Gary Michael Wassermann
September 2008
Computer Science

Techniques and Tools for Engineering Secure Web Applications

Abstract

With the rise of the Internet, web applications, such as online banking and web-based email, have become integral to many people's daily lives. Web applications have brought with them new classes of computer security vulnerabilities, such as SQL injection and cross-site scripting (XSS), that in recent years have exceeded previously prominent vulnerability classes, such as buffer overflows, in both reports of new vulnerabilities and reports of exploits. SQL injection and XSS are both instances of the broader class of input validation-based vulnerabilities. At their core, both involve one system receiving, transforming, and constructing string values, some of which come from untrusted sources, and presenting those values to another system that interprets them as programs or program fragments. These input validation-based vulnerabilities therefore require fundamentally new techniques to characterize and mitigate them.

This dissertation addresses input validation-based vulnerabilities that arise in the context of web applications, or more generally, in the context of metaprogramming. This dissertation provides the first principled characteriztion, based on concepts from programming languages and compilers, for such vulnerabilities, with formal definitions for SQL injection and XSS in particular. Building on this characterization, the dissertation also contributes practical algorithms for runtime protection, static analysis, and testing-based analysis of web applications to identify vulnerabilities in application code and prevent attackers from exploiting them. This dissertation additionally reports on implementations of these algorithms, showing them to be effective for their respective settings. They have low runtime overhead, validate the definitions, scale to large code bases, have low false-positive rates, handle real-world application code, and find previously unreported vulnerabilities.

Techniques and Tools for Engineering Secure Web Applications

By

Gary Michael Wassermann
B.S. (University of California, Davis) 2002

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Zhendong Su (Chair)

Professor Premkumar T. Devanbu

Professor Ronald Olsson

Committee in Charge
2008

© Gary Wassermann, 2008. All rights reserved.

To my Lord and Savior, Jesus Christ
and to my wife, Jaci, who has been a help and a joy to me.

Contents

List of Figures

List of Tables

Abstract

With the rise of the Internet, web applications, such as online banking and web-based email, have become integral to many people's daily lives. Web applications have brought with them new classes of computer security vulnerabilities, such as SQL injection and cross-site scripting (XSS), that in recent years have exceeded previously prominent vulnerability classes, such as buffer overflows, in both reports of new vulnerabilities and reports of exploits. SQL injection and XSS are both instances of the broader class of input validation-based vulnerabilities. At their core, both involve one system receiving, transforming, and constructing string values, some of which come from untrusted sources, and presenting those values to another system that interprets them as programs or program fragments. These input validation-based vulnerabilities therefore require fundamentally new techniques to characterize and mitigate them.

This dissertation addresses input validation-based vulnerabilities that arise in the context of web applications, or more generally, in the context of metaprogramming. This dissertation provides the first principled characteriztion, based on concepts from programming languages and compilers, for such vulnerabilities, with formal definitions for SQL injection and XSS in particular. Building on this characterization, the dissertation also contributes practical algorithms for runtime protection, static analysis, and testing-based analysis of web applications to identify vulnerabilities in application code and prevent attackers from exploiting them. This dissertation additionally reports on implementations of these algorithms, showing them to be effective for their respective settings. They have low runtime overhead, validate the definitions, scale to large code bases, have low false-positive rates, handle real-world application code, and find previously unreported vulnerabilities.

Acknowledgments and Thanks

Many have provided invaluable support and assistance throughout my graduate studies. First, I thank God who has met my biggest need by redeeming me and calling me to know himself. In addition, He gave me the abilities that I have, directed me to UC Davis, and answered my prayers for an advisor, for a research topic, and for direction and employment after completing my degree.

More directly related to this dissertation, I thank my adviser, Prof. Zhendong Su, for his hard work and personal attention as he trained me to be a researcher. Prof. Su invested significantly in me as his first Ph.D. student. He provided me with ideas, critiqued my ideas, trained me in writing and presenting research, maintained a high standard for work, procured funding that allowed me significant freedom in the research I pursued, connected me with people and opportunities in my professional development, and set an example for excellence and integrity in work and life. For all these things I thank him and am indebted to him.

I thank also the other members of my committee, Prof. Prem Devanbu and Prof. Ron Olsson. Prof. Devanbu's practical perspective on Software Engineering problems helped improve the work presented in this dissertation. I also appreciate his help with professional networking and his low-stress approach even to academic life. Prof. Olsson's carefully prepared and delivered lectures sparked some of my initial interest in the field of Programming Languages, and his meticulous reading of earlier drafts of this dissertation helped to catch many errors and infelicities. I thank also the anonymous reviewers who provided helpful comments on the papers on which Chapters 2 through 5 are based.

I thank my pastor, Rev. P.G. Mathew, who, through his unique and uncompromising commitment to preaching the gospel, has presented to me the God who saved me and has equipped me for the matters of this life, including graduate school, and of the life of the world to come. In addition, I thank him for praying for me and for counseling me at

crucial junctures in my personal life and academic career. I thank Mr. Suman Jha, who has often borne my burdens, counseled me in various decisions, and kept me accountable for how I live. I thank Dr. Richard Spencer, Dr. Dawson Kesling, Mr. Ron Guly, and Dr. David Mobley for their counsel and assistance in my academic and professional decisions. In addition to those listed above, I thank Mr. Mike Moroski, Dr. Nick Faber, and Mr. Greg Perry for their encouragements and exhortations during my graduate studies. I thank also the other leaders and members of Grace Valley Christian Center and especially those of the Jhas' home group, whose fellowship is invaluable.

I thank my labmates who, both in the lab and at conferences, provided valuable camaraderie. They also gave much useful feedback on ideas, paper drafts, and practice talks, and this feedback improved the quality of my work significantly. These people include my predecessors in 2249 Kemper, Eric Wohlstadter, Stoney Jackson, and Brian Toone, as well as my contemporaries and successors, Lingxiao Jiang, Jed Crandall, Earl Barr, Matt Roper, Chris Bird, Mark Gabel, Ghassan Misherghi, Andreas Saebjoernsen, Jim Apple, Taeho Kwan, Dennis Xu, and Sophia Sun

I thank my wife, Jaci, for her constant support during these six years of graduate school, which were also our first six years of marriage. The stability she helped to provide at home significantly eased the task of persevering through graduate school. Her seemingly bottomless well of energy and joy have refreshed and sustained me many times. I thank also my children, Sharon and Tabitha, who have given added joy and purpose to my graduate career. I thank my parents for their support throughout my prior education and upbringing, and I thank my parents and my parents-in-law their encouragement and support during my graduate studies.

Chapter 1

Introduction

This dissertation proposes principled techniques to address input validation errors in web applications. Many analysis techniques have been applied to programs that run on single machines, for example to find memory errors or to verify structural properties of heap data structures. In contrast to traditional applications, web applications are distributed and dynamic, and they have appeared relatively recently; hence they present a new domain with additional challenges for analysis techniques. This chapter introduces the significance of web applications and their major classes of errors, and presents the structure of the rest of this dissertation.

1.1 Web Applications

Web applications enable much of today's online business including banking, shopping, university admissions, email, social networking, and various governmental activities. They have become ubiquitous because users only need a web browser and an Internet connection to receive a system-independent interface to some dynamically generated content. The data web that applications handle, such as credit card numbers and product inventory data, typically has significant value both to the users and to the service providers. Additionally, users care that web pages behave as trusted servers intend because web browsers run on

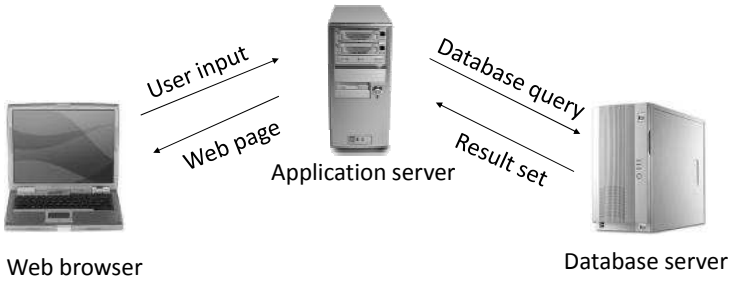


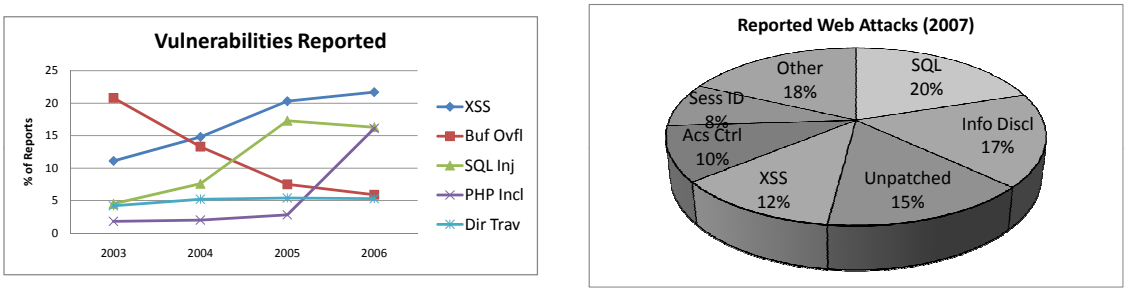
Figure 1.1: Web application system architecture.

users' local systems and often have bugs that could be exploited by maliciously written pages.

Figure 1.1 shows a typical system architecture for web applications. This three-tiered architecture consists of a web browser, which functions as the user interface; a web application server, which manages the business logic; and a database server, which manages the persistent data. The web application server receives input in the form of strings from both other tiers: user input from the browser and result sets from the database server. It typically incorporates some of this input into the output that it provides to the other tiers, again in the form of strings: queries to the database server, and HTML documents to the browser, both of which get executed by their respective tiers. The web application server constructs code dynamically, so the code for the entire web application does not exist in any one place at any one time for any one entity to regulate. The flow of data among tiers gives rise to the *input validation* problem for the web application server; it must check and/or modify incoming strings before processing them further or incorporating them into output that it passes to other tiers to execute. Failure to check or sanitize input appropriately can compromise the web application's security.

1.2 Input Validation-Based Vulnerabilities

The two most prominent classes of input validation errors are cross-site scripting (XSS) and SQL injection. XSS and SQL injection are the classes of vulnerabilities in which an

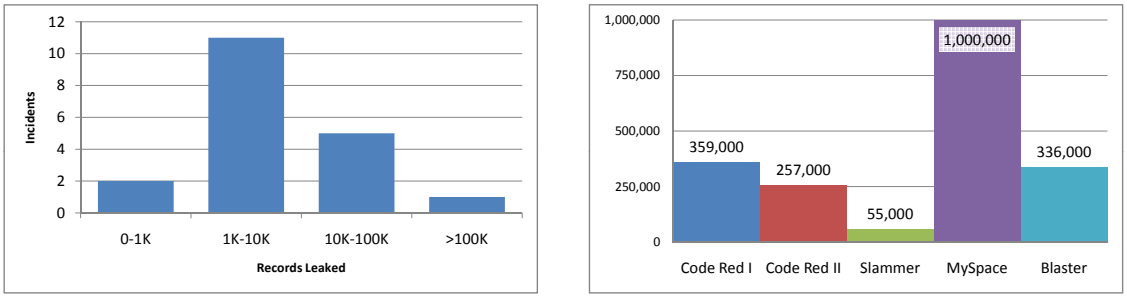


(a) (b)

Figure 1.2: Recent vulnerability and attack numbers.

attacker causes the web application server to produce HTML documents and database queries, respectively, that the application programmer did not intend. They are possible because, in view of the low-level APIs described above for communication with the browser and database, the application constructs queries and HTML documents via low-level string manipulation and treats untrusted user inputs as isolated lexical entities. This is especially common in scripting languages such as PHP, which generally do not provide more sophisticated APIs and use strings as the default representation for data and code. Some paths in the application code may incorporate user input unmodified or unchecked into database queries or HTML documents. The modifications/checks of user input on other paths may not adequately constraint the input to function in the generated query or HTML document as the application programmer intended. In that sense, both XSS and SQL injection are integrity violations in which low-integrity data is used in a high-integrity channel; that is, the browser or the database executes code from an untrusted user, but does so with the permissions of the application server. However, both problems involve more than naive integrity level or taintedness tracking because the output gets parsed and interpreted rather than treated as an atomic value. Sections 2.3 and 4.1 present precise definitions of SQL injection and XSS employing sentential forms and integrity levels.

To highlight both how ubiquitous web applications have become and how prevalent their problems are, Figure 1.2a shows, for each year from 2003 to 2006, the percentage of



(a) (b)

Figure 1.3: Results of recent attacks.

newly reported security vulnerabilities in five vulnerability classes (this data comes from Mitre's report [13]): XSS, SQL injection, PHP file inclusions, buffer overflows, and directory traversals. These were the five most reported vulnerabilities in 2006. All of these except buffer overflows are specific to web applications. Note that XSS and SQL injection are consistently at or near the top: 21.7% and 16% of the reported vulnerabilities in 2006, respectively. Some web security analysts speculate that because web applications are highly accessible and databases often hold valuable information, the percentage of SQL injection attacks being executed is significantly higher than the percentage of reported vulnerabilities would suggest [86]. Empirical data supports this hypothesis. Figure 1.2b shows percentages of reported web attacks for the year 2007 (this data comes from the Web Hacking Incidents Database [84]). Although many attacks go unreported or even undetected, this chart shows that 12% and 20% of the web-based attacks that made the press in 2007 were XSS and SQL injection, respectively. This chart also includes attacks on higher-level logic errors, such as weak session IDs.

Both kinds of attacks can cause severe damage. Typical uses of SQL injection leak confidential information from a database, by-pass authentication logic, or add unauthorized accounts to a database. Although XSS vulnerabilities are more prevalent than SQL injection vulnerabilities, some XSS vulnerabilities cannot be exploited in damaging ways, whereas most SQL injection vulnerabilities can. Figure 1.3a shows the number of reported

information leakage attacks in 2007 that leaked the number of records in each of four ranges split by a log-scale (this data comes from the Web Hacking Incidents Database [84]). A news article from 2005 gives an example of the kinds of records that this figure includes: an attacker retrieved personal information via an SQL injection vulnerability in an admissions page about hundreds of thousands of applicants to a major university [57]. The university had to notify every applicant whose records were in the database about the possibility that the applicant was now the victim of identity theft. This consequence was both an expense and a blow to public relations for the university. Most attacks leaked more than one thousand records, and one leaked more than one hundred thousand records. This illustrates how many users a single attack typically affects.

Typical uses of XSS attacks leak information, such as authentication credentials to a bank website; exploit browser vulnerabilities and perhaps load other malware onto users' systems; or contribute to a social engineering effort to trick users into revealing information, such as passwords. Not only can a single XSS exploit do the things listed above, but an XSS vulnerability can be used to create rapidly spreading malware, and thus compound multiply the damage caused by a single exploit. Figure 1.3b shows the number of systems infected in 24 hours by five prominent pieces of malware (this data comes from Grossman [30]). What has come to be known as the "MySpace worm" spread by means of an XSS vulnerability, and it reached approximately one million users' systems in 24 hours. By contrast, the other four spread by means of memory errors and reached dramatically fewer systems in the same time period. The reason for this disparity is that the four memory error-based worms quickly flooded the network and prevented their own spread, while the MySpace worm followed users' activity and only sent network traffic to real systems. The MySpace worm did not perform any malicious activity beyond spreading, but it highlights what could happen.

1.3 Common Defenses

Usually developers approach input validation by handling each input in isolation. This approach leaves two major opportunities for error: the validation may be omitted, and the validation may be incorrect. Most web application programming languages provide an unsafe default for passing untrusted input to the client or database. Typically, including the untrusted input directly in the output page or query is the most straightforward way of passing such data. Nothing in the web application programming language, compiler, or runtime system alerts the developer that validation is omitted. Hence, static or dynamic information flow analysis is needed to ensure that all untrusted inputs are validated.

The validation routines used may also have errors. Many suggested techniques for input validation are signature-based, including enumerating known "bad" strings necessary for injection attacks, limiting the length of input strings, or more generally, using regular expressions for filtering. An alternative for preventing SQL injection is to alter inputs, perhaps by adding slashes in front of quotes to prevent the quotes that surround literals from being closed within the input (*e.g.*, with PHP's `addslashes` function and PHP's `magic_quotes` setting, for example). All of these techniques are an improvement over unregulated input, but they all have weaknesses. None of them can say anything about the syntactic structure of the generated queries or web pages, and all may still admit bad input; for example, regular expression filters may be under-restrictive. Determining the language of potentially dangerous strings is especially problematic for XSS, as Section 4.1 explains. Often if a single function is designated as the input validation function, it is designed to prevent attacks in all settings. It is therefore overly restrictive for certain settings and is sometimes intentionally omitted. Hence, an analysis of string values or of the string values that validation routines produce is needed in order to ensure attacks cannot occur.

1.4 Dissertation Structure

Dynamic code generation and execution is one form of metaprogramming, and we view the input validation problems described above as metaprogramming problems. The general approach we take for each problem is to formulate first a formal characterization of the problem. SQL injection is conceptually simpler and cleaner than XSS, so we address it first, presenting a formal characterization of SQL injection in Chapter 2 [85]. Then we design, implement, and experiment with approaches to address the problem. Runtime techniques to prevent attacks are the simplest because only a single concrete value must be considered at a time. Chapter 2 continues with a sound, complete, and low-overhead runtime approach to prevent SQL injection. Although runtime prevention can be effective, static analysis has the advantage that it can be used to catch errors early in the software development lifecycle and enable cheaper solutions, it can analyze code fragments or code modules, it can provide useful feedback to software developers, and it can remove the runtime overhead of general approaches by determining that in certain cases general checks are not needed. Chapter 3 presents a sound, precise, and scalable static analysis to find SQL injection vulnerabilities in web application code [94].

Chapters 2 and 3 lay the foundations for runtime checks and static analysis for input validation errors, and Chapter 4 applies those foundations to XSS [95]. Chapter 4 introduces a characterization of XSS and explores the system-related details involved in it. Because this characterization can be used in a runtime context analogous to the one described in Chapter 2, it is not explored further. Chapter 4 does, however, explore how to apply the characterization to a static analysis and studies the nature and pervasiveness of XSS vulnerabilities in real-world web applications.

As Chapters 3 and 4 show, static analysis can be very effective for finding vulnerabilities in the restricted metaprogramming domain of web applications. However, even in this domain, the logic behind the generation of code can become too complex for effective static analysis in some cases. In particular, when web applications make significant use of dynamic

features, a static analyzer will have a difficult time determining what exactly the source code is. To overcome the hurdle of dynamic features and yet maintain the benefit of early bug detection, Chapter 5 proposes a directed, automated test input generation, or concolic, technique to find SQL injection vulnerabilities [96]. The concolic testing framework involves running a program both concretely and symbolically on a given input, gathering constraints, and resolving those constraints to generate a new input. In contrast to previous work on concolic testing, we approximate constraints by considering only a single variable occurrence per constraint, and thus support a richer constraint language than previous approaches. Chapter 5 presents results on finding known vulnerabilities using this approach. Finally, Chapter 6 concludes and describes some future directions based on this work.

Chapter 2

SQL Injection: Characterization and Runtime Checking

We argued in Chapter 1 that SQL injection attacks are common and potentially very damaging and that the current programming practice is ill-suited to preventing them. Current techniques treat inputs as isolated strings and neither the APIs nor the runtime system provides any means of ensuring that the sanitization performed indeed prevents attacks. This is understandable in part because no principled characterization of SQL injection attacks has previously been available. This chapter presents a formal definition of SQL injection that combines information flow with sentential forms. Based on this definition, we then present a runtime approach to ensure that the web application server will never send an SQL injection attack to the backend database. This runtime technique has no false positives, no false negatives, and imposes low runtime overhead.

2.1 Introduction

An SQL command injection attack (SQLCIA) injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer intended. For example, if a database

contains user names and passwords, the application may contain code such as the following:

```
query = "SELECT * FROM accounts WHERE name='"
      + request.getParameter("name") + "' AND password='"
      + request.getParameter("pass") + "'";
```

This code generates a query intended to be used to authenticate a user who tries to login to a web site. However, if a malicious user enters "`badguy`" into the name field and "`'OR' a'='a`" into the password field, the query string becomes:

```
SELECT * FROM accounts WHERE name='badguy' AND password='' OR 'a'='a'
```

whose condition always evaluates to true, and the user will bypass the authentication logic.

The problem goes beyond simply failing to check input that is incorporated into a query. Even web applications that perform some checks on every input may be vulnerable. For example, if the application forbids the use of the single-quote in input (which may prevent legitimate inputs such as "`O'Brian`"), SQLCIAs may still be possible because numeric literals are not delimited with quotes. The problem is that web applications generally treat input strings as isolated lexical entities. Input strings and constant strings are combined to produce structured output (SQL queries) without regard to the structure of the output language (SQL).

A number of approaches to dealing with the SQLCIA problem have been proposed, but to the best of our knowledge, no formal definition for SQLCIAs has been given. Consequently, the effectiveness of these approaches can only be evaluated based on examples, empirical results, and informal arguments. This chapter fills that gap by formally defining SQLCIAs and presenting a sound and complete algorithm to detect SQLCIAs using this definition combined with parsing techniques [1].

```
<%!
  // database connection info
  String dbDriver = "com.mysql.jdbc.Driver";
  String strConn = "jdbc:mysql://sport4sale.com/sport";
  String dbUser = "manager";
  String dbPassword = "athltpass";

  // generate query to send
  String sanitizedName = replace(request.getParameter("name"),"'","''");
  String sanitizedCardType =
     replace(request.getParameter("cardtype"),"'","''");
  String query = "SELECT cardnum FROM accounts" + " WHERE uname='"
     + sanitizedName + "'" + " AND cardtype=" + sanitizedCardType + ";";

  try {
    // connect to database and send query
    java.sql.DriverManager.registerDriver(
        (java.sql.Driver) (Class.forName(dbDriver).newInstance()));
    javaq.sql.Connection conn =
        java.sql.DriverManager.getConnecion(strConn, dbUser, dbPassword);
    java.sql.Statement stmt = conn.createStatement();
    java.sql.ResultSet rs = stmt.executeQuery(query);

    // generate html output
    out.println("<html><body><table>");
    while(rs.next()) {
      out.println("<tr> <td>");
      out.println(rs.getString(1));
      out.println("</td> </tr>");
    }
    if (rs != null) { rs.close(); }
    out.println("</table> </body> </html>");
  } catch (Exception e)
  { out.println(e.toString()); }
\%>
```

Figure 2.1: A JSP page for retrieving credit card numbers.

2.2 Overview of Approach

Web applications have injection vulnerabilities because they do not constrain syntactically the inputs they use to construct structured output. Consider, for example, the JSP page

in Figure 2.1. The context of this page is an online store. The website allows users to store credit card information so that they can retrieve it for future purchases. This page returns a list of a user's credit card numbers of a selected credit card type (*e.g.*, Visa). In the code to construct a query, the quotes are "escaped" with the `replace` method so that any single quote characters in the input will be interpreted as literal characters and not string delimiters. This is intended to block attacks by preventing a user from ending the string and adding SQL code. However, `cardtype` is a numeric column, so if a user passes "`2 OR 1=1`" as the card type, all account numbers in the database will be returned and displayed.

We approach the problem by addressing its cause: we track through the program the substrings from user input and constrain those substrings syntactically. The idea is to block queries in which the input substrings change the syntactic structure of the rest of the query. Such queries are *command injection attacks* (SQLCIAs, in the context of database back-ends). We track the user's input by using meta-data, displayed as '⟪' and '⟫,' to mark the beginning and end of each input string. This meta-data follows the string through assignments, concatenations, etc., so that when a query is ready to be sent to the database, it has matching pairs of markers identifying the substrings from input. We call this annotated query an *augmented query*.

We want to forbid input substrings from modifying the syntactic structure of the rest of the query. To do this we construct an *augmented grammar* for augmented queries based on the standard grammar for SQL queries. In the augmented grammar, the only productions in which '⟪' and '⟫' occur have the following form:

$$nonterm \texttt{ ::= } \langle\!\langle \; symbol \; \rangle\!\rangle$$

where *symbol* is either a terminal or a non-terminal. For an augmented query to be in the language of this grammar, the substrings surrounded by '⟪' and '⟫' must be syntactically confined. By selecting only certain symbols to be on the *rhs* of such productions, we can specify the syntactic forms permitted for input substrings in a query.

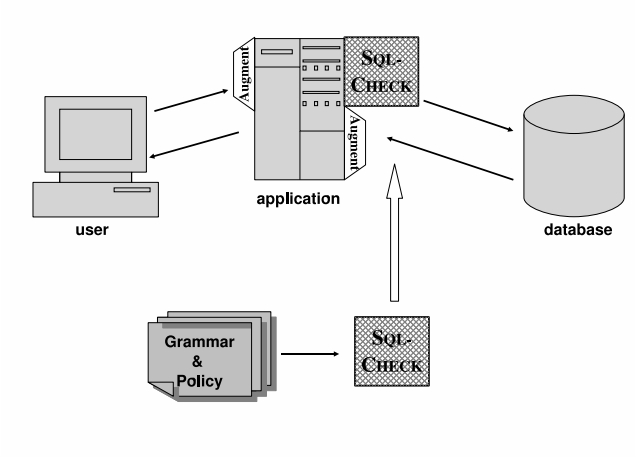


Figure 2.2: System architecture of SqlCheck.

One reason to allow input to take syntactic forms other than literals is for stored queries. Some web applications read queries or query fragments in from a file or database. For example, Bugzilla, a widely used bug tracking system, allows the conditional clauses of queries to be stored in a database for later use. In this context, a tautology is not an attack, since the conditional clause simply serves to filter out uninteresting bug reports. Persistent storage can be a medium for second order attacks [2], so input from them should be constrained, but if stored queries are forbidden, applications that use them will break. For example, in an application that allows conditional clauses to be stored along with associated labels, a malicious user may store "`val = 1; DROP TABLE users`" and associate a benign-sounding label so that an unsuspecting user will retrieve and execute it.

