The Dissertation Committee for Sooel Son
certifies that this is the approved version of the following dissertation:

Toward Better Server-side Web Security

Committee:

Vitaly Shmatikov, Supervisor

Kathryn S. McKinley, Supervisor

Don Batory

Miryung Kim

V.N. Venkatakrishnan

Toward Better Server-side Web Security

by

Sooel Son, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Acknowledgments

I deeply appreciate Vitaly Shmatikov and Kathryn S. McKinley for their invaluable advice and support. Without their encouragement and support, I cannot imagine myself writing this dissertation.

V.N. Venkatakrishnan has inspired our work and gave invaluable feedback for us to improve the last piece of this dissertation. I am deeply grateful for help and personal support provided by Daehyeok Kim.

I thank my family for their love and personal support which they have provided throughout my doctoral studies. I also appreciate my wife, Kayoung Lee, for tolerating me. She not only motivated me to continue my research but encouraged me with kind words and love.

Toward Better Server-side Web Security

Publication No. _____

Sooel Son, Ph.D.
The University of Texas at Austin, 2014

Supervisors: Vitaly Shmatikov
Kathryn S. McKinley

Server-side Web applications are constantly exposed to new threats as new technologies emerge. For instance, forced browsing attacks exploit incomplete access-control enforcement to perform security-sensitive operations (such as database writes without proper permission) by invoking unintended program entry points. SQL command injection attacks (SQLCIA) have evolved into NoSQL command injection attacks targeting the increasingly popular NoSQL databases. They may expose internal data, bypass authentication or violate security and privacy properties. Preventing such Web attacks demands defensive programming techniques that require repetitive and error-prone manual coding and auditing.

This dissertation presents three methods for improving the security of server-side Web applications against forced browsing and SQL/NoSQL command injection attacks. The first method finds incomplete access-control enforcement. It statically identifies access-control logic that mediates security-sensitive operations and finds missing access-control checks without an *a priori*

specification of an access-control policy. Second, we design, implement and evaluate a static analysis and program transformation tool that finds access-control errors of omission and produces candidate repairs. Our third method dynamically identifies SQL/NoSQL command injection attacks. It computes shadow values for tracking user-injected values and then parses a shadow value along with the original database query in tandem with its shadow value to identify whether user-injected parts serve as code.

Remediating Web vulnerabilities and blocking Web attacks are essential for improving Web application security. Automated security tools help developers remediate Web vulnerabilities and block Web attacks while minimizing error-prone human factors. This dissertation describes automated tools implementing the proposed ideas and explores their applications to real-world server-side Web applications. Automated security tools are effective for identifying server-side Web application security holes and a promising direction toward better server-side Web security.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Web Threats

Web applications continue to gain wide adoption. Over three-fourths of the population in North America surfs the Internet [24]. Web applications, such as e-commerce, blogging, and media software, typically consist of *client-side* programs running in a Web browser as well as a *server-side* program that (1) converts clients' requests into queries to a backend database and (2) returns HTML content after processing user input and retrieved database records. Thus, server-side Web applications work with user input from the untrusted public on the Internet. As Web applications become more prevalent, security of server-side Web applications is becoming a major concern for many service providers.

Unfortunately, Web applications have been targeted by many Web attacks such as cross-site scripting (XSS), cross-site request forgery (CSRF), forced browsing, server-side includes injection (SSI), SQL/NoSQL injection and execution after redirection (EAR) [8, 16, 20, 56, 57, 78]. Server-side Web attacks are often devastating. For example, SQL injection attacks on retail stores owned by TJX Companies compromised more than 45 million credit and

debit numbers in 2005–2007 [78]. In 2012, SQL injection attacks also exposed 450,000 Yahoo! passwords [57]. Furthermore, a hacker group who conducted SQL injection attacks on several banks stole 160 million credit card numbers, causing more than 200 million dollars in losses [56]. In 2010, a CSRF attack that exploited a known vulnerability in an open-source Web application compromised Rackspace databases [16]. Client-side Web applications are also not free from Web attacks. In 2010, Twitter was overrun with spam posts due to client-side XSS attacks [60]. Thus, the presence of adversaries who exploit Web application vulnerabilities is inarguable.

This dissertation focuses on server-side Web threats under a common adversary model. We assume that the adversary is capable of (1) tampering with user input values and (2) examining code of a victim Web application. Recent research has confirmed such adversaries in the wild. Canali and Balzarotti showed that a honeypot Web site that contains known SSI and SQL injection vulnerabilities was attacked only 5 hours after its first deployment [8]. To protect sites from such adversaries, the Open Web Application Security Project (OWASP) is equipping developers with defensive coding techniques [49]. However, WhiteHat security reported that an average industry Web site has 79 vulnerabilities with 231 exposure days on average [20]. These statistics manifest that today's Web security is fragile.

This dissertation focuses on removing the causes of two attack vectors: *forced browsing* and *SQL/NoSQL command injection*. Forced browsing attacks take advantage of incomplete access-control enforcement. For instance,

unauthorized users execute privileged database operations by exploiting an access-control vulnerability. It is a vulnerability ranked as the fourth most likely to appear in an average industry Web site [20]. Automatically finding omitted or incorrect access-control logic is challenging. What complicates an automated approach is the absence of either a universal standard pattern or a predefined syntactic template for access-control logic in server-side Web applications. Each Web application implements its own particular logic.

Therefore, many researchers have examined the following options: take advantage of an external annotation that indicates access-control logic [7, 74], extract benign access-control traces from dynamic program executions [12, 15, 19, 23, 82] or exploit characteristic syntax [17, 31, 55, 65, 71]. However, large legacy applications are often not accompanied by a specification, which motivates our research on the following question: How do we find an access-control bug when we do not know what the access-control logic is?

Remediating identified access-control bugs is also challenging. Dynamic remediation of access-control bugs only fixes the symptom by enforcing predefined security policies. However, it does not repair the bugs present in Web applications. Static repair of access-control bugs at the source-code level has several challenges such as identifying locations to update, conducting interprocedural program transformation, and validating repaired code. No previous approach has attempted to automatically repair identified access-control bugs at the source-code level.

SQL command injection is a notorious attack vector that occurs when

untrusted user input is executed as code parts of a SQL query. Because successful SQL command injection attacks result in executing arbitrary queries, the adversary may steal, tamper with or remove sensitive data in backend databases. NoSQL injection attacks are a new class of emerging threats that aim to inject code into a NoSQL database. NoSQL and SQL injection attacks share a common factor—a vulnerable server-side Web application allows an adversary to inject malicious input which is interpreted by the database as code instead of string or numeric constants. Database queries generated by server-side Web applications are a classic target of code injection, as previously described. Since 2003, SQL injection attack has remained in the top 10 list of CVE vulnerabilities [13]. Furthermore, the 2012 WhiteHat security report states that SQL injection attacks are still the eighth most prevalent attack type [20].

Both static detection and dynamic prevention of code injection attacks are challenging. Static methods start by defining *sanitization* and *sink* methods in their own domains. Sink methods refer to certain methods that output or send application-generated values to other domains. Database methods of invoking SQL queries and HTML output methods are classic sink methods. Sanitization methods change or remove from input values undesirable parts that may cause code injection attacks. Many static detection methods focus on finding unsanitized data flows from input to sensitive sinks [26, 34, 45, 67]. Since incorrectly sanitized input can still cause an injection attack, soundness depends on how precisely the sanitization logic is modeled. However, it is not

trivial to precisely model every operation that involves generating tainted data flows. The proposed static methods conduct coarse-grained taint analyses that decide whether an application-generated query is tainted or not [26, 34, 45, 67]. On the other hand, several dynamic methods identify which parts of a generated query come from user input [22, 40, 45, 54, 81]. Identifying tainted parts of a generated query at the character level requires sophisticated and costly taint-tracking. Because of this, several studies have proposed different approaches, agnostic to the precision of a taint-tracking algorithm. Sekar et al. infer tainted parts by measuring the similarity between the generated query and user input [64]. Bandhakavi et al. perform mirror executions to build a benign candidate query and compare it with the actual query [3]. However, all these approaches are subject to false positives and negatives. No prior work directly considers NoSQL.

Figure 1.1 illustrates the different causes of SQL/NoSQL injection and forced browsing attacks. Forced browsing attacks take advantage of *control-flow* vulnerabilities that allow the adversary to bypass access-control logic. SQL/NoSQL injection attacks exploit *data-flow* vulnerabilities that cause illegitimate data flows from user input to database operations. Identifying or repairing both vulnerabilities involves tedious auditing and coding that often brings error-prone human factors. Furthermore, individual proficiency in implementing secure Web applications varies with developers' expertise as Web security becomes complicated. These problems motivate the creation of an automated tool that can ease developers' burden and improve the robustness

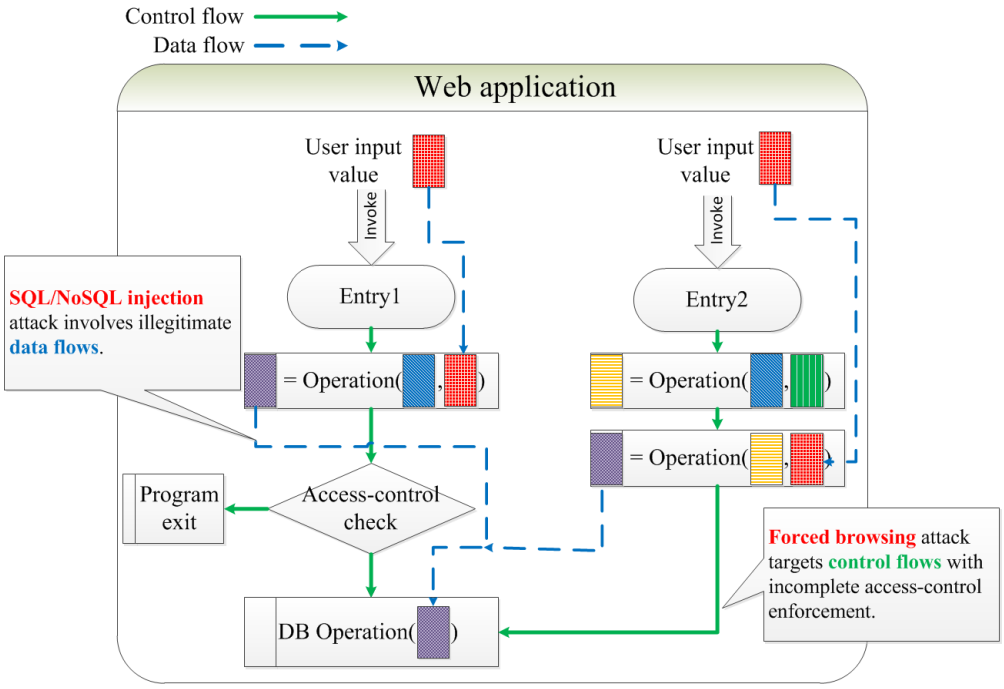

Figure 1.1: The attack model of forced browsing and SQL/NoSQL command injection attacks.

of Web applications. Automated bug-finding and attack-blocking tools are promising ways for better server-side Web security.

1.2 Better Server-side Web Security

Forced browsing and SQL/NoSQL injection attacks stem from access-control bugs and illegitimate data flows in server-side Web applications, respectively. Client-side remediation of server-side application vulnerabilities is infeasible because in design clients in the Internet are incapable of changing server-side application code. Blocking suspicious requests at the client side

is also impracticable because it is hard to manage and enforce client-side defenses for many clients. This dissertation presents *server-side* security tools that directly remediate access-control bugs and identify illegitimate data flows for forced browsing and SQL/NoSQL injection attacks.

To identify access-control bugs, we present a static tool called RoleCast, which finds missing access-control checks without *a priori* knowledge of which methods or variables implement access-control checks. Each Web application implements access-control checks in a different, often idiosyncratic way. Even within the application, access-control checks varies based on the users *role*, e.g., regular users versus administrators. RoleCast infers such application- and role-specific access-control checks by exploiting common software engineering patterns —the code that implements distinct user role functionality and its security logic typically resides in distinct methods and files. It then finds missing access-control checks before security-sensitive operations within a *role*. RoleCast partitions the set of file contexts (a coarsening of calling contexts) on which security-sensitive operations are control dependent into roles. For each role, RoleCast identifies critical variables that affect the reachability of security-sensitive operations. It then applies role-specific variable consistency analysis to find missing access-control checks. RoleCast found 13 access-control bugs with 3 false positives in 11 real-world Web applications.

RoleCast only notifies developers of its findings. It makes no attempt to repair source code. Unfortunately, fixing access-control bugs is a tedious

and repetitive task that involves replicating existing access-control policies. To help developers fix these bugs, we designed and implemented FixMeUp, a static program transformation tool that makes repairs at the source-code level. FixMeUp starts by obtaining role-specific access-control policy examples from manual or inferred annotations. From these annotations, FixMeUp then computes an access-control template (ACT). Next, it replicates a template in vulnerable contexts that implement incorrect access-control policies. Finally, FixMeUp validates its repair and suggests the repaired code to developers. Our experiments show that FixMeUp correctly repaired 30 access-control bugs with one warning in 10 Web applications.

The final Web threat that this dissertation addresses is code injection. We present a dynamic SQL/NoSQL injection detection tool, Diglossia. A SQL/NoSQL injection attack occurs when user input is interpreted as code in a database query generated by a Web application. Diglossia dynamically blocks database queries that contains user-injected code.

The challenges in detecting injected code are (1) recognizing code in the generated query and (2) determining which parts of the query are tainted by user input. To identify tainted characters, Diglossia dynamically maps all application-generated characters to shadow characters that do not occur in user input and computes shadow values for all input-dependent strings. Any original characters in a shadow value are exactly the taint from user input. To detect injected code in a generated query, Diglossia parses the query in tandem with its shadow and checks that (1) the two parse trees are syntactically

isomorphic and (2) all code in the shadow query is in shadow characters and, therefore, originated from the application itself, as opposed to user input. We evaluated DIGLOSSIA on 10 PHP Web applications. DIGLOSSIA accurately detected 25 SQL and NoSQL injection attacks while avoiding the false positives and false negatives of prior methods [3, 22, 45, 72, 81].

1.3 Contributions and Impacts

The dissertation presents three novel ideas to remediate forced browsing vulnerabilities and block SQL/NoSQL injection attacks. We believe that programmers, server administrators, and bug-finding tool developers can benefit from our implementations as well as the ideas behind them.

For the first contribution, we demonstrate the effectiveness of ROLECAST in finding access-control vulnerabilities, the root cause of forced browsing attacks. ROLECAST uses a heuristic algorithm that takes advantage of the software engineering patterns to precisely infer application- and role-specific access-control logic.

