(M_1)$,

and also there exists an $(n-1)$-track DFA M' that accepts $L(M_1 \downarrow_i)$, where $M_1 \downarrow_i$ denotes the result of erasing the i^{th} track (by homomorphism) of M_1.

7.1.2 Word Equations

A word equation is an equality relation of two words that concatenate a finite set of variables $\mathbf{X}$ and a finite set of constants $\mathbf{C}$. The general form of word equations is defined as $v_1 \ldots v_n = v'_1 \ldots v'_m$, where $\forall i, v_i, v'_i \in \mathbf{X} \cup \mathbf{C}$.

Let f be a word equation over $\mathbf{X}$= $\{X_1, X_2, \ldots, X_n\}$, $f[c/X]$ denotes a new equation where X is replaced with c for all X that appears in f. We say that an n-track DFA M under-approximates f if for all $w \in L(M)$, $f[\hat{w}[1]/X_1, \ldots, \hat{w}[n]/X_n]$ holds. We say that an n-track DFA M over-approximates f if for any $s_1, \ldots, s_n \in \Sigma^*$ where $f[s_1/X_1, \ldots, s_n/X_n]$ holds, there exists $w \in L(M)$ such that for all $1 \leq i \leq n, \hat{w}[i] = s_i$. We call M precise with respect to f if M both under-approximates and over-approximates f.

Definition 8 *A word equation f is regular expressible if and only if there exists a multi-track DFA M such that M is precise with respect to f.*

Linear Word Equations: A linear word equation is a word equation where either side of the equation contains at most one variable. A general form of linear word equation is $c_1 X_1 c_2 = d_1 X_2 d_2$. Any linear word equation is equivalent to one of the following:

- $c'_1 X_1 c'_2 = X_2$ if $c_1 = d_1 c'_1$ and $c_2 = c'_2 d_2$,
- $c'_1 X_1 = X_2 d'_2$ if $c_1 = d_1 c'_1$ and $d'_2 c_2 = d_2$,
- $X_1 c'_2 = d'_1 X_2$ if $c_1 d'_1 = d_1$ and $c_2 = c'_2 d_2$,
- $X_1 = d'_1 X_2 d'_2$ if $c_1 d'_1 = d_1$ and $d'_2 c_2 = d_2$,
- *false* otherwise.

It follows that all linear equations can be reduced into two forms: (1) $X_1 = cX_2d$ or (2) $cX_1 = X_2d$, which are equivalent to $\exists X_k.X_1 = cX_k \wedge X_k = X_2d$ and $\exists X_k.cX_1 = X_k \wedge X_k = X_2d$.

Theorem 9 *Linear word equations and Boolean combinations of these equations can be expressed using equations of the form $X_1 = X_2c$ and $X_1 = cX_2$, Boolean combinations of such equations and existential quantification.*

Non-linear Word Equations: A non-linear word equation is a word equation where at least one side of the equation has at least two variables. There are two basic forms of non-linear equations: $c = X_1X_2$ and $X_1 = X_2X_3$.

Theorem 10 *Non-linear word equations and Boolean combinations of these equations can be expressed using equations of the form $c = X_1X_2$ and $X_1 = X_2X_3$, Boolean combinations of such equations and existential quantification.*

For example, $X_1 = X_2 d X_3 X_4$ is equivalent to $\exists X_{k_1}, X_{k_2}.X_1 = X_2 X_{k_1} \wedge X_{k_1} = dX_{k_2} \wedge X_{k_2} = X_3 X_4$.

In the following, we show how to construct the corresponding multi-track DFAs for the basic forms of linear and non-linear word equations: (1) $X_1 = X_2 c$, (2) $X_1 = cX_2$, (3) $c = X_1 X_2$, and (4) $X_1 = X_2 X_3$. Note that, based on the fact that multi-track DFAs are closed under intersection, disjunction, complementation, and homomorphism, we can construct the corresponding multi-track DFAs for all word equations both linear and non-linear, as well as their Boolean combinations based on the constructions for these basic forms. Note that the boolean operations conjunction, disjunction and negation can be handled with intersection, disjunction, and complementation of the multi-track automata, respectively. Existential quantification on the other hand, can be handled using homomorphism, where given a word equation f and a multi-track automaton M such that M is precise with respect to f, then the multi-track automaton $M \downarrow_i$ is precise with respect to $\exists X_i.f$.

Before delving into these constructions, we summarize our results in the following theorem:

Theorem 11 *(1) Linear word equations are regular expressible, as well as their Boolean combinations. (2) $X_1 = cX_2$ is regular expressible but the corresponding M has exponential number of states in the length of c. (3) $X_1 = X_2 X_3$ is not regular expressible.*

7.1.3 Construction of Multi-track DFAs for Word Equations

Given a DFA $M = \langle Q, \Sigma, \delta, I, F\rangle$, Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $I \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (accepting) states. We say a state $q \in Q$ is a $sink$ state if $q \notin F$ and $\forall a \in \Sigma, \delta(q, a) = q$. The $sink$ states are also extended to multi-track DFAs. In the following constructions, we ignore transitions that go to sink states, and assume that all unspecified transitions go to sink states.

Before we give the constructions, we generalize the problem of constructing multi-track DFAs for word equations as follows. We assume that each variable in $\mathbf{X}= \{X_1, X_2, \ldots, X_n\}$ is associated with an automaton $M_i = \langle Q_i, \Sigma, \delta_i, I_i, F_i\rangle$, where $L(M_i)$ denotes the set of values that the variable X_i can take. Then, given a word equation f over $\mathbf{X}= \{X_1, X_2, \ldots, X_n\}$, we say that *an n-track DFA M under-approximates f within $M_1, \ldots M_n$*, if for all $w \in L(M)$, $f[\hat{w}[1]/X_1, \ldots, \hat{w}[n]/X_n]$ holds and for all $1 \leq i \leq n$, $\hat{w}[i] \in L(M_i)$. We say that *an n-track DFA M over-approximates f within $M_1, \ldots M_n$*, if for any $s_1, \ldots, s_n \in \Sigma^*$ where $f[s_1/X_1, \ldots, s_n/X_n]$ holds and for all $1 \leq i \leq n$, $s_i \in L(M_i)$, there exists $w \in L(M)$ such that for all $1 \leq i \leq n, \hat{w}[i] = s_i$. Note that, for either case, for any word $w \in L(M)$, for all $1 \leq i \leq n, \hat{w}[i] \in L(M_i)$.

The Construction of $X_1 = X_2 c$: Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1\rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2\rangle$ be two DFAs that accept possible values of variables X_1 and X_2, respectively. We

present the construction of a 2-track DFA $M = \langle Q, \Sigma, \delta, I, F \rangle$, such that M is precise with respect to $X_1 = X_2 c$ within M_1, M_2.

Let $sink_1$ be the sink state of M_1, and $sink_2$ be the sink state of M_2. Let $c = a_1 a_2 \ldots a_n$, where $\forall 1 \leq i \leq n, a_i \in \Sigma$ and n is the length of the constant string c. $M = \langle Q, \Sigma^2, \delta, q_0, F \rangle$ is constructed as:

- $Q \subseteq Q_1 \times Q_2 \times \{0, \ldots, n\}$,
- $I = (I_1, I_2, 0)$,
- $\forall a \in \Sigma, \delta((r, p, 0), (a, a)) = (\delta_1(r, a), \delta_2(p, a), 0)$, if $\delta_1(r, a) \neq sink_1$ and $\delta_2(p, a) \neq sink_2$
- $\forall a_i, p \in F_2, \delta((r, p, i), (a_i, \lambda)) = (\delta_1(r, a_i), p, i + 1)$,
- $F = \{(r, p, i) \mid r \in F_1, p \in F_2, i = n\}$.

Note that M simulates M_1 and M_2 making sure that both tracks are the same until a final state of M_2 is reached. Then, the second track reads the symbol λ while the first track reads the constant c, and the automaton goes to a final state when c is consumed. $|Q|$ is $O(|Q_1| \times |Q_2| + n)$ since in the worst case Q will contain all possible combinations of states in Q_1 and Q_2 followed with a tail of n states for recognizing the constant c. For the automaton M resulting from the above construction we have, $w \in L(M)$ if and

only if $\hat{w}[1] = \hat{w}[2]c$, $\hat{w}[1] \in L(M_1)$ and $\hat{w}[2] \in L(M_2)$, i.e., M is precise with respect to $X_1 = X_2 c$ (within M_1, M_2), hence, $X_1 = X_2 c$ is regular expressible.

The Construction of $X_1 = cX_2$: Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1 \rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2 \rangle$ be two DFAs that accept possible values of variables X_1 and X_2, respectively. Below we present the construction of a 2-track DFA M, such that M is precise with respect to $X_1 = cX_2$ within M_1, M_2. Let $c = a_1 a_2 \ldots a_n$, where $\forall 1 \leq i \leq n, a_i \in \Sigma$ and n is the length of the constant string c.

The intuition behind the construction of M is as follows. In the initial stage (denoted as *init* below), M makes sure that the first track matches the constant c, while recording the string that is read in the second track in a buffer (a vector of symbols) stored in its state. After c is consumed, M goes to the next stage (denoted as *match* below) and matches the symbols read in the first track with the next symbol stored in the buffer while continuing to store the symbols read in the second track in the buffer. Note that, the kth symbol read in track 2 has to be matched with the $(k+n)$th symbol read in track 1. So, the buffer stores the symbols read in track 2 until the corresponding symbol in track 1 is observed.

Let $\vec{v}$ be a size n vector. For $1 \leq i \leq n, \vec{v}[i] \in \Sigma \cup \{ \perp \}$. The vector $\vec{v}' = \vec{v}[i := a]$ is defined as follows: $\vec{v}'[i] = a$ and $\forall j \neq i$, $\vec{v}'[j] = \vec{v}[j]$. $M = \langle Q, \Sigma^2, \delta, I, F \rangle$ is constructed as:

- $Q \subseteq Q_1 \times Q_2 \times \{1, \ldots, n\} \times (\Sigma \cup \{\perp\})^n \times \{init, match\}$,

- $I = (I_1, I_2, 1, \vec{v}_\perp, init)$, where $\forall i, \vec{v}_\perp[i] = \perp$,
- $\forall a \in \Sigma, 1 \leq i < n, \delta((r, p, i, \vec{v}, init), (a_i, a)) = (\delta_1(r, a_i), \delta_2(p, a), i + 1, \vec{v}[i := a], init)$,
- $\forall a \in \Sigma, i = n, \delta((r, p, i, \vec{v}, init), (a_i, a)) = (\delta_1(r, a_i), \delta_2(p, a), 1, \vec{v}[i := a], match)$,
- $\forall a, b \in \Sigma, 1 \leq i < n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, b)) = (\delta_1(r, a), \delta(p, b), i+1, \vec{v}[i := b], match)$,
- $\forall a, b \in \Sigma, i = n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, b)) = (\delta_1(r, a), \delta(p, b), 1, \vec{v}[i := b], match)$,
- $\forall a \in \Sigma, p \in F_2, 1 \leq i < n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, \lambda)) = (\delta_1(r, a), p, i+1, \vec{v}[i := \perp], match)$,
- $\forall a \in \Sigma, p \in F_2, i = n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, \lambda)) = (\delta_1(r, a), p, 1, \vec{v}[i := \perp], match)$,
- $F = \{(r, p, i, \vec{v}_\perp, match) \mid r \in F_1, p \in F_2\}$.