We use a parser generator to build a parser for the augmented grammar and attempt to parse each augmented query. If the query parses successfully, it meets the syntactic constraints and is legitimate. Otherwise, it fails the syntactic constraints and either is a command injection attack or is meaningless to the interpreter that would receive it.

Figure 2.2 shows the architecture of our runtime checking system. After SqlCheck is built using the grammar of the output language and a policy specifying permitted syntactic forms, it resides on the web server and intercepts generated queries. Each input that is to be propagated into some query, regardless of the input's source, gets augmented with

the meta-characters '⟪' and '⟫.' The application then generates augmented queries, which SqlCheck attempts to parse. If a query parses successfully, SqlCheck sends it sans the meta-data to the database. Otherwise, the query is blocked.

2.3 Formal Descriptions

This section formalizes the notion of a web application, and, in that context, formally defines an SQLCIA.

2.3.1 Problem Formalization

A web application has the following characteristics relevant to SQLCIAs:

- It takes input strings, which it may modify;
- It generates a string (*i.e.*, a query) by combining filtered inputs and constant strings. For example, in Figure 2.1, `sanitizedName` is a filtered input, and `"SELECT cardnum FROM accounts"` is a constant string for building dynamic queries;
- The query is generated without respect to the SQL grammar, even though in practice programmers write web applications with the intent that the queries be grammatical; and
- The generated query provides no information about the source of its characters/substrings.

In order to capture the above intuition, we define a *web application* as follows:

Definition 2.1 (Web Application)**.** *We abstract a* web application $P : \langle \Sigma^*, \ldots, \Sigma^* \rangle \rightarrow \Sigma^*$ *as a mapping from user inputs (over an alphabet* Σ*) to query strings (over* Σ*). In particular,* P *is given by* $\{\langle f_1, \ldots, f_n \rangle, \langle s_1, \ldots, s_m \rangle\}$ *where*

- $f_i : \Sigma^* \rightarrow \Sigma^*$ *is an* input filter;
- $s_i : \Sigma^*$ *is a* constant string.

The argument to P is an n-tuple of input strings $\langle i_1, \ldots, i_n \rangle$*, and P returns a* query $q = q_1 + \ldots + q_\ell$ *where, for* $1 \leq j \leq \ell$*,*

$$q_j = \begin{cases} s & \text{where } s \in \{s_1, \ldots, s_m\} \\ f(i) & \text{where } f \in \{f_1, \ldots, f_n\} \ \wedge \ i \in \{i_1, \ldots, i_n\} \end{cases}$$

That is, each q_j is either a static string or a filtered input.

Definition 2.1 says nothing about control-flow paths or any other execution model, so it is not tied to any particular programming paradigm. This definition does not require web applications to define recursively enumerable functions—the i^{th} substring used to construct a query need not be the same constant string or the same filtering function applied to the input at the same position for each input tuple. The single requirement that this definition makes on web applications is that for each query that a web application generates, certain substrings come from untrusted input and certain substrings come from the application's source. same constant string or the same

In order to motivate our definition of an SQLCIA, we return to the example JSP code shown in Figure 2.1. If the user inputs "`John`" as his user name and perhaps through a dropdown box selects credit card type "`2`" (both expected inputs), the generated query will be:

```
SELECT cardnum FROM accounts WHERE uname='John' AND cardtype=2
```

As stated in Section 2.2, a malicious user may replace the credit card type in the input with "`2 OR 1=1`" in order to return all stored credit card numbers:

```
SELECT cardnum FROM accounts WHERE uname='John' AND cardtype=2 OR 1=1
```

Figure 2.3 shows a parse tree for each query. Note that in Figure 2.3a, for each substring from input there exists a node in the parse tree whose descendant leaves comprise the entire input substring and no more: *lit* for the first substring and *num_lit*/*value* for the second, as shown with shading. No such parse tree node exists for the second input substring in

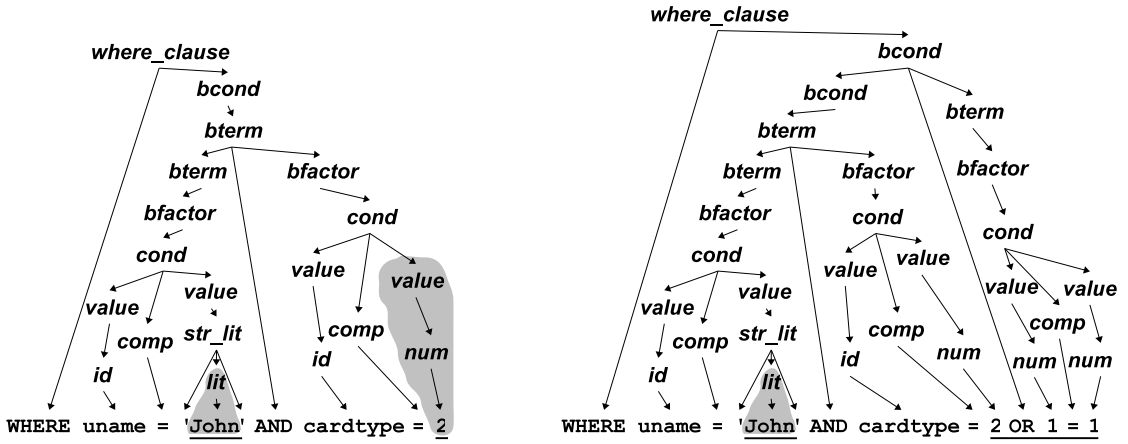


Figure 2.3: Parse trees for `WHERE` clauses of generated queries. Substrings from user input are underlined.

Figure 2.3b. This distinction is common to all examples of legitimate vs. malicious queries that we have seen. The intuition behind this distinction is that the malicious user attempts to cause the execution of a query beyond the constraints intended by the programmer, while the normal user does not attempt to break any such constraints. We use this distinction as our definition of an SQLCIA, and we state our definition in terms of sentential forms.

Let $G = (V, \Sigma, S, R)$ be a context-free grammar with nonterminals V, terminals Σ, a start symbol S, and productions R. Let '$\Rightarrow_G$' denote "derives in one step" so that $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ if $A \rightarrow \gamma \in R$, and let '$\Rightarrow_G^*$' denote the reflexive transitive closure of '$\Rightarrow_G$.' If $S \Rightarrow_G^* \gamma$, then γ is a "sentential form." The following definition formalizes syntactic confinement:

Definition 2.2 (Syntactic Confinement)**.** Given a grammar $G = (V, \Sigma, S, R)$ and a string $\sigma = \sigma_1\sigma_2\sigma_3 \in \Sigma^*$, σ_2 is *syntactically confined* in σ iff there exists a sentential form $\sigma_1 X \sigma_3$ such that $X \in (V \cup \Sigma)$ and $S \Rightarrow_G^* \sigma_1 X \sigma_3 \Rightarrow_G^* \sigma_1\sigma_2\sigma_3$.

We define our policy such that user-provided substrings must be syntactically confined.

Definition 2.3 (SQL Command Injection Attack)**.** Given a web application $P =$

$\{\langle f_1, \ldots, f_n\rangle, \langle s_1, \ldots, s_m\rangle\}$ and a query string q constructed from the input $\langle i_1, \ldots, i_k\rangle$, q is a *command injection attack* if there exists $i \in \{i_1, \ldots, i_k\}$ and $f \in \{f_1, \ldots, f_m\}$ such that $q = q_1 + f(i) + q_2$ and $f(i)$ is not syntactically confined in q with respect to the SQL grammar.

We can parameterize this definition with a policy $U \subseteq (V \cup \Sigma)$ by modifying Definition 2.2 to require that $X \in U$ rather than $X \in (V \cup \Sigma)$.

Moreover, below the top level of "insert," "update," and "delete" statements, SQL is a functional language—no subqueries can have side effects. Hence, expressions parsed under a single nonterminal generally evaluate to a single value and do not otherwise influence the evaluation of the rest of the query.

Definition 2.3 requires the user input to be derived from some symbol within the context of a sentential form. This requirement is significant because the same substring may have multiple syntactic forms when considered in isolation. For example, "`DROP TABLE employee`" could be viewed either as a `DROP` statement or as string literal data if not viewed in the context of a whole query. A column name could be a *select_sublist* or it could be *row_value_constructor* depending on the context, and a policy may allow one and forbid the other.

False negatives: Note that these definition do not include all forms of dangerous or unexpected behavior. Definition 2.1 provides no means of altering the behavior of the web application (*e.g.*, through a buffer overflow). Definition 2.3 assumes that the portions of the query from constant strings represent the programmer's intentions. If a programmer mistakenly includes in the web application a query to drop a needed table, that query would not be considered an SQLCIA.

False positives: An untrusted user could add innocuous code to a query. By our definition such a query may be considered an attack even though the substring from the user does not cause any harm. We do not consider this to be a limitation because if the added

code indeed does nothing, then there is no reason to allow it. Additionally, Definition 2.3 constrains the web application to use input only where a syntactically confined substring is permitted. By Definition 2.2, a syntactically confined substring has a unique root in the query's parse tree. Consider, for example, the following query construction:

```
query = "SELECT * FROM tbl WHERE col " + input;
```

If the variable `input` had the value `"> 5"`, the query would be syntactically correct. However, if the grammar uses a rule such as "$e \rightarrow e\ op_r\ e$" for relational expressions, then the input cannot have a unique root, and this construction will only generate SQLCIAs.

However, we believe these limitations are appropriate in this setting. By projecting away the possibility of the application server getting attacked, we can focus on the essence of the SQLCIA problem. Regarding the programmer's intentions, none of the literature we have seen on this topic ever calls into question the correctness of the constant portions of queries (except Fugue [20], Gould et al.'s work on static type checking of dynamically generated query strings [28], and our earlier work on static analysis for web application security [92], which consider this question to some limited degree). Additionally, programmers generally do not provide formal specifications for their code, so taking the code as the specification directs us to a solution that is fitting for the current practice. Finally, we have not encountered any examples either in the literature or observed in practice of constructed queries where the input cannot possibly be a valid syntactic form.

2.3.2 Algorithm for Runtime Enforcement

Given web application P, query string q generated by P, and input $\langle i_1, \ldots, i_n \rangle$, we need an algorithm $\mathcal{A}$ to decide whether q is an SQLCIA. The algorithm $\mathcal{A}$ must check whether the substrings $f_j(i_j)$ in q are syntactically confined, but the web application does not automatically provide information about the source of a generated query's substrings. Since the internals of the web application are not accessible directly, we need a means of tracking the input through the web application to the constructed query. For this purpose we use

meta-characters '$(\!|$' and '$|\!)$,' which are not in Σ. We modify the definition of the filters such that for all filters f,

- $f : (\Sigma \cup \{(\!|, |\!)\})^* \rightarrow (\Sigma \cup \{(\!|, |\!)\})^*$; and
- for all strings $\sigma \in \Sigma^*$, $f((\!|\sigma|\!)) = (\!|f(\sigma)|\!)$.

By *augmenting* the input to $\langle (\!|i_1|\!), \ldots, (\!|i_n|\!) \rangle$, we can determine which substrings of the constructed query come from the input.

Definition 2.4 (Augmented Query)**.** *A query q^a is an* augmented query *if it was generated from augmented input,* i.e.*, $q^a = P((\!|i_1|\!), \ldots, (\!|i_n|\!))$.*

We now describe an algorithm for checking whether a query is an SQLCIA. This algorithm is initialized once with the SQL grammar and a policy stated in terms of allowable terminals and nonterminals, with which the algorithm constructs an *augmented grammar*.

Definition 2.5 (Augmented Grammar)**.** *Given a grammar $G = \{V, \Sigma, S, R\}$ and a policy $U \subseteq V \cup \Sigma$, an* augmented grammar *G^a has the property that an augmented query $q^a = P((\!|i_1|\!), \ldots, (\!|i_n|\!))$ is in $\mathcal{L}(G^a)$ iff:*

- *The query $q = P(i_1, \ldots, i_n)$ is in $\mathcal{L}(G)$; and*
- *For each matching pair of metacharacters in q^a such that $q^a = \sigma_1(\!|\sigma_2|\!)\sigma_3$, there exists a symbol $X \in U$ such that $S \Rightarrow_G^* \sigma_1 X \sigma_3 \Rightarrow_G^* \sigma_1\sigma_2\sigma_3 = q$, where q is q^a with all metacharacters removed.*

A natural way to construct an augmented grammar G^a from G and U is to create a new production rule for each $u \in U$ of the form $u^a \rightarrow (\!|u|\!) \mid u$, and replace all other *rhs* occurrences of u with u^a. We give our construction in Algorithm 2.6.

Algorithm 2.6 (Grammar Augmentation)**.** *Given a grammar $G = \langle V, \Sigma, S, R \rangle$ and a policy $U \subseteq V \cup \Sigma$, we define G's augmented grammar as:*

$$G^a = \langle V \cup \{v^a | v \in U\}, \Sigma \cup \{(\!|, |\!)\}, S, R^a \rangle$$

select_stmt ::= `SELECT` *select_list from_clause*
| `SELECT` *select_list from_clause where_clause*
select_list ::= *id_list* | `*`
id_list ::= *id* | *id* `,` *id_list*
from_clause ::= `FROM` *tbl_list*
tbl_list ::= *id_list*
where_clause ::= `WHERE` *bool_cond*
bcond ::= *bcond* `OR` *bterm* | *bterm*
bterm ::= *bterm* `AND` *bfactor* | *bfactor*
bfactor ::= `NOT` *cond* | *cond*
cond ::= *value comp value*
value ::= *id* | *str_lit* | *num*
str_lit ::= `'` *lit* `'`
comp ::= `=` | `<` | `>` | `<=` | `>=` | `!=`

Figure 2.4: Simplified grammar for the `SELECT` statement.

where v^a *denotes a fresh non-terminal. Given* $rhs = v_1 \dots v_n$ *where* $v_i \in V \cup \Sigma$, *let* $rhs^a = w_1 \dots w_n$ *where*

$$w_i = \begin{cases} {v_i}^a & \text{if } v_i \in U \\ v_i & \text{otherwise} \end{cases}$$

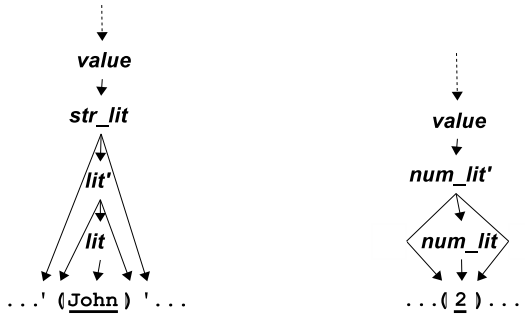
R^a *is given by:*

$$\begin{aligned} R^a &= \{v \to rhs^a \mid v \to rhs \in R\} \\ &\cup \{v^a \to v \mid v \in U\} \quad \cup \quad \{v^a \to (\!|v|\!) \mid v \in U\} \end{aligned}$$

To demonstrate this algorithm, consider the simplified grammar for SQL's `SELECT` statement in Figure 2.4. This grammar was used to generate the parse trees in Figure 2.3. If a security policy of $U = \{cond, id, num, lit\}$ is chosen, the result of Algorithm 2.6 is shown in Figure 2.5. Suppose the queries shown in Figure 2.3 were augmented. Using the augmented grammar, the parse tree for the first query would look the same as Figure 2.3a, except that the subtrees shown in Figures 2.6a and 2.6b would be substituted in for the first and second input strings, respectively. No parse tree could be constructed for the second augmented query.

select_stmt ::= SELECT *select_list from_clause*
| SELECT *select_list from_clause where_clause*
select_list ::= *id_list* | *
id^a ::= *id* | ⟪ *id* ⟫
id_list ::= id^a | id^a , *id_list*
from_clause ::= FROM *tbl_list*
tbl_list ::= *id_list*
where_clause ::= WHERE *bcond*
bcond ::= *bcond* OR *bterm* | *bterm*
bterm ::= *bterm* AND *bfactor* | *bfactor*
bfactor ::= NOT $cond^a$ | $cond^a$
$cond^a$ ::= *cond* | ⟪ *cond* ⟫
cond ::= *value comp value*
value ::= id^a | *str_lit* | num^a
num^a ::= *num* | ⟪ *num* ⟫
lit^a ::= *lit* | ⟪ *lit* ⟫
str_lit ::= ' lit^a '
comp ::= = | < | > | <= | >= | !=

Figure 2.5: Augmented grammar for grammar shown in Figure 2.4. New/modified productions are shaded.



(a) (b)

Figure 2.6: Parse tree fragments for an augmented query.

A GLR (generalized LR) parser generator [61] can be used to generate a parser for an augmented grammar G^a.

Algorithm 2.7 (SQLCIA Prevention)**.** *Here are steps of our algorithm $\mathcal{A}$ to prevent SQLCIAs and invalid queries:*

1. *Intercept augmented query q^a;*
2. *Attempt to parse q^a using the parser generated from G^a;*
3. *If q^a fails to parse, raise an error;*
4. *Otherwise, if q^a parses, strip all occurrences of '$\Uparrow$' and '$\Downarrow$' out of q^a to produce q and output q.*

2.3.3 Correctness and Complexity

We now argue that the algorithms given in Section 2.3.2 are correct with respect to the definitions given in Section 2.3.1. We also state the complexity of our runtime enforcement algorithm.

Theorem 2.8 (Soundness and Completeness)**.** *For all $\langle i_1, \ldots, i_n \rangle$, Algorithm 2.7 will permit query $q = P(i_1, \ldots, i_n)$ iff $q \in \mathcal{L}(G)$ and q is not an SQLCIA.*

Proof. Step 2 of Algorithm 2.7 attempts to parse the augmented query $q^a = P(\Uparrow i_1 \Downarrow, \ldots, \Uparrow i_n \Downarrow)$. If q^a does not include any metacharacters, then it cannot be an SQLCIA, and it will be sent to the database iff it parses. Algorithm 2.6 specifies that for each matching pair of metacharacters that q^a includes, q^a parses under the augmented grammar iff the following derivations exist: $S \Rightarrow^*_{G^a} \sigma_1^a X^a \sigma_3^a \Rightarrow_{G^a} \sigma_1^a \Uparrow X \Downarrow \sigma_3^a \Rightarrow^*_{G^a} \sigma_1^a \Uparrow \sigma_2^a \Downarrow \sigma_3^a = q^a$, where $X \in U$. From such a derivation, we can construct the derivation $S \Rightarrow^*_G \sigma_1 X \sigma_3 \rightarrow^*_G \sigma_1\sigma_2\sigma_3 = P(i_1, \ldots, i_n) = q$. This derivation shows that the query is not a command injection attack according to Definition 2.3. Conversely, for any query q that is not a command injection attack, a derivation for q^a under G^a can be constructed from q's derivation under G. Therefore, q^a fails to parse iff either q is a command injection attack or q does not parse and hence is garbled. If q^a fails to parse, step 3 will prevent q from being executed. If q^a parses, step 4 causes the query to be executed. □

Theorem 2.9 (Time Complexity)**.** *The worst-case time bound on Algorithm 2.7 is:*

$$\left.\begin{array}{l} O(|q|) \\ O(|q|^2) \\ O(|q|^3) \end{array}\right\} \; \textit{if } G^a \textit{ is } \left\{\begin{array}{l} \textit{LALR} \\ \textit{not LALR but is deterministic} \\ \textit{non-deterministic} \end{array}\right.$$

Proof. These time bounds follow from known time-bounds for classes of grammars [1]. Achieving them for Algorithm 2.7 is contingent on the parser generator being able to handle each case without using an algorithm for a more expressive class of grammar. □

2.4 Applications

Although we have so far focused on examples of SQL command injections, our definition and algorithm are general and apply to varying degrees to other settings that generate structured, meaningful output from user-provided input. We discuss three other common forms of command injections.

2.4.1 Cross Site Scripting

Web sites that display input data are subject to XSS attacks. Chapter 3 addresses XSS directly, but we discuss here the possibility of applying our characterization of SQL injection to XSS. As an example of XSS, consider an auction website that allows users to put items up for bid. The site displays a list of item numbers where each item number links to a URL to bid on the corresponding item. Suppose that an attacker enters as the item to add:

```
><script>document.location=
  'http://www.xss.com/cgi-bin/cookie.cgi?
  '%20+document.cookie</script
```

When a user clicks on the attacker's item number, the text in the URL will be parsed and interpreted as JavaScript. This script sends the user's cookie to `http://www.xss.com/`, the attacker's website. Note that the string provided by the attacker is not syntactically confined, since the first character completes a preceding tag. However, other instances

of XSS may, for example, add a scriptable attribute to a tag and set the script to leak confidential information as in the above example. Hence syntactic confinement can catch some but not all XSS attacks.

2.4.2 XPath Injection

A web application that uses an XML document/database for its back-end storage and accesses the data through dynamically constructed XPath expressions may be vulnerable to XPath injection attacks [50]. This is closely related to the problem of SQLCIAs, but the vulnerability is more severe because:

- XPath allows one to query all items of the database, while an SQL DBMS may not provide a "table of tables," for example; and
- XPath provides no mechanism to restrict access on parts of the XML document, whereas most SQL DBMSs provide a facility to make parts of the database inaccessible.

The following piece of ASP code is vulnerable to XPath injection:

```
XPathExpression expr = nav.Compile("string(//user[name/text()='"
                       + TextBox1.Text + "' and password/text()='"
                       + TextBox2.Text + "']/account/text()");
```

Entering a tautology as in Figure 2.3b would allow an attacker to log in, but given knowledge of the XML document's node-set, an attacker could enter:

```
NoUser'] | P | //user[name/text()='NoUser
```

where `P` is a node-set. The surrounding predicates would always be false, so the constructed XPath expression would return the string value of the node-set `P`. In this attack, the user input is not syntactically confined, but other attacks may succeed despite being syntactically confined. A syntactically confined XPath sub-expression will not be able to introduce new

side effects, but by using various axes (*e.g.*, parent, child, etc.), the sub-expression from user input can access a part of the XML document other than what has been specified by the rest of the expression.

2.4.3 Shell Injection

Shell injection attacks occur when input is incorporated into a string to be interpreted by the shell. For example, if the string variable `filename` is insufficiently sanitized, the PHP code fragment:

```
exec("open(".\$filename.")");
```

will allow an attacker to be able to execute arbitrary shell commands if `filename` is not a valid syntactic form in the shell's grammar. This vulnerability is not confined to web applications. A `setuid` program with this vulnerability allows a user with restricted privileges to execute arbitrary shell commands as `root`. Checking the string to ensure that each substring from input is syntactically confined would prevent this attack. However, using backticks a subexpression could do damage to a user's system without being syntactically unconfined.

2.5 Implementation

We implemented the query checking algorithm as SqlCheck. We had a few goals in implementing our approach; we wanted to: (1) see whether we would get any false positives or false negatives in our tests in order to evaluate our characterization against real-world data, (2) measure the runtime overhead, and (3) understand the usability of our approach. SqlCheck is generated using an input file to `flex` and an input file to `bison`. For meta-characters, we use two randomly generated strings of four alphabetic characters: one to represent '⦇' and the other to represent '⦈.' We made this design decision based on two considerations: (1) the meta-characters should not be removed by input filters, and (2) we can "escape" instances of those strings in user input.

First, we selected alphabetic characters because some input filtering functions restrict or remove certain characters, but generally alphabetic characters are permitted. The common exceptions are filters for numeric fields which allow only numeric characters. In this case either the meta-characters can be added after applying an filter, or they can be stripped off leaving only numeric data which cannot change the syntactic structure of the generated query. We added them after the filter, where applicable.

Second, we can check, at the point where the strings are added, whether those strings occur in the user input. If they do, we can prepend a designated character to them so that SqlCheck will not interpret them as metacharacters. If the augmented query parses successfully, SqlCheck will strip out the metacharacters and remove the prepended escape character from other strings that would otherwise represent metacharacters.

The input to `flex` requires roughly 70 lines of manually written C code to distinguish meta-characters from string literals, column/table names, and numeric literals when they are not separated by the usual token delimiters.

The algorithm allows for a policy to be defined in terms of which non-terminals in the SQL grammar are permitted to be at the root of a user input substring in a derivation. For the evaluation we selected literals, names, and arithmetic expressions to be permitted. Additional symbols can be added to the policy at the cost of one line in the `bison` input file per symbol, a find-and-replace on the added symbol, and a token declaration. Additionally, if the DBMS allows SQL constructs not recognized by SqlCheck, they can be added straightforwardly by updating the `bison` input file. The `bison` utility includes a `glr` (generalized LR) mode, which can be used if the augmented grammar is not LALR. For the policy choice used here, the augmented grammar is LALR.

2.6 Evaluation

This section presents the setup and results of our evaluation of SqlCheck. Whereas Sections 2.6.1 and 2.6.2 evaluate SqlCheck quantitatively, Section 2.6.3 evaluates Sql-

Subject	LOC		Query Checks Added	Query Sites	Metachar Pairs Added	External Query Data
	PHP	JSP				
Employee Directory	2,801	3,114	5	16	4	39
Events	2,819	3,894	7	20	4	47
Classifieds	5,540	5,819	10	41	4	67
Portal	8,745	8,870	13	42	7	149
Bookstore	9,224	9,649	18	56	9	121

Table 2.1: Subject programs used in our empirical evaluation.

CHECK qualitatively primarily in terms of design decisions.

2.6.1 Evaluation Setup

To evaluate our implementation, we selected five web applications that have been used for previous evaluations in the literature [32]. Each of these web applications was produced with a web application generator and so is provided in multiple web-programming languages. We used the PHP and JSP version of each to evaluate the applicability of our implementation across different languages. Although the notion of "applicability across languages" is somewhat qualitative, it is significant: the more language-specific an approach is, the less it is able to address the broad problem of SQLCIAs (and command injections in general). For example, an approach that involves using a modified interpreter [70, 73] is not easily applicable to a language like Java (*i.e.*, JSP and servlets) because Sun is unlikely to modify its Java interpreter for the sake of web applications. To the best of our knowledge, this is the first evaluation in the literature run on web applications written in different languages.