Furthermore, we demonstrate an automated static technique that generates candidates that repair access-control vulnerabilities. Instead of inserting one- or two-line patches, our FIXMEUP tool aggressively creates interprocedural repairs and suggests them to developers. To the best of our knowledge, FIXMEUP is the first program repair tool that reuses pre-existing statements in a target context. FIXMEUP helps programmers improve code productivity as well as enforce consistent role-specific access-control logic. Because the

repair algorithms are agnostic to vulnerability types, FixMeUp is also applicable for repairing other access-control vulnerabilities such as cross-site request forgery [84].

The last technique, Diglossia, is based on new ideas for identifying user-injected code and avoids the flaws of prior detection methods that use syntactic structures and regular expression filters. Diglossia takes advantage of dual parsing and value shadowing to determine whether user-injected string values are interpreted as code in computed SQL/NoSQL queries. By consolidating dual parsing and value shadowing, we gain performance improvement and better precision while avoiding the false positives and false negatives of prior tools. Diglossia does not require any changes in either legacy applications or database environment. It only requires installing a PHP plug-in.

Today's Internet is a dangerous place, with serious and subtle Web attacks that demand security expertise from defenders. Many developers need help identifying application bugs and evaluating the security of their services as Web security becomes more complex and applications become larger. We demonstrate the effectiveness of the automated tool-guided approaches for remediating causes of Web attacks.

Chapter 2

Web Attacks

This chapter presents the background on server-side Web applications and NoSQL databases, which are the major subjects of this thesis. It describes an adversary model and presents two security properties that prevent server-side Web application vulnerabilities. This chapter also explains access-control policies in server-side Web applications and characterizes various major Web attack vectors, including forced browsing and SQL/NoSQL injection attacks. Finally, in Section 2.6, it describes and compares previous work that addressed these vulnerabilities.

2.1 Server-side Web Applications

Web applications are classified by their deployment scenario as either server-side or client-side applications. A Web browser runs *client-side* Web applications fetched from Web servers. They are typically coded in JavaScript, ActionScript or Java. *Server-side* Web applications run at Web servers. As Figure 2.1 shows in detail, a script interpreter, a module of a Web server, accepts clients' input values delivered by HTTP requests as well as cookies, and then invokes the user's choice of Web application with the input values.

The invoked Web application transforms the input values into SQL/NoSQL queries to perform backend-database operations and builds an HTML page, which is returned to a client. Therefore, it is an intrinsic feature of server-side Web applications to interact with untrusted user input values. Among the many Web programming languages, our tools analyze Web applications implemented in the PHP and JSP programming languages.

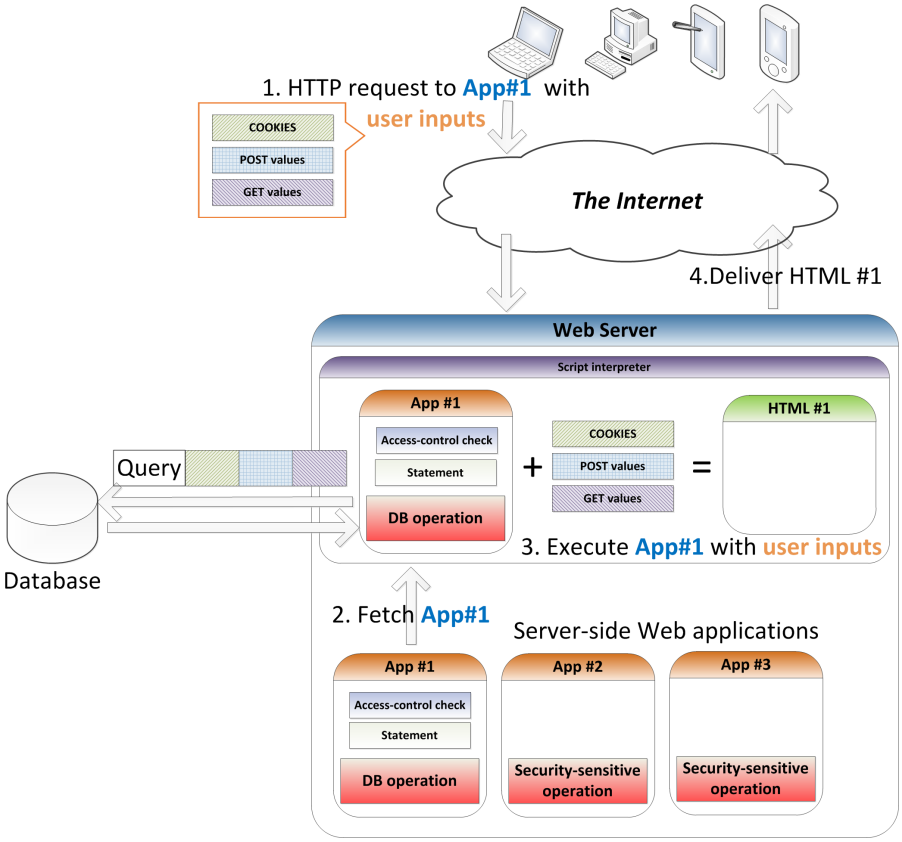

Figure 2.1: An overview of server-side Web applications

2.1.1 PHP and JSP

The PHP (**P**HP: **H**ypertext **P**reprocessor) scripting language is designed for dynamically generating Web pages [52]. PHP is commonly used to implement Web applications with user-generated content or content stored in backend databases (as opposed to static HTML pages). A recent survey of 12 million domains found that 59% use PHP to generate HTML content [53].

PHP borrows syntax from Perl and C. In PHP programs, executable statements responsible for generating content are mixed with XML and HTML tags. PHP provides basic data types, a dynamic typing system, rich support for string operations, some object-oriented features, and automatic memory management with garbage collection. Instead of a module or class system, PHP programs use a flat file structure with a designated main entry point. Consequently, (1) a network user can directly invoke *any* PHP file by providing its name as part of the URL, and (2) if the file contains executable code outside of function definitions, this code will be executed. These two features of PHP require defensive programming of each entry point and are a source of forced-browsing attacks.

JSP (**J**ava **S**erver **P**ages) is a Java technology for dynamically generating HTML pages [29]. It adds scripting support to Java and mixes Java statements with XML and HTML tags. Scripting features include libraries of page templates and an expression language. JSP supports dynamic types, but because it builds on Java, it has more object-oriented features than PHP. JSP executes in a Java Virtual Machine (JVM).

2.1.2 NoSQL and MongoDB

NoSQL is a new, more flexible approach for large data stores, and an alternative to relational DBMSs using Structured Query Language (SQL). NoSQL stores data in key and value pairs, and their implementations target for distribution and scalability [46]. Many NoSQL families including DynamoDB, MongoDB, CouchDB, Cassandra and others are growing in popularity.

Both SQL and NoSQL database vendors provide server-side Web application APIs for searching and managing stored data. Because the main objective of server-side Web applications is to dynamically generate Web pages by retrieving persistent database data and user input, the interactions between an application and a database are carried out through such APIs

We chose to analyze MongoDB NoSQL database to demonstrate the effectiveness of DIGLOSSIA. MongoDB is an open-source document-oriented NoSQL database [39]. MongoDB has been gaining wide adoption among corporations, including Foursquare and Craigslist [38]. MongoDB provides an API (MongoDB PHP driver) for a PHP application to interact with a MongoDB database. MongoDB supports two query languages: JavaScript and JSON. JavaScript is a dynamic weakly-typed scripting language heavily used in many HTML Web pages. JSON is a data structure of consisting of one or more key-value pairs [28]. The MongoDB PHP driver uses a string or array type value in PHP to represent respectively a JavaScript or JSON query in MongoDB.

mongodbadmin.php

```
1  <?
2  ...
3  if (!$document) {
4    $document = findMongoDbDocument($_REQUEST['search'], $_REQUEST['db'],
         $_REQUEST['collection'], true);
5    $customId = true;
6  }
7  ...
8  function findMongoDbDocument($id, $db, $collection, $forceCustomId = false)
9  {
10   ...
11   $collection = $mongo->selectDB($db)->selectCollection($collection);
12   ...
13   $document =$collection->findOne(array('_id' => $id));
14   ...
15 }
16 ...
17 ?>
```

Figure 2.2: Example of building a JSON type query

Figure 2.2 shows a PHP code instance that sends a JSON query to a MongoDB database. The code in line 13 of *mongodbadmin.php* builds an array value where the key is "_id" and its corresponding value is user input obtained by referencing $\$\_REQUEST[`search'].$ The MongoDB PHP driver then transforms the array value into a JSON query and sends it to a MongoDB.

2.2 Adversary Model

This section describes the adversary model that we assume throughout this dissertation. We make the standard assumption that the adversary has the following capabilities:

1. The adversary may examine server-side source code deployed at a victim server. However, the adversary cannot modify such code.

2. The adversary may send a victim server forged HTTP requests. The adversary can also entice or trick honest users into visiting a malicious site and send—via a spam message, advertising and so on—a forged HTTP request on the visitor's behalf.

Both assumptions are realistic because many Web servers use open source Web applications. Common behaviors for Internet surfers include clicking on advertisements and visiting untrusted Web pages. The adversary only takes advantages of application bugs by tampering with input parameters. The adversary does not have the capabilities to inspect logs or network data of a victim server.

2.3 Security Properties

Security-aware developers generally seek to establish the following two security properties:

1. Only authorized users should be able to access a security-sensitive operation.
2. Any untrusted input should not serve as executable code.

A security-sensitive operation is a privileged procedure that the author of an application wants to protect from untrusted users. Security-sensitive operations include updating database records, deleting disk files or letting a Web server send emails. Therefore, server-side Web applications should vet with correct access-control logic whether an incoming request leading to the

execution of sensitive operations has the proper privilege. The first security property restricts reachable control flows for security-sensitive operations.

The violation of the second property directly leads to a code injection attack that destroys the intended purpose of an application. To enforce the second property, developers usually confine user input to serve as string and numeric constants in database queries. A code injection attack occurs when an adversary injects code into an application by exploiting incorrect confinement of user input. Generally, code injection attacks can be characterized as illegitimate data-flow attacks from user input to sensitive sink methods that execute application-generated commands.

2.4 Access-control Policies

By design, server-side Web applications interact with untrusted users and receive untrusted network inputs. Therefore, they must not perform security-sensitive operations, unless users hold the proper permissions. This mechanism is known as *security mediation*. In Web applications, security mediation is typically implemented by *access-control policies*.

In general, an *access-control policy* requires some *checks* prior to executing *security-sensitive operations*. These access-control checks vary with applications and user roles even within the same application. For example, an online store may have customers and administrators, while a blogging site may have blog owners, publishers, and commenters. Thus, different calling contexts associated with different user roles often require different checks.

Add.php

```
1  <? ...
2  session_start();
3  dbConnect();
4  if (!verifyuser() ) { // access-control check
5    header( "Location: ./login.php" );
6    exit;
7  }
8  ...     // security-sensitive operation
9  $sql = "INSERT INTO blogdata SET user_id='$id', subject='$subject',
       message='$message', datetime='$datetime'";
10 $query = mysql_query($sql);
11 ...
12 function verifyuser() {
13     ....
14     session_start();
15     global $user, $pwd;
16     if (isset($_SESSION['user']) && isset($_SESSION['pwd'])) {
17        $user = $_SESSION['user'] ;
18        $pwd  = $_SESSION['pwd'] ;
19        $result = mysql_query( "SELECT user, password FROM blogusername
              WHERE user='$user' AND BINARY password='$pwd'" );
20        if( mysql_num_rows( $result ) == 1 )
21           return true;
22     }
23     return false;
24 }
25 ?>
```

Figure 2.3: *minibloggie*: Access-control check

Figures 2.3 and 2.4 show examples of application-specific access-control checks in real-world PHP applications. Figure 2.3 shows a correct check (line 4) in *Add.php* from *minibloggie*. *Add.php* invokes a dedicated *verifyuser* function that queries the user database with the username and password. If verification fails, the application returns the user to the login page. Figure 2.4 shows a different access-control check (line 3) performed by *AcceptBid.php* in the DNscript application. It reads the hash table containing the session state and checks the *member* flag. Both access-control policies protect the same operation—a *mysql_query* call site that updates the backend database—but with very different logic.

AcceptBid.php

```
1  <?
2  session_start();
3  if (!$_SESSION['member']) { // access-control check for member
4      header('Location: login.php');
5      exit;
6  }
7  include 'inc/config.php';
8  include 'inc/conn.php';
9  ...              // security-sensitive operation
10 $q5 = mysql_query("INSERT INTO close_bid(item_name, seller_name,
       bidder_name, close_price) ".$sql5);
11 ...
12 ?>
```

DelCb.php

```
1  <? // No access-control check
2  include 'inc/config.php';
3  include 'inc/conn.php';
4  // security-sensitive operation
5  $delete = mysql_query("DELETE FROM close_bid where item_name = '".
       $item_name."'");
6  if($delete) {
7     mysql_close($conn);
8     ...
9  }
10 ?>
```

Del.php for the "administrator" role

```
1  <?
2  session_start();
3  if ($_SESSION['admin'] != 1) { // access-control check for administrator
4      header('Location: login.php');
5      exit;
6  }
7  include 'inc/config.php';
8  include 'inc/conn.php';
9  ...              // security-sensitive operation
10 $sql = mysql_query("DELETE FROM domain_list WHERE dn_name = '".$dn_name."'
       ");
11 ...
12 ?>
```

Figure 2.4: DNscript: Correct access-control check in AcceptBid.php and a missing check in DelCb.php

The access-control checks are role-specific. For example, the DNscript application has two roles. *AcceptBid.php* in Figure 2.4 shows the check (line 3) for the "regular user" role and Line 3 of *Del.php* shows the different check for the "administrator" role.

2.5 Web Attack Classification

The Open Web Application Security Project (OWASP) is warning developers about various Web attacks, including cross-site scripting (XSS), cross-site request forgery (CSRF), forced browsing, server-side include injection (SSI), SQL/NoSQL code injection and execution after redirection (EAR) [49]. This section examines forced browsing and SQL/NoSQL injection attacks in relation to other major Web attacks.