Since M accepts the set $\{w \mid \hat{w}[1] = c\hat{w}[2], \hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2)\}$, $X_1 = cX_2$ is regular expressible. However, the number of states of M is exponential in c. Below, we show that the exponential number of states is inevitable.

Intractability of $X_1 = cX_2$: Consider the equation $X_1 = cX_2$, where c is a constant string of length n. Let $L(M_1)$ and $L(M_2)$ be regular languages. Define the 2-track language:

$$L = \{(x_1x_2, y_1y_2\lambda^n) \mid x_1x_2 \in L(M_1), y_1y_2 \in L(M_2), k \geq n, |x_1x_2| = k, |x_1| = |y_1| = n, x_1 = c, x_2 = y_1y_2\}$$

Note that any automaton m that accepts the language L defined above will be precise with respect to the the equation $X_1 = cX_2$ (within M_1 and M_2).

Theorem 12 *Any nondeterministic finite automaton (NFA) M needs at least 2^n states to accept L.*

Proof 5 *Let $c = 1^n$ and consider the regular languages $L(M_1) = (0+1)^+$ and $L(M_2) = (0+1)^+$. Suppose M is an NFA accepting L. Consider any pair of distinct strings y_1 and y_1' of length n. Then M will accept the following 2-track strings:*

$(1^nx_2, y_1y_2\lambda^n)$, *where* $x_2, y_1, y_2 \in (0+1)^+$, $k \geq n$, $|1^nx_2| = k, |y_1| = n, x_2 = y_1y_2$, *and*

$(1^n x_2', y_1' y_2' \lambda^n)$, *where* $x_2', y_1', y_2' \in (0+1)^+$, $k \geq n$, $|1^n x_2'| = k, |y_1'| = n, x_2' = y_1' y_2'$

Suppose in processing $(1^n x_2, y_1 y_2 \lambda^n)$, M *enters state* q *after processing the initial 2-track segment* $(1^n, y_1)$, *and in processing* $(1^n x_2', y_1' y_2' \lambda^n)$, M *enters state* q' *after processing the initial 2-track segment* $(1^n, y_1')$. *Then* $q \neq q'$*; otherwise,* M *will also accept* $(1^n x_2, y_1' y_2 \lambda^n)$. *This is a contradiction, since* $x_2 \neq y_1' y_2$.

Since there are 2^n *distinct strings* y *of length* n, *it follows that* M *must have at least* 2^n *states.*

The Construction of $c = X_1 X_2$**:** Below we briefly describe the construction of a 2-track DFA M, such that M is precise with respect to $c = X_1 X_2$ within the given regular sets characterizing possible values of X_1 and X_2. Assume that $c = a_1 \ldots a_n$. We can split c to two strings $a_1 \ldots a_k$ and $a_{k+1} \ldots a_n$ so that $c = a_1 \ldots a_k a_{k+1} \ldots a_n$. There are $n+1$ such splits. For each of them, if $a_1 \ldots a_k \in L(M_1)$ and $a_{k+1} \ldots a_n \in L(M_2)$, then if $k \geq n - k$, $(a_1 \ldots a_k, a_{k+1} \ldots a_n \lambda^{2k-n})$ should be accepted by M and if $k < n - k$, $(a_1 \ldots a_k \lambda^{n-2k}, a_{k+1} \ldots a_n)$ should be accepted by M. We can construct an automaton M with $O(n^2)$ states that accepts this language by explicitly checking each of these

$n + 1$ cases. Since we can construct this 2-track DFA, it follows that $c = X_1X_2$ is regular expressible.

Non-Regularity of $X_1 = X_2X_3$: We first show that $X_1 = X_2X_3$ is not regular expressible, and later we give constructions of 3-track DFAs that over-approximate or under-approximate $X_1 = X_2X_3$.

Given M_1, M_2, M_3, let $L = \{w \mid \hat{w}[1] = \hat{w}[2]\hat{w}[3], \hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2), \hat{w}[3] \in L(M_3)\}$.

Theorem 13 *L is not necessarily a regular language.*

Proof 6 *Let $L(M_1) = a^+b^+$, $L(M_2) = a^+$, and $L(M_3) = b^+$. Suppose L is regular and is accepted by a 3-track DFA M. Then M when given a 3-track string consisting of:*

a^sb^t

$a^i\lambda^{s+t-i}$

$b^j\lambda^{s+t-j}$

accepts if and only if $s = i$ and $t = j$. Clearly, we can construct a 3-track DFA M' which accepts 3-track strings of the form:

a^sb^t

$a^i\lambda^{s+t-i}$

$b^i\lambda^{s+t-i}$

We can then construct another 3-track DFA M'' which accepts $L(M) \cap L(M')$. But $L(M'')$ consists of 3-track strings of the form:

$a^i b^i$

$a^i \lambda^{s+t-i}$

$b^i \lambda^{s+t-i}$

It follows that we can construct a 1-track NFA from M'' which accepts the language $\{a^i b^i \mid i \geq 1\}$ (by erasing the second and third tracks by homomorphism), which is not regular and leads to a contradiction.

The Approximation of $X_1 = X_2 X_3$: Below we propose an over approximation construction for $X_1 = X_2 X_3$. Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1 \rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2 \rangle$, and $M_3 = \langle Q_3, \Sigma, \delta_3, I_3, F_3 \rangle$ accept values of X_1, X_2, and X_3 respectively. $M = \langle Q, \Sigma^3, \delta, I, F \rangle$ is constructed as follows.

- $Q \subseteq Q_1 \times Q_2 \times Q_3 \times Q_3$,
- $I = (I_1, I_2, I_3, I_3)$,
- $\forall a, b \in \Sigma$, $\delta((r, p, s, s'), (a, a, b)) = (\delta_1(r, a), \delta_2(p, a), \delta_3(s, b), s')$,
- $\forall a, b \in \Sigma$, $p \in F_2, s \notin F_3, \delta((r, p, s, s'), (a, \lambda, b)) = (\delta_1(r, a), p, \delta_3(s, b), \delta_3(s', a))$,
- $\forall a \in \Sigma$, $p \in F_2, s \in F_3, \delta((r, p, s, s'), (a, \lambda, \lambda)) = (\delta_1(r, a), p, s, \delta_3(s', a))$,
- $\forall a \in \Sigma$, $p \notin F_2, s \in F_3, \delta((r, p, s, s'), (a, a, \lambda)) = (\delta_1(r, a), \delta_2(p, a), s, s')$,

- $F = \{(r, p, s, s') \mid r \in F_1, p \in F_2, s \in F_3, s' \in F_3\}$.

$|Q|$ is $O(|Q_1| \times |Q_2| \times |Q_3| + |Q_1| \times |Q_3| \times |Q_3|)$. For all $w \in L(M)$, the following properties hold:

- $\hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2), \hat{w}[3] \in L(M_3)$,
- $\hat{w}[1] = \hat{w}[2]w'$ and $w' \in L(M_3)$,

Note that w' may not be equal to $\hat{w}[3]$ (i.e., there exists $w \in L(M)$, $\hat{w}[1] \neq \hat{w}[2]\hat{w}[3]$), and hence M is not precise with respect to $X_1 = X_2X_3$. On the other hand, for any w such that $\hat{w}[1] = \hat{w}[2]\hat{w}[3]$, we have $w \in L(M)$, hence M is a regular *over*-approximation of $X_1 = X_2X_3$.

Below, we show a regular *under*-approximation construction of $X_1 = X_2X_3$. Note that if $L(M_2)$ is a finite set language, one can construct the DFA M that satisfies $X_1 = X_2X_3$ by explicitly taking the union of the construction of $X_1 = cX_3$ for all $c \in L(M_2)$. If $L(M_2)$ is an infinite set language, we can still use this idea to construct a regular *under*-approximation of $X_1 = X_2X_3$ by considering a (finite) subset of $L(M_2)$ where the length is bounded. Formally speaking, for each $k \geq 0$ we can construct M_k, so that $w \in L(M_k), \hat{w}[1] = \hat{w}[2]\hat{w}[3]$, $\hat{w}[1] \in L(M_1)$, $\hat{w}[3] \in L(M_3)$, $\hat{w}[2] \in L(M_2)$ and $|\hat{w}[2]| \leq k$. It follows that M_k is a regular *under*-approximation of $X_1 = X_2X_3$. The following lemma holds by construction.

Lemma 14 *$L(M_{k_1}) \subseteq L(M_{k_2})$ if $k_1 \leq k_2$.*

To sum up, if $L(M_2)$ is a finite set language, there exists k (the length of the longest accepted word) so that $L(M_k)$ is precise with respect to $X_1 = X_2X_3$. If $L(M_2)$ is an infinite set language, there does not exist such k so that $L(M_k)$ is precise with respect to $X_1 = X_2X_3$, as we have proven non-regularity of $X_1 = X_2X_3$.

We say a regular under-approximation M_κ is *tightest* if $L(M_\kappa)$ is an under-approximation of $X_1 = X_2X_3$ and for all M' where M' is an under-approximation of $X_1 = X_2X_3$ we have $L(M') \subseteq L(M_\kappa)$. Since the precision of a regular under-approximation can be always improved by adding new words to the language, the tightest regular under-approximation does not exist if $L(M_2)$ is not finite.

7.2 Symbolic Reachability Analysis on Multi-track Automata

Our symbolic reachability analysis involves two main steps: forward fixpoint computation and summarization.