Table 2.1 lists the subjects, giving for each subject its name, the number of lines of code in the PHP and JSP versions, the number of pairs of meta-characters added, the number of input sites, the number of calls to SQLCHECK added, and the number of points at which complete queries are generated. The number of pairs of meta-characters added was less than the number of input sites because in these applications, most input parameters were passed through a particular function, and by adding a single pair of meta-characters in

this function, many inputs did not need to be instrumented individually. For a similar reason, the number of added calls to SqlCheck is less than the number of points at which completed queries are generated: In order to make switching DBMSs easy, a wrapper function was added around the database's `SELECT` query function. Adding a call to SqlCheck within that wrapper ensures that all `SELECT` queries will be checked. Calling SqlCheck from the JSP versions requires a Java Native Interface (JNI) wrapper. We report both figures to indicate approximately the numbers of checks that need to be added for web applications of this size that are less cleanly designed. For this evaluation, we added the meta-characters and the calls to SqlCheck manually; in the future, we plan to automate this task using a static flow analysis.

In addition to real-world web applications, the evaluation needed real-world inputs. To this end we used a set of URLs provided by Halfond and Orso with assistance from another student who had experience with penetration testing of web applications [32]. These URLs were generated by first compiling one list of attack inputs, which were gleaned from CERT/CC advisories and other sources that list vulnerabilities and exploits, and one list of legitimate inputs. The data type of each input was also recorded. Then each parameter in each URL was annotated with its type. Two lists of URLs were then generated, one Attack list and one Legit list, by substituting inputs from the respective lists into the URLs in a type consistent way. Each URL in the Attack list had at least one parameter from the list of attack inputs, while each URL in the Legit list had only legitimate parameters. Finally, the URLs were tested on unprotected versions of the web applications to ensure that the Attack URLs did, in fact, execute attacks and the Legit URLs resulted in normal, expected behavior.

The machine used to perform the evaluation runs Linux kernel 2.4.27 and has a 2 GHz Pentium M processor and 1 GB of memory.

Lang.	Subject	Queries		Timing (ms)	
		Legitimate (attempted/ allowed)	Attacks (attempted/ prevented)	Mean	Std Dev
PHP	Employee Directory	660 / 660	3937 / 3937	3.230	2.080
	Events	900 / 900	3605 / 3605	2.613	0.961
	Classifieds	576 / 576	3724 / 3724	2.478	1.049
	Portal	1080 / 1080	3685 / 3685	3.788	3.233
	Bookstore	608 / 608	3473 / 3473	2.806	1.625
JSP	Employee Directory	660 / 660	3937 / 3937	3.186	0.652
	Events	900 / 900	3605 / 3605	3.368	0.710
	Classifieds	576 / 576	3724 / 3724	3.134	0.548
	Portal	1080 / 1080	3685 / 3685	3.063	0.441
	Bookstore	608 / 608	3473 / 3473	2.897	0.257

Table 2.2: Precision and timing results for SqlCheck.

2.6.2 Results

Table 2.2 shows, for each web application, the number of attacks attempted (using URLs from the Attack list) and prevented, the number of legitimate uses attempted and allowed, and the mean and standard deviation of times across all runs of SqlCheck for that application. SqlCheck successfully prevented all attacks and allowed all legitimate uses. Theorem 2.8 predicted this, but these results serve as a sanity check for both the characterization and the implementation. Additionally, the timing results show that SqlCheck is quite efficient. Round trip time over the Internet varies widely, but 80–100ms is typical. Consequently, SqlCheck's overhead is imperceptible to the user, and is also reasonable for servers with heavier traffic.

In addition to the figures shown in Table 2.2, our experience using SqlCheck provides experimental results. Even in the absence of an automated tool for inserting metacharacters and calls to SqlCheck, this technique could be applied straightforwardly. Most existing techniques for preventing SQLCIAs either cannot make syntactic guarantees (*e.g.*, regular expression filters) or require a tool with knowledge of the source language. For example, a type-system based approach requires typing rules in some form for each construct in the source language. As another example, a technique that generates automata for use

in dynamic checking requires a string analyzer designed for the source language. Forgoing the use of the string analyzer would require an appropriate automaton for each query site to be generated manually, which most web application programmers cannot/will not do. In contrast, a programmer without a tool designed for the source language of his choice can still use SqlCheck to prevent SQLCIAs.

2.6.3 Discussions

We now discuss some of our design decisions of the current implementation.

First, we used a single policy U for all test cases. In practice we expect that a simple policy will suffice for most uses. In general, a unique policy can be defined for each pair of input site (by choosing a different pair of strings to serve as delimiters) and query site (by generating an augmented grammar according to the desired policy for each pair of delimiters). However, even if U were always chosen to be $V \cup \Sigma$, SqlCheck would restrict the user input to syntactic forms in the SQL language. In the case where user input is used in a comparison expression, the best an attacker can hope to do is to change the number of tuples returned; no statements that modify the database, execute external code, or return columns other than those in the column list will be allowed.

Second, because the check is based on parsing, it would be possible to integrate it into the DBMS's own parser. From a software engineering standpoint, this does not seem to be a good decision. Web applications are often ported to different environments and interface with different backend DBMSs, so the security guarantees could be lost without the programmer realizing it.

Finally, the test cases used for the evaluation were generated by an independent research group from real-world exploits. However, they were not written by attackers attempting to defeat the particular security mechanism we used. It would be interesting to expose our implementation to motivated attackers, but, as this chapter argues, we expect that no attacks would succeed and no legitimate usage would fail.

2.7 Related Work

2.7.1 Input Filtering Techniques

Improper input validation accounts for most security problems in database and web applications. Many suggested techniques for input validation are signature-based, including enumerating known "bad" strings necessary for injection attacks, limiting the length of input strings, or more generally, using regular expressions for filtering. An alternative is to alter inputs, perhaps by adding slashes in front of quotes to prevent the quotes that surround literals from being closed within the input (*e.g.*, with PHP's `addslashes` function and PHP's `magic_quotes` setting, for example). Recent research efforts provide ways of systematically specifying and enforcing constraints on user inputs [7, 80, 81]. A number of commercial products, such as Sanctum's AppShield [79] and Kavado's InterDo [46], offer similar strategies. All of these techniques are an improvement over unregulated input, but they all have weaknesses. None of them can say anything about the syntactic structure of the generated queries, and all may still admit bad input; for example, regular expression filters may be under-restrictive. More significantly, escaping quotes can also be circumvented when subtle assumptions do not hold, as in the case of the second order attacks [2]. In the absence of a principled analysis to check these methods, security cannot be guaranteed.

2.7.2 Syntactic Structure Enforcement

Other techniques deal with input validation by enforcing that all input will take the syntactic position of literals. Both bind variables and parameters in stored procedures are parts of database programming APIs, and they function as placeholders for literals within queries, so that whatever they hold will be treated as literals and not as arbitrary code. SQLrand, a recently proposed instruction set randomization for SQL in web applications, has a similar effect [6]. It relies on a proxy to translate instructions dynamically, so SQL keywords entered as input will not reach the SQL server as keywords. The main disadvantages of such a system are its complex setup and security of the randomization key. Halfond and Orso

address SQL injection attacks through first building a model of legal queries and then ensuring that generated queries conform to this model via runtime monitoring [32], following a similar approach to Wagner and Dean's work on Intrusion Detection Via Static Analysis [19]. The precision of this technique is subject to both the precision of the statically constructed model and the tokenizing technique used. Given how their model is generated, user inputs are confined to statically defined syntactic positions. These techniques for enforcing syntactic structure do not extend to applications that accept or retrieve queries or query-fragments, such as those that retrieve stored queries from persistent storage (*e.g.*, a file or a database).

2.7.3 Runtime Enforcement

Many real-world web applications have vulnerabilities, even though measures such as those mentioned above are used. Vulnerabilities exist because of insufficiency of the technique, improper usage, incomplete usage, or some combination of these. Sections 5.6.2 and 3.6.2 survey testing and static analysis approaches, respectively, but this section surveys proposed techniques for preventing SQLCIAs at runtime. The most basic policy conceptually for runtime enforcement is string-level taint tracking, in which the runtime system designates inputs as tainted and select functions as sanitizers, and raises an exception if the argument string to the query function is tainted. Perl provides such a taint-tracking capability [91]. Note that string-level taint tracking differs from substring-level taint tracking in that the former assigns a single integrity level to a given string, whereas the latter can assign a different integrity level to each substring in an arbitrary partitioning of the string.

AMNESIA, by Halfond and Orso, uses Christensen *et al.*'s Java string analyzer to construct a policy requiring user inputs to be single tokens in constructed queries, and enforces that policy at runtime [32]. The effectiveness of this approach is limited by the string analysis' precision. Buehrer *et al.* also enforce a policy that user input must be a single token in the query, but they do not rely on a static analysis [8]. They bound user input, parse the query, and check whether the parse tree retains the same structure when the user

input is replaced by a single dummy node. Java provides a `PreparedStatement` API, which forces inputs in queries built with it to be string or numeric literals. Boyd and Keromytis sought to enforce this via instruction set randomization [47], *i.e.*, by randomizing the SQL keywords in the web application, so that users could not guess the keywords [6]. This technique cannot provide guarantees, because a user may guess the randomization key.

Bandhakavi *et al.* propose a somewhat different runtime technique that they call CANDID. They consider that a query is safe if all of the numeric inputs used to construct it have the value 1 and all the string values are strings of a's. At runtime, CANDID records the sequence of commands used to construct a given query, replays those commands replacing the inputs with 1's and strings of a's, and if the parse trees of the two queries differ, CANDID blocks the query. While this approach offers some flexibility in dealing with certain program constructs, it essentially ensures that all tainted substrings in a query take the syntactic position of literals.

Both Nguyen-Tuong *et al.* [70] and Pietraszek and Berghe [73] propose to enforce the same policy for PHP more rigorously. They modify the PHP interpreter to track taint information at the character level, tokenize the completed query, and check whether any tainted characters appear in any tainted characters. A modified interpreter has the advantage that it can add security guarantees to arbitrary web applications, but practical issues of deployment and system maintenance limit such a technique's effectiveness. Xu *et al.* propose a source-to-source translator for C that adds taint tracking [100]. It can be used to ameliorate the system maintenance problem by adding taint-tracking to new versions of the PHP interpreter's source code.

WASP, by Halfond *et al.*, enforces approximately the same policy for Java, but they use *positive tainting*, *i.e.*, they taint trusted strings, and allow only tainted characters in keywords unless the programmer specifies with a regular expression that user input may include certain keywords [34]. Additionally, instead of modifying the JVM, they provide a byte code instrumenter. In earlier work, we use delimiters to track user input into generated queries, and parse the queries based on a modified grammar to check whether the user

input is parsable under any of a permitted set of nonterminals within the query [85]. The policy we enforce allows user input to be parsable under any nonterminal, but in principle we could limit the allowable nonterminals. Although these runtime techniques to prevent SQL injection attacks are more precise than static analyses in general, some SQLCIVs are indicative of larger programming errors. Static analysis can help to find and fix such errors prior to deployment. Additionally, general runtime enforcement techniques incur more runtime overhead than appropriate, well-placed filters, which static analysis can check.

This work also relates to some recent work on security analysis for Java applications. Naumovich and Centonze propose a static analysis technique to validate role-based access control policies in J2EE applications [68]. They use a points-to analysis to determine which object fields are accessed by which EJB methods to discover potential inconsistencies with the policy that may lead to security holes. Koved *et al.* study the complementary problem of statically determining the access rights required for a program or a component to run on a client machine [52] using a dataflow analysis [45, 48].

2.7.4 Meta-Programming

To be put in a broader context, our research can be viewed as an instance of providing runtime safety guarantee for meta-programming [88]. Macros are a very old and established meta-programming technique; this was perhaps the first setting where the issue of correctness of generated code arose. Powerful macro languages comprise a complete programming facility, which enable macro programmers to create complex meta-programs that control macro-expansion and generate code in the target language. Here, basic syntactic correctness, let alone semantic properties, of the generated code cannot be taken for granted, and only limited static checking of such meta-programs is available. The levels of static checking available include none, syntactic, hygienic, and type checking. The widely used `cpp` macro pre-processor allows programmers to manipulate and generate arbitrary textual strings, and it provides no syntactic or semantic checking. The programmable syntax macros of Weise and Crew [97] work at the level of correct abstract-syntax tree (AST) fragments, and

guarantee that generated code is syntactically correct with respect (specifically) to the C language. Weise and Crew macros are validated via standard type checking: static type checking guarantees that AST fragments (e.g., Expressions, Statements, etc.) are used appropriately in macro meta-programs. Because macros insert program fragments into new locations, they risk "capturing" variable names unexpectedly. Preventing variable capture is called hygiene. Hygienic macro expansion algorithms, beginning with Kohlbecker *et al.* [51] provide hygiene guarantees. Recent work, such as that of Taha & Sheard [88], focuses on designing type checking of object-programs into functional meta-programming languages. A number of other proposals provide type-safe APIs for dynamic SQL, including, for example Safe Query Objects [14], SQL DOM [60], and Xen [5, 62]. These proposals suggest better programming models, but require programmers to learn a new API. In contrast, our approach does not introduce a new API, and it is suited to address the problems in the enormous number of programs that use existing database APIs. Other research efforts focus on type-checking polylingual systems [25, 29], but they do not deal with applications interfacing with databases such as web applications.

Chapter 3

Static Analysis for SQL Injection

In Chapter 1 we argued that SQL injection attacks are a common and significant problem, and in Chapter 2 we presented a formal definition of SQL injection attacks and a runtime technique to prevent them. This technique is effective for each place where it is used, but it does not guarantee that every query site is protected. Static analysis can guarantee that all query sites are protected, and it has other advantages (discussed below). This chapter presents a sound static analysis for finding SQL injection vulnerabilities based on the definition presented in the last chapter. This static analysis scales to large, real-world codes bases and has a low false-positive rate. Our evaluation of our implementation revealed previously unknown SQL injection vulnerabilities in real-world code.

3.1 Introduction

Our approach for runtime enforcement prevents SQL injection effectively in deployed software, but static approaches are desirable during software development and testing for three reasons. First, a single programming error often manifests itself as multiple different bugs, so statically verifying code to be free from one kind of error (*e.g.*, static type checking) helps to reduce the risk of other errors. Second, the overhead that general techniques incur significantly exceeds the overhead of appropriate, well-placed checks on untrusted input.

Even if the network latency dominates the overhead of a runtime check for a single user, the added overhead can prevent a server from functioning effectively under a heavy load of requests. Finally, some runtime techniques [70, 73] require a modified runtime system, which constitutes a practical limitation in terms of deployment and upgrading.

Static analyses to find SQL command injection vulnerabilities (SQLCIVs) have also been proposed, but none of them runs without user intervention and can guarantee the absence of SQLCIVs. String analysis-based techniques [12, 64] use formal languages to characterize conservatively the set of values a string variable may assume at runtime. They do not track the source of string values, so they require a specification, in the form of a regular expression, for each query-generating point or *hotspot* in the program—a tedious and error-prone task that few programmers are willing to do. Static taint analyses [43, 59, 99] track the flow of tainted (*i.e.*, untrusted) values through a program and require that no tainted values flow into hotspots. Because they use a binary classification for data (tainted or untainted), they classify functions as either being sanitizers (*i.e.*, all return values are untainted) or being security irrelevant. Because the policy that these techniques check is context-agnostic, it cannot guarantee the absence of SQLCIVs without being overly conservative. For example, if the `escape_quotes` function (which precedes quotes with an "escaping" character so that they will be interpreted as character literals and not as string delimiters) is considered a sanitizer, an SQLCIV exists but would not be found in an application that constructs a query using escaped input to supply an expected numeric value, which need not be delimited by quotes. Additionally, static taint analyses for PHP, a language that is widely used for web applications and is ranked fourth on the TIOBE programming community index [9], typically require user assistance to resolve dynamic includes (a construct in which the name of the included file is generated dynamically).

This chapter proposes a sound, automated static analysis algorithm to overcome the limitations described above. It is grammar-based; we model sets of string values as context free grammars (CFGs) and string operations as language transducers following Minamide [64]. This string analysis-based approach tracks the effects of string operations and retains the

structure of the values that flow into hotspots (*i.e.*, where query construction occurs). If all of each string in the language of a nonterminal comes from a source that can be influenced by a user, we label the nonterminal with one of two labels. We assign a "direct" label if a user can influence the source directly (as with `GET` parameters) and a "indirect" label if a user may be able to influence the source indirectly (as with data returned by a database query). Such labeling tracks the source of string values. We use our syntax-based definition of SQL injection attacks [85], which requires that input from a user be syntactically isolated within a generated query. This policy does not need user-provided specifications. Finally, we check policy conformance by first abstracting the labeled subgrammars out of the generated CFG to find their contexts. We then use regular language containment and context free language derivability [89], to check that each subgrammar derives only syntactically isolated expressions.

We have implemented this analysis for PHP, and applied it to several real-world web applications. Our tool scales to large code bases—it successfully analyzes PHP web applications of up to ∼300K loc as compared to ∼90K loc, the largest previously analyzed in the literature. It discovered many vulnerabilities, some previously unknown and some based on insufficient filtering, and generated few false positives.

3.2 Overview

In order to motivate our analysis, we first present an example web application with an SQLCIV and then give an overview of how our analysis checks such web applications against our policy.

3.2.1 Example Vulnerability

Figure 3.1 shows a code fragment excerpted from Utopia News Pro, a real-world news management system written in PHP; we will use this code to illustrate the key points of our algorithm. This code authenticates users to perform sensitive operations, such as

users.php

```
312 isset ($_GET['userid']) ? $userid = $_GET['userid'] : $userid = '';
313 if ($USER['groupid'] != 1)
314 { // permission denied
315     unp_msg($gp_permserror);
316     exit;
317 }
318 if ($userid == '')
319 {   unp_msg($gp_invalidrequest);
320     exit;
321 }
322 if (!eregi('[0-9]+', $userid))
323 {   unp_msg('You entered an invalid user ID.');
324     exit;
325 }
326 $getuser = $DB->query("SELECT * FROM `unp_user`"
327                       ."WHERE userid='$userid'");
328 if (!$DB->is_single_row($getuser))
329 {   unp_msg('You entered an invalid user ID.');
330     exit;
331 }
```

Figure 3.1: Example code with an SQLCIV.

managing user accounts and editing news sources. Initially, the variable `$userid` gets assigned data from a `GET` parameter, which a user can easily set to arbitrary values. The code then performs two checks on the value of `$userid` before incorporating it into an SQL query. The query should return a single row for a legitimate user, and no rows otherwise. From line 322 it is clear that the programmer intends `$userid` to be numeric, and from line 327 it is clear that the programmer intends that `$userid` evaluate to a single value in the SQL query for comparison to the `userid` column. However, because the regular expression on line 322 lacks anchors ('^' and '$' for the beginning and end of the string, respectively), any value for `$userid` that has at least one numeric character will be included into the generated query. If a user sets the `GET` parameter to "`1'; DROP TABLE unp_user; --`", this code will send to the database the following query:

```
SELECT * FROM `unp_user` WHERE userid='1'; DROP TABLE unp_user; --'
```

and delete user account data.

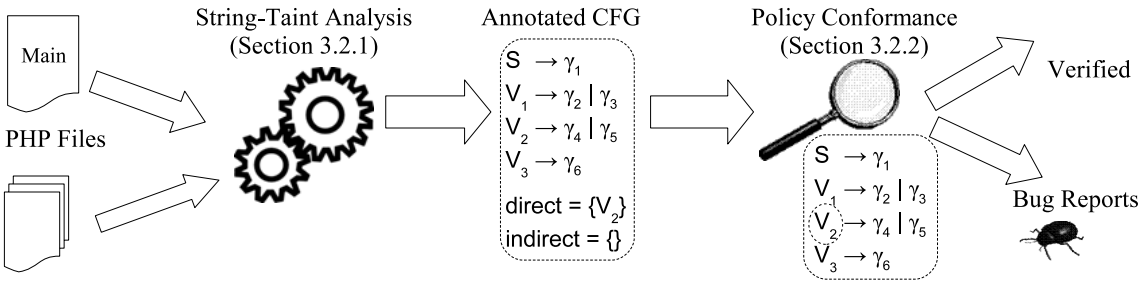


Figure 3.2: SQLCIV analysis workflow.

3.2.2 Analysis Overview

Our analysis takes PHP files as input and returns as output either a list of bug reports or the message "verified."

In order to provide useful bug reports, we first categorize sources of untrusted input as being either *direct* or *indirect*. Direct sources, such as `GET` parameters, provide data immediately from users; indirect sources, such as results from a database query, provide data from a source whose data may come from untrusted users. In practice, the risk of attacks from indirect sources is less severe than that of standard injection attacks for two reasons. First, programs often regulate which data is allowed to go into the database (or other sources), and second, attackers must pass through more steps and take more time to execute an indirect attack than to execute a standard injection attack.

Figure 3.2 shows a high-level overview of our analysis algorithm. It has two main phases. The first phase generates a conservative, annotated approximation of the query strings a program may generate; the annotations show which substrings in the query string are untrusted, *i.e.*, are from either direct or indirect sources. This phase is based on existing string analysis techniques [64] augmented to propagate taint information. The string-taint analyzer takes as input a PHP file that provides the top-level code for a web page (analogous to a `main` function in C). As it encounters dynamic include statements, it determines the possible string values of the argument to the `include`, and analyzes those files as well. The string-taint analyzer represents the set of query strings using an annotated

$$
\begin{array}{ll}
\mathit{query} & \rightarrow \mathit{query1}\texttt{'} \\
\mathit{query1} & \rightarrow \mathit{query2}\ \mathit{userid} \\
\mathit{query2} & \rightarrow \mathit{query3}\ \texttt{WHERE userid='} \\
\mathit{query3} & \rightarrow \texttt{SELECT * FROM `unp\_user`} \\
\mathit{userid} & \rightarrow \mathit{GETuid} \\
\mathit{GETuid} & \rightarrow \Sigma* \ [0\text{–}9]\ \Sigma*
\end{array}
$$

$$\mathsf{direct} = \{\mathit{GETuid}\} \quad \mathsf{indirect} = \{\}$$

Figure 3.3: Grammar productions of possible query strings from Figure 3.1.

context free grammar (CFG) — the nonterminals whose sub-languages represent untrusted strings are labeled with "direct" or "indirect," as appropriate. We choose to represent sets of strings with CFGs for several reasons: (1) tainted substring boundaries can be represented simply by labeling certain nonterminals; (2) our policy is grammar-based, and a CFG representation can capture context-free query construction that follows the policy; (3) regular expression-based string operations (common in PHP) can be represented as finite state transducers (FSTs), and the image of a CFG over an FST is context free.

The second phase of our analysis takes the annotated CFG produced by the first phase, and checks whether all strings in the language of the CFG are safe, *i.e.*, they are not SQL command injection attacks according to Definition 2.3. This analysis checks for common cases (of both SQL command injection attacks and attack-free grammars) efficiently by (1) abstracting the subgrammars that represent untrusted substrings out of the larger CFG, (2) determining the syntactic contexts of those subgrammars within the larger CFG, and (3) checking for (the absence of) policy violating strings in the languages of the subgrammars. For large grammars, this is significantly more efficient than checking the language of the generated CFG as a whole. If the policy conformance checker finds any violations, it issues a bug report. Because this algorithm is sound, if it does not issue any bug reports, the PHP code is guaranteed to be free from SQLCIVs.

To illustrate this algorithm on the example code in Figure 3.1, the string-taint analysis will produce the grammar productions shown in Figure 3.3; the annotations are shown in terms of sets of nonterminals annotated with "direct" and "indirect," respectively. The

regular expression notation on the right hand side of the last rule is notational shorthand intended to simplify the presentation. The grammar for *userid* reflects the regular expression match on line `14`, because the string-taint analyzer propogates the regular expression predicate. The nonterminal *GETuid* has the label "direct," because it represents strings from a `GET` parameter.

The policy-conformance checker then receives this labeled grammar. The check first replaces the annotated *GETuid* nonterminal with a new terminal $t \notin \Sigma$. By intersecting this modified grammar with an appropriate regular language, the checker finds that for all sentential forms $\sigma_1.GETuid.\sigma_2$ derivable from *query*, *GETuid* is between quotes in the syntactic position of a string literal. The checker therefore uses another regular language intersection to check the language rooted at *GETuid* for un-escaped quotes. When it finds them, it issues a bug report. The checker does not only check for the case of string literals, but that suffices for this example.

3.3 Analysis Algorithm

3.3.1 String-Taint Analysis

The first phase of our analysis combines ideas from static taint analysis with string analysis.

Adapting String Analysis: String analysis has the goal of producing a representation of all strings values that a variable may assume at a given program point. This goal does not imply any relationship between the structure of that representation and the way that the program produces those values. If the string analysis represents languages as finite automata and it determinizes intermediate results, the final DFA will have little relation to the program's dataflow [12]. Our analysis has the goal of producing not only a representation of all string values that a variable may assume, but also a function from string values to substrings whose values come from direct or indirect sources. In terms of Definition 2.3, we need to identify occurrences of $f(i)$ in each query string q.