Cause	Application Type	
	Server-side	Client-side
A violation of security property #1 (Incorrect access-control logic)	**Forced browsing** Cross-site request forgery (CSRF) Execution after redirection (EAR)	
A violation of security property #2 (Illegitimate data flows)	**SQL/NoSQL command injection** Reflected cross-site script injection (XSS) Server-side include injection (SSI) HTTP response splitting	Client-side XSS (CXSS)
Application logic error	Parameter tampering	Browser or plugin exploits

Table 2.1: Web attack classification

Table 2.1 arranges Web attacks into groups based on the cause and the target application type. The cause of forced browsing, CSRF and EAR attacks lies in absent or incorrect access-control logic. However, they exploit different bugs in access-control logic. Forced browsing attacks take advantage of omitted or incorrect access-control logic. EAR harnesses a common mistake

when the programmer does not place a program termination call after a page redirection call. A CSRF attack exploits the absence of checks to validate origins of HTTP requests.

Reflected XSS, server-side includes injection (SSI), HTTP response splitting and SQL/NoSQL injections attacks share the same cause, illegitimate data flows, but target different method calls. The reflected XSS injects malicious input into HTML output methods. The HTTP response splitting attack injects forged HTTP headers into a header output method. The SSI and SQL/NoSQL injections attacks inject code into file inclusion and database operation functions, respectively.

Whereas reflected XSS causes a vulnerable server-side Web application to generate a Web page with malicious payloads, client-side XSS attacks take advantage of bugs in client-side JavaScript code. They inject forged script code into *eval*, *setTimeout*, *postMessage* or client-side JavaScript code that dynamically generates HTML Document Object Model(DOM) objects [63, 68].

Parameter-tampering attacks usually refer to a general attack that exploits a server-side application logic bug with forged input parameters. For instance, Bisht et al. identify server-side application logic bugs that do not validate input parameters, but perform validations in client-side JavaScript code [5].

Among the Web attacks listed above, this dissertation focuses on forced browsing and SQL/NoSQL injection attacks. The following sections describe

in detail the two attacks as well as examples of vulnerabilities.

2.5.1 Forced Browsing Attack

Web applications typically perform *access-control checks* to protect *security-sensitive operations*. Thus, the absence of such checks directly leads to successful forced browsing attacks that enable an adversary to gain unauthorized access to security-sensitive operations. It is a clear violation of the first security property described in Section 2.3.

Since every file included in a PHP application can be used as an alternative entry, developers must replicate access-control logic on every path that accesses security-sensitive operations. In particular, even if the code contained in some file is intended to be called only from other files, it must still be programmed defensively because an attacker may invoke it directly via its URL. Thus, access-control bugs are control-flow vulnerabilities. They enable an attacker to execute a sensitive operation, which may or may not be accompanied by illegitimate data injection.

Figure 2.4 shows an instance of code that is vulnerable to forced browsing attacks. The check on $\$\_SESSION$ at line 3 is present in *AcceptBid.php*, but missing in *DelCb.php*. The access-control check protects the sensitive operation—a *mysql_query* call site that updates the backend database. A malicious user can directly invoke *DelCb.php* by supplying its URL to the Web server and execute a DELETE query because *DelCb.php* is missing a check on the $\$\_SESSION$ variable.

Finding and repairing access-control vulnerabilities require manual, repetitive audits of source code. In complex applications, the number of unintended entry points can be very large, which greatly complicates manual inspection and motivates the need for automated analysis. As Section 2.4 describes, access-control logic has no standard pattern and differs even within the same application. Therefore, enforcing a single access-control pattern across the entire program may subvert the intended semantics.

login.php in *minibill*

```
1 <?
2 $Q = "SELECT * FROM users
3       WHERE email='{$_REQUEST['email']}'
4       AND password='{$_REQUEST['password']}'
5       LIMIT 1";
6 $res = mysql_query($Q);
7 ?>
```

Attack URL

http://victimHost/login.php?email=any'; OR 1=1 –'&password=any

Actual query

SELECT * FROM users WHERE email='any' OR 1=1 - -'' AND password='any' LIMIT 1

Figure 2.5: SQL injection attack on *minibill*.

2.5.2 SQL Injection Attack

SQL injection attack exploits incorrect or absent sanitization logic on user input, which may lead to arbitrary code execution. For instance, Figure 2.5 shows a PHP code instance that retrieves database records matching a given email and password. However, the application does not sanitize user input obtained from $\$\_REQUEST[\text{`}email\text{'}]$ and $\$\_REQUEST[\text{`}password\text{'}]$. This mistake allows the adversary to escape quotes that supposedly confine

user input to string literals. The injected *"OR 1=1"* string makes the resulting query to become the tautology and the following *"–"* string comments out the remaining query. Thus, the resulting query returns all stored records, which is a violation of the second security property.

2.5.3 NoSQL Injection Attack

NoSQL databases are as vulnerable to code injection attacks as SQL databases. For example, we found four PHP MongoDB-based applications in GitHub with injection vulnerabilities (see Table 5.2).

mongodbadmin.php

```
1  <?
2  ...
3  if (!$document) {
4    $document = findMongoDbDocument($_REQUEST['search'], $_REQUEST['db'],
         $_REQUEST['collection'], true);
5    $customId = true;
6  }
7  ...
8  function findMongoDbDocument($id, $db, $collection, $forceCustomId = false)
9  {
10   ...
11   $collection = $mongo->selectDB($db)->selectCollection($collection);
12   ...
13   $document =$collection->findOne(array('_id' => $id));
14   ...
15 }
16 ...
17 ?>
```

Attack URL

http://victimHost/mongodbadmin.php?search[$ne]=1&db=test&collection=test

Figure 2.6: JSON injection vulnerability.

Figure 2.6 shows a PHP application with a JSON injection vulnerability. Line 13 of *mongodbadmin.php* in Figure 2.6 builds an array consisting of a

single key-value pair, where the key is "_id" and the value is equal to the user input obtained from $\$\_REQUEST[`search']$. The Mongo API transforms this array into a JSON query and sends it to MongoDB. The intention is to return all database items whose *_id* field is equal to the user-supplied value.

A malicious user, however, can set the *search* variable to be an array value, $array(\$ne => 1)$. In the resulting JSON query, line 13 of Figure 2.6 no longer compares *_id* for equality with $\$id$, but instead interprets the first element of $\$id$ as a function, $(\$ne)$, the second element, (1), as the argument to this function, and returns all database items whose *_id* is not equal to 1. In this case, user input is supposed to be a string constant, but instead symbols $\$ne$ are interpreted as code in the query.

Figure 2.7 shows another vulnerable PHP application. Lines 3 to 18 build a query string from user input, Line 21 sends the resulting JavaScript program to MongoDB. MongoDB evaluates this program on every key-value pair in the database and returns the pairs on which the program evaluates to "true". The query is supposed to retrieve data whose privilege keys are the same as *userType*. The malicious URL, however, tricks the application into generating a tautology query that always returns "true". Note that user-injected symbols ;, *return*, }, and // are parsed into code in the JavaScript query:

$$function\ q()\{\ var\ default\_user = `normal';$$
$$var\ admin\_passwd = `guessme';$$
$$var\ userType = \underline{1;\ return\ true;\}//}....$$

vulfquery.php

```
<?
// Build a JavaScript function query from user input
$query_body = "
  function q() {
    var default_user = 'normal';
    var admin_passwd = 'guessme';
    var userType = " . $_GET['user'] . ";
    var userPwd = " . $_GET['passwd'] . ";
    if(userType == '_admin_' && userPwd == admin_passwd)
      userType = 'admin';
    else
      userType = 'normal';

    if( this.showprivilege == userType )
      return true;
    else
      return false;
  }";
....
// Initiate a function query
$result = $collection->find( array('$where' => $query_body) );
?>
```

Attack URL

http://victimHost/vulfquery.php?user=1;return true;}//

Figure 2.7: JavaScript injection vulnerability.

2.6 Related Work

Researchers have suggested many ideas to prevent an adversary from abusing Web application bugs. Against each Web threat described in Section 2.5, Table 2.2 groups the proposed studies according to two criteria: (1) client or server side where the defense is intended to be deployed and (2) the analysis type. Static methods find or repair vulnerabilities before Web applications are deployed. Dynamic methods identify and then block Web attacks at runtime. Hybrid methods use both static and dynamic information.

Cause	Web attacks	Server-sides Static	Server-sides Dynamic	Server-sides Hybrid	Client-sides Dynamic
Access-control bugs	Forced browsing	[2, 66, 74]	[7, 15, 23]	[21]	
	CSRF			[21, 84]	[4, 62]
	EAR	[17]			
Illegitimate data flows	Client-side XSS	[63]			[27, 37]
	Reflected XSS	[26, 34, 67]	[45, 54, 64]		[27, 37, 81]
	SQL command injection	[26, 34, 67, 79]	[22, 45, 54, 64, 81]		
	SSI		[64]		
	HTTP response splitting	[34]	[64]		
Application logic errors	Parameter tampering	[5, 75]			

Table 2.2: Related work classification

Table 2.2 positions our methods among related work. RoleCast and FixMeUp are static methods for finding forced browsing vulnerabilities and repairing such vulnerabilities, respectively. Diglossia is a dynamic framework that protects a server-side Web application from SQL/NoSQL injection attacks.

The following sections explain other related static and dynamic approaches. The advantages of static methods are two-fold. First, they find vulnerabilities before application deployment. They also require no runtime performance overhead. Unfortunately, static bug-finding tools often suffer from false positives because they rely on over-approximations of dynamic program behavior. Their expensive interprocedural analyses are also a hindrance for wide deployment. However, static tools can find vulnerabilities with reasonable false positives with intentional coarse approximations.

In general, dynamic methods are precise in identifying application attacks due to their ability to track dynamic program behaviors. However, a dynamic analysis also has challenges, such as incomplete code coverage and runtime performance overhead.

2.6.1 Static Detection and Remediation of Access-control Bugs

Detection of access-control bugs requires a correct access-control policy that specifies access-control checks and security-sensitive operations. Some static detection methods require developers to specify access-control policies. Because of difficulties in analyzing and specifying access-control policies in

large legacy applications, other methods infer policies by using software engineering patterns or auxiliary information.

Using specified access-control policies. Several approaches employ access-control specifications to identify forced-browsing vulnerabilities [2, 11, 74]. Sun et al. require programmers to specify the variable states used at the intended checks for each application role and then automatically find vulnerable execution paths with unchecked access to the role's privileged pages [74]. Balzarotti et al. propose a method for finding workflow violations caused by unintended entry points [2]. They derive the intended access-control checks from user specified variables. Both methods heavily rely on identifying link relations between pages by statically approximating HTML outcome with link graphs or context free grammars. However, they are not robust because the difficulty of resolving dynamic values in generating HTML outcome varies greatly between applications. Chlipala finds security violations by statically verifying whether the application's behavior is consistent with a policy specified as a database query [11]. However, his verification only works on Ur [10], a strictly typed functional programming language, which is not directly applicable to legacy Web applications.

No prior work conducts a source-level remediation on forced-browsing vulnerabilities. By contrast, FixMeUp builds source-level patches that repair access-control bugs. For automatic remediation of CSRF vulnerabilities, Zhou et al. suggest a hybrid method that statically inserts token validation checks and dynamically transforms an HTTP outcome page to initiate an HTTP

request along with CSRF tokens [84]. They identify valid locations for code changes with annotated token validation checks and sensitive operations.

Static inference of access-control policy. Without a programmer-provided specification, several static methods (1) infer the application's access-control policies and (2) find violations of such inferred policies. They define an access-control policy as a mapping of security-sensitive operations to access-control checks that must precede them. Tan et al. use interprocedural analysis to find missing access-control checks in SELinux [76]. However, they only check the presence of access-control checks, but overlook whether the preceding checks must or may protect sensitive operations. Pistoia et al. [31, 55] and Sistla et al. [65] propose techniques for finding missing security checks in Java library code. Doupé et al. use a universal security policy—page redirection calls must be followed by program termination calls—to find EAR vulnerabilities. These papers assume that a certain policy must hold everywhere for events of a given type.

As described in Section 2.5.1, access-control logic in Web applications is more sophisticated than simple "this check must always precede that operation" patterns. They are *role-* and *context-sensitive*, with different policies enforced on different execution paths. Simple pattern matching will not find violations of such policies. RoleCast infers application and role-specific access-control checks without specification by exploiting software engineering conventions common in Web applications.

Inferring security policies using auxiliary information. Inference and verification of security policies implemented by a given program often benefit from auxiliary information and specifications. For example, Livshits et al. find errors by mining software revision histories [33]. Srivastava et al. use independent implementations of the same Java class library API to find discrepancies in security policies between implementations [71].

2.6.2 Dynamic Detection and Remediation of Access-control Bugs

Several dynamic security analyses find security violations or enforce a given security policy by tracking program execution. Hallé et al. dynamically validate whether page navigation within a given application conforms to the state machine specified by a programmer [23]. GuardRails, Nemesis and RESIN require a developer to provide explicit access-control policies and enforce them dynamically [7, 15, 82].

Alternatives to explicit specification include learning the state machine by observing benign runs and then relying on anomaly detection to find violations [12], or using static analysis of the client code to create a conservative model of legitimate request patterns and detecting deviations from these patterns at runtime [21]. Violations caused by missing access-control checks are an example of generic "execution omission" bugs. Zhang et al. presented a general dynamic approach to detect such bugs [83].

For CSRF attacks, Barth et al. dynamically attach to all HTTP requests origin headers that show what sites send HTTP requests [4]. Ryck et

al. dynamically tag each HTTP request with a browsing context state and then check whether cross-origin requests should be rejected [62].

In addition to the usual challenges of dynamic analysis, such as incomplete coverage, once dynamic enforcement of access-control policies detects a violation, it is limited in what it can do. Typically, the runtime enforcement mechanism terminates the application. After all, when an access-control check fails, the mechanism does not know what the programmer intended the application to do.

2.6.3 Static Detection of Illegitimate Data Flows

Many researchers have proposed static methods for identifying illegitimate data flows. They perform pointer and taint analyses to find unsanitized data flows from user input to sensitive sinks [26, 34, 67]. These methods can verify that a sanitization routine is always called on tainted inputs, but not whether sanitization is performed correctly. Since incorrectly sanitized input may still cause an injection attack, it is essential to precisely model the semantics of string operations performing sanitization. Wassermann and Su model string operations as transducers and check whether non-terminals in the query are tainted by user input [79].

2.6.4 Dynamic Detection and Remediation of Illegitimate Data Flows

Most dynamic detection methods aim to precisely track the source of every byte and thus determine which parts of the application-generated code

come from tainted user input and which come from a trusted source [22, 45, 54, 81]. All of these tools use a simple, imprecise definition of "code" and consequently suffer from false positives and false negatives when detecting SQL injection attacks (see Table 5.1).