7.2.1 Forward Fixpoint Computation

The first phase of our analysis is a standard forward fixpoint computation on multi-tack DFAs. Each program point is associated with a single multi-track DFA, where each track is associated with a single string variable $X \in \mathbf{X}$. We use $M[l]$ to denote

the multi-track automaton at the program label l. The forward fixpoint computation algorithm is a standard work-queue algorithm. Initially, for all labels l, $L(M[l]) = \emptyset$. We iteratively compute the post-images of the statements and join the results to the corresponding automata. For a $stmt$ in the form: X:= $sexp$, the post-image is computed as:

$$\text{POST}(M, stmt) \equiv (\exists X.M \cap \text{CONSTRUCT}(X' = sexp, +))[X/X'].$$

CONSTRUCT(exp, b) returns the DFA that accepts a regular approximation of exp, where $b \in \{+, -\}$ indicates the direction (*over* or *under*, respectively) of approximation if needed. During the construction, we recursively push the negations ($\neg$) (and flip the direction) inside to the basic expressions ($bexp$), and use the corresponding construction of multi-track DFAs discussed in the previous section. We use function summaries to handle function calls. Each function f is summarized as a finite state transducer, denoted as M_f, which captures the relations among input variables (parameters), denoted as X_p, and return values. The return values are tracked in the output track, denoted as X_o. We discuss the generation of the transducer M_f below. For a $stmt$ in the form X:= `call` $f(e_1, \ldots, e_n)$, the post-image is computed as:

$$\text{POST}(M, stmt) \equiv (\exists X, X_{p_1}, \ldots X_{p_n}.M \cap M_I \cap M_f)[X/X_o],$$

where M_I = CONSTRUCT($\bigwedge_i X_{p_i} = e_i$, $+$). The process terminates when we reach a fixpoint. To accelerate the fixpoint computation, we extend our automata widening operator [58], denoted as ∇, to multi-track automata. We identify equivalence classes according to specific equivalence conditions and merge states in the same equivalence class [4, 7]. The following lemma shows that the equality relations among tracks are preserved while widening multi-track automata.

Lemma 15 *if* $L(M) \subseteq L(x = y)$ *and* $L(M') \subseteq L(x = y)$, $L(M \nabla M') \subseteq L(x = y)$.

Algorithm 9 CONSTRUCT(exp, b)

```
1: if exp is exp1 ∧ exp2 then
2:     return CONSTRUCT(exp1, b) ∩ CONSTRUCT(exp2, b);
3: else if exp is ¬(exp1 ∧ exp2) then
4:     return CONSTRUCT(¬exp1, b) ∪ CONSTRUCT(¬exp2, b);
5: else if exp is ¬(¬exp1) then
6:     return CONSTRUCT(exp1, b);
7: else if exp is bexp then
8:     return CONSTRUCT(bexp, b);
9: else if exp is ¬bexp then
10:    return COMPLEMENT(CONSTRUCT(bexp, b̄));
11: end if
```

For a $stmt$ in the form: X:= $sexp$, the post-image is computed as follows:

$$\text{POST}(M, stmt) \equiv (\exists X.M \cap \text{CONSTRUCT}(X' = sexp, +))[X/X'].$$

We use function summaries to handle function calls. Each function f is summarized as a finite state transducer, denoted as M_f, which captures the relations among input

variables (parameters), denoted as X_p, and return values. The return values are tracked in the output track, denoted as X_o. For a $stmt$ in the form X:= `call` $f(e_1, \ldots, e_n)$, POST(M, $stmt$) returns the result of $(\exists X, X_{p_1}, \ldots X_{p_n}.M \cap M_I \cap M_f)[X/X_o]$, where $M_I = \text{CONSTRUCT}(\bigwedge_i X_{p_i} = e_i, +)$.

During the fixpoint computation, we report assertion failures if $M[l]$ accepts some string that violates the assertion labeled l. Note that at line 21 we compute an under approximation of the assertion expression to ensure the soundness of our analysis. Finally, a program label l is not reachable if $L(M[l])$ is empty. Our analysis is sound but incomplete due to the following approximations: (1) regular approximation for non-linear word equations, (2) the widening operation and (3) summarization.

Algorithm 10 PROPAGATE(m, l)

```
1: m' := (m ∪ M[l])∇M[l];
2: if m' ⊈ M[l] then
3:     M[l] := m';
4:     WQ.enqueue(l);
5: end if
```

7.2.2 Summarization

Similar to analysis on single track automata, we compute procedure summaries in order to handle procedure calls. We assume parameter-passing with call-by-value semantics and we are able to handle recursion. Each function f is summarized as a

Algorithm 11 FORWARDRECAHABILITYANALYSIS(l_0)

```
1:  Init(M);
2:  queue WQ;
3:  WQ.enqueue(l_0 : stmt_0);
4:  while WQ ≠ NULL do
5:     e := WQ.dequeue(); Let e be l : stmt;
6:     if stmt is seqstmt then
7:        m := POST(M[l], stmt);
8:        PROPAGATE(m, l + 1);
9:     end if
10:    if stmt is if exp goto l' then
11:       m := M[l]∩ CONSTRUCT(exp, +);
12:       if L(m) ≠ ∅ then
13:          PROPAGATE(m, l');
14:       end if
15:       m := M[l]∩ CONSTRUCT(¬exp, +);
16:       if L(m) ≠ ∅ then
17:          PROPAGATE(m, l + 1);
18:       end if
19:    end if
20:    if stmt is assert exp then
21:       m := CONSTRUCT(exp, −);
22:       if L(M[l]) ⊈ L(m) then
23:          ASSERTFAILED(l);
24:       else
25:          PROPAGATE(M[l],l + 1);
26:       end if
27:    end if
28:    if stmt is goto L then
29:       for l' ∈ L do
30:          PROPAGATE(M[l],l');
31:       end for
32:    end if
33: end while
```

multi-track DFA, denoted as M_f, that captures the relation among its input variables and return values.

During the summarization phase, (possibly recursive) functions are summarized as unaligned multi-track DFAs that specify the relations among their inputs and return values. The summarization approach has been discussed in Chapter 4. Briefly, we first build (possibly cyclic) dependency graphs to specify how the inputs flow to the return values. Each node in the dependency graph is associated with an unaligned multi-track DFA that traces the relation among inputs and the value of that node. An unaligned multi-track DFA is a multi-track DFA where λs might not be right justified. Return values of a function are represented with an auxiliary output track. Given a function f with n parameters, M_f is an unaligned $(n+1)$-track DFA, where n tracks represent the n input parameters and one track X_o is the output track representing the return values. We iteratively compute post images of reachable relations and join the results until we reach a fixpoint. Upon termination, the summary is the union of the unaligned DFAs associated with the return nodes. The main difference is that to compose these summaries at the call site, we also propose an alignment algorithm to *align* (so that λ's are right justified) an unaligned multi-track DFA. In Section 7.2.3, we discuss some theoretical results about alignment problems and propose an approximate algorithm to align unaligned multi-track automata.

Once the summary DFA M_f has been computed, it is not necessary to reanalyze the body of f. To compute the post-image of a call to f we intersect the values of input parameters with M_f and use existential quantification to obtain the return values. Let M be a one-track DFA associated with X where $L(M) = \{b\}$. POST(M, $X :=$ `call` $f(X)$) returns M' where $L(M') = ba^*$ for the example function shown above. As another example, let M be a 2-track DFA associated with X, Y that is precise with respect to $X = Y$. Then POST(M, $X :=$ `call` $f(X)$) returns M' which is precise with respect to $X = Y.a^*$ precisely capturing the relation between X and Y after the execution of the function call. As discussed above, M' is computed by $(\exists X, X_{p_1}.M \cap M_I \cap M_f)[X/X_o]$, where $L(M_I) =$ CONSTRUCT($X_{p_1} = X$, $+$).

7.2.3 Alignment

In general, $M_{\mathbf{f}}$ can be an unaligned multi-track DFA (λs are not right justified) and needed to be *aligned* before composition. Theorem 16 shows that an unaligned multi-track DFA may not be definable by an aligned multi-track DFA.

Theorem 16 *For any $n \geq 2$, there exists a language L accepted by an unaligned n-track DFA M that cannot be converted to any aligned DFA M'.*

Proof 7 *Let $L = \{(a\lambda)^i(cc)^k \mid i, k \geq 1\}$. Clearly, L can be accepted by an unaligned 2-track DFA M. Suppose we can convert M to an aligned 2-track DFA M'. Let M'*

have s states. Consider the string $w = (ac)^s(c\lambda)^s$. Then w is accepted by M'. Then there exist $i, k \geq 0$ and $j \geq 1$ such that w decomposes into $w = (ac)^i(ac)^j(ac)^k(c\lambda)^s$, where $i+j+k = s$, and $(ac)^i(ac)^{mj}(ac)^k(c\lambda)^s$ is accepted by M' for every $m \geq 0$. Let $m = 2$. Then $w' = (ac)^i(ac)^{2j}(ac)^k(c\lambda)^s$ is accepted by M'. But now, the first track of w' contains the string $a^{s+j}c^s$, and the second track contains c^{s+j}. Since $j \geq 1$, this is a contradiction since the number of c's in the first track is less than the number of c's in the second track.

Given an unaligned multi-track DFA M and a bound k, we construct M' that accepts an *over* or *under* approximation of $L(M)$ based on k. The construction is shown in Appendix. The basic idea is to associate a bounded FIFO queue (up to size k) with the states of M' to record the symbols seen on the track that is being aligned when a transition that contains the symbol λ for that track is taken. Later, when a non-λ symbol is seen on that track, it has to match the symbol that is at the head of the queue if the queue is not empty. During the construction, if no queue exceeds size k, then we say M is *k-alignable*, and the construction returns the precise aligned M' such that $L(M') = L(M)$. If M is not *k-alignable*, the *under*-approximation construction rejects all words that cause a queue to exceed k and returns an M' such that $L(M') \subseteq L(M)$, while the *over*-approximation construction accepts those words that partially match the

contents of the queue (up to size k) and returns an M' such that $L(M) \subseteq L(M')$. The precision improves when we increase k.

Our approximate construction: We propose an approximate *k-alignment* construction. Given an unaligned multi-track DFA M and a bound k, we construct M' that accepts an *over* or *under* approximation of $L(M)$ based on k.

Let $M = \langle Q, \Sigma^n, \delta, I, F \rangle$ and $\Sigma^n \subseteq (\Sigma \cup \{\lambda\}) \times \ldots \times (\Sigma \cup \{\lambda\})$. For $\alpha \in \Sigma^n$, $\alpha[i] \in \Sigma \cup \{\lambda\}$ denotes the i^{th} character of α and $\alpha[i := a]$ denotes $\alpha' \in \Sigma^n$ such that $\alpha'[i] = a$ and $\forall i \neq j, \alpha'[j] = \alpha[j]$. We align one track of M at a time. To align M completely, we iteratively align each track. Given a bound k and a track i, we construct M' such that the track i is aligned in M'. We assume that there is a sink state and all unspecified transitions go to the sink state. Let $\varrho_\perp$ be an empty queue and $*$ denote $+$ or $-$. We construct $M' = \langle Q', \Sigma^n, \delta', I', F' \rangle$ as follows:

- $Q' \subseteq Q \times Q_{queue}$, where $Q_{queue} \subseteq \{+, -\} \times \Sigma^k$.
- $I' = (I, (+, \varrho_\perp))$.
- $F' = \{(q, (*, \varrho_\perp)) \mid q \in F)\}$

For each $\delta(q, \alpha) = q'$,

- if $\alpha[i] \in \Sigma$,
  - $\delta'((q, (*, \varrho_\perp)), \alpha) = (q', (-, \varrho_\perp))$,

  - $\delta'((q, (*, \varrho)), \alpha[i := \lambda]) = (q', (-, \varrho'))$, if $\alpha[i] = \varrho$.head and $\varrho' = \varrho$.dequeue.