(a)

```
$X = $UNTRUSTED;
if ($A) {
  $X = $X."s";
} else {
  $X = $X."s";
}
$Z = $X;
```

(b)

```
$X1 = $UNTRUSTED;
if ($A) {
  $X2 = $X1."s";
} else {
  $X3 = $X1."s";
}
$X4 = φ($X2, $X3);
$Z = $X4;
```

(c)

$$
\begin{array}{ll}
\mathit{UNTRUSTED} & \rightarrow \Sigma^* \\
X_1 & \rightarrow \mathit{UNTRUSTED} \\
X_2 & \rightarrow X_1\mathtt{s} \\
X_3 & \rightarrow X_1\mathtt{s} \\
X_4 & \rightarrow X_2 \mid X_3 \\
Z & \rightarrow X_4
\end{array}
$$

Figure 3.4: Grammar reflects dataflow.

Section 3.2.2 gives as one reason for using CFGs to represent sets of query strings that tainted substring boundaries can be represented by labeling certain nonterminals—thus the strings' derivations provide the function from strings to tainted substrings. In addition to that reason, a natural way to design and implement a CFG-based string analysis produces CFGs that reflect the program's dataflow, so taint annotations applied at untrusted sources appear in the final CFG. Minamide designed his string analysis this way, so we review the main steps of his analysis here and show how to adapt it to track taint information [64].

The contrived example program in Figure 3.4a serves to show that the generated grammar reflects the program's dataflow. The first step of the string analysis translates the program into static single assignment form, as shown in Figure 3.4b. SSA form makes data-dependencies explicit, so translating each assignment statement into a grammar production yields a CFG that reflects the program's dataflow, as in Figure 3.4c. By simply annotating the nonterminals corresponding to direct and indirect sources appropriately, we have a string-taint anlysis for programs with concatenation, assignments, and control flow.

In general, right hand sides of assignment statements may contain string functions, such as `escape_quotes()`, which adds a slash before each quote character in its argument.

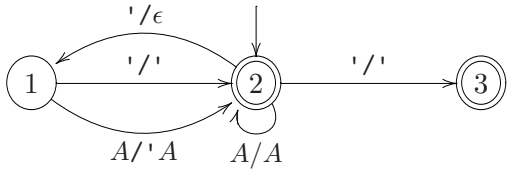


Figure 3.5: A finite state transducer equivalent of the function `str_replace("'", "'", $B)`; $A \in \Sigma \setminus \{'\}$.

Translating assignments into grammar productions then yeilds an extended CFG that has functions in its productions' right hand sides. Converting extended CFGs into standard CFGs requires some approximation, and Minamide models string operations as finite state transducers (FSTs) in order to capture their effects and make the approximation reasonably precise. Section 3.3.1 describes how FSTs can model string operations and how we track annotated sources through them in more detail.

Tracking Substrings through Filters: Definition 2.1 specifies that web applications can apply functions on strings to untrusted inputs. In order to avoid reporting many false positives, the string-taint analyzer must model the effects of filters and propagate annotations through them. This section reviews how Minamide's string analysis models the effects of filters and then describes how we adapt these techniques.

A transducer is an automaton with output. A finite state transducer is similar to a Mealy machine, except that a finite state transducer has one or more final states and may be non-deterministic. Many string operations that PHP provides as library functions behave as finite state transducers. For example, `str_replace` takes three strings as arguments: a pattern, a replacement, and a subject. The FST in Figure 3.5 describes the effects of `str_replace` when the pattern is "`'`" and the replacement is "`'`." The notation 'c_1/c_2' on the transitions means that on input character c_1, the transition can be taken and it will output c_2. In Figure 3.5, A matches any character except "`'`." The string analysis converts

a grammar production with a string operation, such as

$$x \rightarrow \texttt{escape\_quotes}(y)$$

into a standard grammar production by finding the image of the CFG rooted at the operation's argument (y) over the FST that the string operations represents.

A CFG has a *cycle* if a nonterminal X can derive a sentential form that contains X. If an extended grammar has the production shown above, and if

$$y \Rightarrow_G^* \alpha x \beta$$

then the `escape_quotes` operation occurs in a cycle. String operations that occur in cycles within the extended CFG must be approximated because the complete CFG rooted at the operation's argument (y) cannot be constructed independently of the string operation.

Some string operations are more expressive than finite state, or even context free, transducers. For example, PHP provides a regular expression-based replace function, `preg_replace`. Its three arguments are: a regular expression pattern, a parameterized replacement, and a subject. Within the replacement, an occurrence of "\n," where n is a number, represents the string matched by the expression between the n^{th} open parenthesis and its matching close parenthesis in the pattern. As an example,

```
preg_replace("/a([0-9]*)b/", "x\\1\\1y", "a01ba234b") = "x0101yx234234y"
```

The "`\\1`" puts the substring matched by the expression within the first pair of parentheses (because the number is `1`) into the output. Although the image of a CFL under a regular expression replacement is not necessarily context free (because of the ability to insert multiple copies of a regular expression match, as above), Mohri and Sproat describe how to approximate it using two FSTs [65].

The string analysis also uses a similar technique to maintain precision from conditional expressions when constructing the extended CFG. If the condition is a regular expression

```
INTERSECT(G = ⟨V_0, Σ_0, S_0, R_0⟩, FSA = ⟨Q, Σ, δ, q_0, q_f⟩)
 1  ⟨V, Σ, S, R⟩ ← NORMALIZE(⟨V_0, Σ_0, S_0, R_0⟩)
 2  V' ← ∅
 3  R' ← ∅
 4  for each (q_i, σ, q_j) in δ
 5  do V' ← V' ∪ {σ_ij}
 6     R' ← R' ∪ {σ_ij → σ}
 7  /* |rhs| = 0 */
 8  for each X → ϵ in R
 9  do for each q_i in Q
10        do V' ← V' ∪ {X_ii}
11           R' ← R' ∪ {X_ii → ϵ}
12  WkLst ← V'
13  for each α_ij in WkLst
14  do WkLst ← WkLst \ {α_ij}
15     /* |rhs| = 1 */
16     for each X → α in R
17     do if not (X_ij in V')
18           then WkLst ← WkLst ∪ {X_ij}
19                V' ← V' ∪ {X_ij}
20                TAINTIF(X, X_ij)
21        R' ← R' ∪ {X_ij → α_ij}
22     /* |rhs| = 2 */
23     for each X → αβ in R
24     do for each β_jk in V'
25        do if not (X_ik in V')
26              then WkLst ← WkLst ∪ {X_ik}
27                   V' ← V' ∪ {X_ik}
28                   TAINTIF(X, X_ik)
29           R' ← R' ∪ {X_ik → α_ij β_jk}
30  if S_0f in V'
31    then return ⟨V', Σ, S_0f, R'⟩
32    else return ⟨{S'}, {}, S', {}⟩
```

```
NORMALIZE(G = ⟨V, Σ, S, R⟩)
 1  R' ← ∅
 2  WkLst ← R
 3  for each X → [γ] in WkLst
 4  do WkLst ← WkLst \ {X → [γ]}
 5     if length[γ] > 2
 6       then X' ← FRESHVAR()
 7            V ← V ∪ {X'}
 8            R' ← R' ∪
 9              {X → head[γ]X'}
10            WkLst ← WkLst ∪
11              {X' → tail[γ]}
12       else R' ← R' ∪ {X → [γ]}
13  return ⟨V, Σ, S, R'⟩
```

```
TAINTIF(X_1, X_2)
1  if HASLABEL(X_1, direct)
2    then ADDLABEL(X_2, direct)
3  if HASLABEL(X_1, indirect)
4    then ADDLABEL(X_2, indirect)
```

Figure 3.6: Taint propagation in CFG-FSA intersection.

match, as on line 14 in Figure 3.1, the string analysis adds an intersection with the condition's regular expression to the beginning of the `then` branch and an intersection with the complement of the regular expression to the `else` branch.

An adaptation of the standard context free language-reachability algorithm [63] computes the intersection of a CFG and an FSA as a CFG without constructing an intermediate push-down automaton, and we add to the algorithm to propagate annotations. Figure 3.6 shows the algorithm with our additions: the function TAINTIF() and the two calls to it

$$
\begin{array}{lcl}
\texttt{explode}(s_1,\, s_2) & = & \texttt{expld}(s_1,\, s_2,\, \texttt{[ ]},\, \epsilon) \\
\texttt{expld}(s_1,\, s_1,\, L,\, s) & = & L@\texttt{[}s\texttt{]} \\
\texttt{expld}(s_1,\, s_2,\, L,\, s),\ |s_2| \leq |s_1| & = & L@\texttt{[}s\texttt{\^{}}s_2\texttt{]} \\
\texttt{expld}(s_1,\, s_1\texttt{\^{}}s_2,\, L,\, s) & = & \texttt{expld}(s_1,\, s_2,\, L@\texttt{[}s\texttt{]},\, \epsilon) \\
\texttt{expld}(s_1,\, c\texttt{\^{}}s_2,\, L,\, s),\ |c| = 1 & = & \texttt{expld}(s_1,\, s_2,\, L,\, s\texttt{\^{}}c)
\end{array}
$$

Figure 3.7: Semantics of `explode`.

on lines 20 and 28 of INTERSECT(). The following theorem states that this algorithm propagates annotations appropriately.

Theorem 3.1. Given $C' = \textsc{Intersect}(C, F)$, s $\in \mathcal{L}(C) \cap \mathcal{L}(F)$, and a parse tree p of s under C, there exist s_1, s_2, and s_3 such that $s_1 s_2 s_3 = s$ and s_2 is derivable from a direct-labeled nonterminal in p iff there exists a parse tree p' of s under C' such that s_2 is derivable from a direct-labeled nonterminal in p'.

The proof is by a straightforward induction on the height of the derivation of s. The theorem and proof for "indirect" are identical to the theorem and proof for "direct" except that "indirect" replaces "direct."

The algorithm for finding the image of a CFG over an FST is similar to the CFG-FSA intersection algorithm, except that the FST's output symbols replace the CFG's terminals as they match the FST's input symbols. The modifications for propogating taint information are the same for that algorithm as in Figure 3.6, and the proof of correctness is analogous to the proof of Theorem 3.1.

Definition 2.1 only allows string concatenation after input filters. The taint-propagating algorithm in Figure 3.6 correctly propagates tainted substring boundaries even for filters that operate on inputs concatenated with other strings. Thus we extend the definition of SQLCIVs to web applications with operations beyond those that Definition 2.1 allows and still check for SQLCIVs with high precision.

Handling Other String Operations: Real-world web applications also perform operations involving strings that do not simply map strings to strings. For each such operation,

we must determine how substrings in the input map to substrings in the output and propagate annotations accordingly. We use as a straightforward but representative example the `explode` function, which takes two string arguments: a delimiter and a subject. It returns an array of substrings formed by splitting the subject on boundaries formed by the delimiter. Figure 3.7 shows the semantics of `explode`. Because the strings that it returns are taken directly from the subject, the meaning of untrusted substring flow is clear.

The string analysis models the effects of `explode` accurately, except that it loses the order of the strings in the returned array—it produces a grammar whose language is that set of strings. The algorithm (due to Minamide [64]) uses two FSTs constructed from the delimiter, and because we propagate labels through FSTs correctly (see Section 3.3.1), we track tainted substrings accurately through the `explode` function.

3.3.2 Policy-Conformance Analysis

The second phase of our analysis checks the generated, annotated grammar for SQL injection attacks. In most cases, programmers intend that inputs take the syntactic position of literals. Section 3.3.2 describes our checks for this case, and Section 3.3.2 presents our approach for the case when the input may be derived from an arbitrary nonterminal in the reference (SQL) grammar.

Untrusted Substrings as Literals: This section describes how we attempt for each annotated nonterminal X either to verify that all strings derivable from X are syntactically confined or to find that some string derivable from X are not syntactically confined. We apply the algorithm described in Section 3.3.2 to nonterminals for which the checks in this section fail to provide conclusive results.

The first check attempts to find untrusted substrings that cannot be syntactically confined in any SQL query. In particular, because quotes delimit string literals in SQL, if an untrusted substring has an odd number of un-escaped quotes preceding it (escaped quotes represent characters rather than delimiters), it cannot be syntactically confined. The gram-

mar generated by the string-taint analysis reflects the program's dataflow, so the strings derivable from labeled nonterminals are the possible untrusted substrings in generated SQL queries. Let V_l be the set of labeled nonterminals in V. For each $X \in V_l$, if

$$\emptyset \neq \mathcal{L}(V, \Sigma, X, R) \cap \mathcal{L}(\texttt{/\^{}(([\^{}']|\textbackslash')*[\^{}\textbackslash])?'(([\^{}']|\textbackslash')*[\^{}\textbackslash])?'} \\ \texttt{(([\^{}']|\textbackslash')*[\^{}\textbackslash])?')*([\^{}']|\textbackslash')*\$/})$$

then there exists a string derivable from X that is not syntactically confined (the Perl regular expression matches strings with an odd number of unescaped quotes), and we remove X from V_l.

The second check finds the nonterminals in V_l that occur only in the syntactic position of string literals and, for each one, either verifies it as safe or finds that it derives some unconfined string. The algorithm identifies the syntactic position of labeled nonterminals by creating from the grammar production set R a new production set R_t: for each labeled nonterminal $X \in V$, replace right-hand-side occurrences of X in R with a fresh terminal $t_X \notin \Sigma$ and add t_X to Σ. For each labeled $X \in V$, if for all strings $\sigma_1 t_X \sigma_2 \in \mathcal{L}(V, \Sigma, S, R_t)$, σ_1 has an odd number of unescaped quotes, then X only occurs in the syntactic position of a string literal. The following implements this check:

$$\emptyset = \mathcal{L}(V, \Sigma, S, R') \cap \mathcal{L}(\texttt{/\^{}[\^{}']*('(([\^{}']|\textbackslash')*} \\ \texttt{(([\^{}\textbackslash][\textbackslash\textbackslash]+)|[\^{}'\textbackslash]))?'[\^{}']*)*}t_X\texttt{.*\$/})$$

For each $X \in V_l$ for which the test above succeeds, if any $\sigma \in \mathcal{L}(V, \Sigma, X, R)$ has unescaped quotes in it, X derives unconfined strings; otherwise X is safe. We then remove X from V_l.

The third check attempts to identify those remaining nonterminals in V_l that only derive numeric literals. For each $X \in V_l$, if

$$\emptyset = \mathcal{L}(V, \Sigma, X, R) \cap \mathcal{L}(\texttt{/\^{}(([\^{}0-9.+-].*[\^{}0-9.]/)|([.].*[.]))})$$

then X derives only numeric literals and is safe; remove X from V_l.

Finally, if X can produce a non-numeric string outside of quotes, it likely represents an SQLCIV. To confirm this, we check whether X can derive any of a given set of strings that cannot be syntactically confined (*e.g.*"`DROP WHERE`," "`--`," etc.). If it can, then X is unsafe, and we remove it from V_l.

Untrusted Substrings Confined Arbitrarily: If any nonterminals remain in V_l after the checks in Section 3.3.2, we wish to check whether each string derivable from them is derivable from some nonterminal in the SQL grammar. In general, context free language inclusion is undecidable, but we can approximate it by checking grammar derivability, *i.e.*, whether the generated grammar is derivable from the SQL grammar [89].

Definition 3.2 (Derivability)**.** Grammar $G_1 = (V_1, \Sigma, S_1, R_1)$ is *derivable* from grammar $G_2 = (V_2, \Sigma, S_2, R_2)$ iff

$$\begin{aligned}
&\exists\, \Phi : (V_1 \cup \Sigma) \rightarrow (V_2 \cup \Sigma) \\
&\quad \Phi(S_1) = S_2 \wedge \\
&\quad \forall\, s \in \Sigma \;\; \Phi(s) = s \wedge \\
&\qquad \forall\, (X \rightarrow \gamma) \in R_1 \;\; \Phi(X) \Rightarrow^*_{G_2} \Phi^*(\gamma)
\end{aligned}$$

where Φ^* is Φ lifted to $(V_1 \cup \Sigma)^*$, *i.e.*,

$$\begin{aligned}
\Phi^*(\epsilon) &= \epsilon \\
\Phi^*(\alpha) &= \Phi(\alpha) \quad \text{for } \alpha \in V_1 \cup \Sigma \\
\Phi^*(\alpha\beta) &= \Phi^*(\alpha)\Phi^*(\beta)
\end{aligned}$$

Lemma 3.3. *If G_1 is derivable from G_2, then $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$.*

We check derivability using an extension of Earley's parsing algorithm [21] that parses sentential forms and treats nonterminals in G_1 as variables that range over terminals and nonterminals. This algorithm is inspired by and is similar to Thiemann's algorithm [89]. We do not require that the entire generated grammar be derivable from the SQL grammar;

we require derivability for the subgrammar rooted at X and all sentential forms that include X. If the derivability check fails, we consider X to be unsafe.

3.3.3 Soundness

We state and sketch the proof of a soundness result here.

Theorem 3.4 (Soundness)**.** If our analysis algorithm does not report any SQLCIVs for a given web application P, then P has no SQLCIVs.

Proof. The string analysis produces a CFG G from web application P that derives all strings that P may generate as query strings [64]. The algorithm for constructing G reflects P's dataflow so that for assignments and concatenation, labels on nonterminals from untrusted sources accurately identify untrusted substrings. By Theorem 3.1, the CFGs constructed as the intersection of a CFG and an FSA, or the image of a CFG over an FST, is labeled to reflect the boundaries of untrusted substrings. The conformance checking algorithm from Section 3.3.2 generates an error message on each labeled nonterminal unless the algorithm can verify it to derive only syntactically confined strings, as required by Definition 2.3. □

3.4 Implementation

We implemented our technique for PHP, using and modifying Minamide's string analyzer. In addition to the changes described previously (adding information flow tracking and checks on the generated grammars), we made the analyzer more automated in two ways. First, we added specifications for 243 PHP functions. Second, we enhanced its support for dynamic includes. Previously, the analyzer would fail if it reached an include statement, and the grammar it had generated for the include statement's argument had an infinite language. For example, if the analyzer recorded the possible values for `$choice` as being Σ^*, the analyzer would fail at:

```
include("e107_languages/lan_".$choice.".php");
```

Name (version)	Files	Lines	Grammar Size		Time (h:m:s) String Analysis	Time (h:m:s) SQLCIV Check
			$\|V\|$	$\|R\|$		
Claroline	1144	169,232	74,425	545,243	3:04:11.42	2:22.23
e107	741	132,850	62,350	377,348	3:39:26.23	35:36.12
EVE Activity Tracker	8	905	57	1628	0.40	0.06
Tiger PHP News System	16	7,961	82,082	1,078,768	3:14:06.95	5.39
Utopia News Pro	25	5,611	5,222	336,362	25:00.08	2:08.69
Warp Content MS	42	23,003	1,025	73,543	21.10	0.08

Table 3.1: Resource usage from evaluation.

We address this by considering the file and directory layout to be part of the specification. If the analyzer encounters such an include statement, it builds a regular expression representation of the directory layout starting from the analyzed project's root. It then intersects the (finite) language of this regular expression with the language of the grammar to find the list of files to include. This language-based approach does not model the full semantics of paths (*e.g.*, "`..`" as parent directory), but we believe this choice to be appropriate for two reasons. First, we have not encountered cases where the programmer-intended values of variables like `$choice` include "`..`"; and second, security exploits on dynamic inclusion vulnerabilities generally reveal sensitive information stored in files and do not facilitate SQL command injection attacks.

The string analyzer does not support all features for PHP. For example, it includes only limited support for references. We plan to add support for these features, but until full-support is available, we manually approximate unsupported lines of PHP code and verify that the changes do not remove potential errors.

3.5 Evaluation

This section presents the setup and results of our evaluation.

Name (version)	Errors		
	Direct		Indirect
	Real	False	
Claroline (1.5.3)	30	11	24
e107 (0.7.5)	1	0	4
EVE Activity Tracker (1.0)	4	0	1
Tiger PHP News System (1.0 beta 39)	0	3	2
Utopia News Pro (1.3.0)	14	2	12
Warp Content MS (1.2.1)	0	0	0
Totals	49	16	41

Table 3.2: Evaluation results.

```
isset($_GET['newsid']) ?
        $getnewsid = $_GET['newsid'] : $getnewsid = false;
if(($getnewsid != false) && (!preg_match('^[\d]+$', $getnewsid)))
{   unp_msg('You entered an invalid news ID.');
    exit;
}
...
if(!$showall && $getnewsid)
{ $getnews = $db->query("select * from `unp_news`"
    ."where `newsid` = '$getnewsid' order by `date` desc limit 1");
}
```

Figure 3.8: Source of a false positive.

3.5.1 Test Subjects

We evaluated our tool on five real-world PHP web applications in order to test its scalability and its false positive rate, and to see what kinds of errors it would find and what would cause false positives. We use the following subjects in our evaluation: Claroline is a collaborative learning management system; e107 and Warp Content Management System are content management systems; EVE Activity Tracker is an activity tracker for integration into existing IGB homepages; and Tiger PHP News System and Utopia News Pro are news management systems. Table 3.1 lists the size of each of these web applications in terms of the number of files and the number of lines of PHP code. The test suite for another PHP analysis tool [99] includes an earlier version of e107, and the only database-backed PHP

```
$newsposter = $USER['username'];
$newsposterid = $USER['userid'];
// Verification
if (unp_isEmpty($subject) || unp_isEmpty($news))
{
    unp_msg($gp_allfields);
    exit;
}
if (!preg_match('^[\d]+$', $newsposterid))
{
    unp_msg($gp_invalidrequest);
    exit;
}
$submitnews = $DB->query("INSERT INTO `unp_news`"
    ."(`date`, `subject`, `news`, `posterid`, `poster`) VALUES "
    ."('$posttime', '$subject', '$news',"
        ."'$newsposterid', '$newsposter')");
```

Figure 3.9: Source of an indirect error report.

web application we know of that has more lines of code is Claroline. We ran the analysis on a machine with a 3GHz processor and 8GB of RAM running Linux – Fedora Core 5.

3.5.2 Accuracy and Bug Reports

The code in Figure 3.1 shows a vulnerability that our tool found by modeling regular expressions precisely. Two others in Utopia News Pro are similar to this one. Although some of the SQLCIVs that our tool found were trivial, others crossed file and class boundaries. For example, the SQLCIV in e107 comes from a field read from a cookie, which a user can modify, that is used in a query in a different file.

As Table 3.2 shows, our tool had a $(16/(49+16)) = 24.6\%$ false positive rate across this test suite. This false positive rate demonstrates that our approach is effective for finding SQLCIVs and verifying the absence of them.

Our tool produced the false positives that it did because of it does not track information with sufficient precision through type conversions. Figure 3.8 shows one of the two false positives from Utopia News Pro (the other is similar). The PHP runtime system will dynamically cast between any of the scalar types without complaint. It casts a value of

type string to a value of type boolean producing a value of `false` if the string is "" (empty) or "0," and `true` otherwise. To avoid the false positive shown in Figure 3.8, the analyzer would have to model this conversion in the first conditional expression and propagate its implications beyond the "then" branch. The three false positives from Tiger PHP News System resulted from a hand-written string sanitizing routine. Depending on a character's ASCII value, this routine will either encode it or keep it as is. The string analyzer does not have a map from characters to their ASCII values, so it failed to track the precise effects of this routine. Both of these types of false positives could be avoided by equipping the string analyzer with more information about type conversions.

Evaluating whether indirect error reports represent real errors is difficult because it requires making assumptions about what data can flow into the source (*e.g.*, the database). However, Figure 3.9 shows one example of an indirect error report that seems to represent a true vulnerability. Both `$newsposter` and `$newsposterid` are assigned from the `$USER` array, which is populated elsewhere from the results of a database query. The fact that `$newsposterid` is checked and not `$newsposter` seems to indicate the possibility of unexpected values, and at the least it represents inconsistent programming.

3.5.3 Scalability and Performance

As stated in Section 3.4, our string analyzer currently has some limitations in terms of the PHP constructs it supports. Nevertheless, on three of the subjects in our test suite (EVE Activity Tracker, Utopia News Pro, and Warp Content Management System) the analyzer ran successfully. The others include certain currently unsupported constructs, and we manually modified the code to allow the analyzer to continue but without causing any potential errors to be missed. The unsupported construct that we encountered most frequently was the `str_replace` function with array-type arguments, which were generally given statically. We expanded these `str_replace` statements into sequences of `str_replace` statements, each with scalar arguments. These unsupported constructs do not represent a shortcoming in our technique, but only a current limitation in our prototype.

Regarding scalability, we note first that our tool successfully analyzed all of the web applications in our test suite. Table 3.1 lists the size of the grammars representing SQL queries that our tool generates in terms of the number of nonterminals ($|V|$) and the number of production rules ($|R|$). Next to the grammar size, it lists the time spent on string analysis and the time spent checking the generated grammars for SQLCIVs. Analyzing web applications is different in one key respect. Unlike for many program analysis settings where a code base has a single top-level function that can be passed to an analyzer, each file that represents one page in a web application defines a top level function. In many web applications, most files defining top-level functions include and use the same helper functions in other files, and our tool re-analyzes these included files each time. In such cases, our tool analyzes most of the code in a small fraction of the time required to analyze the whole web application. This also illustrates that memoization or concurrent executions of the analyzer could improve the performance dramatically in some cases.

A few points are particularly noteworthy here. First, the grammar size is not necessarily proportional to the web application size. The query grammar generated from Tiger PHP News System is significantly larger than that of e107, which is over an order of magnitude larger in terms of lines of code. This reflects in some sense the size of the web application devoted to database queries.

Second, the string analysis time is not necessarily proportional to the grammar size. The grammar size reported is only for the grammar representing possible database queries. The string analysis works "eagerly," analyzing some string expressions that have no influence on the generated database queries. This eager analysis introduces significant unnecessary overhead in web applications that process user input for marked up display, such as in an online bulletin board or forum. Tiger PHP News System includes such code that substitutes html tags for forum equivalents (*e.g.*, `<bold>` for `[bold]`) and designated character sequences for "emoticon" links. Tiger PHP News System is designed to be secure, and it includes a forum with such code. Each regular expression or string replacement function (potentially) causes its argument's grammar to increase by some factor, so that a sequence of these replacement

expressions leads to exponential growth in the number of replacements. We removed two sections of such code from Tiger PHP News System in order to speed up the analysis, but in principle the analyzer could use a backward dataflow analysis to determine which variables may influence a database query, and refrain from analyzing the rest. We expect that this would speed up the analysis significantly. Additionally, dynamic file inclusions can lead to combinatorial growth in analysis time. Each time a file is included, it is inserted *in situ* and its top-level scope is merged into the scope where it is included. If one file has an include statement whose argument is entirely unspecified statically, the analyzer will try to include every other file in the project and with each of them, whichever files they may include. In the case of e107 with 741 files, we had to provide file names for two include statements.