To avoid the expense of byte-level taint tracking, several dynamic methods modify and examine inputs and generated queries. For example, Su and Wassermann wrap user input with meta-characters, propagate meta-characters through string operations in the program, parse the resulting query, and verify that if a meta-character appears in the parse tree, then it is in a terminal node and has a parent non-terminal such that the meta-characters wrap the descendant terminal nodes in their entirety [72]. This approach suffers from false positives and false negatives because how to wrap input (e.g., the entire input string, each word, and/or each numeric value) depends on the application generating the query.

To infer the tainted parts of the query, Sekar proposes to measure similarity between the query and user input [64], while Liu et al. compare the query to previous queries generated from benign inputs [32]. In addition to being unsound, these heuristics do not use a precise definition of code and non-code and thus suffer from false positives and false negatives.

CANDID performs a shadow execution of the program on a benign input "aaa...a", compares the resulting query with the actual query, and reports a code injection attack if the queries differ syntactically [3]. As Ray and Ligatti point out, this analysis is insufficient to differentiate code from

non-code [59]. CANDID cannot tell which parts of the query came from user input and which came from the application itself, and thus cannot detect injected identifiers (where user input injects a bound variable name that occurs elsewhere in the query), injected method invocations, and incorrect types of literals—see examples in Section 5.1.2.

Randomization and complementary encoding. To prevent the injection of SQL commands, SQLrand remaps SQL keywords to secret, hard-to-guess values [6]. Applications must be modified to use the remapped keywords in the generated queries, and database middleware must be modified to decrypt them back to original keywords. The mapping must remain secret from all users. This approach requires pervasive changes to applications and database implementations and is thus difficult to deploy.

Mui et al. suggest using complementary encoding for user input [40]. The goal is to strictly separate the character set appearing in user input from the character set used by the system internally. This approach cannot be deployed without changing databases, Web browsers, and all other systems dealing with user input. By contrast, DIGLOSSIA is a simple PHP extension that does not require any modifications to applications or databases.

2.6.5 Static Detection of Application Logic Bugs

A popular bug-finding approach is to mine the program for patterns and look for bugs as deviations or anomalies. This approach typically finds frequently occurring local, intraprocedural patterns [18]. Sun et al. find appli-

cation logic bugs in E-commerce Web applications [75]. For a given application along with annotations of which variables implement payment logic, they conduct a symbolic execution to model work flows among payment participants. Then, they check with a static taint analysis whether payment invariants are forgeable.

Several tools learn from a developer-provided fix and help apply similar fixes elsewhere. They perform the same syntactic edit on two clones [42], or suggest changes for API migration [1], or do not perform the edit [44]. Meng et al. ask users where to apply the edit [35] or find locations to apply edits automatically [36]. These approaches only apply local edits and none of them consider the interprocedural edits that are required to repair access-control logic. In the more limited domain of access-control bugs, FixMeUp automates both finding the missing logic and applying the fix.

2.6.6 Dynamic Detection and Remediation of Application Logic Bugs

Bisht et al. propose NoTamper, which finds omitted input validation logic in server-side Web applications [5]. After identifying target input parameters that have client-side validation checks, they check whether benign and tampered input parameters produce different server responses. If they produce similar responses, NoTamper reports that server-side applications do not have proper input validation logic.

BLUEPRINT and ConScript block XSS attacks by dynamically enforc-

ing given security policies in a client-side Web application instead of identifying illegitimate data flows [27, 37]. However, both methods involve heavy augmentation of server-side or client-side Web applications to specify security policies.

Dynamic program repair fixes the symptom, but not the cause of the error [9, 15, 23, 27, 37, 43, 61, 82]. For example, dynamic repair allocates a new object on a null-pointer exception, or ignores out-of-bounds references instead of terminating the program. The dynamic fixes, however, are not reflected in the source code and require a special runtime.

Chapter 3

Automatically Finding Missing Access-control Checks in Web Applications

This chapter introduces a robust method for finding missing access-control checks in Web applications. The main challenge is that each application—and even different roles within the same application, such as administrators and regular users—implements checks in a different, often idiosyncratic way, using different variables to determine whether the user is authorized to perform a particular operation. Finding missing checks is easier if the programmer formally specifies the application's security policy, e.g., via annotations or data-flow assertions [15, 82], but the overwhelming majority of Web applications today are not accompanied by specifications of their intended authorization policies.

Previous techniques for finding missing access-control checks without a programmer-provided policy take the syntactic definition of checks as input. Therefore, they must know *a priori* the syntactic form of every check. For example, Java programs perform security mediation by calling predefined methods in the SecurityManager class from the Java libraries [31, 55, 65, 71]. This approach is suitable for verifying security mediation in library and system

code, for which there exists a standard protection paradigm, but it *does not work for finding missing authorization checks in applications* because there is no standard set of checks used by all applications or even within the same application.

This chapter presents RoleCast, a static tool that finds omission of access-control checks. Given a Web application, RoleCast automatically infers (1) the set of user roles in this application and (2) the access-control checks—specific to each role—that must be performed prior to executing security-sensitive operations such as database updates. RoleCast then (3) finds missing access-control checks.

RoleCast does not rely on programmer annotations or an external specification that indicates the application's intended authorization policy, nor does it assume *a priori* which methods or variables implement access-control checks. RoleCast exploits common software engineering patterns in Web applications. A typical Web application has only a small number of sources for authorization information (e.g., session state, cookies, results of reading the user database). Therefore, all authorization checks involve a conditional branch on variables holding authorization information. Furthermore, individual Web pages function as program modules and each role within the application is implemented by its own set of modules (i.e., pages). Because each page is typically implemented by one or more program files in PHP and JSP applications, the sets of files associated with different user roles are largely disjoint.

Our static analysis that exploits the above properties has four phases. Phase I performs flow- and context-sensitive interprocedural analysis to collect calling contexts on which security-sensitive operations are control dependent. For each context, RoleCast analyzes interprocedural control dependencies to identify *critical variables*, i.e., variables that control reachability of security-sensitive operations. It then uses branch asymmetry to eliminate conditional statements that are unlikely to implement access-control checks because they do not contain branches corresponding to abnormal exit in the event of a failed check. This step alone is insufficient, however, because many critical variables (e.g., those responsible for logging) are unrelated to security.

Phase II performs role inference. This step is the key analysis and is critical because different roles within the same application often require different checks. For example, prior to removing an entry from the password database, a photo-sharing application may check the session variable to verify that the user performing the action is logged in with administrator privileges, but this check is not needed for updating the content database with users' photos. RoleCast infers application-specific roles by analyzing the modular structure of the application. As mentioned above, in PHP and JSP applications this structure is represented by program files. Phase II partitions *file contexts* that use critical variables into *roles*. A file context is simply a coarsened representation of a calling context. The partitioning algorithm arranges file contexts into groups so as to minimize the number of files shared between groups. We call each group a role.

Phase III of RoleCast determines, within a role, which critical variables are checked *consistently* and classifies this subset of the critical variables as *security-critical variables*. Phase IV then reports potential vulnerabilities in the following cases: (1) if a calling context reaches a security-sensitive operation without a check; or (2) if the role contains a single context and thus there is no basis for consistency analysis; or (3) if a check is performed inconsistently (in the majority, but not all calling contexts of the role).

Because our approach infers the Web application's authorization logic under the assumption that the application follows common code design patterns, it may suffer from both false positives and false negatives. This imprecision is inevitable because there is no standard, well-defined protection paradigm for Web applications. Furthermore, no fixed set of operations is syntactically recognizable as access-control checks (in contrast to prior approaches). Instead, RoleCast partitions the program into roles and infers, for each role, the access-control checks and security-relevant program variables by recognizing how they are used consistently (or almost consistently) within the role to control access to security-sensitive operations. When evaluated on 11 substantial, real-world PHP and JSP applications, RoleCast discovered 13 previously unreported security vulnerabilities with only 3 false positives, demonstrating its usefulness for practical security analysis of Web applications.

RoleCast demonstrates that it is possible to accurately infer the security logic of Web applications at the level of individual user roles by static

analysis, without using any programmer annotations or formally specified policies, but relying instead on common software engineering patterns used by application developers.

3.1 Access-control Logic in Web Applications

This section starts by explaining static analysis tools that we used to implement RoleCast. It also describes the common design patterns for application- and role-specific access control logic.

3.1.1 Translating Web Applications into Java

Translating scripting languages into Java is becoming a popular approach because it helps improve performance by taking advantage of mature JVM compilers and garbage collectors. We exploit this practice by (1) converting Web applications into Java class files, and (2) extending the Soot static analysis framework for Java programs [69] with new algorithms for static security analysis of Web applications.

To translate JSP and PHP programs into Java class files we use, respectively, the Tomcat Web server [77] and Quercus compiler [58]. Tomcat produces well-formed Java; Quercus does not. PHP is a dynamically typed language and the target of every callsite is potentially bound at runtime. Instead of analyzing calls in the PHP code, Quercus translates each PHP function into a Java class that contains a main method and methods that initialize the global hash table and member variables. Every function call is translated

by Quercus into a reflective method call or a lookup in the hash table. This process obscures the call graph.

Because our security analysis requires a precise call graph, we must reverse-engineer this translation. We resolve the targets of indirect method calls produced by Quercus using a flow- and context-insensitive intraprocedural symbol propagation.

3.1.2 Application-specific Access-control Logic

Our security analysis targets interactive Web applications such as blogs, e-commerce programs, and user content management systems. Interactive applications of this type constitute the overwhelming majority of real-world Web applications. Since the main purpose of these applications is to display, manage, and/or update information stored in a backend database(s), access control on database operations is critical to their integrity.

Security-sensitive operations. We consider all operations that may affect the *integrity* of database to be *security-sensitive operations*. These include all queries that *insert*, *delete*, or *update* the database. Web applications typically use SQL to interact with the backend database. Therefore, RoleCast marks INSERT, DELETE, and UPDATE mysql_query statements in PHP code as security-sensitive operations. Note that statically determining the type of a SQL query in a given statement requires program analysis. RoleCast conservatively marks all statically unresolved SQL queries as sensitive. For JSP, RoleCast marks java.sql.Statement.executeQuery and .executeUpdate calls ex-

ecuting INSERT, DELETE, or UPDATE SQL queries as security-sensitive operations.

We deliberately do not include SELECT and SHOW queries which retrieve information from the database in our definition of security-sensitive operations. Many Web applications intend certain SELECT operations to be reachable without any prior access-control checks. For example, during authentication, a SELECT statement may retrieve a stored password from the database in order to compare it with the password typed by the user. Without a programmer-provided annotation or specification, it is not possible to separate SELECT operations that need to be protected from those that may be legitimately accessed without any checks. To avoid generating a prohibitive number of false positives, we omit SELECT and SHOW operations from our analysis of Web applications' access-control logic.

Access control logic as a software design pattern. To identify application- and role-specific access-control logic, we take advantage of the software engineering patterns commonly used by the developers of Web applications. A Web application typically produces multiple HTML pages and generates each page by invoking code from several files. The following three observations guide our analysis.

Our first observation about access-control logic is that *when an access-control check fails, the program quickly terminates or restarts.* Intuitively, when a user does not hold the appropriate permissions or his credentials do

not pass verification, the program exits quickly.

Our second observation about correct access-control logic is that *every path leading to a security-sensitive operation from any program entry point must contain an access-control check*. This observation alone, however, is not sufficient to identify checks in application code because different paths may involve different checks and different program variables.

Our third observation is that *distinct application-specific roles usually involve different program files*. Since the main purpose of interactive Web applications is to manage user-specific content and to provide services to users, users' privileges and semantic roles determine the services that are available to them. Therefore, the application's file structure, which in a Web application represents the module structure, reflects a clear distinction between roles defined by the user's privileges. For instance, blog applications typically include administrator pages that modify content and register new user profiles. On the other hand, regular blog users may only read other users' content, add comments, and update their own content. In theory, developers could structure their applications so that one file handles multiple user roles, but this is not the case in real-world applications.

In the Web applications that we examined, individual program files contained only code specific to a single user role. Figure 3.1 shows a representative example with a simple page structure taken from the DNscript application. DNscript supports two types of users: an administrator and a regular user. All administrator code and pages are in one set of files, while all

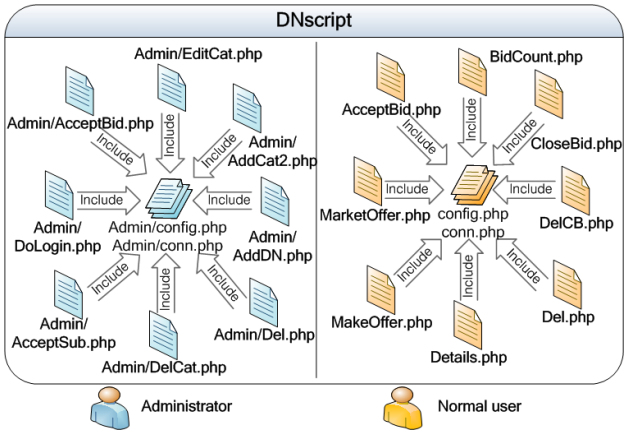

Figure 3.1: File structure of DNscript

user code and pages are in a different set of files.

Our analysis exploits the observation that access-control checks within each role are usually very similar. Inferring the roles requires automatic partitioning of contexts based on commonalities in their access-control logic.

3.2 Implementation

RoleCast has four analysis phases as Figure 3.2 demonstrates. Phase I identifies *critical* variables that control whether security-sensitive operations execute or not. Phase II partitions contexts into groups that approximate application-specific user roles. For each role, Phase III computes the subset of critical variables responsible for enforcing the access-control logic of that role. Phase IV discovers missing access-control checks by verifying whether the relevant variables are checked consistently within the role.