- if $\alpha[i] = \lambda$,

  - $\delta'((q, (-, \varrho_\perp)), \alpha) = (q', (-, \varrho_\perp))$,

  - $\delta'((q, (+, \varrho)), \alpha) = (q', (-, \varrho))$,

  - $\forall a \in \Sigma_\varrho$, $\varrho' = \varrho$.enqueue(a) and $|\varrho'| \leq k$, $\delta'((q, (+, \varrho)), \alpha[i := a]) = (q', (+, \varrho'))$.

$\Sigma_\varrho \subseteq \Sigma$ is the set of characters that can be reached in track i after seeing the sequence of symbols stored in ϱ. Precisely, let $M_i = \langle Q_i, \Sigma, \delta, I_i, F_i \rangle$ accept $\{\hat{w}[i] \mid w \in L(M)\}$, then $\Sigma_\varrho = \{a \mid q' \neq sink, \delta_i(I, \varrho a) = q'\}$. Using Σ_ϱ (instead of Σ) prevents the construction from adding useless states that will end up transitioning to the sink state.

The above construction returns an *under*-approximation if M is not *k-alignable*. To return an *over*-approximation, we make the following modifications. We first add two extra states to the queue, $\{e, e'\}$, to denote that the queue capacity has been exceeded. After the queue capacity is exceeded, we will stop enqueuing symbols to the queue when we see λ. We continue to match and dequeue when we see $a \in \Sigma$ until the queue is empty. In both cases, we can output arbitrary character $a \in \Sigma$ or λ (e), but once we output λ, we can only output λ thereafter (e').

For each $\delta(q, \alpha) = q'$,

- if $\alpha[i] \in \Sigma$,

  - $\delta'((q, (\{e, e'\}, \varrho_\perp)), \alpha[i := \lambda]) = (q', (e', \varrho_\perp))$,
  - $\forall a \in \Sigma, \delta'((q, (e, \varrho_\perp)), \alpha[i := a]) = (q', (e, \varrho_\perp))$,
  - if $\alpha[i] = \varrho$.head and $\varrho' = \varrho$.dequeue.
    - $\delta'((q, (\{e, e'\}, \varrho)), \alpha[i := \lambda]) = (q', (e', \varrho')$,
    - $\forall a \in \Sigma, \delta'((q, (e, \varrho)), \alpha[i := a]) = (q', (e, \varrho'))$,

- if $\alpha[i] = \lambda$,
  - if $|\varrho| = k$,
    - $\forall a \in \Sigma, \delta'((q, (+, \varrho)), \alpha[i := a]) = (q', (e, \varrho))$,
    - $\delta'((q, (+, \varrho)), \alpha) = (q', (e', \varrho))$,
  - $\delta'((q, (\{e, e'\}, \varrho_\perp)), \alpha) = (q', (e', \varrho_\perp))$,
  - $\forall a \in \Sigma, \delta'((q, (e, \varrho)), \alpha[i := a]) = (q', (e, \varrho))$.

7.3 String Abstractions

We present two string abstraction techniques, *alphabet abstraction* and *relation abstraction*, and show that they can be combined to form different abstraction classes with different levels of precision.

7.3.1 Alphabet Abstraction

Let Σ, a finite alphabet, be the concrete alphabet, and $\star \notin \Sigma$ be a special symbol to represent characters that are abstracted away. An abstract alphabet of Σ is defined as $\Sigma' \cup \{\star\}$, where $\Sigma' \subset \Sigma$. The concrete alphabet Σ and its abstract alphabets form a complete lattice, denoted as L_Σ, of which the bottom, denoted as $\sigma_\perp$, is $\{\star\}$ and the top, denoted as $\sigma_\top$, is Σ. The partial order of L_Σ is defined as follows. Let σ_1, σ_2 be two elements in L_Σ. We say

$$\sigma_1 \sqsubseteq \sigma_2, \text{ if } (\sigma_1 \setminus \star) \subseteq (\sigma_2 \setminus \star), \quad \text{and} \quad \sigma_1 \sqsubset \sigma_2, \text{ if } \sigma_1 \sqsubseteq \sigma_2 \text{ and } \sigma_1 \neq \sigma_2.$$

Let $\sigma_2 \sqsubset \sigma_1$. An alphabet abstraction function over σ_1, σ_2, denoted as $\alpha_{\sigma_1,\sigma_2}$ is a function from σ_1 to σ_2, such that for any $a \in \sigma_1$, $\alpha_{\sigma_1,\sigma_2}(a) = a$ if $a \in \sigma_2$; $\alpha_{\sigma_1,\sigma_2}(a) = \star$, otherwise. An alphabet concretization function over σ_1, σ_2, denoted as $\gamma_{\sigma_1,\sigma_2}$, is a function from σ_2 to σ_1, such that for any $a \in \sigma_2$, $\gamma_{\sigma_1,\sigma_2}(a) = a$ if $a \neq \star$; otherwise there exists a c where $\gamma_{\sigma_1,\sigma_2}(a) = c$ and $c \in \sigma_1 \setminus (\sigma_2 \setminus \{\star\})$.

An alphabet transducer over σ_1 and σ_2 is a 2-track DFA $M_{\sigma_1,\sigma_2} = \langle Q, \sigma_1 \times \sigma_2, \delta, q_0, F \rangle$, where

- $Q = \{q_0, sink\}$, $F = \{q_0\}$, and
- $\forall a \in \sigma_2, \delta(q_0, (a, a)) = q_0$,
- $\forall a \in \sigma_1 \setminus \sigma_2, \delta(q_0, (a, \star)) = q_0$.

Let M be a single track DFA over σ_1 with track X. $M_{\sigma_1,\sigma_2}(X, X')$ denotes the alphabet transducer over σ_1 and σ_2 where X and X' correspond to the input and output tracks, respectively. We define the abstraction and concretization functions on automata as:

- $\alpha_{\sigma_1,\sigma_2}(M) \equiv (\exists X.M \cap M_{\sigma_1,\sigma_2}(X, X'))[X/X']$, and
- $\gamma_{\sigma_1,\sigma_2}(M) \equiv \exists X'.(M[X'/X] \cap M_{\sigma_1,\sigma_2}(X, X'))$.

The definition can be extended to multi-track DFAs. Let M be a multi-track DFA over σ_1^n associated with $\{X_i | 1 \leq i \leq n\}$. $\alpha_{\sigma_1,\sigma_2}(M)$ returns a multi-track DFA over σ_2^n. On the other hand, while M is a multi-track DFA over σ_2^n, $\gamma_{\sigma_1,\sigma_2}(M)$ returns a multi-track DFA over σ_1^n. We add $\delta(q_0, (\lambda, \lambda)) = q_0$ to M_{σ_1,σ_2} to deal with the padding symbol λ. We use $M_{\sigma_1,\sigma_2}(X_i, X_i')$ to denote the alphabet transducer associated with tracks X_i and X_i'. The abstraction and concretization of a multi-track DFA M is done track by track as follows:

- $\alpha_{\sigma_1,\sigma_2}(M) \equiv \forall X_i.(\exists X_i.M \cap M_{\sigma_1,\sigma_2}(X_i, X_i'))[X_i/X_i']$, and
- $\gamma_{\sigma_1,\sigma_2}(M) \equiv \forall X_i.(\exists X_i'.M[X_i'/X_i] \cap M_{\sigma_1,\sigma_2}(X_i, X_i'))$.

7.3.2 Relation Abstraction

Let $\mathbf{X} = \{X_1, \ldots X_n\}$ be a finite set of variables. Let $\chi \subseteq 2^{\mathbf{X}}$ where $\emptyset \notin \chi$. We say χ defines a relation of $\mathbf{X}$ if (1) for any $\mathbf{x}, \mathbf{x}' \in \chi$, $\mathbf{x} \not\subseteq \mathbf{x}'$, and (2) $\bigcup_{\mathbf{x} \in \chi} \mathbf{x} = \mathbf{X}$. The set

of χ that defines the relations of $\mathbf{X}$ form a complete lattice, denoted as $L_{\mathbf{X}}$, of which the bottom, denoted as $\chi_\perp$, is $\{\{X_1\}, \{X_2\}, \{X_3\}, \ldots . \{X_n\}\}$ (which corresponds to the case where, for each program point, n single-track automata are used and each automaton represents the set of values for a single string variable) and the top, denoted as $\chi_\top$, is $\{\{X_1, X_2, \ldots, X_n\}\}$ (which corresponds to the case where, for each program point, a single multi-track automaton is used to represent the set of values for all string variables where each string variable corresponds to one track). The partial order of $L_{\mathbf{X}}$ is defined as follows: Let χ_1, χ_2 be two elements in $L_{\mathbf{X}}$. We say

- $\chi_2 \sqsubseteq \chi_1$, if for any $\mathbf{x} \in \chi_2$, there exists $\mathbf{x}' \in \chi_1$ such that $\mathbf{x} \subseteq \mathbf{x}'$.
- $\chi_2 \sqsubset \chi_1$ if $\chi_2 \sqsubseteq \chi_1$ and $\chi_1 \neq \chi_2$.

Let $\chi_2 \sqsubset \chi_1$. A relation abstraction function over χ_1, χ_2, denoted as α_{χ_1,χ_2}, is a function from χ_1 to χ_2, such that for any $\mathbf{x} \in \chi_1$, $\alpha(\mathbf{x}) = \mathbf{x}'$, where $\mathbf{x}' \subseteq \mathbf{x}$ and $\mathbf{x}' \in \chi_2$. A relation concretization function over χ_1, χ_2, denoted as γ_{χ_1,χ_2}, is a function from χ_2 to χ_1, such that for any $\mathbf{x}' \in \chi_2$, $\gamma_{\chi_1,\chi_2}(\mathbf{x}') = \mathbf{x}$, where $\mathbf{x}' \subseteq \mathbf{x}, \mathbf{x} \in \chi_1$.

For each $\mathbf{x} \in \chi$, we use a $|\mathbf{x}|$-track DFA, denoted as $M_{\mathbf{x}}$, where each track is associated with a variable in $\mathbf{x}$. For $\mathbf{x}' \subseteq \mathbf{x}$, $M_{\mathbf{x}} \downarrow_{\mathbf{x}'}$ is defined as the $|x'|$-track DFA that accepts $\{w' \mid w \in L(M_{\mathbf{x}}), \forall X_i \in \mathbf{x}', w'[i] = w[i])\}$, and $M_{\mathbf{x}'} \uparrow_{\mathbf{x}}$ is defined as the $|x|$-track DFA that accepts $\{w \mid w' \in L(M_{\mathbf{x}'}), \forall X_i \in \mathbf{x}', w[i] = w'[i])\}$.