Finally, the SQLCIV checking phase is relatively efficient. Although the grammars had more than one million production rules in some cases, SQLCIV checking never took more than a few minutes, and usually took less.

3.6 Related Work

In this section we survey closely related work by looking at the development of two strains of static analysis, string analysis in Section 3.6.1 and taint checking in Section 3.6.2.

3.6.1 Static String Analysis

The study of static string analysis grew out of the study of text processing programs. An early work to use formal languages (*viz.* regular languages) to represent string values is XDuce [38], a language designed for XML transformations. Tabuchi *et al.* designed regular expression types for strings in a functional language with a type system that could handle certain programming constructs with greater precision than had been done before [87].

Christensen *et al.* introduced the study of static string analysis for imperative (and real-world) languages by showing the usefulness of string analysis for analyzing reflective code in Java programs and checking for errors in dynamically generated SQL queries [12]. They

designed an analysis for Java that has FSAs as its target language representation; they chose FSAs because FSAs are closed under the standard language operations. They also applied techniques from computational linguistics to generate good FSA approximations of CFGs [65]. Their analysis, however, does not track the sourced of data, and because it must determinize the FSAs between each operation, it is less efficient than other string analyses and not practical for finding SQLCIVs. Gould *et al.* used this analysis to type check dynamically generated queries, but made approximations that would cause them to miss SQLCIVs [28].

Minamide borrowed techniques from Christensen *et al.* to design a string analysis for PHP that does not approximate CFGs to FSAs, so it can be more efficient and more accurate [64]. He also utilized techniques from computational linguistics (*viz.* language transducers) [66] to improve the precision of his analysis and model the effects of string operations, which are used frequently in scripting languages. His analysis does not track the source of data explicitly, and it is designed to validate dynamically generated HTML, which has a flatter grammar than SQL. For both Minamide and Christensen *et al.*'s analyses, the user must provide regular expression specifications of the permitted queries at each query location. We avoid the need for manually written specifications first by using a general policy based on both dataflow and string structure, and second by adding explicit dataflow information to the grammar's nonterminals in Minamide's analysis.

3.6.2 Static Taint Checking

Static taint checking is essentially information flow analysis specialized to determine whether data from an untrusted source flows into a sensitive sink. Static taint checking has a long history, but Huang *et al.* were perhaps the first to apply it to SQLCIVs [40]. They used a CQual-like [24] type system to propagate taint information through PHP programs. Livshits and Lam [59] used a precise points-to analysis for Java [98] and queries specified in PQL [55] to find paths in Java programs that allow "raw" input to flow into SQL queries. Both of these tools are sound with respect to the policy they enforce and the language

features they support, and both find many vulnerabilities, but both consider all values returned from designated filtering functions to be safe. Because the policy they use says nothing about the context of the user input and the structure of the query, both techniques may miss real SQLCIVs. Additionally, Huang *et al.*'s type system does not support some of PHP's more dynamic features, in part because it does not track string values at all and supporting these features would result in excessively many false positives.

Jovanovic *et al.* sought to address this last shortcoming with Pixy [43, 44], a static taint analysis for PHP that propagates limited string information and implements a finely tuned alias analysis. Xie and Aiken designed a more precise and scalable analysis for finding SQLCIVs in PHP by using block- and function-summaries [99]. The precision they gained comes at the expense of automation — the user must provide the filenames when the analysis encounters a dynamic include statement, and the user must tell the analysis whether each regular expression encountered in a filtering function is "safe." We are able to make a stronger guarantee about the absence of SQLCIVs because we analyze the possible values of the strings and check conformance to a policy that takes into account the query's structure.

Subsequent to the publication of our work in this chapter (as [94]), Pixy was modified and extended to become Saner [3]. Saner, like the work we present in this chapter, aims to find SQL injection vulnerabilities while taking into account the semantics of input validation routines. Saner uses FSTs to model string functions, although some of the FSTs Saner uses are less precise than their counterparts in this work, and Saner represents sets of strings as regular languages. Saner checks against the simpler policy that tainted strings must be literals and so differs slighly from our work both in precision and expressivity. However, Saner does have a dynamic component whereby it attempts to provide witnesses of vulnerabilities in order to counter the problem of false positives that static analyses generally have.

Chapter 4

Static Analysis for Detecting XSS Vulnerabilities

As we mentioned in Section 1.2, XSS is related to SQL injection in that both result from input validation errors in web applications, but of the two, XSS is more prevalent. XSS attacks can leak confidential information, such as session identifiers for secure sessions, or serve as launching pads for other, more severe attacks on web users' local systems. This chapter describes the nature of XSS vulnerabilities and builds on the static analysis presented in Chapter 3 to formulate and experiment with the first practical, static, server-side analysis for detecting XSS vulnerabilities that takes into account the semantics of input validation routines.

4.1 Causes of XSS Vulnerabilities

Section 1.2 provides an initial description of XSS as an integrity vulnerability in web applications. More specifically, we consider XSS to be the class of vulnerabilities in which an attacker causes the web application server to produce HTML documents in which some substring that did not come from the application's source code causes the web browser's JavaScript interpreter to be invoked. Section 4.3.2 provides more discussion on this policy.

Several factors contribute to the prevalence of XSS vulnerabilities. First, the system requirements for XSS are minimal: XSS afflicts web applications that display untrusted input, and most do. Second, most web application programming languages provide an unsafe default for passing untrusted input to the client. Typically, printing the untrusted input directly to the output page is the most straightforward way of displaying such data. Static taint analysis addresses this second factor. It marks data values assigned from untrusted sources as tainted and reports a vulnerability if the application may display that data without first sanitizing it. The analysis considers untrusted data sanitized if that data has passed through one of a designated set of sanitizing functions. This chapter addresses a third factor: proper validation for untrusted input is difficult to get right, primarily because of the many, often browser-specific, ways of invoking the JavaScript interpreter.

The MySpace worm, which infected more systems quickly than previous Internet worms (see Figure 1.3a), exploited a weakness in the MySpace input filters. The MySpace input filters prohibited all occurrences of the strings "`<script>`" and "`javascript`;" however, Internet Explorer concatenates strings broken by newlines and allows JavaScript within cascading style sheet tags, so the worm introduced its script by means of a string like:

```
<div style="background:url('java
script:...')">.
```

As if to further illustrate the point about proper input filtering being difficult to get right, several online services sell website visitor-tracking code for sites that officially forbid JavaScript, such as Xanga and MySpace. The sites that sell this code provide their service by continually finding new XSS vulnerabilities, so that whenever their current exploit stops working, they can use another vulnerability to inject their code.

Browser-specific ways of invoking the JavaScript interpreter exist because browsers handle permissively pages that are not standards compliant. In the early days of the web, this design decision on the part of the browser implementers seemed good because it enabled browsers to make a "best effort" to display poorly written pages. However, as the JavaScript

language became more standardized and its use increased, this decision has exacerbated the XSS problem.

4.2 Overview

This section provides an overview of our approach and introduces a running example to be used for a more detailed presentation of our approach in the next section.

4.2.1 Current Practice

Currently, XSS scanners rely on either testing or static taint analysis. Automated testing is ill-suited to finding errors in input validation code, because even flawed validation code catches most malicious uses, and, in order for an exploit to work, it must be crafted specifically for a certain validation's weakness. Taint analysis takes as input a list of functions designated as sanitizers, but it does not perform any analysis on them, so it will not catch errors caused by weak input validation.

4.2.2 Our Approach

This chapter proposes an approach to finding not only XSS vulnerabilities due to unchecked untrusted data but also XSS vulnerabilities due to insufficiently-checked untrusted data. The approach has two parts: (1) an adapted string analysis to track untrusted substring values, and (2) a check for untrusted scripts based on formal language techniques.

The string taint analysis phase is largely the same as in Section 3.3.1, but this analysis produces a representation of the language of HTML documents rather than queries that a web application produces.

The second phase of our approach enforces the policy that generated web pages include no untrusted scripts. In order to generate the right low-level description of this high-level policy, we must consider how web browsers' layout engines parse web documents, and under which circumstances they invoke the JavaScript engine. In order to generate the policy

lib.inc.php

```
481 function stop_xss($data) {
524  /* Get attribute="javascript:foo()" tags. Catch spaces in the regex
525   * /(=|url\()("?)[^>]*script:/ */
526   $preg = '/(=|(U\s*R\s*L\s*\())\s*("|\')?[^>]*\s*' .
527           'S\s*C\s*R\s*I\s*P\s*T\s*:/i';
528   $data = preg_replace(  $preg, 'HordeCleaned', $data);
543  /* Get all on<foo>="bar()". NEVER allow these. */
544   $data = preg_replace('/([\s"\']+on\w+)\s*=/i',
545                        'HordeCleaned=', $data);
550  /* Remove all scripts. */
551   $data = preg_replace('|<script[^>]*>.*?</script>|is',
552                        '<HordeCleaned_script />', $data);
553   return $data;
554 }
```

projects_stats_pop.php

```
21 $use     = (int) $_REQUEST['use'];
22 $module  = stop_xss($_REQUEST['module']);
62 if ($use) {
63   echo '<br>';
71 } else {
72   $hiddenfields = "<input type='hidden' "
73     . "name='module' value='$module'/>\n";
74   echo '
75     <form action="stats.php" method="get">
76       <div style="width:60%;float:right">
77         <input name="speichern" />
78         '.get_buttons().'
79       </div>
80       '.$hiddenfields.'
81     </form>';
82 }
```

Figure 4.1: Vulnerable PHP code.

description, we studied the Gecko layout engine [27], which Firefox [23] and Mozilla [67] use, looked at the W3C recommendation for scripts in HTML documents [39], and looked at online documentation for how other browsers handle HTML documents [78]. We represent the policy using regular languages and check whether untrusted parts of the document can invoke the JavaScript interpreter using language inclusion.

4.2.3 Running Example

Section 4.3 presents our analysis algorithm using the PHP code in Figure 4.1 as input. This section explains the example code and describes its vulnerability.

The program fragment in Figure 4.1 is pared down and adapted for display from PHProjekt 5.2.0, a modular application for coordinating group activities and sharing information and documents via the web [72]. It is widely used and can be configured for 38 languages and 9 DBMSs. The untrusted data comes from the `$_REQUEST` array on lines 21 and 22. Line 22 assigns untrusted data to the `$module` variable after passing it through the `stop_xss` function.

The `stop_xss` function, which PHProjekt uses to perform input validation, uses Perl-style regular expressions to remove dangerous substrings from the input. The complete function also checks for alternate character encoding schemes, likely phishing attacks, and other dangerous tags, but these checks are omitted here for brevity. The primary ways to invoke the JavaScript interpreter are through script URIs; event handlers, all of which begin with "`on`"; and "`<script>`" tags. The function `stop_xss` removes these three cases with the regular expression replacements on lines 528, 544, and 551, respectively.

The W3C recommendation for HTML attributes specifies that white space characters may separate attribute names from the following '`=`' character. The regular expression on line 545 reflects this specification: '`\w`' represents word characters (word characters include alphanumeric characters, '`_`', and '.'), and '`\s`' represents white space characters. However, the Gecko layout engine's HTML parser permits arbitrary non-word characters between the attribute name and the '`=`' character. Consequently, if an input string includes a substring such as "`' onload#=`" followed by arbitrary script, it will pass the filter on line 544 but will cause untrusted JavaScript to be executed when the output page is viewed in Firefox.

4.3 Analysis Algorithm

This section focusses on line 545 of the program in Figure 4.1 as it presents the algorithm.

```
$data1 = $_REQUEST['module'];
$data2 = preg_replace('/([\s"\']+on\w+)\s*=/i','HordeCleaned=',$data1);
$module = $data2;

if ($use) {
  $output1 = '<br>';
} else {
  $hiddenfields = "<input value='$module'/>";
  $output2 = ' </div> '.$hiddenfields;
}
$output3 = φ($output1, $output2);
```

Figure 4.2: Code in SSA form.

$REQUESTmodule^T$	$\rightarrow$	Σ^*
data1	$\rightarrow$	*REQUESTmodule*
data2	$\rightarrow$	preg_replace(`/([\s"\']+on\w+)\s*=/i`, `HordeCleaned=`, *data1*);
module	$\rightarrow$	*data2*
output1	$\rightarrow$	` `
hiddenfields	$\rightarrow$	`<input value='`*module*`'/>`
output2	$\rightarrow$	`</div>` *hiddenfields*
output3	$\rightarrow$	*output1* | *output2*

Figure 4.3: Productions for extended CFG.

4.3.1 String-taint Analysis

The string-taint phase of our analysis comes from Section 3.3.1; we review the main steps here to illustrate it on an example relevant to XSS and set up the running example for the next section. The first phase of the string-taint analysis translates output statements (*e.g.*, `echo` statements) into assignments to an added output variable, and translates the program into static single assignment (SSA) form [17] in order to encode data dependencies. Figure 4.2 illustrates this translation on the example code, omitting and simplifying several parts of the program for the sake of presentation.

Because the SSA form encodes data dependencies, the next phase of the string-taint analysis drops control structures, translates assignment statements into grammar productions, and labels untrusted data sources. This phase constructs an extended context free

$REQUESTmodule^T$	$\rightarrow$	$\diamondsuit$
data1	$\rightarrow$	*REQUESTmodule*
data2	$\rightarrow$	*data1*;
module	$\rightarrow$	*data2*
output1	$\rightarrow$	` `
hiddenfields	$\rightarrow$	`<input value='`*module*`'/>`
output2	$\rightarrow$	`</div>` *hiddenfields*
output3	$\rightarrow$	*output1* | *output2*

Figure 4.4: CFG with untrusted substrings summarized; see Section 4.3.2 for an explanation of '$\diamondsuit$.'

grammar (CFG); it is extended in the sense the grammar productions' right hand sides may contain string functions. Figure 4.3 shows the extended CFG for our example, with *output3* as the start symbol. In Figure 4.3, *REQUESTmodule* is labeled with a taint annotation indicating that substrings derived from that non-terminal are untrusted. The only string function in our example grammar is `preg_replace`, which takes three arguments: a pattern, a replacement, and a subject. It searches for instances of the pattern in the subject and replaces them with the replacement. In general the replacement may reference and include parts of the string that the regular expression pattern matched, but in this example it does not.

We transform the extended CFG into a standard CFG using FSTs as described previously. In the extended CFG in Figure 4.3, the subject argument to `preg_replace` has Σ^* as its language, so the output language of the `preg_replace` function is:

$$\Sigma^* \setminus \mathcal{L}(\texttt{(\textbackslash s|"|')}^+\texttt{(o|O)(n|N)\textbackslash w}^+\texttt{\textbackslash s}^*\texttt{=}).$$

We use regular expression notation to simplify the presentation of this example. The '$\diamondsuit$' character in Figure 4.4 replaces the regular expression shown above in the grammar. The next section discusses this substitution in more detail.

4.3.2 Preventing Untrusted Script

As Section 4.1 states, we seek to enforce the policy that no untrusted input may invoke the browser's JavaScript interpreter. This is the standard policy that server-side security mechanisms attempt to enforce, and the policy that untrusted input should not include any HTML mark-up subsumes it. JavaScript's highly dynamic nature as a prototype language inhibits checking from the server side whether untrusted JavaScript will stay within some safe boundaries. How to formalize "safe" JavaScript is an open question.

Challenges: Enforcing this policy to prevent XSS is challenging, and in particular, it is more difficult than preventing SQL injection. SQL injection, the second most reported vulnerability of 2006, is another input validation-based, web application vulnerability, and usually programmers attempt to enforce the policy that untrusted input can only be literals in generated SQL queries, although our syntactic policy (Definition 2.3) has several advantages. The biggest challenge in preventing XSS is that web browsers support many ways of invoking the JavaScript interpreter, some of those according to the W3C recommendation, and some browser-specific. Web application programmers must account for even the browser-specific ways because they cannot control which browsers clients will use to view their pages. Additionally, if a web application programmer wants to allow some HTML mark-up, then every character has some legitimate use, so no single character can be escaped to prevent XSS. In contrast, the lexical definition of SQL literals and delimiters is relatively simple and standard, and web application programmers need not worry about interfacing with arbitrary DBMSs. Because of the many ways of invoking the JavaScript interpreter, statically checking sufficient input validation is more expensive and requires more careful engineering than checking for SQL injection vulnerabilities.

Constructing the Policy: In order to enumerate the ways an HTML document can invoke a browser's JavaScript interpreter, we examined three sources: the W3C recommendation, which serves as the HTML standard; the Firefox source code; and online tutorials

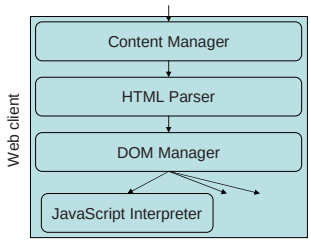


Figure 4.5: Client architecture.

and documents. The source code for Internet Explorer was not available to us. Figure 4.5 shows parts of the browser architecture in the context of the web document workflow that influence how the JavaScript interpreter can be invoked. After the input passes through the content manager, which handles downloads, caching, and protocols, it goes to the HTML parser. From the W3C recommendation, we gathered lexical rules for defining and separating tokens, and from an examination of Firefox's HTML parser, we modified the initially gathered lexical rules to allow non-word characters where previously only whitespace characters were allowed, as described in Section 4.2.3.

The HTML parser sends tokens to the DOM manager, which, among other tasks, constructs the DOM and calls the JavaScript interpreter. The W3C recommendation specifies two ways of including script in HTML: the "`<script>`" tag and event handlers. We found that the Firefox DOM manager also calls the JavaScript interpreter on the URI attribute of "`iframe`," "`meta`," and other tags. However, all sources we examined show that only tokens within a tag context (*i.e.*, between "`<`" and "`>`") can cause the DOM manager to call the JavaScript interpreter.

Because we are interested in whether untrusted input can invoke the JavaScript interpreter and not the string value of untrusted JavaScript code, we construct our policy in terms of the language of untrusted strings permitted or not permitted in a tag context. For example, we describe the language of tags whose names invoke the JavaScript interpreter using regular expressions such as:

```
[Ss][Cc][Rr][Ii][Pp][Tt]([^a-zA-Z0-9_.].*)?
```

The W3C recommendation includes eighteen intrinsic events (*e.g.*, `load`) and 31 events in total. Handlers for intrinsic events can be specified as attributes (*e.g.*, `onload`), but handlers for other events, such as DOM 2 events, can only be registered using "`addEventListener`" in a script. We therefore only check for handlers for intrinsic events. Firefox adds extra events (*e.g.*, `error`) and supports 36 for which handlers can be defined as attributes, all of which begin with "`on`." In order to simplify the regular expressions needed to identify these attributes, we state the policy not in terms of the whole tag, but only from the beginning of the attribute name onwards. To describe the language of attribute names that invoke the JavaScript interpreter (*i.e.*, event handlers and other attributes, such as `src`, that can introduce scripts), we therefore construct regular expressions such as

```
[Oo][Nn][Ll][Oo][Aa][Dd][^a-zA-Z0-9_.]*=.*
```

Note that this description incorporates the lexical rules we gathered in order to describe what may separate the attribute name from '`=`.'

Checking the Example: This section returns to our running example to show how we check the generated grammar against the policy we constructed. As Section 4.3.2 states, only tokens from within tags of an HTML document can invoke the JavaScript interpreter. Our algorithm consists of three main steps: (1) decode encoded characters within the grammar; (2) extract the string values of tags where all or part of the string is untrusted; (3) check those strings for script-inducing substrings. Character encodings are not relevant to our example. In general, however, an FST can decode encoded characters (*e.g.*, translate "`&#97;`" into "`a`").

Extracting Untrusted Tags: For the second step, we must determine in which of the following syntactic contexts each untrusted substring can appear: character data, tag names, attribute names, and attribute values. Depending on which contexts each untrusted string can appear in, we check the string for different script-introducing values. In order to identify untrusted strings' contexts using formal language techniques, we summarize

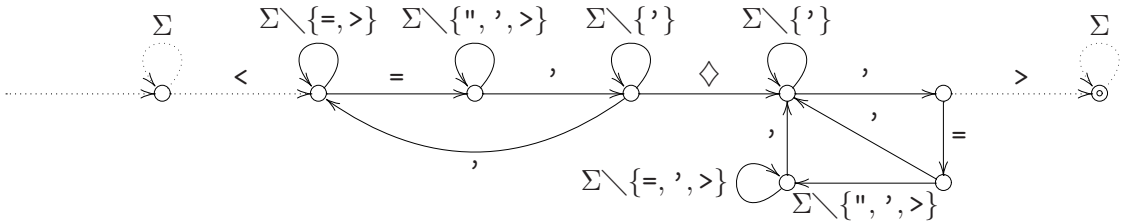


Figure 4.6: FST describing tags with untrusted data summarized by ◇ in an attribute value.

languages of untrusted strings by replacing labeled nonterminals' derivations with "fresh" terminals, *i.e.*, symbols not in Σ. Figure 4.4 shows the example grammar, where '◇' summarizes the untrusted substrings derivable from $REQUESTmodule^T$.

Figure 4.6 shows a nondeterministic FST that we use to check whether the untrusted substrings summarized by '◇' may appear in the context of single-quoted attribute values. In order to avoid having a cluttered figure, we use a different FST notation for Figure 4.6: solid transitions output the same character they read (*e.g.*, if the transition reads '`a`,' then it outputs '`a`'), and dotted transitions output ϵ. The image of the CFG in Figure 4.4 over this FST is the language of strings within a tag in the language of the original CFG that have an untrusted substring in an attribute value. We construct the CFG representation of this image, and use FSTs to remove all trusted script-introducing substrings from the language. Finally we replace '◇' with the original derivations of $REQUESTmodule^T$ so that any script-introducing substrings in the language are untrusted.

Identifying Script-Introducing Substrings: In order to check the CFG for untrusted script-introducing strings, we check whether it has a non-empty intersection with the regular expressions generated earlier that describe our policy. In the case of our running example, the intersection of the CFG with any of the regular expressions for event handlers is non-empty, so we discover a vulnerability.

Subject	Files	Lines Per File			Total
		mean	std dev	max	lines
Claroline	1144	148	248	5,207	169,232
FishCart	218	230	196	1,182	50,047
GecBBLite	11	29	30	95	323
PhPetition	17	159	75	281	2,701
PhPoll	40	144	112	512	5,757
Warp	44	554	520	2,276	24,365
Yapig	50	170	191	946	8,500

Table 4.1: Statistics on subjects' files.

4.4 Empirical Evaluation

4.4.1 Implementation

We implemented our analysis by extending our implementation from Section 3.4, which builds on Minamide's PHP string analyzer. Minamide's PHP string analyzer In particular, we added an option to make the default value for uninitialized variables be "untrusted any string" as opposed to null. We also implemented our algorithm to check for untrusted input that would invoke the JavaScript interpreter. The implementation of this algorithm consists of approximately 1000 lines of OCaml.

Our goal in tracking untrusted information flow is to track direct information flow, not implicit information flows. In particular, we do not seek to perform an analysis that is sound with respect to covert channels. We choose this goal both because we assume that web users and not web programmers may be malicious and because of precedent, *i.e.*, static taint analysis for input validation vulnerabilities also has this goal.

4.4.2 Test Subjects

In evaluating our implementation, we sought first to answer the question of how well it scales on large, real-world web applications. This is an interesting question even after our evaluation in Section 3.5 because input validation code for XSS typically has different

Subject	Additional Files Included			Memory (MB)		
	mean	std dev	max	mean	std dev	max
Claroline	10.0	16.5	148	71	117	1030
FishCart	3.0	2.8	8	41	13	161
Gec	2.3	2.0	5	30	6	41
Phpetition	5.5	4.1	15	54	23	101
Phpoll	1.0	1.2	5	34	19	135
Warp	2.0	1.4	3	64	106	740
Yapig	6.1	9.9	41	124	188	645

Table 4.2: File and memory results for test subjects.

Subject	String Analysis (h:mm:ss)				Policy Check (h:mm:ss)			
	mean	std dev	max	sum	mean	std dev	max	sum
Claroline	0:00:11	0:01:52	0:21:58	3:20:29	0:00:08	0:00:45	0:18:48	2:08:30
FishCart	0:00:01	0:00:01	0:00:10	0:01:14	0:00:01	0:00:03	0:00:17	0:02:18
Gec	0:00:01	0:00:01	0:00:01	0:00:02	0:00:01	0:00:01	0:00:02	0:00:03
Phpetition	0:00:03	0:00:05	0:00:15	0:00:39	0:00:12	0:00:17	0:00:50	0:02:26
Phpoll	0:00:01	0:00:01	0:00:01	0:00:02	0:00:01	0:00:03	0:00:12	0:00:25
Warp	0:04:40	0:21:14	1:55:28	2:19:54	0:01:50	0:08:04	0:44:04	0:55:06
Yapig	0:00:14	0:00:43	0:04:26	0:09:06	0:07:08	0:13:48	0:46:53	4:45:27

Table 4.3: Timing results for test subjects.

characteristics than input validation code for SQL injection. Additionally, this work is predicated on the assumption that manually written input validation code needs to be checked, so we sought to address the question of how well it checks manually written input validation code, and how common manual input validation errors are.