To identify critical variables, Phase I performs interprocedural, flow-

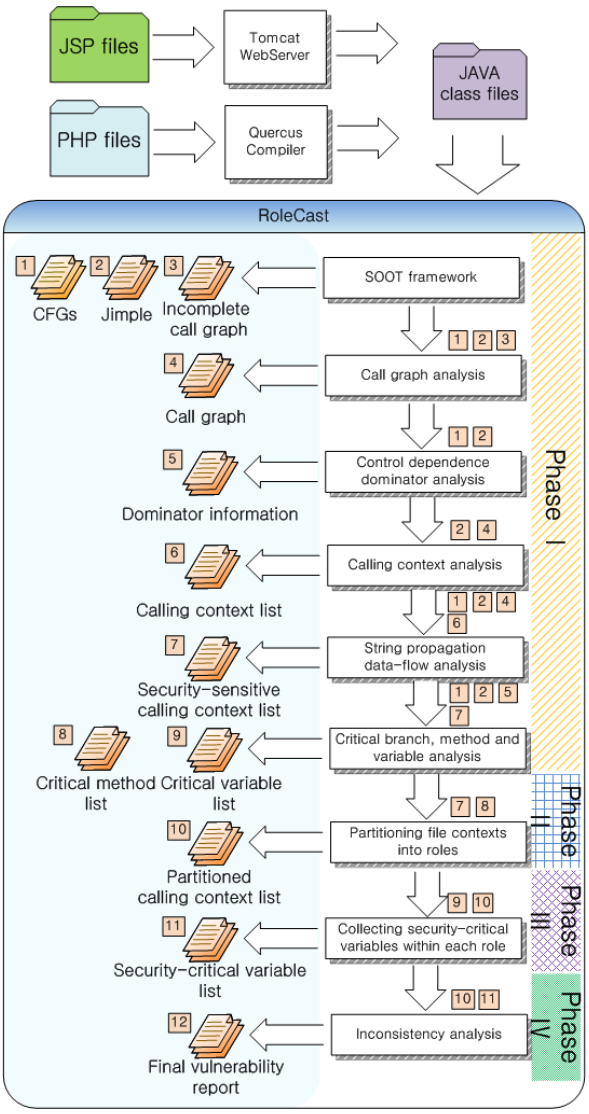

Figure 3.2: Architecture of RoleCast.

and context-sensitive control-dependence and data-flow analysis. It refines the set of critical variables using branch asymmetry, based on the observation that failed authorization checks quickly lead to program exit. To infer

application roles, Phase II maps the set of methods responsible for checking critical variables to program files and partitions them into groups, minimizing the number of shared files between groups. This algorithm seeks to discover the popular program structure in which developers put the code responsible for different application roles into different files. This heuristic is the key new component of our analysis and works well in practice.

Phase III considers each role and computes the subset of critical variables that are used *consistently*—that is, in a sufficiently large fraction of contexts associated with this role—to control reachability of security-sensitive operations in that role. The threshold is a parameter of the system. Phase IV reports a potential vulnerability whenever it finds a security-sensitive operation that can be reached without checking the security-critical variables specific to the role. It also reports all roles that involve a single context and thus preclude consistency analysis, but this case is relatively rare.

3.2.1 Phase I: Finding Security-sensitive Operations, Dominating Calling Contexts, and Critical Variables

Our algorithm for identifying access-control logic takes advantage of the following observations:

1. Any access-control check involves a branch statement on one or more *critical* variables.

2. In the branch corresponding to the failed check, the program does not

reach the security-sensitive operation and exits abnormally. For example, the program calls exit, calls die, or returns to the initial page.

3. The number of program statements in the branch from the check to the abnormal exit is significantly smaller than the number of program statements in the branch leading to the security-sensitive operation.

4. Correct access-control logic must consistently check a certain subset of critical variables prior to executing security-sensitive operations.

This section describes our algorithms that, for each security-sensitive operation, statically compute the following: calling contexts, critical branches (i.e., conditional statements that determine whether or not the security-sensitive operation executes), critical methods (i.e., methods that contain critical branches), and critical variables (i.e., variables referenced by critical branches).

Our analysis is fairly coarse. It only computes *which* variables are checked prior to security-sensitive operations, but not *how* they are checked. Therefore, RoleCast will miss vulnerabilities caused by incorrectly implemented (as opposed to entirely missing) checks on the right variables.

3.2.1.1 Security-sensitive Operations and Calling Contexts

Our analysis starts by identifying security-sensitive operations that may affect the integrity of the database. A typical Web application specifies database operations using a string parameter passed to a generic SQL query

statement. We identify all calls to mysql_query in PHP and java.sql.Statement.-executeQuery and java.sql.Statement.executeUpdate in JSP as candidates for security-sensitive operations. The same call, however, may execute different database operations depending on the value of its string parameter. Therefore, we perform an imprecise context-sensitive data-flow analysis to resolve the string arguments of database calls and eliminate all database operations that do not modify the database (see Section 3.1.2) from our set of security-sensitive operations.

RoleCast computes the set of all calling contexts for each security-sensitive operation e. RoleCast identifies the methods that may directly invoke e, then performs a backward depth-first pass from each such method over the call graph. The analysis builds a tree of contexts whose root is e and whose leaves are program entry points. For each calling context cc corresponding to a call-chain path from e to a leaf, the (cc, e) pair is added to the set of all calling contexts. This analysis records each invoked method only once per calling context, even in the presence of cyclic contexts. This is sufficient for determining whether or not an access-control check is present in the context.

Next, RoleCast propagates the strings passed as parameters in each calling context cc to the candidate operation e. The goal is to eliminate all pairs (cc, e) where we can statically prove that e cannot be a security-sensitive operation, i.e., none of the statically feasible database operations at e affect the integrity of the database because they can only execute SELECT or SHOW

queries.

We find that many Web applications generate SQL queries by assigning a seed string constant to a variable and then concatenating additional string constants. The seed string usually identifies the type of the SQL query (UPDATE, SELECT, *etc.*). Therefore, RoleCast models string concatenation, assignment of strings, and the behavior of string get() and set() methods. It performs forward, interprocedural, context-sensitive constant propagation on the string arguments for each calling context cc of operation e. RoleCast does not model the value of strings returned by method calls not in the calling context. If the string is passed as an argument to some method $m \notin cc$, we conservatively assume that the string is modified and RoleCast marks operation e as security-sensitive. Otherwise, the analysis propagates string values of actual arguments to the formal parameters of methods.

If this analysis proves that e is a SELECT or SHOW query, then RoleCast removes the (cc, e) from the set of calling contexts. The "unresolved" column in Table 3.1, explained in more detail in Section 3.3, shows that fewer than 5% of query types are unresolved, while for at least 95% of all database operations in our sample Web applications, RoleCast successfully resolves whether or not they are sensitive, i.e., whether they can affect the integrity of the database.

3.2.1.2 Critical Branches and Critical Methods

For each calling context and security-sensitive operation pair (cc, e), RoleCast performs an interprocedural control-dependence analysis to find the *critical branches* $B(cc, e)$ performed in the *critical methods* $CM(cc, e)$. These branches determine whether or not e executes. A critical method contains one or more critical branches on which e is interprocedurally control dependent. Note that some critical methods are in cc and some are not. For the reader's convenience, we review the classical intraprocedural definition of *control dependence* [14].

Definition 1. If $G = (N, E)$ is a control-flow graph (CFG) and $s, b \in N$, $b \rightsquigarrow s$ iff there exists at least one path reaching from b to s in G.

Definition 2. If $G = (N, E)$ is a control-flow graph (CFG) and $s, b \in N$, s is ***control dependent*** on b iff $b \rightsquigarrow s$ and s post-dominates all $v \neq b$ on $b \rightsquigarrow s$, and s does not post-dominate b.

Definition 3. If $G = (N, E)$ is a control-flow graph (CFG) and $a, b \in N$, a ***post-dominates*** b iff every path in $b \rightsquigarrow end$ intersects a.

The set of branch statements on which e is control dependent is computed in two steps.

1. RoleCast uses intraprocedural control dependence to identify branch statements in methods from cc that control whether e executes or not. For each method $m_i \in cc$ where m_i calls m_{i+1}, the algorithm finds branch

statements on which the callsite of m_{i+1} is control dependent and adds them to $B(cc, e)$.

2. RoleCast then considers the set of methods N such that the callsite of $n_i \in N$ dominates some method $m \in cc$ or n_i is called unconditionally from $n_j \in N$. Because every method $n_i \in N$ is invoked before reaching e, n_i *interprocedurally dominates* e. For each $n_i \in N$, RoleCast finds branch statements on which the program-exit calls in n_i (if any) are control dependent and adds them to $B(cc, e)$.

Next, RoleCast eliminates statements from B that do not match our observation that failed access-control checks in Web applications terminate or restart the program quickly. To find branch statements in which one branch exits quickly while the other executes many more statements, RoleCast calculates the *asymmetric ratio* for each $b \in B$ as follows.

RoleCast counts the number of statements in, respectively, the shortest path reaching program termination and the shortest path reaching e. Each statement in a loop counts as one statement. The asymmetric ratio is the latter count divided by the former. The larger the value, the more asymmetric the branches are. If the calculated asymmetric ratio for b is less than a threshold θ_{asymm}, we remove b from B because b does not have a branch that causes the program to exit quickly and thus is not likely to be security-critical. Our experiments use 100 as the default θ_{asymm} threshold when the calculated ratio for one or more branch statements is greater than 100; otherwise, we use the

median ratio of all branch statements. As Table 3.3 shows, the results are not very sensitive to the default value. In our sample applications, applying the default filter reduces the number of critical branches by 36% to 83% (54% on average).

After this step, the set $B(cc, e)$ contains critical branch statements. We map $B(cc, e)$ to the set of critical methods $CM(cc, e)$ that contain one or more branches from $B(cc, e)$. Recall that some critical methods are in cc and some are not. Critical methods are a *superset* of the methods responsible for implementing the application's security logic.

3.2.1.3 Critical Variables

Informally, a program variable is *critical* if its value determines whether or not some security-sensitive operation is reached. All *security-critical* variables involved in the program's security logic (e.g., variables holding user permissions, session state, *etc.*) are critical, but the reverse is not always true: critical variables are a superset of security-critical variables. We derive the set of critical variables from the variables referenced directly or indirectly by the critical branches $B(cc, e)$. Section 3.2.4 further refines the set of critical variables into the set of security-critical variables.

Given $B(cc, e)$, we compute the set of critical variables $V(cc, e)$ where $v \in V$ iff $\exists b \in B$ that references v directly or indirectly through an intraprocedural data-flow chain. We use a simple reaching definitions algorithm to compute indirect references within a method. We compute the backward in-

traprocedural data-flow slice of v for all v referenced by b. Thus if b references v and v depends on u (e.g., $v = foo(u)$), we add v and u to V.

We found that intraprocedural analysis was sufficient for our applications, but more sophisticated and object-oriented applications may require interprocedural slicing.

3.2.2 Phase II: Partitioning Into Roles

This section describes how RoleCast partitions applications into roles. We use role partitioning to answer the question: "Which critical variables should be checked before invoking a security-sensitive operation e in a given calling context?"

Without role partitioning, critical variables are not very useful because there are a lot of them and they are not always checked before every security-sensitive operation. Reporting a vulnerability whenever some critical variable is checked inconsistently results in many false positives (see Section 3.3). Role partitioning exploits the observation made in Section 3.1.2 that Web applications are organized around distinct user roles (e.g., administrator and regular user). We note that (1) applications place the code that generates pages for different roles into different files, and, furthermore, (2) the files containing the security logic for a given role are distinct from the files containing the security logic for other roles. These observations motivate an approach that focuses on finding sets of (cc, e) in which the critical methods $CM(cc, e)$ use the same *files* to enforce their security logic.

RoleCast starts by coarsening its program representation from methods and calling contexts to files. This analysis maps each set of critical methods $CM(cc, e)$ to a set we call the *critical-file context* CF. A file $cf \in CF(cc, e)$ if cf defines any method $m \in CM(cc, e)$. We also define the *file context* $F(cc, e)$. A file $f \in F(cc, e)$ if f defines any method m which interprocedurally dominates e. F is a superset of CF—some files in $F(cc, e)$ are critical and some are not. We refer to the set of all file contexts F of all security-sensitive operations in the program as $\widehat{F}$ and to the set of all critical-file contexts CF as $\widehat{CF}$.

Since we do not know *a priori* which file corresponds to which application role, our algorithm generates candidate partitions of $\widehat{CF}$ and picks the one that minimizes the number of shared files between roles. The goal is to group similar critical-file contexts into the same element of the partition. We consider two critical-file contexts as similar if they share critical files. To generate a candidate partition, the algorithm chooses a "seed" critical file cf_1 and puts all critical-file contexts from $\widehat{CF}$ that reference cf_1 into the same group, chooses another critical file cf_2 and puts the remaining critical-file contexts from $\widehat{CF}$ that contain cf_2 into another group, and so on. The resulting partition depends on the order in which seed critical files are chosen. Our algorithm explores all orders, but only considers frequently occurring critical files. In practice, the number of such files is small, thus generating candidate partitions based on all possible orderings of seed files is feasible.

To choose the best partition from the candidates, our algorithm evaluates how well the candidates separate the more general file contexts $\widehat{F}$. The

insight here is that since programmers organize the *entire* application by roles (not just the parts responsible for the security logic), the correct partition of security-related file contexts should also provide a good partition of all file contexts. RoleCast thus prefers the partition in which the groups are most self-contained, i.e., do not reference many files used by other groups.

More formally, RoleCast's partitioning algorithm consists of the following five steps.

1. For each $CM(cc, e)$, compute the critical-file context $CF(cc, e)$ and file context $F(cc, e)$.

2. Eliminate critical files that are common to all $CF(cc, e)$, i.e., if file f belongs to the critical-file context cf for all $cf \in \widehat{CF}$, then remove f from all contexts in $\widehat{CF}$. Since these files occur in every critical-file context, they will not help differentiate roles.

3. Extract a set of *seed files* (SD) from $\widehat{CF}$. We put file f into SD if it occurs in at least the θ_{seed} fraction of critical-file contexts $cf \in \widehat{CF}$. In our experiments, we set $\theta_{seed} = 0.2$. We use only relatively common critical files as the seeds of the partitioning algorithm, which helps make the following steps more efficient.

4. Generate all ordered permutations SD_i. For each SD_i, generate a partition $P_i = \{G_1, \ldots G_k\}$ of $\widehat{CF}$ as follows. Let $SD_i = \{f_1, \ldots f_n\}$. Let $\widehat{TF} = \widehat{CF}$ and set $k = 1$. For $i = 1$ to n, let $C_i \subseteq \widehat{TF}$ be the set of

all critical-file contexts from $\widehat{TF}$ containing file f_i. If C_i is not empty, remove these contexts from $\widehat{TF}$, add them to group G_k, and increment k.

5. Given candidate partitions, choose the one that minimizes the overlap between files from different groups. Our algorithm evaluates each candidate $\{G_1, \ldots G_k\}$ as follows. First, for each group of critical-file contexts G_j, take the corresponding set of file contexts F_j. Then, for each pair F_k, F_l where $k \neq l$, calculate the number of files they have in common: $|F_k \cap F_l|$. The algorithm chooses the partition with the smallest $\sum_{k<l} |F_k \cap F_l|$.