Let $\mathbf{M}_\chi = \{M_\mathbf{x} \mid \mathbf{x} \in \chi\}$ be the set of DFAs of χ. The set of string values represented by $\mathbf{M}_\chi$ is defined as: $L(\bigcap_{\mathbf{x}\in\chi} M_\mathbf{x} \uparrow_{\mathbf{x_u}})$, where $\mathbf{x_u} = \{X_1, X_2, \ldots, X_n\}$. That is, we extend the language of every automaton in $\mathbf{M}_\chi$ to all string variables and then take their intersection.

Let $\chi_2 \sqsubset \chi_1$. $\alpha_{\chi_1,\chi_2}(\mathbf{M}_{\chi_1})$ returns an instance of the set of DFAs of χ_2, i.e., $\{M_{\mathbf{x}'}|\mathbf{x}' \in \chi_2\}$, where for each $\mathbf{x}' \in \chi_2$, $M_{\mathbf{x}'} = (\bigcap_{\mathbf{x}\in\chi_1,\mathbf{x}'\cap\mathbf{x}\neq\emptyset} M_\mathbf{x} \uparrow_{\mathbf{x_u}}) \downarrow_{\mathbf{x}'}$, where $\mathbf{x_u} = \{X_i \mid X_i \in \mathbf{x}, \mathbf{x} \in \chi_1, \mathbf{x}' \cap \mathbf{x} \neq \emptyset\}$.

$\gamma_{\chi_1,\chi_2}(\mathbf{M}_{\chi_2})$ returns an instance of the set of DFAs of χ_1, i.e., $\{M_\mathbf{x}|\mathbf{x} \in \chi_1\}$, where for each $\mathbf{x} \in \chi_1$, $M_\mathbf{x} = (\bigcap_{\mathbf{x}'\in\chi_2,\mathbf{x}'\cap\mathbf{x}\neq\emptyset}(M_{\mathbf{x}'} \uparrow_{\mathbf{x_u}})) \downarrow_\mathbf{x}$, where $\mathbf{x_u} = \{X_i \mid X_i \in \mathbf{x}', \mathbf{x}' \in \chi_2, \mathbf{x}' \cap \mathbf{x} \neq \emptyset\}$.

Both the alphabet and relation abstractions are conservative in the sense that the automata generated by the abstraction functions recognize more possible values for the string variables than the input automata. On the other hand, the concretization functions do not lose any precision, in the sense that, the automata generated by the concretization functions recognize the same possible values for the string variables as the input automata.

7.3.3 Heuristics for Abstraction Selection

An *abstraction class* is defined as a pair (χ, σ) where $\chi \in L_\mathbf{X}$ and $\sigma \in L_\Sigma$. The abstraction classes of $\mathbf{X}$ and Σ also form a complete lattice, of which the partial order

is defined as: $(\chi_1, \sigma_1) \sqsubseteq (\chi_2, \sigma_2)$ if $\chi_1 \sqsubseteq \chi_2$ and $\sigma_1 \sqsubseteq \sigma_2$. Given Σ and $\mathbf{X} = \{X_1, \ldots, X_n\}$, we can adjust the precision and performance of our analysis by selecting different abstraction classes.

If we select the abstraction class $(\chi_\top, \sigma_\top)$, we conduct our most precise relational string analysis. All the relations among $\mathbf{X}$ that are regular expressible will be kept using one n-track DFA at each program point. If we select $(\chi_\bot, \sigma_\bot)$, we only keep track of the *length* of each string variable individually. Though we abstract away almost all string relations and contents, this kind of path-sensitive (w.r.t length conditions on a single variable) size analysis can be used to detect buffer overflow vulnerabilities [20, 47]. If we select $(\chi_\top, \sigma_\bot)$, then we will be conducting relational size analysis. Finally, earlier string analysis techniques that use DFA, e.g., our previous work [58], correspond to the abstraction class $(\chi_\bot, \sigma_\top)$, where multiple single-track DFAs over Σ are used to encode reachable states. As shown in [56, 58], this type of analysis is useful for detecting XSS and SQLCI vulnerabilities.

Given a string system and a property we propose a heuristic for selecting an abstraction class (χ, σ). Let $\mathbf{x_p}$ denote the set of variables involved in the property we wish to check and C_p denote the set of characters. If the cardinality of $\mathbf{x_p}$ is less than or equal to one, we set χ to $\chi_\bot$. That is, we abstract away all the relations among the string variables. If there are more than one variables involved, χ is selected as follows: For each $X_i \in \mathbf{x_p}$, we generate its dependency graph. Let $\mathbf{x_i}$ denote the set of variables and

C_i be the set of characters that are associated with the nodes in the dependency graph of X_i. We select χ as the least element of $L_{\mathbf{X}}$, such that there exists $\mathbf{x} \in \chi$, $\mathbf{x_p} \subseteq \mathbf{x}$, as well as for each $X_i \in \mathbf{x_p}$, there exists $\mathbf{x}' \in \chi$, $\mathbf{x_i} \subseteq \mathbf{x}'$. We select σ as the least element of L_Σ, such that $C_p \cup_i C_i \subseteq \sigma$.

Once an abstraction class is selected, we perform our reachability analysis using the corresponding abstraction/concretization functions and the operations defined below. Let α_σ denote $\alpha_{\sigma_\top,\sigma}$, and α_χ denote $\alpha_{\chi_\top,\chi}$. We use $\log(|\sigma| + 1)$ bits to encode the selected alphabet (including the padding symbol λ).

- For $\mathbf{M}_\chi$ *bop* $\mathbf{M}'_\chi$, *bop* $\in \{\cup, \cap, \nabla\}$, we return $\{M_{\mathbf{x}}$ *bop* $M'_{\mathbf{x}} \mid \mathbf{x} \in \chi\}$. The result can be made more precise by refining the automata that have overlapping variable sets so that their projections to the same set of variables are equal.

- For CONSTRUCT(exp, $+$), we return α_σ(CONSTRUCT(exp, $+$)).

- For CONSTRUCT(exp, $-$), we return α_σ(CONSTRUCT(exp, $+$)) $\cap M_{\bar{\star}}$, where $M_{\bar{\star}}$ accepts arbitrary non-$\star$ words (i.e., $L(M_{\bar{\star}}) = \{w \mid w \in (\sigma \setminus \{\star\})^*\}$).

- For $\mathbf{M}_\chi \cap M_{exp}$, let $\mathbf{x}_{exp}$ denote the set of variables that are associated with M_{exp}. $\mathbf{x_u} = \cup_{\mathbf{x}\in\chi, \mathbf{x}\cap\mathbf{x}_{exp}\neq\emptyset}\{X_i \mid X_i \in \mathbf{x}\}$. We first generate a $|\mathbf{x_u}|$-track DFA $M_{\mathbf{x_u}} = M_{exp} \uparrow_{\mathbf{x_u}} \bigcap_{\mathbf{x}\subseteq\mathbf{x_u}} M_{\mathbf{x}} \uparrow_{\mathbf{x_u}}$. We return $\mathbf{M}'_\chi = \{M_{\mathbf{x_u}} \downarrow_{\mathbf{x}} \mid \mathbf{x} \subseteq \mathbf{x_u}, \mathbf{x} \in \chi\} \cup \{M_{\mathbf{x}} \mid \mathbf{x} \not\subseteq \mathbf{x_u}, \mathbf{x} \in \chi\}$.

7.3.4 Handling Complex String Operations

We extend our analysis to other complex string operations, e.g., replacement, that have been defined using single-track automata [58]. To do so, we first extract the values of each argument from $\mathbf{M}_\chi$ as a single track DFA. We compute the result of the string operation using these single-track DFAs accordingly. The post image of $\mathbf{M}_\chi$ against the operation can then be computed using the result. We also modify these operations to ensure the soundness of our approach while using an abstract alphabet. Consider REPLACE(M_1, M_2, M_3) [58] that returns the DFA accepts $\{w_1c_1w_2c_2 \dots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \dots w_kx_kw_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$. Assume M_1, M_2, M_3 over σ. We return α_σ(REPLACE($\gamma_\sigma(M_1)$, $\gamma_\sigma(M_2)$, $\gamma_\sigma(M_3)$) if $L(M_1) \not\subseteq L(M_{\bar{\star}})$ and $L(M_2) \not\subseteq L(M_{\bar{\star}})$, so that all possible results in the concrete domain are included in the abstract domain after abstraction. We return REPLACE(M_1, M_2, M_3), otherwise.

7.4 Experiments

We evaluate our approach against three kinds of benchmarks: 1) Basic benchmarks, 2) MFE benchmarks, and 3) XSS benchmarks. These benchmarks represent typical

string manipulating programs along with string properties that address severe web vulnerabilities.

Basic benchmarks: These examples demonstrate that our approach can prove implicit equality properties of string systems. We wrote two small programs. CheckBranch (B1) has if branch ($X_1 = X_2$) and else branch ($X_1 \neq X_2$). In the else branch, we assign a constant string c to X_1 and then assign the same constant string to X_2. We check at the merge point whether $X_1 = X_2$. CheckLoop (B2) is similar to the example from Section 2, where we assign X_1 and X_2 the same constant string at the beginning, and iteratively append another constant string to both in an infinite loop. We check at the end point of the loop whether $X_1 = X_2$. Let M accept the values of X_1 and X_2 upon termination. The equality assertion holds when $L(M) \subseteq L(M_a)$, where M_a is CONSTRUCT($X_1 = X_2$, $-$).

MFE benchmarks: This set of benchmarks show that the precision that is obtained using multi-track DFAs can help us in removing false positives generated by single-track automata based string analysis. These benchmarks represent *malicious file execution* (MFE) attacks. Such vulnerabilities are caused because developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. We systematically searched web applications for program points that execute file functions (include, fopen, etc) whose arguments may be influenced

by external inputs. At these program points, we check whether the retrieved files and the external inputs are consistent with what the developers intend. For instance, in pblguestbook.php distributed with Pblguestbook-1.32, one possible violation is that `$_GET['type']` is `A` but the retrieved file is `pblguestbook_back_up_B.txt`. We manually generate a multi-track DFA M_{vul} that accepts a set of possible violations for each benchmark, and apply our analysis on the sliced program segments. Upon termination, we report that the file function is vulnerable if $L(M) \cap L(M_{vul}) \neq \emptyset$. M is the composed DFA of the listed single-track DFAs in the single-track analysis. M1: PBLguestbook-1.32, pblguestbook.php (536). 536 denotes the line number of the sink function in the PHP script. M2: MyEasyMarket-4.1, prod.php (94). M3: MyEasyMarket-4.1, prod.php (189). M4: php-fusion-6.01, db_backup.php (111). M5: php-fusion-6.01, forums_prune.php (28). These test applications are available at `http://www.cs.ucsb.edu/~vlab/application/test-apps.tar.gz`.