We selected various web applications as test subjects in order to address each of these questions. In order to address the first question, we selected several PHP web applications of varying sizes whose names and sizes are listed in Table 4.1; Table 4.4 includes version numbers as well. One of the web applications, Claroline 1.5.3, is one of the largest open source, PHP web applications we have found (169 Kloc), and this particular version has several known vulnerabilities (CVE-2005-1374).

In order to evaluate the second question, we searched for PHP functions with “xss” in

their name on the assumption that these functions likely represent manually written input validation specifically designed to prevent XSS. The functions we analyzed come from the following projects: VLBook, a light-weight guest book; Sendcard, an e-card system; Drupal, a content management system; LinPHA, a photo archive; Sugar Suite, a customer relationship management system; BASE, an engine to search and process a database of security events; FishCart, an online shopping cart and catalog management system; PHPlist, a simple mailing list system; and PHProjekt, a groupware suite.

4.4.3 Evaluation Results

This section presents the results of our empirical evaluation to address the questions listed above.

Results on Programs: As stated in Section 3.4, we can analyze all PHP pages by giving the analyzer the top-level files. However, we found it easier to run the analyzer on each file (with dynamic includes still being resolved as before) even though this would involve some duplication of work. Running the analyzer on each file skews the average time and memory usage down (because many of the files only define values and are intended for inclusion in other files) and skews the total up. Tables 4.2 and 4.3 show the resource usage per file. In most cases the time for performing the string-taint analysis dominated the total time. The cases that took the longest for the string-taint analysis had string operations with cyclic dependencies. The cases that took the longest for the policy checking had many labeled nonterminals in the output grammar; each labeled nonterminal had to be checked individually. The included files column shows the average number of included files the analyzer parsed and analyzed on a given input file.

Table 4.4 shows the breakdown of bug reports from our experiments. As in Section 3.2.2, vulnerabilities are *direct* if an untrusted user can provide the data directly, whereas vulnerabilities are *indirect* if the data comes from a source such as a file or a database where untrusted data may have entered, but users cannot provide the value directly. “Get-Post-

Subject	Direct Flow				Indirect Flow
	Get-Post-Cookie		Uninitialized		
	Real	False	Real	False	
Claroline 1.5.3	32	43	38	25	42
FishCart 3.1	2	2	30	12	2
GecBBLite 0.1	1	1	0	0	7
PhPetition 0.3.1b	0	0	7	8	7
PhPoll 0.96 beta	5	6	0	0	0
Warp CMS 1.2.1	1	1	22	19	18
Yapig 0.95b	15	13	9	1	14

Table 4.4: Bug reports.

Cookie" (GPC) vulnerabilities come from GET, POST, or COOKIE variables, which the user can set. "Uninitialized" vulnerabilities come from uninitialized variables being used for output. If "export globals" is set, then each key in the associative GET, POST, and COOKIE arrays becomes the name of a variable, and its initial value is the value it maps to in the array. Therefore, if "export globals" is set and an uninitialized variable's values is displayed, a user can provide a GET parameter with that variable's name and the server will include the untrusted data into the generated page. For both GPC and uninitialized vulnerabilities, vulnerability reports are classified as either real or false. We do not attempt to distinguish true and false vulnerabilities from indirect sources because we cannot determine whether or not it is possible for untrusted data to enter a given source.

Figure 4.7 shows one of the previously unreported vulnerabilities that our analysis discovered in Claroline; it is not due to weak input validation, but because the untrusted data passes function and file boundaries and is passed through an array, it would be easy to miss in a manual inspection. To illustrate the benefit of automated analysis, our analysis found 32 true GPC vulnerabilities, whereas CVE-2005-1374 lists only 10, although it does indicate that its list is not exhaustive. Most of the false positives our tool produced come from either spurious paths, or untrusted input being used in a conditional expression and the "taintedness" being propagated into the condition's branches. We could reduce the number of false positives by modifying the tool to output reports from untrusted conditionals as a

Subject	Time (h:mm:ss)		Memory	Vulnerability	
	String Analysis	Policy Check	(MB)	Reported	Present
PHPlist 2.10.2	0:00:01	0:00:01	36	yes	yes
PHProjekt 5.2.0	0:00:36	0:00:39	167	yes	yes
Sendcard 3.2.2	0:00:15	1:01:11	2822	yes	yes
VLBook 1.21	0:00:01	0:00:27	232	yes	yes
Drupal 4.2.0	—	—	—	failed	yes
BASE 1.2.5	0:00:01	0:00:01	33	no	no
FishCart 3.1	0:00:01	0:00:01	39	no	no
SugarSuite 4.2.1	0:00:01	0:00:01	36	no	no
LinPHA 1.3.0	—	—	—	failed	no

Table 4.5: Analysis results for manual input validation functions. "failed" = "failed to analyze."

user_access_details.php

```
43 switch ($_GET['cmd'])
44 {
45   ...
57   case 'doc':
58     $toolTitle['subTitle'] =
59         $langDocument.$_GET['data'];
60   ...
69 }
70 claro_disp_tool_title($toolTitle);
```

claro_main.lib.php

```
435 function claro_disp_tool_title(
436     $titleElement, $helpUrl = false)
437 {
438   ...
474   if ($titleElement['subTitle'])
475   {
476     echo '<br><small>'.
477       $titleElement['subTitle'].'</small>';
478   }
479   echo '</h3>';
480 }
```

Figure 4.7: A vulnerability in Claroline 1.5.3.

different class of warnings.

In addition to the vulnerabilities listed in Table 4.4, Claroline has 77 vulnerabilities

that do not fit naturally into any of the categories in the table. Claroline has a debugging mode that can be turned on and off by the administrator. When it is on, it displays all SQL queries before they are sent to the database, and that is the source of these 77. They are neither clearly true vulnerabilities, since Claroline would not normally run in debugging mode, and when it does, it would be under close control, nor are they false positives, because under specific circumstances XSS is possible with them. Note that these do not necessarily represent SQL injection vulnerabilities because an escaping function that properly sanitizes input for inclusion in SQL queries may not be adequate for preventing XSS.

Our tool failed to analyze some of the web applications we tried it on. It failed to analyze e107 0.75 (132,863 lines) because it failed to resolve certain alias relationships between variables whose values are used for dynamic features, including dynamic file inclusions. This could be addressed by using a more conservative alias analysis. Our tool exceeded its memory limit of 4.5GB when attempting to analyze Phorum 5.1.16a (30,871 lines), because Phorum uses `preg_replace` tens of times consecutively in several cycles and with variables in all arguments. We expect that by redesigning the string analysis to retain only the precision it needs to check our policy, the memory requirement of the analysis would be substantially reduced in such cases.

Manual Validation: We had two goals in checking manually written input validation code: we wanted to see how our tool would perform in terms of time and memory usage and precision (can the tool do the job it is supposed to do?), and we wanted to get a sense of how prevalent insufficient input validation errors are (is the tool's job necessary?). Each of the nine test subjects we selected for this section had one function that performed all, or nearly all, of the application's input validation. We identified these functions, wrote small test files that call them, and sent those files to the analyzer.

Table 4.5 reports on how the tool performed on each of the nine subjects. SendCard stands out as being much more expensive to analyze than the rest. It uses several parameterized regular expression replacements (*i.e.*, the replacement includes a reference to

Subject	Allows HTML	XSS Vuln.	Vulnerability description
PHPlist 2.10.2	yes	yes	Filter only removes "script" tag
PHProjekt 5.2.0			Event handler filter only matches handlers with white space between the handler name and the '='
Sendcard 3.2.2			Event handler filter only matches handlers with no characters between the handler name and the '='
VLBook 1.21			Event handler filter only matches handlers preceded with a space (' ')
Drupal 4.2.0			Event handler filter only matches handlers with white space between the handler name and the '='
BASE 1.2.5	no	no	Filter wraps htmlspecialchars, prevents untrusted HTML mark-up
FishCart 3.1			Filter removes special characters, prevents untrusted HTML mark-up
SugarSuite 4.2.1			Filter encodes special characters, prevents untrusted HTML mark-up
LinPHA 1.3.0			Filter permits only alphabetic characters

Table 4.6: Explanation of (absense of) vulnerabilities for manual input validation functions.

a substring that the parameter matched) that cause the string analysis to generate a large and complex CFG to represent the possible strings SendCare may generate. Some of the other subjects use regular expression replacements, but they are not parameterized. Our tool failed to analyze two of the files because they use PHP features that the tool does not support, and no simple modification to the files would retain their semantics and be analyzable by the tool. Except for the case of SendCard, the analyzer runs relatively efficiently on these subjects and produced precise results, so it appears to be practical.

Table 4.6 describes the weakness in the vulnerable filters and explains the effects of those that are not vulnerable. With the exception of the vulnerability in PHProjekt, these vulnerabilities were previously unknown. We manually inspected these input validation routines to determine whether they allow any HTML mark-up (*e.g.*, the `<b>` tag). Notably, all five of the nine subjects that allow any HTML mark-up from untrusted input have vulnerabilities. The only ones without vulnerabilities prevent all untrusted mark-up. This suggests that writing web applications correctly is a difficult software engineering problem

and that principled checking is necessary in real-world web applications.

4.4.4 Current Limitations

We discuss here some of the current limitations of our analysis.

Three main kinds of XSS exist: stored, reflected, and DOM-based; our analysis currently does not detect DOM-based XSS. Stored XSS occurs when the server stores untrusted data and later displays it; this kind of XSS commonly afflicts forums and online bulletin boards. Reflected XSS occurs when a server echos back untrusted input; this kind of XSS usually shows up in error messages. Unlike stored and reflected XSS, DOM-based XSS reads malicious data from the DOM, and the malicious data need not ever appear on the server. Detecting DOM-based XSS requires an analysis of the generated web page's semantics, not just its syntax. We expect that a reasonably precise approximation could be added on top of our framework, but currently our analysis does not include that check.

Our analysis checks web applications against the policy that no untrusted data should invoke the JavaScript interpreter, and we represent this policy as a black-list rather than a white-list. Omissions in black-list policies usually manifest themselves as difficult-to-detect security vulnerabilities, whereas omissions in white-list policies usually appear as disruptions of functionality, which show up rather quickly. We believe that our policy representation is correct for Gecko-based browsers, but we do not have a formal proof of its correctness with respect to the Gecko source code. Although a white-list policy could prove effective when designed for specific web applications that expect an easy-to-represent language of inputs, one main factor inhibits the use of a white-list policy in the general case. A regular language representation of all input that is valid HTML and does not invoke the JavaScript interpreter would be huge and likely impractical for language inclusion/intersection checks. Additionally, a white-list policy would always report errors in manually written input validation routines that enforce black-list policies, as the manually written code that we have seen does. However, even a weak black-list policy based solely on the W3C recommendation will help to uncover more vulnerabilities than a standard taint

analysis will.

Our string analysis-based tool cannot handle arbitrarily complex and dynamic code. For example, because it does not track information flow across web page visits, it loses precision when the web application performs operations and calls functions based on the values of session variables. The tool also cannot verify input validation routines based on manually written HTML parsers and manipulators. Finally, the tool does not support some of PHP's features, such as array arguments in string replacement functions.

4.5 Related Work

We classify related work into server-side (Section 4.5.1) and client-side (Section 4.5.2) techniques for preventing XSS.

4.5.1 Server-Side Validation

Because XSS and SQL injection are closely related problems, much of the server-side work on XSS has also been applied to SQL injection and so is surveyed in Section 3.6. We add some additional comments here regarding relevant aspects of that work to XSS.

As previously described in Section 3.6.2, Xie and Aiken designed an SQL injection vulnerability analysis that uses block- and function-summaries [99]. Their analysis requires some interaction with the user—the user must provide the filenames when the analysis encounters a dynamic include statement, and the user must tell the analysis whether each regular expression encountered in a filtering function is "safe." Asking the user about regular expression filters may be acceptable for SQL injection vulnerabilities where the regular expressions enforce relatively simple lexical rules, but this will not be acceptable for the sequences of complex regular expressions that programmers use to prevent XSS. Many manually written input validation routines include a series of functions, none of which are intended to be sanitizers in isolation. By analyzing the possible string values according to a formal specification using formal language techniques, we are able to make a stronger and

more reliable guarantee that a given program is free of XSS vulnerabilities.

Minamide designed a static string analyzer that is foundational to our work [64]. He suggested using this analysis to check for XSS vulnerabilities, but his proposed technique checks the whole document for the presence of the "`<script>`" tag. Because web applications often include their own scripts, and because many other ways of invoking the JavaScript interpreter exist, this approach is not practical for finding XSS vulnerabilities.

4.5.2 Client-Side Mitigation

We organize our discussion of client-side approaches into those that enforce policies on the local behavior of JavaScript code and those that regulate outbound traffic based on information gained during the JavaScript's execution.

Local Behavior Enforcement: Hallaraker and Vigna use logging and auditing integrated into the JavaScript interpreter to enforce any policy specified with JavaScript code [36]. Yu *et al.* describe formally how to enforce arbitrary policies using interposition in a JavaScript-like language that is capable of code generation [101]. Both of these approaches impose low overhead because they integrate the enforcement mechanism into the JavaScript engine. However, they have no way to distinguish legitimate JavaScript from malicious JavaScript, and they leave open the question of which policy to enforce. BrowserShield prevents known browser vulnerabilities from being exploited by receiving a vulnerability description from a central server and interposing JavaScript wrappers to enforce the given policy [75]. BrowserShield has the advantage that it prevents real exploits, but it does not address the more general XSS problem, and because its enforcement mechanism consists of JavaScript wrappers to the JavaScript code, it may impose significant overhead.

Beep (Browser-Enforced Embedded Policies) overcomes the problem of distinguishing trusted from untrusted JavaScript by providing a mechanism for the client to enforce either a black-list or white-list policy that the server sends specifying which scripts are trusted [42].

This coarse-grained policy language is similar to script signing, where servers can sign the scripts that they intend to be executed and request the clients execute only those. Deployment poses a practical limitation for BEEP, because both the client and the server must use it in order for it to work.

Outbound Traffic Regulation: NOXES regulates activity that occurs over the network, but it does not address local behavior of JavaScript code [49]. Its default rule prohibits dynamically constructed links from being followed, because these are the primary mechanism attackers use to communicate sensitive information. It enforces this policy by adding JavaScript code underneath the received web document.

Vogt *et al.* propose a client-side, information flow-based policy to mitigate the effects of XSS [90]. Their approach involves marking the client's confidential data as tainted, tracking tainted data through the client's browser, and only allowing tainted data to be sent to sites that have permission to access that data. This approach augments the same-origin policy that browsers already enforce. The same origin policy only permits servers to access information on a user's system that the server "owns" (*e.g.*, the cookie for that site); common examples of XSS exploits use injected code to send the data that a server owns elsewhere, and this policy prevents that.

This client-side mitigation complements server-side analysis, in the sense that server-side analysis protects many clients of one server, whereas Vogt *et al.*'s approach protects one client from many servers. However, even applying their approach universally does not suffice to solve the XSS problem completely. First, their approach addresses only one class of XSS attack; it does not mitigate the damage of other XSS-based attacks, such as port-scanning (where the sensitive information does not appear in the form of data), browser vulnerability exploitation, web page defacement, and browser resource consumption. Second, as web applications move closer to the desktop, identifying confidential data becomes a bigger and more error-prone task. For example, the Google Desktop indexes a user's local system and runs a web server on it in order to provide efficient search capabilities. It had an XSS

vulnerability that, when coupled with an XSS vulnerability in Google that exposes the Google Desktop key, exposes the user's private data to remote attackers. Preventing this attack by tainting confidential data at the client-side would require designating confidential data at as fine a granularity as the DOM element level and as broad a scope as the user's whole system.

Chapter 5

Concolic Testing of Web Applications

In Chapter 3 we argued that static analysis for detecting injection vulnerabilities has the advantage that it can help programmers to correct errors early rather than leaving errors to be discovered in the field. However, static analysis has limitations of its own, such as program constructs that render static analysis highly imprecise or even useless. The pre-deployment complement to purely static analysis is testing-based analysis which will not find false positives and is well-suited to handle the program constructs that hinder static analysis. This chapter presents a testing-based analysis to find SQL injection vulnerabilities. This analysis is based on a novel "SQL predicate" that guides the analysis to vulnerabilities. Our evaluation demonstrates that this analysis can find vulnerabilities in real-world code.

5.1 Introduction

Static analysis has the significant advantage over runtime enforcement that it can find errors prior to deployment. However, some program constructs impede static analysis by making it lose too much information to be useful, and that was our experience with the analysis presented in Chapters 3 and 4. In order to retain the benefit of pre-deployment

bug discovery on code that proves difficult to analyze statically, we propose a testing-based analysis to find input validation errors. Specifically, we apply our analysis to SQL injection, although by modifying underlying policy, we could apply the analysis to XSS as well.

Testing is a widely used approach for identifying bugs and for providing concrete inputs and traces that developers use for fixing bugs. However, manual testing requires extensive human effort, which comes at significant cost. Additionally, QA (quality assurance) testing usually attempts to ensure that the software can do everything it ought to do, but it does not check whether the software can do things it ought not to do; such functionality usually constitutes security holes.

Our goal in this chapter is to help automate the process of web application testing. In particular, we seek to generate test cases automatically that will achieve a designated code-coverage metric, branch coverage or (bounded) path coverage, and in the process, find SQL injection vulnerabilities. Previous work on concolic testing has helped to automate test input generation for desktop applications written in C or Java [82,83], but web applications written in scripting languages such as PHP pose different challenges that we must address.

First, PHP is a scripting language and not a compiled language. Such languages, especially in the context of web applications, encourage a style of programming that is more string- and array-centric as opposed to languages like Java where numeric values and data structures play a more central role. In the limit, scripting languages allow for arbitrary meta-programming, although most PHP programs only make moderate use of dynamic features. Additionally, PHP web applications receive all user input in the form of strings, and many string manipulation and transformation functions may be applied to these values.

Second, in order for automatic test input generation to be useful, we need test oracles that will identify when common classes of errors have occurred. Several common classes of errors in C programs are memory errors; Java has eliminated most memory errors, but Java programs may still have null-pointer dereference errors. On the other hand, PHP programs are entirely free of memory corruption errors (barring bugs in the interpreter). Hence other kinds of test oracles are needed.

Finally, much of the previous work on concolic testing is designed only for unit testing. Because many value-based and information flow-based errors (as opposed to memory-based errors, for example) span multiple functions, we need to make the concolic testing approach scale beyond single functions.

This chapter presents the first application of concolic testing to web applications. We address the first challenge by first borrowing techniques from our static analysis to model string operations using finite state transducers and second employing a constraint resolution algorithm that resolves constraints on string values. We address the second challenge by developing a novel algorithm to check string values against our policy on SQL injection attacks. We address the third challenge by using values collected at runtime to construct a backward slice from where queries are constructed and recording constraints only from that slice.

In order to accomplish our goal of applying concolic testing to PHP web applications to find SQL injection vulnerabilities, we had to address several challenges. In PHP, not only do many library functions take arguments of one type and return values of another, but the runtime system itself readily performs many different dynamic type casts. Consequently, subexpressions of the constraints we generate may be over types other than string. The constraints we gather include some expressed through FSTs. To solve the constraints, we leverage the property of FSTs that they can be inverted and borrow from existing work on language equations [53]. The concolic testing framework helps to resolve constraints by supplying values from the program's execution in place of intractable subexpressions. In order to model precisely the semantics of PHP's many library functions and to support runtime type casts in generated constraints, we approximate expressions by considering only one variable occurrence per expression at a time. In our evaluation, none of the expressions had more than one variable occurrence,[1] so this approximation did not introduce any imprecision. We evaluated our implementation on real-world PHP programs with

[1]Web applications often handle each input individually rather than comparing inputs or combining them in predicates. The scope of our evaluation was somewhat limited, however, and a broader study could more effectively show the trends in web programming practice.

user.php

```
312 isset ($_GET['userid']) ? $userid = $_GET['userid'] : $userid = '';
313 if ($USER['groupid'] != 1)
314 { // permission denied
315   unp_msg($gp_permserror);
316   exit;
317 }
318 if ($userid == '')
319 { unp_msg($gp_invalidrequest);
320   exit;
321 }
322 $userid = "00".$userid;
323 if (!eregi('00[0-9]+', $userid))
324 { unp_msg('You entered an invalid user ID.');
325   exit;
326 }
327 $getuser = $DB->query("SELECT * FROM"
328     ."`unp user` WHERE userid='$userid'");
329 if (!$DB->is_single_row($getuser))
330 { unp_msg('You entered an invalid user ID.');
331   exit;
332 }
333 ...
```

Figure 5.1: Example PHP code.

known SQL injection vulnerabilities that existing static analysis tools fail to find. Our implementation took between three and thirteen minutes on these subjects and successfully found the vulnerabilities (details appear in Section 5.4).

5.2 Overview

Section 5.2.1 presents some example code in order to illustrate some of the significant points about our approach. Sections 5.2.2 and 5.2.3 show on this code how we generate and resolve constraints. Section 5.2.4 shows how we determine whether a given run succeeds or fails, and finally Section 5.2.5 describes an optimization we used to make this analysis scalable.

5.2.1 Example Code

Figure 5.1 shows some sample PHP source code that we will use to present our approach. This code takes a user ID, and attempts to authenticate the user to perform other actions. If the user's ID does not appear in the database, the program exits with an error message. This particular code fragment does not use dynamic features, but it still serves to illustrate some of the main points of our algorithm. In the presence of dynamic features, we simply record the concrete values of interpreted strings and use those values in our constraint generation and resolution.

5.2.2 Constraint Generation

As in the case of standard concolic testing, we instrument the program in order to execute it both concretely, using the standard runtime system, and symbolically. The testing framework's top-level loop executes the program, records symbolic constraints, and uses the constraints to generate new inputs that will drive the program along a different path on the next iteration. Symbolic execution takes place at the level of a RAM machine, which means that we maintain maps from names to symbolic locations, and a map from symbolic locations to symbolic values. Therefore the analysis does not require any offline alias analysis. The testing framework records a symbolic constraint for each conditional expression that appears in the program's execution.

On the first iteration, the testing framework executes the program without providing any input parameters. When it encounters the `isset` conditional on line 312, it records the constraint:

$$GET[userid] \in \emptyset$$

and the program reaches line 320 and exits. This constraint says that no value exists for the variable $GET[userid]$. Each of the constraints it gathers is expressed as a language

membership constraint. For the next run, the testing framework inverts this constraint:

$$GET[userid] \notin \emptyset \quad \Leftrightarrow \quad GET[userid] \in \Sigma^*$$

It then finds ϵ, the empty string, as the shortest value in Σ^*, and reruns the program with the `GET` parameter's "userid" set to "". This illustrates a useful feature of our approach: we do not need to specify or infer via static analysis the interfaces for the PHP programs we test. When the program expects a parameter that our testing framework omits, that parameter will show up in a constraint that, when inverted, will cause the parameter to be included in the next run. This is not only the case when explicit conditionals check whether variables are set, but also when any uninitialized variables are used.

For this example, we assume that the condition on line 313 holds. On the second iteration, the framework gathers the constraints:

$$[GET[userid] \in \Sigma^*,\ GET[userid] \in \{\epsilon\}]$$

The execution reaches line 320 and exits. The testing framework inverts the last constraint to perform a depth-first search of the program's computation tree:

$$[GET[userid] \in \Sigma^*,\ GET[userid] \notin \{\epsilon\}] \quad \Leftrightarrow \quad GET[userid] \in \Sigma^+$$

Again the testing framework selects some shortest value in Σ^+, in this case '`a`.'

On the third iteration, the framework gathers the constraints:

$$[GET[userid] \in \Sigma^*,\ GET[userid] \notin \{\epsilon\},\ 00.GET[userid] \notin \mathcal{L}(A_{\texttt{00[0-9]+}})]$$

Inverting constraints such as the last one here requires techniques beyond those that have been proposed in the testing literature because this constraint includes a string operation, *viz.* concatenation.

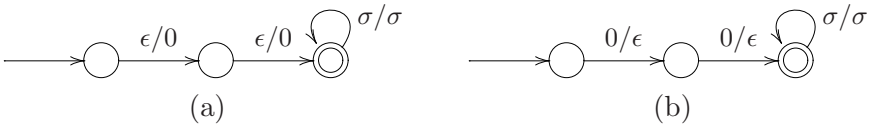


Figure 5.2: An FST representation of concatenating "00" to another string, and the FST's inverse; $\sigma \in \Sigma$.

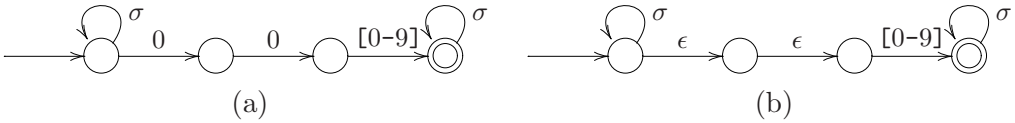


Figure 5.3: An FSA representation of the `ereg` "`00[0-9]+`" and its image over the FST in Figure 5.2b; $\sigma \in \Sigma$.

5.2.3 Constraint Resolution

The problem of satisfiability of word equations with regular language constraints is PSPACE-complete [74]. However, our constraint language is more expressive than this because we include nondeterministic rational relations, expressed as FSTs, and many classes of language constraints are undecidable [53]. Consequently, we cannot solve precisely every constraint in the language of constraints that may be generated. However, a benefit of the concolic testing framework is that the constraint resolution algorithm can be incomplete or even wrong, and no false positives will be reported. We design our algorithm for the common case in which input variables appear only on the left-hand side of each language membership constraint.

Here we show by example how our algorithm uses finite state transducers (FSTs) to invert string operations. Figure 5.2a shows an FST that represents the curried function "`00.`", *i.e.*, the function that prepends the string "`00`" to its argument. The first two transitions each read nothing and output "`0`" and the third transition outputs whatever it reads. FSTs can be inverted by swapping the input symbol with the output symbol on each transition. Figure 5.2b shows the FST inverted.