Figure 3.3 gives an example of the partitioning process. At the top, we show five initial file contexts and critical-file contexts produced by Step 1. Step 2 removes the files common to all critical-file contexts, common.php in this example, producing three files, process.php, user.php and admin.php. Step 3 selects these three files as the seed files, because they appear in 1/5, 3/5, and 2/5 of all critical-file contexts, respectively. There are 6 permutations of this set, thus Step 4 produces 6 candidate partitions. In Figure 3.3, we compare two of them: (admin.php, user.php, process.php) and (process.php, admin.php, user.php). The corresponding candidates are $P_1 = \{(A, C), (B, D, E)\}$ and $P_2 = \{(B), (A, C), (D, E)\}$. To decide which one best approximates user roles, Step 5 computes the intersection of file contexts between every pair of groups in each partition and selects the partition with the smallest intersection. In

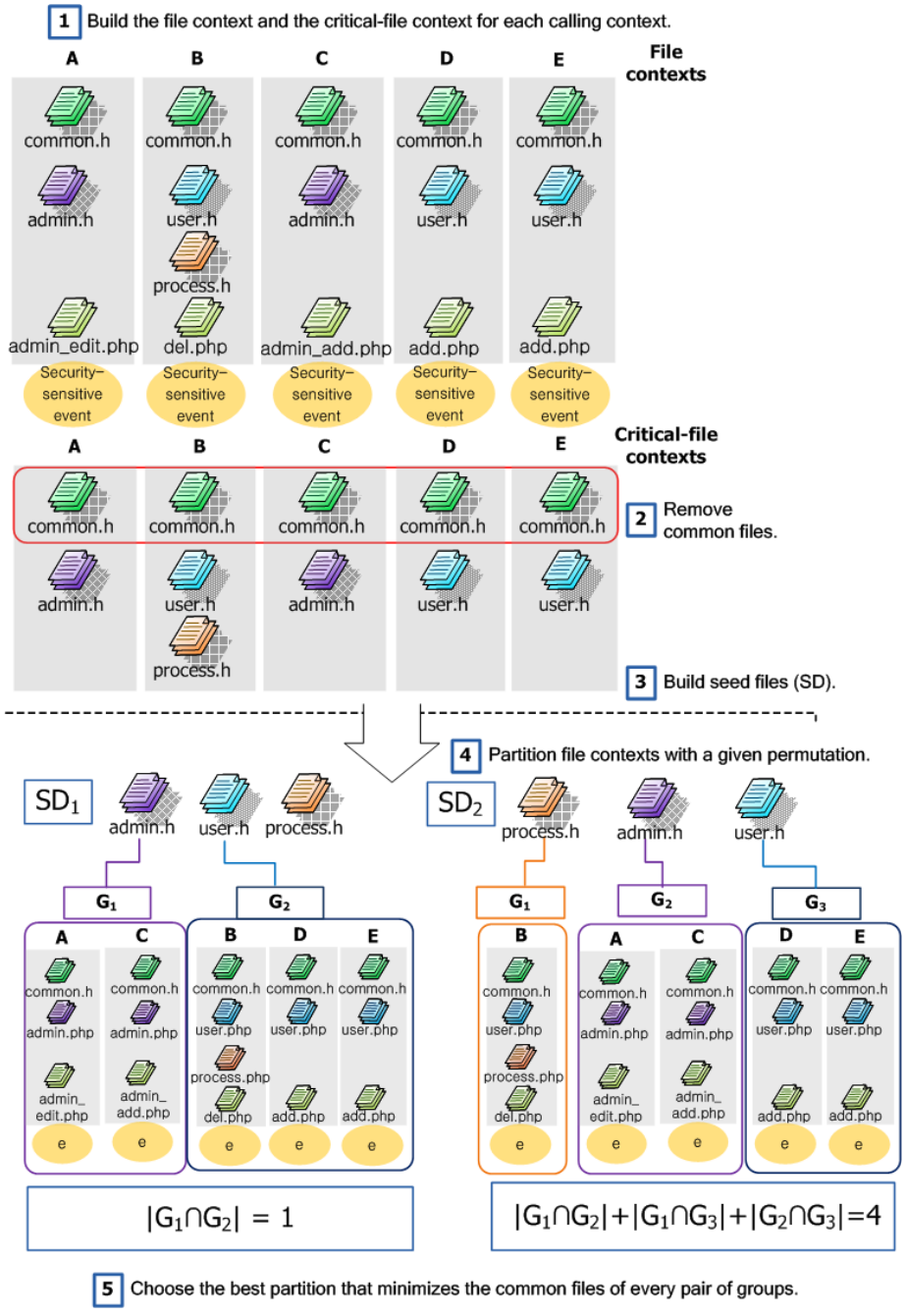

Figure 3.3: Example of partitioning contexts

Algorithm 1: Partitioning file contexts into roles

$P \leftarrow \phi$ { initialize partition candidate set }
$n \leftarrow |SD|$ { get the size of SD }
for each $SD_i \in$ n permutation of SD **do**
  $\widehat{TF} \leftarrow \widehat{CF}$ { initialize work list with all critical-file contexts }
  $k \leftarrow 1$
  for each $f_j \in SD_i$ **do**
    { put all critical-file context that contain seed file f_j into G_k }
    for each $CF \in \widehat{TF}$ **do**
      if $f_j \in CF$ **then**
        $G_k \leftarrow G_k \cup CF$
        $\widehat{TF} \leftarrow \widehat{TF} - \{CF\}$
      end if
    end for
    if G_k is not empty **then**
      $k \leftarrow k + 1$ { increment the group index }
    end if
    if $\widehat{TF}$ is empty **then**
      break { if work list is empty, then break }
    end if
  end for
  $P_i \leftarrow \{G_1 \dots G_k\}$ { store the current groups into P_i }
  $DS_i \leftarrow 0$
  { compute the number of common files among all pairs in G_i }
  for each pair $(G_a, G_b)_{a<b}$ from $\{G_1 \dots G_k\}$ **do**
    F_a = the file contexts corresponding to critical-file contexts in G_a
    F_b = the file contexts corresponding to critical-file contexts in G_b
    $DS_i \leftarrow DS_i + |F_a \cap F_b|$
  end for
  $P \leftarrow P \cup \{P_i\}$ { add the current partition to the candidate list }
  for $x = 1$ to k **do**
    $G_x \leftarrow \phi$ { reset groups for the next candidate SD_{i+1} }
  end for
end for
Pick the best $P_i \in P$ that has a minimum DS_i

our example, since P_1 has the fewest files (one) in the intersection, RoleCast chooses it as the best partition.

The accuracy of the partitioning step depends on using good seed files. We select files that contain critical variables and appear in many critical-file contexts, based on our observation that Web applications use common security logic in most contexts that belong to the same role. One concern

is that exploring all permutations of the seed-file set is exponential in the number of seed files. The selection criteria for seed files are fairly stringent, and therefore their actual number tends to be small. It never exceeded four in our experiments. If the number of seed files grows large, the partitioning algorithm could explore fewer options. For example, it could prioritize by how often a seed file occurs in the file contexts.

3.2.3 Phase III: Finding Security-critical Variables

The groups in the partition computed by Phase II approximate semantic user roles in the application. Phase III computes the *security-critical variables* for each role: the subset of the critical variables that enforce the role's security logic.

We assume that the application correctly checks the security-critical variables in at least some fraction of the critical-file contexts and use this observation to separate security-critical variables from the rest of the critical variables. Recall that there is a one-to-one mapping between each critical-file context $CF(cc, e)$ and the set of its critical variables $V(cc, e)$. Given the partition $\{G_1, \ldots, G_k\}$ of $\widehat{CF}$, let V_i be the set of all critical variables from the critical-file contexts in G_i. We initialize the set of security-critical variables $SV_i = V_i$. RoleCast then removes all variables $v \in SV_i$ that appear in fewer than a $\theta_{consistent}$ fraction of the critical-file contexts in G_i. We use $\theta_{consistent} = 0.5$ as our default. Table 3.5 shows that our results are not very sensitive to this parameter. We define the remaining subset SV_i to be the security-critical

variables for role i.

3.2.4 Phase IV: Finding Missing Access-control Checks

This phase reports vulnerabilities. Within each role, all contexts should consistently check the same security-critical variables before performing a security-sensitive operation e. Formally, given $sv \in SV_i$, RoleCast examines every (cc, e) in the group of contexts corresponding to role i and verifies whether $sv \in V(cc, e)$ or not. If $sv \notin V(cc, e)$, RoleCast reports a potential security vulnerability. To help developers, RoleCast also reports the file(s) that contains the security-sensitive operation e for the vulnerable (cc, e).

RoleCast reports a potential security vulnerability in two additional cases: (1) for each file that executes a security-sensitive operation and does not check any critical variables whatsoever, and (2) for each singleton critical-file context (i.e., a role with only one critical-file context). Because there is nothing with which to compare this context, RoleCast cannot apply consistency analysis and conservatively signals a potential vulnerability.

3.3 Evaluation

We evaluated RoleCast by applying it to a representative set of open-source PHP and JSP applications. All experiments were performed on a Pentium(R) 3GHZ with 2G of RAM. Table 3.1 shows the benchmarks, lines of original source code, lines of Java source code produced by translation, and analysis time. We have not tuned our analysis for performance.

Web applications	LoC	Java LoC	analysis time	DB operations (|contexts|) candidates	sensitive	unresolved	critical branches candidates	asymm
minibloggie 1.1	2287	5395	47 sec	13	3	0	12	7
DNscript	3150	11186	47 sec	99	26	0	27	5
mybloggie 1.0.0	8874	26958	74 min	195	26	0	135	58
FreeWebShop 2.2.9	8613	28406	110 min	699	175	0	186	82
Wheatblog 1.1	4032	11959	2 min	111	30	0	31	20
phpnews 1.3.0	6037	13086	166 min	80	14	3	65	15
Blog199j 1.9.9	8627	18749	75 min	195	68	2	104	54
eBlog 1.7	13862	24361	410 min	677	261	0	136	17
kaibb 1.0.2	4542	21062	197 min	676	160	0	306	152
JsForum (JSP) 0.1	4242	4242	52 sec	60	32	0	6	1
JSPblog (JSP) 0.2	987	987	16 sec	6	3	0	0	0

Table 3.1: Benchmarks and analysis characterization

For example, we unnecessarily re-analyze every context, even if we have analyzed a similar context before. Memorizing analysis results and other optimizations are likely to reduce analysis time. The three columns in the middle of the table show the number of contexts for all database operations (candidate security-sensitive operations), operations that can affect the integrity of the database (security-sensitive operations), and database operations whose type could not be resolved by our analysis. Comparing these three columns shows that our string propagation is effective at resolving the type of database operations and rarely has to assume an operation is security-sensitive because it could not resolve the string argument determining its type. The last two columns show the number of critical branch statements before and after eliminating statements that are not sufficiently asymmetric.

Table 3.2 shows the results of applying RoleCast to our benchmarks. As described in Section 3.2.2, RoleCast partitions calling contexts containing security-sensitive operations into groups approximating application-specific user roles. For each role, RoleCast finds critical variables that are checked in at least the $\theta_{consistency}$ fraction of the contexts in this role. If such a critical variable is not checked in one of the contexts, RoleCast reports a potential vulnerability. RoleCast also reports a potential vulnerability if a security-sensitive operation is reachable without any checks at all. We examined each report by hand and classified it as a false positive or real vulnerability.

Note the importance of role partitioning in Table 3.2 for reducing the

Web applications	false positives roles	false positives no roles	no auth.	vuln.
minibloggie 1.1	0	0	0	1
DNscript	1	5	0	3
mybloggie 2.1.6	0	0	0	1
FreeWebShop 2.2.9	0	1	0	0
Wheatblog 1.1	1	0	1	0
phpnews 1.3.0	1	12	0	0
Blog199j 1.9.9	0	1	0	0
eBlog 1.7	0	4	2	0
kaibb 1.0.2	0	11	1	0
JsForum (JSP) 0.1	0	0	0	5
JSPblog (JSP) 0.2	0	0	0	3
totals	3	34	4	13

Table 3.2: Accuracy ($\theta_{consistency} = .5$). Note the reduction in false positives due to role partitioning.

number of false positives. Without role partitioning, a conservative analysis might assume that critical variables should be checked consistently in all program contexts. All contexts associated with roles that do not require a particular access-control check would then result in false positives.

The number of false positives after role-specific consistency analysis is very small. There are several reasons for the remaining false positives. First, if a role contains only one context, RoleCast cannot apply consistency analysis and conservatively reports a potential vulnerability. Second, a Web application may use a special set of critical variables only for a small fraction of contexts (this case is rare). Consider Figure 3.5. Both the post2() method call on Line 11 in index.php and the fullNews() method call on Line 4 in news.php

contain security-sensitive operations. A large fraction of calling contexts use $auth variable to enforce access control (Line 10 in auth.php). On the other hand, a small fraction of contexts leading to the sensitive database operation in fullNews use only $Settings (Line 9 in news.php). Because RoleCast decides that $auth is the variable responsible for security enforcement due to its consistent presence in the contexts of security-sensitive operations, it decides that the few contexts that only use $Settings are missing a proper access-control check.

Table 3.2 distinguishes between two kinds of unauthorized database operations. Some database updates may be relatively harmless, e.g., updating counters. Nevertheless, if such an update is executed without an access-control check, it still enables a malicious user to subvert the intended semantics of the application. Therefore, we do not consider such updates as false positives and count them in the 3rd column of Table 3.2, labeled "no auth.". The 4th column of Table 3.2, labeled "vuln.," reports database updates that allow a malicious user to store content into the database without an access-control check. Because these vulnerabilities are both severe and remotely exploitable, we notified the authors of all affected applications.

Figure 3.4 shows two files from DNscript that RoleCast reports as vulnerable. Neither file contains any access-control checks, thus a malicious user can alter the contents of the backend database by sending an HTTP request with the name of either file as part of the URL. RoleCast reports that the security-sensitive operations in DelCB.php and admin/AddCat2.php

admin/AddCat2.php

```
1  <?php
2  // No security check. It should have been checked with $_SESSION['admin']
3  include 'inc/config.php';
4  include 'inc/conn.php';
5  $values = 'VALUES ("'.$_POST['cat_name'].'")';
6  // Security-sensitive operation
7  $insert = mysql_query("INSERT INTO gen_cat(cat_name) " .$values);
8  if($insert)
9  {
10   mysql_close($conn);
11   ...
12 }
13
14 ?>
```

DelCB.php

```
1  <?php
2  // No security check. It should have been checked with $_SESSION['member']
3  include 'inc/config.php';
4  include 'inc/conn.php';
5  // Security-sensitive operation
6  $delete = mysql_query("DELETE FROM close_bid where item_name = '".
       $item_name."'");
7  if($delete)
8  {
9    mysql_close($conn);
10   ...
11 }
12 ?>
```

Figure 3.4: Detected vulnerable files in DNscript

should be protected by checking $_SESSION['member'] and $_SESSION['admin'], respectively.