XSS benchmarks: In this set of benchmarks, we check the existence of Cross-Site Scripting (XSS) vulnerabilities against known vulnerable Web applications. S1: MyEasyMarket-4.1, trans.php (218). S2: Aphpkb-0.71, saa.php(87), and S3: BloggIT 1.0, admin.php (23). We check whether at a specific program point, a sensitive function may take an attack string as its input. If so, we say that the program is vulnerable for the given attack pattern. To identify XSS attacks, we check intersection emptiness against all possible values of the input of the sensitive function at a

	$(\chi_\perp, \sigma_\top)$			
	Result	DFA state(bdd)	Time user+sys(sec)	Memory (kb)
B1	n	33(477)	0.027 + 0.006	410
B2	n	9(120)	0.022+0.008	484
M1	n	56(801)	0.027+0.003	621
M2	n	22(495)	0.013+0.004	555
M3	n	5(113)	0.008+0.002	417
M4	n	1201(25949)	0.226+0.025	9495
M5	n	211(3195)	0.049+0.008	1676

Table 7.1: Experimental Results against Basic and MFE Benchmarks Using Single-track Automata.

given program point and the attack strings specified as a regular language. All three benchmarks are vulnerable. We also modified/inserted sanitization routines to these benchmarks (denoted as S1', S2', and S3'). These test benchmarks are available at `http://www.cs.ucsb.edu/~vlab/stranger`.

Experimental Results: Table 7.1, 7.2, and 7.3 summarize the results for the first two benchmarks where we check properties depending on the relations of variables. The notation is explained as the following: DFA: the final (composed) DFA associated with the checked program point, state: number of states, and bdd: number of bdd nodes.

We start from $(\chi_\perp, \sigma_\top)$ (the analysis proposed in [57, 58]). As shown in Table 7.1, for all these benchmarks, we fail to prove the properties using single-track automata ("n" indicates that the property does not hold). We refine the abstraction class to $(\chi, \sigma_\top)$, where χ is selected for relation concretization by our heuristic, to perform a

	$(\chi, \sigma_\top)$			
	Result	DFA state(bdd)	Time user+sys(sec)	Memory (kb)
B1	y	14(193)	0.070 + 0.009	918
B2	y	5(60)	0.025+0.006	293
M1	y	50(3551)	0.059+0.002	1294
M2	y	21(604)	0.040+0.004	996
M3	y	3(276)	0.018+0.001	465
M4	y	181(9893)	0.784+0.07	19322
M5	y	62(2423)	0.097+0.005	1756

Table 7.2: Experimental Results against Basic and MFE Benchmarks Using Multi-track Automata.

	(χ, σ)			
	Result	DFA state(bdd)	Time user+sys(sec)	Memory (kb)
B1	y	10(61)	0.009 + 0.002	382
B2	y	5(16)	0.001+0.002	135
M1	y	54(556)	0.015+0.004	517
M2	y	22(179)	0.007+0.003	538
M3	y	3(49)	0.003+0.002	298
M4	y	175(4137)	0.218+0.13	5945
M5	y	66(1173)	0.033+0.003	782

Table 7.3: Experimental Results against Basic and MFE Benchmarks Using Multi-track Automata and Alphabet Abstraction.

more precise analysis. As shown in Table 7.2, we prove all properties using multi-track automata (”y” indicates that the property holds). Finally, we use the abstraction class (χ, σ), where σ is also selected for alphabet abstraction by our heuristic. As shown in Table 7.3, we prove all properties with better performance in terms of both time and memory using abstract alphabet. This result indicates that our heuristic picks a good abstraction class that is precise enough to prove the properties while coarse enough to be efficiently computed. Using the presented techniques, we can prove properties that we are not able to prove using multiple single-track automata, and by using our abstraction techniques we can improve the performance.

Table 7.4 and Table 7.5 summarize the results for checking the XSS benchmarks. The property holds if the benchmark is not vulnerable. Again, we start from $(\chi_\perp, \sigma_\top)$. We fail to prove the property for S1, S2, and S3, which might be due to false alarms. We refine the abstraction class to $(\chi, \sigma_\top)$, where χ is manually selected so that all branch conditions are precisely modeled. We still fail to prove the property for S1, S2, and S3. We identify that all these benchmarks include a real vulnerability and, hence, both analyses report correct results without false alarms. We manually insert/modify the sanitization routines to remove the vulnerabilities in S1, S2, and S3. Using $(\chi_\perp, \sigma_\top)$, we are able to prove the property against the modified benchmarks (S1’, S2’, S3’). We change the abstraction class to $(\chi_\perp, \sigma)$, where σ is selected by our heuristic, to perform a more coarse analysis. We are still able to conclude that S1’, S2’, and S3’

	$(\chi_\perp, \sigma_\top)$			
	Result	DFA state(bdd)	Time user+sys(sec)	Memory (kb)
S1	n	17(148)	0.010+0.002	444
S2	n	27(229)	0.035+0.002	895
S3	n	79(633)	0.062+0.005	1696
	$(\chi_\perp, \sigma_\top)$			
S1'	y	17(147)	0.010+0.002	382
S2'	y	17(141)	0.240+0.012	5686
S3'	y	127(1142)	0.436+0.008	6201

Table 7.4: Experimental Results against XSS Benchmarks.

	$(\chi, \sigma_\top)$			
	Result	DFA state(bdd)	Time user+sys(sec)	Memory (kb)
S1	n	65(1629)	0.195+0.150	1231
S2	n	47(2714)	0.153+0.008	2684
S3	n	79(1900)	0.226+0.003	2826
	$(\chi_\perp, \sigma)$			
S1'	y	17(89)	0.004+0.002	287
S2'	y	9(48)	0.036+0.005	2155
S3'	y	125(743)	0.297+0.002	3802

Table 7.5: Experimental Results against XSS Benchmarks Using Multi-track Automata and Alphabet Abstraction.

are not vulnerable but with better performance in terms of both time and memory. The experimental result shows that using the presented abstraction techniques, we can improve the performance of earlier string analysis techniques. It also shows that for this set of benchmarks, it is appropriate using multiple single-track automata, which matches our heuristic.

Chapter 8

Stranger Tool

We present a new tool called STRANGER (STRing AutomatoN GEneratoR) that can be used to check the correctness of string manipulation operations in Web applications with respect to known attacks. STRANGER implements an automata-based approach [56, 58] for automatic verification of string manipulating programs based on symbolic string analysis. String analysis is a static analysis technique that determines the values that a string expression can take during program execution at a given program point.

STRANGER encodes the set of string values that string variables can take as deterministic finite automata (DFAs). STRANGER implements both the *pre*- and *post*-image computations of common string functions on DFAs, including a novel algorithm for *language*-based replacement [58]. This replacement function takes three DFAs as arguments and outputs a DFA and can be used to model PHP replacement commands, e.g.,

`preg_replace()` and `str_replace()`, as well as many PHP sanitization routines, e.g., `addslashes()`, `htmlspecialchars()` and `mysql_real_escape_string()`. STRANGER implements all string manipulation functions using a symbolic automata representation (MBDD representation from the MONA automata package [9]) and leverages efficient manipulations on MBDDs such as determinization and minimization. This symbolic encoding also enables STRANGER to deal with large alphabets.

STRANGER combines forward and backward reachability analyses [56] and is capable of (1) checking the correctness of sanitization routines and proving that programs are free from specified attacks (with respect to attack patterns), and (2) identifying vulnerable programs, as well as generating non-trivial vulnerability signatures. Using forward reachability analysis, STRANGER computes an over-approximation of all possible values that string variables can take at each program point. If this conservative approximation does not include any attack pattern, STRANGER concludes that the program does not contain any vulnerabilities. Otherwise, intersecting these with attack patterns yields the potential attack strings. Using backward analysis STRANGER automatically generates string-based vulnerability signatures, i.e., a characterization that includes all malicious inputs that can be used to generate attack strings. In addition to identifying existing vulnerabilities and their causes, these vulnerability signatures can be used to filter out malicious inputs.

8.1 Tool Description

STRANGER uses Pixy developed by Jovanovic et al. [31] as a front end and the MONA [9] automata package developed by Klarlund et al. 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, SQL Injection, or MFE 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. The architecture of STRANGER is shown in Figure 8.1. The tool consists of the following parts.

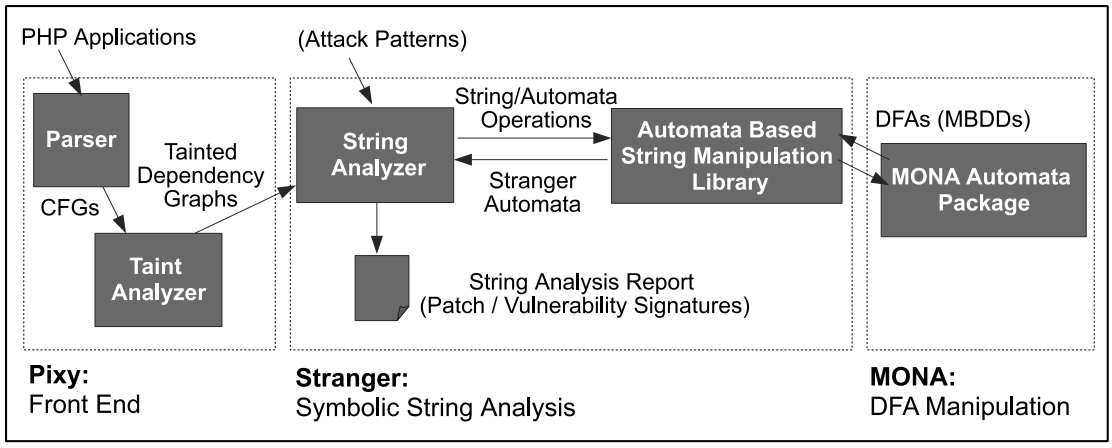


Figure 8.1: The Architecture of STRANGER

8.1.1 PHP Parser and Taint Analyzer

The first step in our analysis is done by Pixy [31], 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. A dependency graph specifies how the inputs flow to a sensitive sink with respect to *string operations*. The number of its nodes is linear in the number of the string operations in the program with respect to a static single assignment representation. Loop structures generate cyclic dependency relations. If no tainted data flow to the sink, taint analysis reports the dependency graph to be secure; otherwise, the dependency graph is tainted and passed to the string analyzer for more inspection.