Figure 5.3a shows an FSA representation of the language of strings that match the

regular expression on line 323 of Figure 5.1. Because the regular expression does not have anchors (‘`^`’ for “beginning of the string” and ‘`$`’ for “end of the string”), the pattern must only appear somewhere in the string, as the FSA shows. Because the FST in Figure 5.2b represents the inverse of prepending “`00`,” this FST can be applied to the FSA in Figure 5.3a to produce the FSA in Figure 5.3b. The language of this FSA represents the language of values for $GET[userid]$ for which the conditional expression on line 323 will evaluate to true. As before, the language of this FSA must be intersected with the languages of the other FSAs for the same variable in order to find the language of values that will cause the program to take a new path in its computation tree. A new value, such as “`0`” can then be selected for the `userid GET` parameter.

5.2.4 Test Oracles

In order to be useful, automatic test input generation requires a *test oracle* that will give feedback on each execution of the program. Typically this feedback takes the form of *pass* or *fail*. In the case of testing C programs, example test oracles include `assert` statements, and tools like Valgrind [69], which monitors the memory and checks for memory errors such as buffer overflows and double-frees. Scripting languages, however, are not vulnerable to memory corruption errors, and although programmers use `exit` statements, in practice, very few use `assert`’s.

The same need for test oracles arises with fuzz testing and testing based on a preset list of inputs, and currently for web applications at least two kinds of oracles are used. First, security testers often see whether the input causes the web browser to pop up an alert window. If it does, this indicates a cross-site scripting (XSS) vulnerability. Second, testers check to see whether corresponding pages of sites written to be configured for multiple natural languages have the same structure. If they do not, this indicates that some data is missing in one of the languages.

These test oracles are available for our setting as well, but because our testing framework has full view of all the data values, we can use a more sophisticated oracle. Grasp [26] is

a modified version of the PHP interpreter that performs character-level tainting [70], and allows security policies to be defined on strings based on tainting. A typical example of such a policy is for SQL injection, and particularly SQL injection as we define it. This can be used as a pass/fail oracle for the tests we generate and execute. We can also use other taint-based policies, such as the policy that tainted strings in the web application's output document must not invoke the clients JavaScript interpreter, as described in Chapter 4.

An advantage to using the taint-based policies we have described is that we can attempt to generate inputs that will result in failing runs and so discover bugs. In the case of SQL injection vulnerabilities, prior to each call to the query function in the database API, we apply an implicit *SQL conditional* to the string value of the query. That conditional does not appear in the program or in the execution of the program; it is simply recorded as a constraint in our symbolic execution. The constraint specifies that substrings in the query from user input are syntactically confined. In order to invert this constraint, we take essentially the same approach as with other constraints, but with one significant difference. With other constraints, the initial representation of the language of values for the variable is the regular language Σ^*, the inverse operations given by the constraints are applied to that. With SQL conditionals, the initial representation of the language of values for the variable is the SQL grammar, and the inverse operations are applied to that. The image of a context-free language represented by a CFG over an FST can be constructed using an adaptation of the CFL-reachability algorithm [76] to construct the intersection of a CFG and an FSA [37].

The structure of the resulting CFG G' corresponds to the structure of the SQL CFG such that for a PHP variable v whose value is used to construct the query string, we extract a sub-grammar G_v from G' such that $v \in \mathcal{L}(G_v)$. The SQL predicate checks symbolically whether all possible values for v are safe based on the structure of G_v. In the case of our running example, this predicate does not hold. Rather than inverting it by taking its complement, we extract from G_v G'_v, the grammar for values for v where $\mathcal{L}(G'_v) \subseteq \mathcal{L}(G_v)$ and every string in $\mathcal{L}(G'_v)$ represents an attack input. G'_v can be constructed because it is

based on the finite structure of G_v.

To resolve the constraints on v, we take the intersection of $\mathcal{L}(G'_v)$ with the intersection of the other regular languages that bound v's range. The result is a CFG, and finding a word in the language of a CFG can be done in linear time. Such a word will then be supplied as input for the next test run, and if it indeed violates the security policy, the runtime system will catch it. Because the intersection of two CFGs cannot be constructed in general, we only handle one SQL constraint on each variable at a time. In the case of our running example, the result of resolving the SQL predicate is too involved to show in a meaningful picture, but the algorithm will produce a string like "`0' OR 'a'='a`," which will result in an attack.

5.2.5 Selective Constraint Generation

In real-world programs, much of the program's execution has little to do with the property of interest. For example, a web application page that constructs a query may also instantiate a timer class to limit the execution time of the page, construct from configuration files HTML fragments that have text in the user's language, and other such operations. Gathering constraints on unrelated parts of the code adds unnecessary overhead both to the constraint generation and to the constraint resolution. Previously, however, automated input test generation efforts did gather constraints for the entire execution because: (1) the points of possible failure may not be known statically; and (2) it may be difficult to compute a backward static slice from failure points that are known.

We propose an iterative approach to narrow the focus of our constraint generation to constraints that are relevant to possible failures. First, we identify points of possible failure. In the case of SQL injection, these are the program points where the API function is called to send a query to the database. We then add all functions in which these points occur to a set of functions to be analyzed. We then execute the program (*i.e.*, load the page) and gather constraints from only the functions in this set. If the initial execution does not encounter a query function and hence gathers no constraints, we instrument the

$$
\begin{array}{lcl}
e & \rightarrow & e\ bin\_op\ e \quad | \quad \mathrm{c} \quad | \quad (cast)\ e \\
 & | & f(e,\ldots) \quad | \quad e[e] \quad | \quad v \\
bin\_op & \rightarrow & str\_op \mid num\_op \mid bool\_op \\
f & \rightarrow & str\_f \mid num\_f \mid bool\_f \mid arr\_f \\
str\_op & \rightarrow & . \quad \text{(concatenation)} \\
str\_f & \rightarrow & \text{trim} \mid \text{add\_slashes} \mid \text{implode} \mid \ldots \\
num\_op & \rightarrow & + \mid - \mid \times \mid \div \mid \% \\
num\_f & \rightarrow & \text{count} \mid \text{strlen} \mid \ldots \\
bool\_op & \rightarrow & = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
bool\_f & \rightarrow & \text{isset} \mid \text{is\_int} \mid \text{ereg} \mid \ldots \\
arr\_f & \rightarrow & \text{explode} \mid \text{split} \mid \text{array\_combine} \mid \ldots \\
cast & \rightarrow & \text{int} \mid \text{float} \mid \text{string} \mid \text{bool} \mid \text{array}
\end{array}
$$

Figure 5.4: Expression language.

top-level file and in successive iterations instrument included files until a query function is encountered. We resolve control dependencies by recording a stack trace at the beginning of the function call. Those functions that invoked the analyzed functions then get added to the set of functions to analyze. We resolve data dependencies by examining symbolic values, for example, the symbolic value of the constructed query, and where branches terminate at function calls, we add those functions to the set as well. We then re-execute the program gathering constraints from all of the functions in the set, and repeat this process until no more dependencies exist. This process approximately computes a backward slice, but we use runtime data to compute it even in the presence of dynamically constructed function call names and variable names. By gathering constraint selectively, we reduce by several orders of magnitude the size of the constraint set to gather and resolve. Reducing the constraint set size facilitates scaling to larger programs, as our evaluation shows.

5.3 Algorithm

Figure 5.4 gives the grammar for the Boolean expressions from a PHP-like language, and the grammar implicitly defines the structure of the expressions' abstract syntax trees. The grammar includes some representative functions that return values of types string, Boolean,

```
SingleOcc(t)
 1  switch t
 2    case v :
 3         return {t}
 4    case c :
 5         return {}
 6    case (cast)e :
 7         r ← ∅
 8         ss ← SingleOcc(e)
 9         for each s in ss
10         do r ← r ∪ {(cast)s}
11         return r
12    case e1 bin_op e2 :
13         r ← ∅
14         s1 ← SingleOcc(e1)
15         c2 ← GetConcrete(e2)
16         for each s in s1
17         do r ← r ∪ {s bin_op c2}
18         s2 ← SingleOcc(e2)
19         c1 ← GetConcrete(e1)
20         for each s in s2
21         do r ← r ∪ {c1 bin_op s}
22         return r
23    case f(args list) :
24         r ← {}
25         cc ← map GetConcrete args
26         for i in 0 to length(args)
27         do s1 ← SingleOcc(args[i])
28            if ∅ ≠ s
29               then cs ← cc
30                    cs[i] ← s1
31                    for each s in s1
32                    do r ← r ∪ {f(cs)}
33         return r
34    case ... :
```

Figure 5.5: Algorithm to construct sets of single variable-occurrence expressions.

and array, as well as values of numeric types. The constraints recorded from the execution of the subject program come from this grammar. Although the grammar does not specify the arity of each function, PHP's runtime system executes only programs in which functions

c	$\rightarrow$	$\bullet \in \mathcal{L}$	$\mathcal{L}$ is a regular language
	$\mid$	$\bullet$ *op i*	integer
	$\mid$	$\bullet$ *op f*	float
	$\mid$	$\bullet = b$	Boolean
	$\mid$	$\bullet$*[index]* c	constraint on array element
	$\mid$	$\mid \bullet \mid$ *op i*	constraint on array length
index	$\rightarrow$	i	integer index
	$\mid$	s	string index
	$\mid$	$\top$	unknown index
op	$\rightarrow$	$= \mid \neq \mid > \mid \geq \mid < \mid \leq$	

Figure 5.6: Constraint language.

have the right number of arguments. Since we analyze constraints collected from a run of the program, the PHP runtime system guarantees that each function will be passed the right number of arguments.

The PHP runtime system performs runtime casts among many types automatically. For example, PHP will not throw an exception when trying to execute (3+ "a"); rather, because the "+" operator takes two integers as arguments, the runtime system will automatically cast "a" to an integer and evaluate the expression. Because the string "a" does not represent a numeric value, the runtime system will cast "a" to 0. In addition, many functions, such as the length function, take arguments of one type and return arguments of another.

Previous work on concolic testing solves constraints by selecting a theory and replacing expressions that lie outside of the chosen theory with the corresponding concrete value gathered from an execution. PHP's propensity to convert between types makes it difficult to identify a decidable theory that covers most of the constraints that a program execution will generate. Consequently, we adopt a different approximation strategy from the approach proposed in previous work on concolic testing. For each variable occurrence in a Boolean control expression encountered in a test execution, we create a copy of the expression and set all other variable occurrences in the expression to their concrete values from the execution. The SingleOcc function in Figure 5.5 shows how this operation is performed on a representative set of expressions. It returns a set of expressions where each expression

in the set has only a single variable occurrence and each subexpression that does not depend on that variable is replaced with its concrete value. Note also that certain subexpressions, such as function names, are replaced automatically with their concrete values.

The constraint gathering phase collects a list of Boolean control expressions along with their runtime values, each one being either true or false. In order to generate a set of input values to take a new program path, we select from the list an expression e to invert and discard the expressions following e in the list. We apply SingleOcc to each expression in the remaining list. The result is a list of expressions, each with a single variable occurrence, and each with a Boolean value to which it ought to evaluate in order for the new input values to drive execution along the designated path.

The function SolveFor in Figure 5.10 generates a constraint on the variable in an expression by solving the expression against a constraint. We pass to it an expression from the list of Boolean control expressions along with the expected Boolean value as a constraint. Figure 5.6 shows the language of constraints for this algorithm. Each constraint form has a hole, designated with "•", for the expression it constrains. The language includes constraints on integer, floating point, Boolean, string, and array values. The constraints on Boolean values are simply equality to Boolean constants. When SolveFor is called with an initial expression and Boolean value, the constraint says that the value of the expression equals the given Boolean value. The constraints on string values are regular language membership constraints. The constraints on arrays include length constraints and element constraints. The language does not, however, include standard Boolean operators, such as conjunction and disjunction, because it is designed for expressions with a single variable occurrence.

The SolveFor function is designed to retain as much precision as possible across arbitrary type conversions. It does not require that a given expression has the same type as the supplied constraint. Rather, it converts the constraint to have the same type as the expression. Converting the type of the constraint requires working backward across PHP's type conversion rules. To illustrate, Figure 5.7 shows the rules for converting Boolean values

cast	true	false
(int)	1	0
(float)	1.0	0.0
(string)	“1”	“”
(array)	[true]	[false]

(a)

step	expression	constraint
1	5 + ereg(“^[a-z]*$”, x)	(= 5)
2a	ereg(“^[a-z]*$”, x)	(= 0)
2b	ereg(“^[a-z]*$”, x)	(false)
3	x	$\left(\overline{\mathcal{L}(\text{^[a-z]*\$})}\right)$

(b)

Figure 5.7: PHP's rules for type conversion from Boolean; an example constraint resolution with a type conversion from Boolean.

MakeBoolConstraint(t)

```
 1  switch t
 2    case b : return b
 3    case op n when ¬(0 op n) and ¬(1 op n) :
 4         raise Unsatisfiable
 5    case op n when 0 op n and ¬(1 op n) :
 6         return false
 7    case op n when ¬(0 op n) and 1 op n :
 8         return true
 9    case op n when 0 op n and 1 op n :
10         raise Unconstrained
11    case op n when x op n ⇒ x ≠ 0 : return true
12    case ∈ L when 1 ∉ L and ε ∉ L :
13         raise Unsatisfiable
14    case ∈ L when 1 ∈ L and ε ∉ L : return true
15    case ∈ L when 1 ∉ L and ε ∈ L : return false
16    case ∈ L when 1 ∈ L and ε ∈ L :
17         raise Unconstrained
18    case length op i when 0 op i :
19         raise Unconstrained
20    case length op i : raise Unsatisfiable
21    case [index]t' : return MakeBoolConstraint(t')
```

Figure 5.8: Algorithm for converting arbitrary constraints to Boolean constraints.

to integer, floating point, string, and array values. Figure 5.8 shows the rules for converting an arbitrary constraint. The following contrived example helps to illustrate this point: The expression and the constraint in step 1 both have type integer, so the 5 can be subtracted from both. The ereg function in step 2a returns a Boolean value, but the constraint is over

```
IMAGE(A, F)
 1   // Image of FSA A over FST F
 2   (Q_1, Σ, s_1, f_1, δ_1, L_1) ← A
 3   (Q_2, Σ, s_2, f_2, δ_2, L_2) ← F
 4   δ ← ∅; Q ← ∅; L ← ∅
 5   δ'_1 ← δ_1; δ'_2 ← δ_2
 6   for each q ∈ Q_1
 7   do δ'_1 ← δ'_1 ∪ {(q, ϵ, q)}
 8   for each q ∈ Q_2
 9   do δ'_2 ← δ'_2 ∪ {(q, ϵ, ϵ, q)}
10   for each (q_s1, i, q_t1) in δ_1
11   do for each (q_s2, i, o, q_t2) in δ_2
12       do δ ← δ ∪ {((q_s1, q_s2), o, (q_t1, q_t2))}
13   for each q_1 in Q_1
14   do for each q_2 in Q_2
15       do Q ← Q ∪ {(q_1, q_2)}
16          if q_1 ∈ domain (L_1)
17             then L ← L ∪ {(q_1, q_2) → L_1(q_1)}
18   (Q, Σ, (s_1, s_2), (f_1, f_2), δ, L)
```

```
INVIMAGE(a : FSA, f : FST)
1   a_2 ← COMPLEMENT(a)
2   f_2 ← INV(f)
3   a_3 ← IMAGE(a_2, f_2)
4   a_4 ← COMPLEMENT(a_3)
5   return a_4
```

Figure 5.9: FSA image and construction and inversion.

integers. According to the type conversion rules in Figure 5.7, false converts to 0, which satisfies the constraint, and true converts to 1, which does not. The constraint then gets converted to false in step 2b. The function ereg returns false when the value of its second argument is not in the language represented by its first argument, so step 3 finishes with the variable x as its expression and a constraint specifying a language in which the value of x lies.

As Figure 5.8 shows, certain type conversions lose the constraint on the value of the expression (as on lines 10 and 17), and other type conversions are inconsistent with the constraint (as on lines 4 and 13). When inconsistencies occur, the algorithm fails to find input values for a certain path. Converting constraints to Boolean constraints does not require approximation, but converting to other types requires approximation in some cases.

Figure 5.9 gives an algorithm for finding the image of an FSA over an FST. Figure 5.9 shows the algorithm that calls IMAGE to find the maximal pre-image of a regular language

```
SolveFor(e : expr, t : constraint)
 1  switch e
 2    case c :
 3         raise ConstantExpression
 4    case v :
 5         return v, t
 6    case e' + c :
 7         op i ← MakeIntConstraint(t)
 8         t' ← op (i − c)
 9         return SolveFor(e', t')
10    case ereg(reg, e') :
11         b ← MakeBoolConstraint(t)
12         fa ← RegToFsa(reg)
13         if not b
14           then fa ← Complement(fa)
15         t' ←∈ L(fa)
16         return SolveFor(e', t')
17    case stripslashes(e') :
18         ∈ L(fa) ← MakeStrConstraint(t)
19         fa ← InvImage(fa, f_stripslashes)
20         t' ←∈ L(fa)
21         return SolveFor(e', t')
22    case count(e') :
23         t_1 ← MakeIntConstraint(t)
24         t_2 ← length t_1
25         return SolveFor(e', t')
```

Figure 5.10: Algorithm to solve for variables.

over an FSA. The SolveFor function uses the InvImage routine for solving over PHP's string functions, as on line 19 in Figure 5.10. Because FSTs may be nondeterministic, applying the inverse of an FST to an FSA directly will not yield the maximal pre-image; the complement of the pre-image of the complement of the language yields the maximal solution.

Section 5.2.4 describes an artificial SQL predicate that our instrumentation inserts into programs wherever the SQL query function is called. This predicate initially takes the form of "$\bullet \in \mathcal{L}(G_{SQL})$," where G_{SQL} is the SQL grammar. Resolving this predicate determines whether SQL injection attacks are possible, and if so, generates input that will cause an

attack. The algorithm for resolving this predicate proceeds initially as the algorithm for resolving other predicates: operations on the single occurrence of a variable are successively inverted and applied to the grammar. The image of a context free grammar over a finite state transducer is again context free. We need only consider each query site individually, so other SQL predicates are excluded from the symbolic expression and the intersection of two context-free grammars will not be needed. This resolution produces a predicate that has the form "$v \in \mathcal{L}(G)$" for some grammar G. If other regular language predicates on the variable v exist, the predicates can be combined by computing the intersection of the regular languages and G.

The CFG representation G_2 of the image of a CFG G_1 over an FST or the intersection of a CFG G_1 and an FSA can be constructed such that each nonterminal in G_2 corresponds to some nonterminal in G_1 (*cf*, Figure 7 in [94]). This means that for a predicate "$v \in \mathcal{L}(G)$," the symbols in G can be related to the symbols in the SQL grammar G_{SQL}. Let $P_{SQL}(X_1) = X_2$ if X_1 is a nonterminal in G, X_2 is a nonterminal in G_{SQL}, and X_1 was constructed based on X_2. The grammar G then describes how the initial values of v, after being transformed by the program and incorporated into queries, will relate to the SQL grammar. G is constructed conservatively, so that it may describe some initial values of v as being incorporated into queries in ways that they will not.

Let G be *precise* if (1) every string in $\mathcal{L}(G)$ is a value for v for which the program will take the same path as in the logged execution provided that the other inputs remain the same, and (2) every string value, after being transformed and incorporated into a query, will be parsed under G_{SQL} as given by the correspondence between the symbols in G and the symbols in G_{SQL}. If $G = (V, \Sigma, R, S)$ is precise, then a string $s \in \mathcal{L}(G)$ will cause an attack iff there exists no X such that the following conditions hold:

- $s \in \mathcal{L}(V, \Sigma, R, X)$ and
- for all sentential forms γ such that $X \Rightarrow^* \gamma \Rightarrow^* s$, if $\gamma = \alpha X_1$ or $\gamma = X_1\alpha$, and if $\alpha \Rightarrow^* s$, then $P_{SQL}(X_1) \Rightarrow^*_{G_{SQL}} \epsilon$.

Such a string $s \in \mathcal{L}(G)$ exists for $G = (V, \Sigma, R, S)$ if for some $X \in V$ and some $X_1 \rightarrow \alpha X \beta$, $\mathcal{L}(V, \Sigma, R, X) \not\subseteq \{\epsilon\}$ and either

- $\beta = \epsilon$, $FOLLOW(X) \not\subseteq \{\epsilon\}$, and $\alpha \Rightarrow^* X_2\alpha'$, or
- $\alpha = \epsilon$, $PRE(X) \not\subseteq \{\epsilon\}$, and $\beta \Rightarrow^* \beta' X_2$,

where $\epsilon \in \mathcal{L}(V, \Sigma, R, X_2)$ and $\epsilon \notin \mathcal{L}(V_{SQL}, \Sigma_{SQL}, R_{SQL}, P_{SQL}(X_2))$, and where $FOLLOW(X)$ is the follow set of X and $PRE(X)$ is the follow set of X in the right-to-left direction. Standard algorithms exist for finding the follow set of a nonterminal and determining whether a nonterminal can derive ϵ. In order to generate a string that will cause an attack, we identify an X as described above and derive a string through it, including a non-empty string from X's follow/pre set and ϵ derived from X_2.

5.4 Evaluation

This section discusses our implementation and the test cases we used, and then presents the results of our evaluation.

5.4.1 Implementation

As previously discussed, our approach has two phases: constraint generation and constraint resolution. PHP is an interpreted language, so the constraint generation phase could be implemented directly in the interpreter. However, it is not clear how the first phase could avoid generating unnecessary constraints if the interpreter has only the web application code. Consequently, we chose to implement constraint generation at the language level. We wrote a plugin to phc, an open source PHP compiler front-end [18], to perform a source-to-source transformation on the PHP code that we want to gather constraints from. The plugin consists of about 2200 lines of C++, and it wraps each statement in a function call. The wrapper functions write to a file a trace log of the program execution. Additionally, on evaluated strings in transformed code, the transformed program first passes the code to

gpc_api.php

```
46 function gpc_get_string( $p_var_name, $p_default = null ) {
47   $args = func_get_args();
48   $t_result = call_user_func_array( 'gpc_get', $args );
49   if ( is_array( $t_result ) ) {
50     error_parameters( $p_var_name );
51     trigger_error( ERROR_GPC_ARRAY_UNEXPECTED, ERROR );
52   }
53   return $t_result;
54 }
```

Figure 5.11: Input handling code in Mantis.

be executed through the source-to-source transformation so that the new code will also be logged.

The second phase reads in the log and symbolically executes the trace. It produces a list of Boolean control expressions where each subexpression is annotated with a concrete value from the execution. This phase is implemented using about 5200 lines of OCaml in addition to Minamide's finite automata and regular expression libraries [64].

5.4.2 Test Subjects

We selected three real-world PHP web applications with known SQL injection vulnerabilities to evaluate our implementation. The first, Mantis 1.0.0rc2, is an open source bug tracking system, similar to Bugzilla. It has an SQL injection vulnerability in its "lost password" page, and the top-level PHP file for this page includes transitively 27 other files for a total of 17,328 lines of PHP in the page. The second, Mambo 4.5.3, is an open source content management system. It has an SQL injection vulnerability in its "submit weblink" page, and the top-level PHP file for this page includes transitively 23 other files for a total of 13,248 lines of PHP in the page. The third, Utopia News Pro 1.3.0, is a news management system. It has an SQL injection vulnerability due to insufficient regular expression filtering in its user-management page. It includes transitively 6 other files for a total 1,529 lines of PHP code.

mambo.php

```
1973 function mosGetParam( &$arr, $name, $def=null, $mask=0 ) {
1974   if (isset( $arr[$name] )) {
1975     if (is_array($arr[$name]))
1976       foreach ($arr[$name] as $key=>$element)
1977         mosGetParam ($arr[$name], $key, $def, $mask);
1978     else {
1979       if (!($mask&_MOS_NOTRIM))
1980         $arr[$name] = trim( $arr[$name] );
1981       if (!is_numeric( $arr[$name] )) {
1982         if (!($mask&_MOS_ALLOWHTML))
1983           $arr[$name] = strip_tags($arr[$name]);
1984         if (!($mask&_MOS_ALLOWRAW)) {
1985           if (is_numeric($def))
1986             $arr[$name] = intval($arr[$name]);
1987         }
1988       }
1989     }
1990     return $arr[$name];
1991   } else {
1992     return $def;
1993   }
1994 }
```

Figure 5.12: Input handling code in Mambo.

Both of the first two web applications we tested use dynamic features for parts of the code that are relevant to query construction. First, both web applications include files dynamically by specifying the names of files to include via dynamically constructed string values. Some static analyzers require user intervention to provide static file names in order to get all of the code that the application will use, although others use constant propagation or related techniques to construct some file names automatically.

Second and more importantly, both use dynamic features in handing user input. Figure 5.11 shows the gpc_get_string function from Mantis' gpc_api.php file ("gpc" stands for GET-POST-COOKIE, the three primary vehicles for delivering user input to the application server). The call to func_get_args() on line 47 returns as an array the list of arguments that was passed to the user-defined function in which it is called. The call to call_user_func_array() on line 48 calls the function named by the string value of the first ar-

Test Case	Translated functions	Log file size	Logging time (s)
Mantis	1	8 KB	1
	2	13 KB	1
	3	18 KB	1
	4	19 KB	1
	all	$\geq$ 2.9 GB	$\geq$ 600

Table 5.1: Trace log file data.

gument passing array in the second argument the function as an argument list. In this case, it is the function gpc_get that retrieves input values directly. Figure 5.12 shows the mosGetParam function, the function used for getting input values, from Mambo's mambo.php file. This function takes as arguments an array reference and the string value of the name of an index and returns the value of the appropriate array element. In some calls to this function, the array passed is itself a dynamically index element of an array (*e.g.*, GLOBALS).