Our analysis uses three thresholds: the branch asymmetry threshold θ_{asymm}, the commonality threshold for seed files θ_{seed}, and the consistency threshold for security-critical variables $\theta_{consistency}$. Tables 3.3 through 3.5 show that the analysis is not very sensitive to these values.

	θ_{asymm}									
	25		50		100		150		200	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	1	3	1	3	1
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	0	0	0	0
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	2	0	1	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	2	0	2	0	1	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	0
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 3.3: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to θ_{asymm}

	θ_{seed}									
	0.2		0.3		0.4		0.5		0.6	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	2	3	1	3	1
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	1	1	1	1
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	0	0	0	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	2	0	2	0	2	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	11
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 3.4: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to θ_{seed}

	$\theta_{consistency}$									
	0.5		0.6		0.7		0.8		0.9	
Web applications	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**	**vl**	**fp**
minibloggie 1.1	1	0	1	0	1	0	1	0	1	0
DNscript	3	1	3	1	3	1	3	0	0	0
mybloggie 1.0.0	1	0	1	0	1	0	1	0	1	0
FreeWebShop 2.2.9	0	0	0	0	0	0	0	0	0	0
Wheatblog 1.1	1	1	1	1	1	1	1	1	1	1
phpnews 1.3.0	0	1	0	1	0	1	0	1	0	1
Blog199j 1.9.9	0	0	0	0	0	0	0	0	0	0
eBlog 1.7	2	0	2	0	1	0	1	0	1	0
kaibb 1.0.2	1	0	1	0	1	0	1	0	1	0
JsForum (JSP) 0.1	3	0	3	0	3	0	3	0	3	0
JSPblog (JSP) 0.2	5	0	5	0	5	0	5	0	5	0

Table 3.5: Sensitivity of actual vulnerabilities (vl) and false positives (fp) to $\theta_{consistency}$

Table 3.3 shows the sensitivity of our results to the branch asymmetry threshold θ_{asymm} as it varies between 25 and 200. The default value is 100. With θ_{asymm} values smaller than 100, RoleCast analyzes more branches, more critical methods, and more critical variables, but still finds the same vulnerabilities, although with two more false positives when $\theta_{asymm} = 25$.

Table 3.4 shows the sensitivity of our results to the value of the seed threshold θ_{seed} used to generate role partitions. The default value is 0.2. For *kaibb*, when θ_{seed} is too large, RoleCast may exclude seed files that actually play an important role in partitioning the application into roles. Note that the results for DNscript do not change monotonically with the value of θ_{seed}. When $\theta_{seed} = .2$ or .3, RoleCast finds three seed files. Two of them correspond to actual user roles (administrator and regular user) and RoleCast produces

the correct partition. When $\theta_{seed} = .4$, there are only two seed files, one of which corresponds to the user role, while the other produces a spurious "role" with a single context, resulting in a false positive. When $\theta_{seed} = .5$ or .6, RoleCast finds a single seed file, which corresponds to the administrator role. The resulting partition has two groups—contexts that use the seed file and contexts that do not use the seed file—which are exactly the same as in the partition created when $\theta_{seed} = .2$ or .3.

Table 3.5 shows how our results change as the $\theta_{consistency}$ threshold varies between 0.5 and 0.9. This threshold controls the fraction of critical-file contexts in which a variable must appear in order to be considered security-critical for the given role. The default value is 0.5. In two applications, increasing the threshold decreases both the number of vulnerabilities detected and the number of false positives (as expected). For most applications, there is no difference because the root cause of many reported vulnerabilities is either the absence of any checks prior to some security-sensitive operation, or roles containing a single context.

In summary, the algorithm is not very sensitive to its three thresholds, requires role analysis to reduce the false positive rate, and finds actual vulnerabilities.

3.4 Conclusion

We designed and implemented RoleCast, a new tool for statically finding missing security checks in the source code of Web applications with-

out an explicit policy specification. RoleCast exploits the standard software engineering conventions used in server-side Web programming to (1) identify security-sensitive operations such as database updates, (2) automatically partition all contexts in which such operations are executed into groups approximating application-specific user roles, (3) identify application- and role-specific security checks by their semantic function in the application (namely, these checks control reachability of security-sensitive operations and a failed check results in quickly terminating or restarting the application), and (4) find missing checks by consistency analysis of critical variables within each role.

When evaluated on a representative sample of open-source, relatively large PHP and JSP applications, RoleCast discovered 13 previously unreported vulnerabilities with only 3 false positives.

RoleCast only reports its findings and delegates responsibility of fixing the found access-control vulnerabilities. However, fixing access-control vulnerabilities involves replicating existing access-control checks at many places, which is repetitive and tedious. This motivates an automatic bug-repairing approach that we address in the next chapter.

index.php

```
if ($_GET['action'] == 'redirect')
{
  ...
}
$time_start = getMicrotime();
define('PHPNews', 1);
session_start();
require('auth.php');
...
// Security-sensitive operation is in post2
post2();
```

auth.php

```
session_start();
...
$result = mysql_query('SELECT * FROM '.$db_prefix. 'posters WHERE username
     = \''.$in_user.'\' AND password = password(\''.$in_password.'\')');
$dbQueries++;
if(mysql_numrows($result) != 0)
{
  $auth = true;
  ...
  // Security check using critical variable $auth
  if(!$auth) {
    exit;
  }
}
```

news.php

```
include('settings.php');
...
else if($_GET['action'] == 'post')
  fullNews();
...
function fullNews(){
  ...
  // Critical variable $Settings
  if($Settings['enablecountviews'] == '1') {
    $countviews = mysql_query("UPDATE ".$db_prefix. "news SET views=views
        +1 WHERE id='".$_GET['id']."'");
  }
   ...
}
```

Figure 3.5: Example of a false positive in phpnews 1.3.0

Chapter 4

Repairing Access-Control Bugs in Web Applications

Chapter 3 explored *finding* access-control bugs. However, *repairing* them is a much harder problem and only recently has some progress been made on semi-automated methods for software repair. Static repair techniques can now fix violations of simple local patterns that need only one- or two-line edits [25, 50], or find one- or two-line changes that pass unit tests [80], or perform user-specified transformations within a single method [1, 35]. Because server-side Web applications often implement access-control logic over multiple methods, repairing them requires an interprocedural approach. None of the prior work addresses interprocedural bugs. Another key issue for repairing access-control bugs is that many, but not all of the statements implementing the access-control logic are often already present in the vulnerable code. None of the prior patch, transformation, refactoring, or repair algorithms check if the statements are already present in the target code.

This chapter introduces a static interprocedural analysis and program transformation tool called FixMeUp. FixMeUp finds violations of access-control policies, produces candidate repairs, eliminates repairs that incorrectly

implement the policy, and suggests the remaining repairs to developers.

FixMeUp starts with annotations to the PHP source code marking (1) access-control checks, (2) the protected sensitive operation, and (3) a tag indicating the *user role* to which the policy applies (e.g., root, admin, or blog poster). FixMeUp assumes that each high-level policy applies throughout the indicated user role.

FixMeUp uses this specification to compute an *access-control template* (ACT). FixMeUp starts with the conditional statement performing the correct access-control check and computes all methods and statements in its backward, interprocedural slice. Given this slice, FixMeUp builds an interprocedural, hierarchical representation of all statements in the check's calling context on which the check depends. The ACT serves as both a low-level policy specification and a program transformation template.

To find missing access-control checks, FixMeUp looks at every calling context in which a sensitive operation may be executed and verifies whether the access-control logic present in this context matches the ACT for the corresponding role. Of course, FixMeUp cannot decide general semantic equivalence of arbitrary code fragments. In practice, the access-control logic of Web applications is usually very stylized and located close to the program entry points. The resulting templates are loop-free, consist of relatively few statements, and have simple control and data dependences (see Table 4.2).

FixMeUp generates candidate repairs by replicating the access-control

logic in program contexts where some or all of it is missing. If FixMeUp finds a vulnerable context that permits execution of some sensitive operation without an access-control check, it transforms the context using the access-control template. This transformation finds and reuses statements already present in the vulnerable code and only inserts the statements from the template that are missing. The repair procedure uses and respects all control and data dependences between statements.

To ensure that the reused statements do not change the meaning of the inserted policy, FixMeUp computes a fresh template starting from the access-control check and matches it against the original template. If the templates do not match, FixMeUp issues a warning. If they match, FixMeUp provides the transformed code to the developer as the suggested repair.

We evaluate FixMeUp on ten real-world Web applications varying in size from 1,500 to 100,000+ lines of PHP code. FixMeUp found 38 access-control bugs and correctly repaired 30 of them. In 7 cases, the inserted access-control check was added to an existing, alternative check. In one case, our repair validation procedure automatically detected an unwanted control dependence and issued a warning. In 28 cases, FixMeUp detected that vulnerable code already contained one or more, but not all, of the statements prescribed by the access-control template and adjusted the repair accordingly. This result shows that detecting which parts of the access-control logic are already present and correct is critical to repairing access-control vulnerabilities. No prior program repair or transformation approach detects whether the

desired logic is already present in the program [1, 25, 35, 50, 80].

FixMeUp guarantees that the repaired code implements the same access-control policy as the template, but it cannot guarantee that the resulting program is "correct." For example, FixMeUp may apply the policy to a context where the developer did not intend to use it, or the repair may introduce an unwanted dependence into the program (adding an access-control check always changes the program's control flow). Static analysis in FixMeUp is neither sound, nor complete because it does not consider language features such as dynamic class loading, some external side effects, or *eval*. The developer should examine the errors found by FixMeUp and the suggested repairs.

Using automated program analysis tools for verification and bug finding is now a well-established approach that helps programmers discover errors and improve code quality in large software systems. No prior tool, however, can repair access-control errors of omission. FixMeUp is a new tool that can help Web developers repair common access-control vulnerabilities in their applications.

4.1 Specifying Access-control Policies

FixMeUp takes as input an explicitly specified or inferred access-control policy. An access-control policy is a set of role-specific mappings from program statements executing security-sensitive operations—such as SQL queries and file operations—to one or more conditional statements that must be executed prior to these operations. The developer marks the access-control

checks and the security-sensitive operations and assigns them a user-role tag. This high-level specification informs FixMeUp that the marked check must be performed before the marked operation in all calling contexts associated with the indicated user role. In Figure 4.1, line 8 of *admin.php* shows an annotation that marks the access-control check with the "admin" role tag. Lines 22 and 26 show the annotations for security-sensitive operations. FixMeUp does not currently support disjunctive policies where operations may be protected by *either* check A *or* check B.

Unlike GuardRails [7], FixMeUp does not require an external specification of all statements involved in access-control enforcement. Instead, FixMeUp automatically computes access-control policies from the annotations marking the checks and the protected operations.

4.2 Implementation

This section describes the implementation of FixMeUp. We implemented all analyses in PHC, an open-source PHP compiler [51], and analyze PHC-generated abstract syntax trees (AST). We started by adding standard call graph, calling context, data dependence, and control dependence analyses to PHC.

4.2.1 Computing Access-control Templates

FixMeUp takes as input an explicit mapping from sensitive operations to correct access-control checks. FixMeUp then performs interprocedural

admin.php

```
1  <?
2    include("configuration.php"); // slice & ACT
3    include("functions.php");
4    require("lang/$language.php");
5    $security = "yes";  // slice & ACT
6    $include_script = "yes";
7    if ($security == "yes") { // slice & ACT
8      //@ACC('admin')
9      if ((!isset($PHP_AUTH_USER))  // slice & ACT
10         || (!isset($PHP_AUTH_PW))
11         || ($PHP_AUTH_USER != 'UT')
12         || ($PHP_AUTH_PW != 'UTCS')) {
13       header('WWW-Authenticate: Basic realm="newsadministration"'); //
             slice & ACT
14       header('HTTP/1.0 401 Unauthorized'); // slice & ACT
15       echo '<html><head><title>Access Denied!</title></head><body>
             Authorization Required.</body></html>'; // slice & ACT
16       exit; // slice & ACT
17   }  }    ...
18   switch($action) {
19     case "check":  check(); break;
20     case "add":    //@SSO('admin')
21        add();
22        break;
23     case "delete": //@SSO('admin')
24        delete();
25        break;
26     ... } ?>
```

configuration.php

```
1  <?php    ...
2   $PHP_AUTH_PW = $_SERVER['PHP_AUTH_PW'];       // slice
3   $PHP_AUTH_USER = $_SERVER['PHP_AUTH_USER']; // slice
4     ... ?>
```

Access-control template for admin users

```
(m0 = admin.php (program entry),
S0 = {
  include("configuration.php");
  $security = "yes";
  if ($security == "yes") {
     if ((!isset($PHP_AUTH_USER))
        || (!isset($PHP_AUTH_PW))
        || ($PHP_AUTH_USER != 'UT')
        || ($PHP_AUTH_PW != 'UTCS')) {
       header('WWW-Authenticate: Basic realm="newsadministration"');
       header('HTTP/1.0 401 Unauthorized');
       echo '<html><head><title>Access Denied!</title></head><body>
            Authorization Required.</body></html>';
       exit;    }  } )
```

Figure 4.1: *Newsscript*: Slice and access-control template

program slicing on the call graph and on the data- and control-dependence graphs of each method to identify the program statements on which each access-control check is data- or control-dependent. FixMeUp converts each slice into a template, which serves as a low-level specification of the correct policy logic and a blueprint for repair. Informally, the template contains all statements in the check's calling context that are relevant to the check: (1) statements on which the check is data- or control-dependent, and (2) calls to methods that return before the check is executed but contain some statements on which the check is dependent.

4.2.1.1 Computing Access-control Slices

Given a conditional access-control *check*, FixMeUp picks an *entry* which has the shortest call depth to *check*. FixMeUp iteratively computes the transitive closure of the statements on which *check* is control- or data-dependent. This analysis requires the call graph, control-flow graphs, intraprocedural aliases, and intraprocedural def-use chains. For each call site, FixMeUp computes an interprocedural summary of side effects, representing the def-use information for every parameter, member variable, and base variable at this site. These analyses are standard compiler fare and we do not describe them further.

In general, slices that perform access-control enforcement are typically loop-free computations that first acquire or retrieve user credentials or session state, and then check them. All of our benchmarks follow this pattern. State-

ments in these slices update only a small set of dedicated variables which are used in the check but do not affect the rest of the program. The exceptions are global variables that hold database connections and session state. These variables are typically initialized before performing access control and read throughout the program. When FixMeUp inserts code to repair vulnerabilities, it takes care not to duplicate statements with side effects.

4.2.1.2 Extracting Access-control Templates

Statements in a slice may be spread across multiple methods and thus do not directly yield an executable code sequence for inserting elsewhere. Therefore, FixMeUp converts slices into templates.