8.1.2 String Analyzer

The string analyzer implements our (forward and backward) vulnerability analysis [56] on the tainted dependency graphs found by the taint analysis. The dependency graphs are pre-processed to optimize the reachability analyses. 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 vul-

nerability 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 [56] on nodes that are part of the SCC represented by the SCC node. During the fixpoint computation we apply automata widening [4] 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 [4] that guarantees the convergence to avoid potential infinite iterations of the fixpoint computation.

8.1.3 String Manipulation Library

The string manipulation library (SML) handles all core string and automata operations such as replacement, concatenation, prefix, suffix, intersection, union, and widen. During the vulnerability analysis, all string and automata manipulation operations that are needed to compute the values of a node in a dependency graph are sent to SML along with the string and/or automata parameters. SML, then, executes the operation and returns back the result as an automaton. A Java class called $StrangerAutomaton$ has been used as the type of the parameters and results. The class follows a well defined interface so that other automata packages can be plugged in and used with our string analyzer instead of SML. SML is also decoupled from the vulnerability analysis component so that it can be used with other string analysis tools. $StrangerAutomaton$ encapsulates $libstranger.so$ shared library that has the actual string manipulation code implemented in C to get a faster computation and a tight control on memory. We used JNA (Java Native Access) to bridge the two languages. Another feature of Stranger is an option to produce a C trace of all string and automaton operations performed during a run to allow us to debug the code directly in gdb. This can be generalized to produce a higher intermediate language that can be used with other string analysis backends that can not be plugged directly into Stranger.

8.2 Experiments and Discussions

We have experimented with STRANGER on several benchmarks extracted from known vulnerable web applications [58]. For each vulnerable benchmark, we also generated a modified version where string manipulation errors are fixed. STRANGER analyzed all benchmarks within a minute. It successfully reported all known vulnerabilities, generated the vulnerability signatures, and verified that the modified version is secure and free from the previously reported vulnerabilities with respect to the attack patterns.

We have conducted a case study on `Schoolmate-1.5.4` - a PHP web application for school administration. `Schoolmate` consists of 63 PHP files and 8181 lines of code. Using a machine with Intel Core 2 Due 2.5GHz with 4 GB of memory running Linux Ubuntu 8.04, it took 22 minutes to analyze the whole application. During the analysis we checked 898 XSS sinks and consumed 281 MB of memory. Stranger reported 153 XSS vulnerabilities with respect to the attack pattern $\Sigma^* < script \Sigma^*$. That is, there are at most 153 sensitive sinks that may take a string that contains $< script$ as its input at run time. We manually inspected these vulnerabilities and identified 105 actual vulnerabilities (48 false alarms). The false positive rate of STRANGER is around 31.3%. 39 of these false alarms are caused by infeasible paths to exploit such vulnerabilities. We can eliminate all these false positives by performing path sensitive analysis using multi-track automata. 6 of the false alarms are due to unmodeled built-in PHP

functions. Three of these functions, e.g., pdf conversion, can be categorized as trusted functions. The remaining unmodelled functions represent a set of functions that may cause a vulnerability but are hard to model using the string operations in our string manipulation library. 3 of the false alarms were the result of unavailable user written functions. These false alarms are caused by different execution entries. Suppose that $a.php$ defines function f, $b.php$ uses the function f but does not include $a.php$ and $c.php$ includes both $a.php$ and $b.php$. When we analyze $a.php$ and $b.php$ as part of $c.php$, we will not get a false alarm. However, when we analyze $b.php$ by itself (since $a.php$ is not included, we were not able to find the definition of the function f), we will conservatively return Σ^* as the return value of function f. On the other hand, while running $b.php$ directly, the PHP interpreter will abort the execution due to an unavailable function f.

We have also conducted another case study on `SimpGB-1.49.0` - a PHP guestbook web application. `SimpGB` consists of 153 php files containing 44000+ lines of code. Using a machine with Intel Core 2 Due 2.5 GHz with 4GB of memory running Linux Ubuntu 8.04, STRANGER took 231 minutes to check XSS vulnerabilities for all entries of executable PHP scripts and concluded 304 possible vulnerabilities out of 15115 sinks. STRANGER took 175 minutes to reveal 172 possible SQL Injection vulnerabilities from 1082 sinks, and 151 minutes to reveal 26 possible MFE vulnerabilities from 236 sinks.

In sum, we presented a string analysis tool for verification of web applications, focusing on SQLI, XSS and MFE attacks. In addition to identifying vulnerabilities and generating vulnerability signatures of vulnerable applications, STRANGER can also verify the absence of vulnerabilities in applications (with respect to attack patterns) that use proper sanitization. Compared to grammar-based string analysis tools [15, 38, 48], STRANGER features specific automata-based techniques including automata widening [4], language-based replacement [58] and symbolic automata encoding and manipulation [9]. STRANGER and several benchmarks are available at `http://www.cs.ucsb.edu/~vlab/stranger`.

Chapter 9

Related Work

Static analysis of strings in programs has been an active research area with the goal of finding and eliminating security vulnerabilities caused by the misuse of string variables. There have been two separate branches of research in this area: 1) *String analysis* that focuses on statically identifying all possible values of a string expression at a program point in order to eliminate vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks [1, 15, 52, 58], and 2) *Size analysis* that focuses on statically identifying all possible lengths of a string expression at a program point in order to eliminate buffer overflow errors [20, 22, 46]. In this chapter, we review the related work on string analysis, size analysis, and composite analysis.

9.1 String Analysis

Due to its importance in security, string analysis has been widely studied. Christensen, Møller and Schwartzbach [15] propose a grammar-based string analysis (implemented in a tool called JSA) to statically determine the values of string expressions in Java programs. They convert the flow graph into a context free grammar where each string variable corresponds to a nonterminal, and each string operation corresponds to a production rule. Then, they convert this grammar to a regular language by computing an over-approximation.

Kirkegaard et al. apply JSA to static analysis of XML transformations in Java programs [35] by using DTD schemas as types and modeling the effects of XML transformation operations. Gould et al. [24] use the grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications [15]. Christodorescu et al. [16] present an implementation of the grammar-based string analysis technique for executable programs for the x86 architecture.

Minamide [38] extends the grammar-based string analysis technique by providing support for string-based replacement operations. He uses finite-state transducers to model replace operations. He describes a string analysis tool similar to JSA to statically detect cross-site scripting vulnerabilities and to validate pages generated by Web

applications written in the PHP language. Instead of approximating the grammar as a regular language, he performs his analysis directly on the context free grammar.

Wassermann et al. [48, 49] propose grammar-based static string analyses to detect SQL injections and XSS, following Minamide's approach [38]. There are some other tools for string analysis [14, 21, 43, 52]. Shannon et al. [43] use forward bounded symbolic execution to perform string analysis on Java programs. Similar to our approach, they use automata to represent path constraints and to encode the values of string variables. They support trim and substring operations. Xie and Aiken [52] support string assignment and validation operations. Fu et al. [21] and Choi et al. [14] support string-based replacement (as opposed to language-based replacement). None of the tools mentioned above address language-based replacement operations. This is a shortcoming that causes the approximations computed by these tools to be too coarse for the analysis of some input sanitization routines.

Language-based replacement has been discussed in computational linguistics [23, 32, 40, 45]. 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 increase in the size of the non-deterministic automaton, and is more likely to blow-up compared to our construction. The transducer-based replacement function [40] has been implemented in Finite State Automata utilities (FSA) [44], where automata are stored and

manipulated using an explicit representation. We use a symbolic DFA representation based on MBDDs. This symbolic encoding enabled us to perform complex automata operations, such as closure, concatenation, replace, and widening, efficiently using the MBDDs.

Balzarotti et al. [1] combine both dynamic and static analysis techniques to verify PHP programs. They support language-based replacement by incorporating FSA [44], but 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 [4] to tackle this problem and obtain a tighter approximation that enables us to verify a larger set of programs.

Choi et al. [14] also investigate a widening method to analyze strings. Their 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 [4]. The intuition behind this widening operator is applicable to any symbolic fixpoint computation that uses automata. In [4] it is shown that, for a restricted class of systems, the widening operator computes the precise fixpoint. We extend this result to our analysis. In our experiments we demonstrate that the over-approximation computed by this widening operator works well in proving the type of properties that we are interested in.

There are several recent string analysis tools that use symbolic string analysis based on DFA encodings [21, 43, 58]. Some of them are based on symbolic execution and use a DFA representation to model and to verify the string manipulation operations in Java programs [21, 43].

None of the previous work we mentioned so far address vulnerability signature and sanitization generation. Wassermann et al. [50] use string analysis in test input generation for Web applications. Their approach is based on concolic execution [42], where results of a concrete execution is used to collect constraints on program execution. These constraints are then used to generate new test cases. They use an automata based backward image computation based on transducers (which is similar to our backward analysis) to propagate constraints on string variables. However, they do not discuss replacement operations which are crucial for string manipulation, and their approach targets test generation rather than generating a sound approximation of all possible inputs that can exploit a vulnerability. Moreover, their approach does not provide a sound approximation in the presence of loops.

Compared to recent work on attack generation (for example [34]), we propose a sound static analysis approach that characterizes all possible inputs that can exploit a given attack pattern, rather than generating concrete attacks using dynamic analysis techniques based on given exploits.

There has been earlier work on vulnerability signature generation [10, 11, 18]. The techniques discussed in [18] and [10] require an input that exploits a vulnerability (i.e., an exploit) in order to generate the vulnerability signatures. For example, in [18], this is obtained by running an instrumented version of the program. Our approach does not need an exploit as input since we combine forward and backward symbolic analyses.

The approach presented in [11] is a backward analysis similar to our backward analysis. However, they require loop invariants to be provided by the user in order to handle loops, whereas we use an automated approach based on widening. Also, they focus on weakest precondition computation for binary programs. None of the earlier results on vulnerability signature generation [10, 11, 18] focus on string manipulation operations. Instead, they use existing symbolic execution engines, which cannot handle the string manipulation operations that we focus on in this dissertation. In order to analyze vulnerabilities of PHP applications, it is necessary to handle string manipulation operations faithfully as we do in our work. We also generate sanitization statements that repair the bad inputs which has not been done before to the best of our knowledge.

Furthermore, all of the results mentioned above use single-track DFAs and encode the reachable configurations of each string variable separately. This can cause two problems: 1) Branch conditions that check relations among different string variables can lead to imprecision in the analysis, resulting in false positives. 2) It is not possible to check invariants that refer to more than one string variable using these earlier

techniques. Our multi-track automata encoding not only improves the precision of the string analysis but also enables verification of properties that cannot be verified with the previous approaches. To the best of our knowledge, our approach is the first relational string analysis technique.

We have also presented alphabet and relation abstractions that enable us to adjust the precision and performance of our analysis. Compared to abstraction techniques on automata [7], our abstractions focus on the values of the string variables and the relations among them. In the heuristic we propose, the selection of a suitable abstraction class can be guided by the constants and relations appearing in the program and the property.