We tried to analyze both web applications using two static analyzers: Pixy [43], and our tool described in Section 3.4, which is based on Minamide's PHP static analyzer [64]. Because of dynamic features such as those shown above, both failed to find any SQL injection vulnerabilities in these files that have known vulnerabilities.

5.4.3 Evaluation

As previously stated, the first phase of our analysis performs a source-to-source translation on PHP files so that the resulting files, when executed, will write to a file a trace log of their execution. For all execution and logging experiments, we set the maximum execution time at 5 minutes per iteration (execute, log, resolve constraints). The first set of experiments that we ran shows that logging the whole trace can be prohibitively expensive. Figure 5.1 shows the execution times and corresponding log sizes for Mantis when increasing numbers of functions were translated and executed. For the assertion we were checking, four functions proved to be sufficient to cover the backward slice. When all of the code was translated and executed, the page failed to load in our browser before timing out. At that point, the

Test Case	Inputs Generated	Time (mm:ss)	Max Log Size (KB)
Mambo	4	13:02	65
Mantis	5	03:38	19
Utopia News Pro	23	05:14	17

Table 5.2: Iterations to find an injection vulnerability.

log file size was 2.9 GB. In this experiment, when a file was to be included, our translation dynamically translated the file, wrote it to a new file, and included the new file. This dynamic translation added to the execution time, but not to the log file size.

In our experiments, only the conditional expressions on constructed queries in our added assert statements had more than one variable occurrence. This means that for the conditional expressions in our experiments, our algorithm did not make any approximations by considering only one variable occurrence per expression instance.

Figure 5.2 shows for each program in our evaluation how long it took to find an input that caused an SQL injection attack in terms of the number of test inputs generated and the total time to generate them. Mambo and Mantis required relatively few test inputs before they generated an attack. This is because in each of their pages, the query constructed at the vulnerable program point plays a central role in the page. If the inputs that get included in the query are not present, the page produces an error message before it has done much else. Once the inputs are provided that cause the page to produce a query successfully, the execution also encounters the implicit conditional inserted by our source-to-source transformation that checks whether the query is an attack. The next input will then produce an attack. In contrast, our implementation produced 22 inputs to Utopia News Pro before producing one that results in an attack. This is because the page we tested performs several roles in the application and essentially has a large switch-case statement on input values to select which action it should take. The vulnerable program point in this page was not reached until several other branches of the switch-case statement had been tried. Although Mambo required the fewest inputs of our test cases to reach an injection

attack, it took the most time. This was because, as indicated by the maximum trace-log size, the page performed more operations before reaching the vulnerable program point than it did for Mantis or Utopia News Pro. Consequently, there were more constraints to be resolved for each path, and for each input generated, our implementation attempted to resolve constraints for several paths that proved to be unsatisfiable.

5.5 Limitations

Previous work on leveraging symbolic and runtime values for input test generation falls back on concrete values when the symbolic value being addressed lies outside the theory of the resolution algorithm's decision procedure. Our constraint resolution algorithm generates constraints only based on one variable instance per value. Therefore it may underapproximate the symbolic values of variables when program predicates depend on multiple variables, and it may miss paths that other resolution algorithms would find. In principle our constraint resolution algorithm could be enhanced to include multivariate constraints in some cases, but we leave that to future work.

Our approach of logging files selectively is effective only when the points of possible failure are known and relatively localized, as is the case with SQL injection, where the possible failure point is where the program sends queries to the database. If the problems of interest are potentially more ubiquitous in the program code, as with arbitrary runtime exceptions, logging selectively will be less effective. Logging the whole execution trace would address that problem, but it is prohibitively expensive. We expect that modifying the PHP interpreter to generate symbolic constraints directly may alleviate some of the expense of execution time, but that may make selective logging difficult.

At present, our implementation is not fully automated. The web page must be manually loaded (*e.g.*, by clicking "go"), the analyzer must be manually invoked, and analyzer writes the next inputs to a file, so they must be manually provided to the URL. However, in principle, nothing about our approach requires user interaction.

5.6 Related Work

In this section, we survey closely related work on test input generation and web application testing. We also consider previous work on static analysis to compare to our constraint resolution approach.

5.6.1 Test Input Generation

Traditional work on testing has generated random values as inputs [16, 56, 71]. Randomly generated input values will often be redundant and will often miss certain program behaviors entirely. Test input generation that leverages runtime values, or concolic testing, has been pursued by multiple groups [10, 11, 15, 31, 82, 83]. These approaches gather both symbolic constraints and concrete values from program executions, and use the concrete values to help resolve the constraints to generate the next input. Previous work on concolic testing handles primarily constraints on numbers, pointer-based data structures, and thread interleavings. This is appropriate for the style of programming that languages like C and Java encourage, but scripting languages, especially when used in the context of web applications, encourage a style in which strings and associative arrays play a more central role.

Perhaps the work most closely related to ours is by Emmi *et al.*, in which they augmented concolic testing to analyze database-backed Java programs. They added support for string equality and inclusion in regular languages specified by SQL `LIKE` predicates [22]. Our work is distinguished from theirs in at least the following aspects. They support a form of multi-lingual programming in which Java programs generate SQL queries, whereas we support a setting in which more general meta-programming is possible. They do not support any string operations, although they mention that string constraints with concatenation can resolved in PSPACE; whereas we support concatenation as well as many other string operations that PHP provides, although this requires us to make some approximations in our constraint resolution algorithm. They check for the same properties as standard concolic checking, whereas we check for security problems common among web applications.

Saner [3], which we discussed in Section 3.6, has a dynamic component, but that simply provides witnesses to validate the static analysis component, so it is somewhat different in spirit than the testing-based approach we presented in this chapter.

5.6.2 Web Application Testing

Some previous work on web application testing has focussed on static webpages and the loosely structured control flow between them (defined by links), and other work has focussed on the server-side code, often carrying over techniques from traditional testing. Early work on web application focussed primarily on static pages and the coverage metric was page-coverage. Ricca and Tonella propose a technique for using UML models of web applications to analyze static web pages via testing [77]. Kung *et al.* model web applications as a graph and develop tests based on the graph in terms of web page traversals [54]. The tool Veriweb explores sequences of links in web applications by nondeterministically exploring action sequences (*i.e.*, sequences of links) [4]. This tool provides data to forms using name-value pairs that provided by the tester.

Other testing techniques that attempt to test the effects of input values on web applications, but they require interface specifications and cannot guarantee code coverage without extensive user interaction. In some cases automated techniques derive the interface specifications [33] and in others developers must provide them [41], but either way, the testing system essentially performs fuzz testing that may be constrained by user-provided value specifications. Other testing mechanisms provide more reliable code coverage, but they repeatedly prompt the user for new inputs, so they sacrifice automation [58].

5.6.3 Static Analysis of PHP Web Applications

Section 3.6 surveys related static techniques, but we add some remarks here regarding how they related to the work presented in this chapter. All of the static techniques for PHP surveyed in Section 3.6 have limited effectiveness, because PHP supports dynamic features, in which the runtime system interprets data values as code, and dynamic features inhibit

static analysis. The standard dynamic features PHP provides allow string values to specify: the name of a file to include, the name of a variable to read/write, the name of a method to invoke, the name of a class to instantiate, and the string representation of code to execute. All of the static analyses for PHP described above either fail on dynamic features, treat them optimistically (*i.e.*, ignore them), ask the user to provide a value for each one, or do some combination of the three. Many PHP applications use dynamic features extensively, for example, to implement dynamic dispatch for dynamically loaded modules or for database handling code. On such code, static analysis fails to produce useful results.

In most real-world PHP programs, however, the values of interpreted strings come only from trusted values such as constant strings within the PHP code, for example in a factory pattern; column names from a known database schema; or field names from a protected configuration file. In such cases, the values of interpreted strings depend only indirectly on user input, and for any given run, the predicates on user inputs are not dynamically constructed.

Chapter 6

Conclusion

This dissertation presents a general framework to understand and prevent input validation-based attacks in a metaprogramming setting, *viz.* the domain of web applications. This dissertation describes the first formal, realistic characterization of SQL injection and XSS and presents principled, practical analyses for identifying vulnerabilities and preventing attacks. The analyses can detect and block real attacks and uncover unknown vulnerabilities in real-world code.

6.1 Summary

We began with a formal definition of SQL injection based on data integrity and sentential forms. Initially, most material presented SQL injection by example and described ad hoc solutions. We proposed the first principled definition of SQL injection by describing the effect that untrusted input is permitted to have on SQL queries that a web application constructs. Our characterization employs two primary notions: sentential forms, an abstraction from parsing algorithms used in compilers; and integrity, a fundamental concept in security that forbids untrusted data from modifying trusted data. Our definition provides a solid foundation for designing web database APIs as well as checking for attacks and vulnerabilities.

Our sentential forms-based definition suggests a parsing-based runtime enforcement mechanism, but parsing alone does not suffice because injection attacks must parse under the SQL grammar to succeed. We needed the insight that delimiting untrusted (i.e., input) strings and adding the delimiters to the SQL grammar provides an elegant, sound, and complete way to distinguish safe queries from attacks. The principal runtime overhead of our technique is the cost of a single parsing, and, in our experiments, our runtime checks prevented all attacks and permitted all instances of normal usage.

Although our runtime checks prevent attacks, we also pursued static analysis because, in software development, static analysis reveals errors that may indicate broader problems, informs programmers about repeatable mistakes, and reduces runtime overhead by removing redundant checks. Previously, taint analysis served to find web application vulnerabilities, but taint analysis cannot guarantee the absence of vulnerabilities given our definition; it does not model string values or string functions' semantics. We modeled both by leveraging synergistically string analysis and taint-tracking. Our analysis relies on novel adaptations of certain formal language algorithms, such as the classical algorithm for context-free language reachability, to track integrity levels and achieve soundness. Where a vulnerability exists, our analysis provides a witness in the form of an attack query.

XSS is a related but fundamentally more difficult problem. Whereas database systems restrict command execution to a well-defined language, web browsers do not. Web browsers parse HTML permissively as a result of early software engineering decisions that seemed beneficial in the short term—they enabled browsers to display poorly written HTML pages—but have now made XSS more difficult to prevent. We examined browser source code and discovered many subtle and undocumented ways for untrusted strings to invoke the JavaScript interpreter. We then constructed a policy that describes these ways using a regular language and employed our string-taint analysis to find XSS vulnerabilities. Our results are sobering: every manually written input validation routine that we analyzed and that allows any HTML fails to prevent XSS.

In the process of designing and implementing our static analysis, we found that many

web applications use dynamic language features, and such features inhibit static analysis. We looked to testing as a complement to static analysis, but previous work on automated test input generation focusses on numeric values and pointer-based data structures, which languages like C and Java emphasize. Web scripting languages, like PHP, promote a style of programming that emphasizes strings and associative arrays. We designed and implemented an algorithm for automated test input generation for web applications. By incorporating values gathered from program executions, we model string operation semantics more precisely than static analyses have done, and we found vulnerabilities in real-world code that every available static analyzer failed to analyze.

6.2 Extensions and Enhancements

Each of the chapters in this dissertation addresses a few central questions and proposes techniques or experimental data to address those questions, but each of the techniques has room for improvement and enhancement. Our runtime technique to enforce that no injection attacks occur involves tracking strings from their source to the web application's output, and we proposed using strings at the level of the program's data to represent delimiters. However, new strings at the level of the program's data may change the program's behavior. Tracking strings' sources via metadata would not introduce this problem. Halfond and Orso proposed a character-level taint tracking for Java that uses the class loader and thus avoids modifying the application source code or the underlying interpreter [35]. This technique partially accomplishes the goal we describe, but our policy involves substring-level tainting and so requires a modification to their approach.

Our static analyzer, while effective for finding bugs, does not produce informative error messages. We expect that by adding line numbers to the grammar productions that it records and altering slightly our policy-conformance algorithm to retain the list of line numbers from source to sink, our analyzer could produce helpful bug reports. Further, it would be interesting to explore how to enhance our static analyzer to suggest changes to

the application code that would fix the vulnerabilities.

In order to check for XSS vulnerabilities, we manually inspected several sources, including the source code of Firefox, in order to determine the set of strings that would invoke the JavaScript interpreter. The possibility remains that our manual inspection includes some mistakes, and even if our policy is correct, deriving it involved a significant effort. It would be interesting to design and implement an automated analysis of browser code to determine which input strings invoke the JavaScript interpreter. Such an analyzer could then be applied even to proprietary browsers by the owners of the source code to provide appropriate policies for our analyzer.

In our testing-based analysis, we have primarily considered a test oracle for SQL injection. In the future, we would like to explore other general test oracles. For example, in many web-based medical records systems, confidential information should not flow to certain classes of users. We are interested in exploring how multiple test executions can be compared to detect information leakage, and how inputs that will leak confidential information can be generated. The challenge will be in determining which predicates to target and in how new input values are selected.

6.3 Outlook

This dissertation focusses on significant security vulnerabilities in web applications, and by showing that those vulnerabilities can be understood, found, and prevented, it directs attention to related and emerging classes of vulnerabilities. Web applications continue to become more feature-rich and more dynamic, in particular with the advent of AJAX-style applications. AJAX (Asynchronous JavaScript And XML) describes a style of web application programming in which fragments of the webpage may load without requiring a reload of the whole page, thus allowing for a web application that feels more like a desktop application to the user. In AJAX-style applications, the client plays a much more active role, asynchronously and dynamically requesting JavaScript and HTML from a server and inte-

grating them and executing them in the current web page. For such applications, analysis designers will have to focus more on the client to identify and prevent XSS effectively.

In other settings, a more refined notion of malicious JavaScript will provide a definition of XSS better suited to the setting. In social networking web sites, for example, the site administrator may want to allow users to create personal pages with limited JavaScript, provided that the JavaScript stays within certain behavioral boundaries. Once again, appropriate analyses and enforcement mechanisms will need to be designed.

Some security researchers have drawn an analogy between XSS and buffer overflows. If we assume a true analogy exists, we must address analogous security problems in web applications to those in which buffer overflows play a role. For example, an informal dual to the problem of how to find and fix buffer overflows is the problem of how to make computing robust and reliable given the inevitability of buffer overflows. The work presented here helps to find and fix XSS vulnerabilities. However, it is interesting to consider how to prevent the malicious behavior from XSS exploits assuming XSS vulnerabilities will always exist. One instance of malicious behavior is JavaScript worms, in which an untrusted principal's low integrity JavaScript on the client writes to a user's high integrity data on the server. Given that this kind of malware spans tiers, it poses an interesting challenge for effective defense.

A central contribution of this dissertation is that web applications can be analyzed effectively as metaprograms. It would be interesting then to explore further analysis of metaprograms. Many program analysis and formal verification problems reduce to validity or satisfiability checking over some logical theories. Consequently, significant effort has been devoted to designing efficient decision procedures for these theories. Traditional program analysis problems address individual programs, so the decision procedures that underlie program analysis algorithms take a single constraint φ. Extending program analysis problems to address potentially infinite languages of programs (as generated by a metaprogram) requires decision procedures that take languages of constraints. We introduced the study of such decision procedures in another line of work [93]. One potential application of de-

cision procedures for language generators is the analysis of unnecessary clauses in queries. If a web application generates a query that has a tautological conditional clause, the database will either do extra work to determine that it can ignore that clause or it will not ignore the clause and process the query in a more expensive way than it otherwise could have done. Work on string analysis has shown how to construct a representation of the language of queries a web application will generate, and the conditional clauses of those queries can be extracted for input to such a decision procedure. The conceptual framework of decision procedures for language generators opens the door to many possible analyses of metaprograms.

Aside from the particular problems discussed above, it is the conceptual contribution of this dissertation that we hope has lasting influence. We have brought principled techniques to an important domain that had been largely overlooked by the research community. As the practice and trends in computing continue to develop and change, we hope that the work presented here will serve both as a useful foundation on which to build and as an instructive example of how principled techniques can prove effective in practical problem domains.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Chris Anley. Advanced SQL Injection in SQL Server Applications. An NGSSoftware Insight Security Research (NISR) publication, 2002. `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`.

[3] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, Oakland, CA, May 2008.

[4] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic Web sites. In *Eleventh International World Wide Web Conference (WWW 2002)*, 2002.

[5] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in C^{ω}. In *The 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 287–311, 2005.

[6] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of Applied Cryptography and Network Security*, pages 292–302, 2004.

[7] Claus Brabrand, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. PowerForms: Declarative Client-Side Form Field Validation. In *Proceedings of the 9th World Wide Web Conference*, pages 205–214. Kluwer Academic Publishers, 2000.

[8] Gregory T. Buehrer, Bruce W. Weide, and Paolo A.G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, pages 106–113, September 2005.

[9] TIOBE Software BV. TIOBE Programming Community Index, September 2007. `http://www.tiobe.com/tpci.htm`.

[10] Cristian Cadar and Dawson R. Engler. Execution Generated Test Cases: How to Make System Code Crash Itself. In *Model Checking Software, 12th International SPIN Workshop*, pages 2–23, 2005.

[11] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 322–335, 2006.

[12] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. `http://www.brics.dk/JSA/`.

[13] Steve Christey. Vulnerability Type Distributions in CVE, October 2006. `http://cwe.mitre.org/documents/vuln-trends.html`.

[14] William R. Cook and Siddhartha Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 97–106, 2005.

[15] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007 (SOSP 2007)*, pages 117–130, 2007.

[16] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software—Practice and Experience*, pages 1025–1050, 2004.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[18] Edsko de Vries, John Gilbert, and Paul Biggar. phc: The Open Source PHP Compiler. `http://www.phpcompiler.com`.

[19] Drew Dean and Dave Wagner. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 156–169, Oakland, CA, May 2001. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[20] Robert DeLine and Manuel Fähndrich. The Fugue Protocol Checker: Is Your Software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research, January 2004. `http://research.microsoft.com/~maf/Papers/tr-2004-07.pdf`.

[21] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102, 1970.

[22] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic Test Input Generation for Database Applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 151–162, 2007.

[23] Firefox web browser—Faster, more secure, & more customizable, 2008. `http://www.mozilla.com/en-US/firefox/`.

[24] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, 2002. ACM Press.

[25] Michael Furr and Jeffrey S. Foster. Checking Type Safety of Foreign Function Calls. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 62–72, 2005.

[26] Ariel Futoransky, Ezequiel Gutesman, and Ariel Waissbein. A Dynamic Technique for Enhancing The Security and Privacy of Web Applications. In *Black Hat USA Briefings*, 2007.

[27] Gecko—MDC, 2008. `http://developer.mozilla.org/en/Gecko`.

[28] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 645–654, May 2004.

[29] Mark Grechanik, William R. Cook, and Karl J. Lieberherr. Static Checking of Object-Oriented Polylingual Systems. Technical report, University of Texas, Austin, Computer Science Dept., March 2005.

[30] Jeremiah Grossman. Cross Site Scripting Worms and Viruses. Whitehat Security Whitepaper, 2007. `http://www.whitehatsec.com/home/resource/whitepapers/xxs.html`.

[31] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 117–127, 2006.

[32] William G. J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of 20th ACM International*

Conference on Automated Software Engineering (ASE), pages 174–183, November 2005.

[33] William G. J. Halfond and Alessandro Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2007)*, pages 145–154, Dubrovnik, Croatia, 2007.

[34] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, pages 175–185, Portland, Oregon, November 2006.

[35] William G. J. Halfond, Alessandro Orso, and Pete Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering (TSE)*, 34(1):65–81, 2008.

[36] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[37] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley, Boston, MA, 2000.

[38] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Typed XML Processing Language (Preliminary Report). In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 226–244, London, UK, 2001. Springer-Verlag.

[39] HTML 4.01 Specification, 1999. `http://www.w3.org/TR/html4/`.

[40] 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, NY, USA, 2004.

[41] Xiaoping Jia and Hongming Liu. Rigorous and Automatic Testing of Web Applications. In *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pages 280–285, 2002.

[42] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Scripting Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th International World Wide Web Conference*, pages 601–610, Banff, Alberta, Canada, 2007. ACM.

[43] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*, pages 258–263, Oakland, CA, May 2006.

[44] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 27–36, Ottowa, Canada, June 2006.

[45] John B. Kam and Jeffery D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[46] Kavado, Inc. InterDo Vers. 3.0, 2003.

[47] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 272–280, Washington D.C., USA, October 2003.

[48] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[49] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[50] Amit Klein. Blind XPath Injection. Whitepaper from Watchfire, 2005.

[51] Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, and Bruce Duba. Hygenic Macro Expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–159, 1986.

[52] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 359–372, 2002.

[53] Michal Kunc. What Do We Know About Language Equations? In *Developments in Language Theory, 11th International Conference (DLT 2007)*, pages 23–27, 2007.

[54] David Kung, Chien-Hung Liu, and Pei Hsia. An Object-Oriented Web Test Model for Testing Web Applications. In *24th International Computer Software and Applications Conference (COMPSAC 2000)*, pages 537–542, 2000.

[55] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-Sensitive Program Analysis as Database Queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–12, Baltimore, Maryland, June 2005.

[56] Yong Lei and James H. Andrews. Minimization of Randomized Unit Test Cases. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 267–276, 2005.

[57] Robert Lemos. Flawed USC Admissions Site Allowed Access to Applicant Data, July 2005. `http://www.securityfocus.com/news/11239`.

[58] J. Jenny Li, David Weiss, and Howell Yee. Code-Coverage Guided Prioritized Test Generation. *Information and Software Technology*, pages 1187–1198, 2006.

[59] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.

[60] Russell A. McClure and Ingolf H. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *27th International Conference on Software Engineering*, pages 88–96, 2005.

[61] Scott McPeak. Elsa: An Elkhound-Based C++ Parser, May 2005. `http://www.cs.berkeley.edu/~smcpeak/elkhound/`.

[62] Erik Meijer, Wolfram Schulte, and Galvin Bierman. Unifying Tables, Objects and Documents. In *Declarative Programming in the Context of OO Languages (DP-COOL)*, pages 139–156, 2003.

[63] David Melski and Thomas Reps. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, Amsterdam, The Netherlands, 1997.

[64] Yasuhiko Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW'05: Proceedings of the 14th International World Wide Web Conference*, pages 432–441, 2005.

[65] Mehryar Mohri and Mark-Jan Nederhof. Regular Approximation of Context-Free Grammars Through Transformation. *Robustness in Language and Speech Technology*, pages 153–163, 2001.

[66] Mehryar Mohri and Richard Sproat. An Efficient Compiler for Weighted Rewrite Rules. In *Meeting of the Association for Computational Linguistics*, pages 231–238, 1996.

[67] Mozilla.org—Home of the Mozilla Project, 2008. `http://www.mozilla.org/`.

[68] Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[69] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, 2007.

[70] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Twentieth IFIP International Information Security Conference (SEC'05)*, pages 124–145, 2005.

[71] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *Object-Oriented Programming, 19th European Conference (ECOOP 2005)*, pages 504–527, 2005.

[72] PHProjekt :: an open source groupware suite. `http://www.phprojekt.com/`.

[73] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[74] Wojciech Plandowski. Satisfiability of Word Equations with Constants is in PSPACE. In *40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, pages 495–500, 1999.

[75] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 61–74, Seattle, Washington, 2006. USENIX Association.

[76] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, San Francisco, CA, 1995. ACM.

[77] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 25–34, 2001.

[78] RSnake. XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`.

[79] Sanctum Inc. AppShield 4.0 Whitepaper., 2002. `http://www.sanctuminc.com`.

[80] David Scott and Richard Sharp. Abstracting Application-Level Web Security. In *Proceedings of the 11th International World Wide Web Conference*, pages 396–407, Honolulu, Hawaii, USA, 2002.

[81] David Scott and Richard Sharp. Specifying and Enforcing Application-Level Web Security Policies. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):771–783, 2003.

[82] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification, 18th International Conference (CAV 2006)*, pages 419–423, 2006. (Tool Paper).

[83] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005)*, pages 263–272, 2005.

[84] Ofer Shezaf. The Web Hacking Incidents Database Annual Report 2007. Breach Security Whitepaper, 2007. `https://bsn.breach.com/downloads/whid/The%20Web%20Hacking%20Incidents%20Database%20Annual%20Report%202007.pdf`.

[85] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages*, pages 372–382, Charleston, SC, January 2006. ACM Press New York, NY, USA.

[86] Michael Sutton. How Prevalent Are SQL Injection Vulnerabilities?, September 2006. `http://portal.spidynamics.com/blogs/msutton/archive/2006/09/26/How-Prevalent-Are-SQL-Injection-Vulnerabilities_3F00_.aspx`.

[87] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular Expression Types for Strings in a Text Processing Language (Extended Abstract). In *Proceedings of TIP'02 Workshop on Types in Programming*, pages 1–18, July 2002.

[88] Walid Taha and Tim Sheard. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

[89] Peter Thiemann. Grammar-Based Analysis of String Expressions. In *2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 59–70, 2005.

[90] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting

and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 37–48, San Diego, CA, February 2007.

[91] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl (3rd Edition)*. O'Reilly, 2000.

[92] Gary Wassermann and Zhendong Su. An Analysis Framework for Security in Web Applications. In *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, pages 70–78, 2004.

[93] Gary Wassermann and Zhendong Su. Validity Checking for Finite Automata over Linear Arithmetic. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 405–416, Kolkata, India, December 2006. Springer-Verlag.

[94] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, San Diego, CA, June 2007. ACM Press New York, NY, USA.

[95] Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, pages 171–180, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.

[96] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic Test Input Generation for Web Applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 249–260, Seattle, WA, July 2008. ACM Press New York, NY, USA.

[97] Daniel Weise and Roger Crew. Programmable Syntax Macros. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165, Albuquerque, New Mexico, United States, 1993.

[98] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, Washington DC, USA, 2004. ACM Press.

[99] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium*, pages 179–192, July 2006.

[100] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, Vancouver, BC, Canada, August 2006.

[101] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–249, Nice, France, 2007. ACM.