The use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [8]). In this dissertation, we focus on verification of string manipulating programs.

There has been some recent work on solving string constraints. Hooimeijer and Weimer [28] present an automata-based decision procedure for solving equations over regular language variables. Our techniques can be used in solving their string constraints. In addition, we can also (conservatively) deal with complex string operations, e.g., replacement. Kiezun et al. [33] present HAMPI, a SAT-based solver for string constraints over bounded string variables. Given a set of constraints (including membership of regular languages), HAMPI outputs a string that satisfies all the constraints, or reports

that the constraints are unsatisfiable. Bjoner et al. [6] present a path feasibility analysis based on solving bounded path conditions for string manipulating programs. Instead of solving string constraints directly, they solve their length constraints using a SMT solver. If the length constraints are unsatisfiable, it implies that the string constraints are unsatisfiable. If the length constraints are satisfiable, they use the satisfying assignment to bound the length of string variables and solve the string constraints over bounded string variables.

Finally, we have investigated the boundary of decidability for the string verification problem. Bjørner et al. [6] show an undecidability result with the replace operation. We show that even when only the concatenation operation is allowed the string verification problem is undecidable for deterministic string systems with only three unary string variables and non-deterministic string systems with only two string variables if the comparison of two variables are allowed.

9.2 Size Analysis

Size analysis is a crucial problem in software security. Various software defects, such as array out of bound errors and memory overflow errors, can be discovered by tracing object sizes without knowing the contents of the objects. Various techniques

that use integer variables and integer constraints to verify size properties have been proposed in the past [13, 30, 54, 55, 60].

Hughes et al. [30] present a sound semantic model of size types to verify the properties of reactive systems. They show that various essential program properties, such as function productivity, memory leaks, array bounds and the termination of some restricted functions, can be reduced to type checking problems. The advantages of type analysis include a) the soundness proof and b) the efficient type checking algorithm. Hughes' work was the first paper on using size types to analyze programs.

Chin et al. [13] extend size types to the verification of object-oriented languages by annotations. They annotate an abstract data type for each object with size invariants, which could then be used to infer size properties among objects. They propose an intermediate language, called OIMP, to capture the size information of programs (such as C++/Java programs) via an annotated type system. One advantage of their approach is that it can handle shared objects.

In our earlier work [60], we apply size analysis to a specification language: Object Constraint Language (OCL), which is part of the Unified Modeling Language (UML). Instead of annotating types on objects, we verify specification consistency on size properties using automata-based symbolic analysis of integer variables. We evaluate our approach against the specification of JAVA card APIs and reveal several unknown er-

rors. Verifying invariants related to integer variables has also been applied to shape analysis [54, 55].

9.3 Composite Analysis

In this dissertation, we present a composite symbolic verification technique [12] that combines string [1, 15, 52, 58] and size [20, 22, 46] analyses with the goal of improving the precision of both. We use a forward fixpoint computation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables. Similar to prior size analysis techniques [20, 22, 46] we associate each string variable with an auxiliary integer variable that represents its length. At each program point, we symbolically compute all possible values of all integer variables (including the auxiliary variables), as well as all possible values of all string variables. The reachable values of all integer variables are over-approximated as a Presburger arithmetic (linear arithmetic) formula and symbolically encoded as *arithmetic automata* [3, 51]. Similar to some prior string analysis techniques [1, 58], the values that string variables can take are over-approximated as regular languages and symbolically encoded as *string automata*. Our composite analysis is a forward fixpoint computation with widening on these arithmetic and string automata. In addition, we improve precision by restricting both representations using lengths of

the values of string variables. To identify length constraints, one can also characterize the Presburger arithmetic formula from the arithmetic automaton by solving the Presburger synthesis problem [37] and further restricting the target string automaton. Our approach is an over approximation that uses the boundary rather than the values of the semilinear set, and it is efficient and simple to implement.

In addition to earlier work on string analysis [1, 15, 52, 58] and size analysis [20, 22, 46] that motivated our work [59], there has been some recent work on analyzing string and integer variables together during symbolic execution [21, 43, 53]. Unlike our approach, these are unsound techniques that target testing and they do not try to compute an over-approximation of the reachable states via widening. Hence, they cannot prove properties of examples that we present in this dissertation. Compared to [25, 26] that use abstract interpretation for reasoning about relational properties among the contents of symbolic intervals of arrays, our analysis traverses concrete values of string and integer variables using automata and addresses language properties.

Finally, symbolic model checking on various variable types has been investigated in [12, 55]. Bultan et al. [12] propose a sound composite framework that combines BDDs and arithmetic constraint representations to analyze systems having boolean (bounded) and integer (unbounded) variables. Leveraging the compactness of both representations on their own domains, Bultan et al. show the effectiveness of the composite framework on several applications. Yavuz-Kahveci and Bultan [55] apply the symbolic

composite model checking framework to concurrent linked list specifications. The analysis combines BDDs, linear arithmetic constraints, and shape graphs to represent values of booleans, integers, and heap variables respectively. They compute both lower and upper approximations of reachable states on this composite representation, and hence their analysis is capable of falsifying or verifying invariants of target programs. Our analysis extends the composite analysis framework to the *string* domain and takes advantage of the fact that arithmetic automata provide a compact representation for Presburger arithmetic constraints and string automata provide a compact representation for regular languages.

Chapter 10

Conclusion

Many security vulnerabilities are caused by inadequate manipulation of string variables. In this dissertation, we presented a formal characterization of the string verification problem and investigated the decidability boundary for string systems. We proposed a conservative symbolic verification approach that computes an over-approximation of the reachable states. The approach features language-based replacement, fixpoint acceleration, and symbolic automata encoding.

We presented a set of techniques that 1) given an attack pattern, identify vulnerabilities that are due to string manipulation, 2) generate a characterization of inputs that can exploit the vulnerability, and 3) generate sanitization statements that eliminate the vulnerability. Our approach is based on automata-based symbolic forward and backward reachability computations. We developed two techniques to generate vulnerabil-

ity signatures that characterize all malicious inputs, and proposed different strategies to prevent attacks that match the given attack patterns by automatically synthesizing effective sanitization routines from the vulnerability signatures.

We presented an automata-based approach for symbolic verification of infinite-state systems with unbounded string and integer variables. Our composite approach that combines string analysis with size analysis is able to verify properties that cannot be verified with either analysis alone. We proposed a novel algorithm to convert unary automata to binary automata and vice versa, and showed how the precision of both string and size analyses can be improved by using length automata and the conversion.

We proposed a novel relational string verification technique using multi-track automata, symbolic reachability analysis, summarization and abstraction. Compared to earlier automata-based string analysis techniques, the presented technique uses a single multi-track DFA to represent all possible values of string variables at a given program point, and enables us to check equality properties among string variables and improves the precision of the string analysis.

Finally, we have developed STRANGER, a public automata-based string analysis tool for verification of PHP web applications, focusing on SQLI, XSS and MFE attacks. In addition to identifying vulnerabilities and generating vulnerability signatures and effective patches of vulnerable applications, STRANGER can also verify the absence of

vulnerabilities in applications that use proper sanitization. We demonstrated the effectiveness of our approach on several examples, as well as some large-scale applications.

Bibliography

[1] 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 (S&P)*, 2008.

[2] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 25–35, New York, NY, USA, 2007. ACM.

[3] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

[4] C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, pages 321–333, 2004.

[5] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, LNCS 1260*. Springer Verlag, 1997.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceeding of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 307–321, 2009.

[7] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, pages 372–386, 2004.

[8] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th International Conference on Computer Aided Verification (CAV)*, pages 403–418, 2000.

[9] BRICS. The MONA project. `http://www.brics.dk/mona/`.

[10] D. Brumley, J. Newsome, D. X. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, pages 2–16, 2006.

[11] D. Brumley, H. Wang, S. Jha, and D. X. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 311–325, 2007.

[12] T. Bultan, R. Gerber, and C. League. Composite model-checking: verification with type-specific symbolic representations. *ACM Trans. Softw. Eng. Methodol.*, 9(1):3–50, 2000.

[13] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *Proceedings of the 27th international conference on Software engineering (ICSE)*, pages 186–195, 2005.

[14] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *Programming Languages and Systems, 4th Asian Symposium (APLAS)*, pages 374–388, 2006.

[15] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[16] 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)*. ACM Press, Sept. 2005.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 117–130, 2007.

[19] CVE. Common Vulnerabilities and Exposures. `http://www.cve.mitre.org`.

[20] N. Dor, M. Rodeh, and M. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, 2003.

[21] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 87–96, Washington, DC, USA, 2007.

[22] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 345–354, 2003.

[23] 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.

[24] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 645–654, 2004.

[25] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 235–246. ACM, Jan. 2008.

[26] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 339–348, 2008.

[27] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 89–110, 1995.

[28] 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.

[29] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[30] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 410–423, New York, NY, USA, 1996. ACM.

[31] 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)*, pages 258–263, 2006.

[32] L. Karttunen. The replace operator. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 16–23, 1995.

[33] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116, 2009.

[34] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, May 20–22, 2009.

[35] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Transactions on Software Engineering*, 30(3), March 2004.

[36] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE)*, pages 389–392, New York, NY, USA, 2007. ACM.

[37] J. Leroux. A polynomial time presburger criterion and synthesis for number decision diagrams. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 147–156, 2005.

[38] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, pages 432–441, 2005.

[39] M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. In *Ann. of Math (74)*, pages 437–455, 1961.

[40] 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.

[41] O. W. A. S. P. (OWASP). Top ten project. `http://www.owasp.org/`, May 2010.

[42] 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)*, pages 263–272, 2005.

[43] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*, pages 13–22, Washington, DC, USA, 2007.

[44] G. van Noord. FSA utilities toolbox. `http://odur.let.rug.nl/~vannoord/Fsa/`.

[45] G. van Noord and D. Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Proceedings of the 4th International Workshop on Implementing Automata (WIA)*, pages 122–139. Springer-Verlag, July 1999.

[46] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium (NDSS)*, pages 3–17, 2000.

[47] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 3–17, 2000.

[48] 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 (PLDI)*, pages 32–41, 2007.

[49] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 171–180, 2008.

[50] 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)*, pages 249–260, 2008.

[51] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 1–19, 2000.

[52] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS)*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.

[53] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2008.

[54] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

[55] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proceedings of the 9th International Symposium on Static Analysis (SAS)*, pages 69–84, 2002.

[56] F. Yu, M. Alkhalaf, and T. Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009.

[57] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[58] 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)*, pages 306–324, 2008.

[59] 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)*, pages 322–336, 2009.

[60] F. Yu, T. Bultan, and E. Peterson. Automated size analysis for ocl. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 331–340, 2007.