An *access-control template (ACT)* is a hierarchical data structure whose hierarchy mirrors the calling context of the access-control check. Each level of the ACT corresponds to a method in the context. For each method, the ACT records the statements in that method that are part of the slice. These statements may include calls to methods that return before the access-control check is executed, but only if the call subgraphs rooted in these methods contain statements that are part of the slice.

The last level of the ACT contains the *access-control check* and the *failed-authorization* code that executes if the check fails (e.g., termination or redirection). The developer optionally specifies the failed-authorization branch. Without such specification, FixMeUp uses the branch that contains a program exit call, such as die or exit. We label each ACT with the programmer-

specified user role from the check's annotation.

Formally, ACT_{role} is an ordered list of (m_i, S_i) pairs, where m_i are method names and $S_i \in m_i$ are ordered lists of statements. Each m_i is in the calling context of *check*, i.e., it will be on the stack when *check* executes. Each statement $s \in S_i$ is part of the access-control logic because (1) the *check* is data- or control-dependent on s, or (2) s is a call to a method n that contains such a statement somewhere in its call graph, but n returns before the *check* executes, or (3) s is a statement in the failed-authorization branch of *check*. Consider the following example:

```
1 main () {
2     a = b;
3     c = credentials(a);
4     if (c) then fail(...);
5     perform security-sensitive operation
6 }
```

The conditional statement if (c) is the access-control check and its calling context is simply main. The computed template ACT_{role} includes the call to credentials, as well as fail(...) in the branch corresponding to the failed check. We add the following pair to the ACT_{role}: (main, { a=b, c=credentials(a), if (c) then fail(...) }).

Figure 4.2 shows the algorithm that, given a calling context and a slice, builds an ACT. The algorithm also constructs data- and control-dependence maps, DD_{ACT} and CD_{ACT}, which represent all dependences between statements in the ACT. FixMeUp uses them to (1) preserve dependences between

```
GetACT (CC, SLICE) {
1    // INPUT
2       CC = {(cs_1, m_0), (cs_2, m_1) ... (check, m_n)}: calling context of the check, where cs_{i+1} ∈ m_i is the
            call site of m_{i+1}
3       SLICE: statements on which the check is data− or control−dependent and statements executed
            when authorization fails
4    // OUTPUT
5       ACT: template {(m_i, s_i)}, where s_i is an ordered list of statements in method m_i
6       DD_ACT, CD_ACT: data and control dependences in ACT
7
8    ACT ← ∅
9    ACT.CC_src ← CC
10   BuildACT (m_0, CC, SLICE)
11   DD_ACT = {(s_k, s_j) s.t. s_{k,j} ∈ ACT and s_k is data−dependent on s_j}
12   CD_ACT = {(s_k, s_j) s.t. s_{k,j} ∈ ACT and s_k is control−dependent on s_j}
13
14   return ACT
15 }

BuildACT (m_i, CC, SLICE) {
1   S_i ← ∅
2   j ← 0
3   for (k = 0 to |m_i|, s_k ∈ m_i) { // |m_i| is the number of statements in m_i
4     if (s_k ∈ SLICE) {
5       S_i[j++] = s_k
6     }
7     if (s_k is a callsite s.t. (s_k, m_{i+1}) ∈ CC) {
8       BuildACT (m_{i+1}, CC, SLICE)
9     }
10  }
11  ACT ← {(m_i, S_i)} ∪ ACT
12 }
```

Figure 4.2: Computing an access-control template (ACT)

statements when inserting repair code, and (2) match templates to each other when validating repairs. Figure 4.1 gives an example of an access-control slice and the corresponding ACT from Newsscript 1.3.

4.2.2 Finding and Repairing Access-control Vulnerabilities

This section firsts give a high-level overview of how FixMeUp finds vulnerabilities, repairs them, and validates the repairs, and then we describe each step in more detail.

FixMeUp considers all security-sensitive operations in the program. Recall that each sensitive operation is associated with a particular user role (see Section 4.1). For each operation, FixMeUp computes all of its calling contexts. For each context, it considers all candidate checks, computes the corresponding access-control template ACT', and compares it with the role's access-control template ACT_{role}. If some context CC_{tgt} is missing the check, its ACT' will not match ACT_{role}. This context has an access-control vulnerability and FixMeUp targets it for repair.

To repair CC_{tgt}, FixMeUp inserts the code from ACT_{role} into the methods of CC_{tgt}. ACT_{role} contains the calling context CC_{src} of a correct access-control check and FixMeUp uses it to guide its interprocedural repair of CC_{tgt}. FixMeUp matches CC_{src} method by method against CC_{tgt}. At the last matching method m_{inline}, FixMeUp inlines all statements from the methods deeper in CC_{src} than m_{inline} into m_{inline}. We call this *adapting* the ACT to a target context. Adaptation produces a method map indicating, for each $m_{src} \in ACT_{role}$, the method $m_{tgt} \in CC_{tgt}$ where to insert statements from m_{src}.

For each statement in ACT_{role}, FixMeUp inserts statements from m_{src} into the corresponding m_{tgt} only if they are missing from m_{tgt}. In the simplest case, when a vulnerable context has only the entry method and no code that corresponds to any code in ACT_{role}, FixMeUp inserts the entire template into the entry method.

A repair can potentially introduce two types of undesired semantic

$method_a(C_0, ..., C_i)$	$method_b(C'_0, ..., C'_i)$	Match if (1) $method_a = method_b$ and (2) all constants $C_k = C'_k$
$localvar_a = C \in m_i$	$localvar_b = C' \in m_k$	Match if (1) $m_i = m_k$ or both methods are entry methods and (2) constants $C = C'$
$globalvar_a = C \in m_i$	$globalvar_b = C' \in m_k$	Match if (1) $globalvar_a = globalvar_b$ and (2) constants $C = C'$

Table 4.1: Matching statements without dependences

changes to the target code. First, statements already present in the target may affect statements inserted from the template. We call these *unintended changes to the inserted policy*. Second, inserted statements may affect statements already present in the target. We call these *unintended changes to the program*. Because our analysis keeps track of all data and control dependences and because our repair procedure carefully renames all variables, FixMeUp prevent most of these errors. As we show in Section 4.4, FixMeUp detects when template statements with side effects are already present in the program and does not insert them.

To validate that there are no unintended changes to an inserted policy, FixMeUp computes a fresh ACT from the repaired code and compares it with the adapted ACT. If they match, FixMeUp gives the repaired code to the developer; otherwise, it issues a warning.

```
isMatchingACT (ACT_x, ACT_y) {
1   // INPUT: two ACTs to be compared
2   // OUTPUT: true if ACT_x and ACT_y match, false otherwise
3
4   if(|ACT_x| ≠ |ACT_y|) return false;
5
6   VarMap ← φ
7   StatementMap ← φ
8   for( s_x ∈ ACT_x in order ) {
9       if( ∃only one (s_x, s_y) s.t. s_y ∈ ACT_y and isMatching(s_x, s_y) ) {
10          StatementMap ← StatementMap ∪ {(s_x, s_y)}
11      } else {
12          return false;
13      }
14  }
15  return true;
16 }
isMatching (s_src, s_tgt) {
1   // INPUT: statements s_src ∈ ACT, s_tgt ∈ m_tgt to be compared
2   // OUTPUT: true if s_src and s_tgt match, false otherwise
3       VarMap: updated variable mappings
4
5   if( ∃(s_src, s_tgt) ∈ StatementMap ) return true
6
7   if (AST structures of s_src and s_tgt are equivalent) {
8     m_src ←method containing s_src ∈ ACT
9     DD_src ← {(s_src, d) s.t. s_src is data−dependent on d ∈ m_src}
10    DD_tgt ← {(s_tgt, d) s.t. s_tgt is data−dependent on d ∈ m_tgt}
11    if (DD_src ≡ φ and DD_tgt ≡ φ) {
12      // no data dependences
13      if ( s_src and s_tgt are one of the types described in Table 1 ) {
14        if (s_src = "v_x = C_x" and s_tgt = "v_y = C_y" and
15              constants C_x and C_y are equal) {
16          VarMap = VarMap ∪ {(v_x, v_y)}
17        }
18        return true
19      } else return false
20    } else if (|DD_src| == |DD_tgt|) {
21      if (∀(s_src, d_x) ∈ DD_src, ∃ (s_tgt, d_y) ∈ DD_tgt and (d_x, d_y) ∈ StatementMap) {
22        if (s_src = "v_x = ..." and s_tgt = "v_y = ...") {
23          VarMap = VarMap ∪ {(v_x, v_y)}
24        }
25        return true
26  } } }
27  return false
28 }
```

Figure 4.3: Matching access-control templates

4.2.2.1 Matching Templates

To find vulnerabilities and validate repairs, FixMeUp *matches* templates. In general, it is impossible to decide whether two arbitrary code se-

quences are semantically equivalent. Matching templates is tractable, however, because ACTs of real-world applications are loop-free and consist of a small number of assignments, method invocations, and conditional statements. Furthermore, when developers implement the same access-control policy in multiple places in the program, they tend to use structurally identical code which simplifies the matching process.

Figure 4.3 shows our template matching algorithm and the statement matching algorithm that it uses. The latter algorithm compares statements based on their data and control dependences, and therefore the syntactic order of statements does not matter. Matching is conservative: two matching templates are guaranteed to implement the same logic.

Let ACT_x and ACT_y be two templates. For every $s_x \in ACT_x$, FixMeUp determines if there exists only one matching statement $s_y \in ACT_y$, and vice versa. The developers may use different names for equivalent variables in different contexts, thus syntactic equivalence is too strict. Given statements $s_x \in ACT_x$ and $s_y \in ACT_y$, FixMeUp first checks whether the abstract syntax tree structures and operations of s_x and s_y are equivalent. If so, s_x and s_y are syntactically isomorphic, but can still compute different results. FixMeUp next considers the data dependences of s_x and s_y. If the dependences also match, FixMeUp declares that the statements match. Table 4.1 shows the matching rules when neither statement has any dependences.

4.2.2.2 Finding Access-control Vulnerabilities

For each security-sensitive operation (*sso*), FixMeUp computes the tree of all calling contexts in which it may execute by (1) identifying all methods that may directly invoke *sso* and (2) performing a backward, depth-first pass over the call graph from each such method to all possible program entries. FixMeUp adds each method to the calling context once, ignoring cyclic contexts, because it only needs to verify that the access-control policy is enforced once before *sso* is executed.

For each calling context CC in which *sso* may be executed, FixMeUp first finds candidate access-control checks. A conditional statement b is a candidate check if it (1) controls whether *sso* executes or not, and (2) is syntactically equivalent to the correct check given by the ACT_{role}. For each such b, FixMeUp computes its slice, converts it into ACT_b using the algorithms in Figure 4.2, and checks whether ACT_b matches ACT_{role}. If so, this context already implements correct access-control logic. Otherwise, if there are no candidate checks in the context or if none of the checks match the correct check, the context is vulnerable and FixMeUp performs the repair.

4.2.2.3 Applying The Template

Formally, $CC_{src} = \{(cs_1, m_0) \dots (check, m_n)\}$, $CC_{tgt} = \{(cs'_1, m'_0) \dots (sso, m'_l)\}$, where $cs_{i+1} \in m_i$, $cs'_{i+1} \in m'_i$ are the call sites of m_{i+1}, m'_{i+1} respectively. For simplicity, we omit the subscript from ACT_{role}.

```
AdaptACT (ACT_src, CC_tgt) {
1    // Adapt ACT_src to the target context CC_tgt
2
3    ACT ←clone ACT_src
4    CC_src = ACT.CC_src
5    l ← 0
6
7    for ( i = 0; i < |CC_src|; i++ ) {
8       // iterate from the entry to the bottom method in CC_src
9       m_i ← i^th method in CC_src
10      m_tgt ← i^th method in CC_tgt
11      if( m_i and m_tgt are entries or m_i == m_tgt ) {
12        MethodMap ← MethodMap ∪ {(m_i, m_tgt)}
13        l ← i
14      } else break;
15   }
16   m_inline ← l^th method in CC_tgt
17   for ( k = l+1 ; k < |CC_src|; k++ ) {
18      inline method m_k from CC_src into m_inline in ACT
19      MethodMap ← MethodMap ∪ {(m_k, m_inline)}
20   }
21   return ACT
22 }
```

Figure 4.4: Adapting ACT to a particular calling context

FixMeUp uses *DoRepair* in Figure 4.5 to carry out a repair. It starts by adapting ACT to the vulnerable calling context CC_{tgt}. If CC_{tgt} already invokes some or all of the methods in ACT, we do not want to repeat these calls because the policy specifies that they should be invoked only once in a particular order. After eliminating redundant method invocations, FixMeUp essentially inlines the remaining logic from ACT into $ACT_{adapted}$.

Formally, the algorithm finds common method invocations in CC_{src} and CC_{tgt} by computing the deepest $m_{inline} \in CC_{src}$ such that for all $i \leq inline$ m_i matches m'_i. For $i = 0$, m_0 and m'_0 match if they are both entry methods. For $i \geq 1$, m_i and m'_i match if they are invocations of exactly the same method. The first for loop in *AdaptACT* from Figure 4.4 performs this process.

```
DoRepair (ACT, CC_tgt) {
1    // INPUT
2       ACT: access−control template specification
3       CC_tgt = {(cs'_1, m'_0), (cs'_2, m'_1) ... (sso, m'_n)}: calling context of the vulnerable security−sensitive
              operation sso
4    // OUTPUT
5       RepairedAST: repaired program AST
6       MatchCount: number of ACT statements already in the target
7
8    MethodMap ← φ // Initialize maps between ACT and target context
9    StatementMap ← φ
10   VarMap ← φ
11
12   ACT_adapted = AdaptACT (ACT, CC_tgt)
13   (RepairedAST, InsertedCheck, MatchCount) ←
14         ApplyACT (ACT_adapted, CC_tgt)
15   if (ValidateRepair (ACT_adapted, InsertedCheck)) {
16      return (RepairedAST, MatchCount)
17   }
18   return warning
19 }
ValidateRepair (ACT_orig, InsertedCheck) {
1    // INPUT
2       ACT_orig: applied access−control template
3       InsertedCheck: inserted access−control check
4    // OUTPUT:
5       true if extracted ACT from the repaired code matches ACT_orig
6
7    SEEDS ← {InsertedCheck, exit branch of InsertedCheck}
8    newSLICE ← doSlicing (SEEDS)
9    newCC ← calling context of InsertedCheck
10   ACT_repair ← GetACT (newSLICE, newCC)
11   return isMatchingACT (ACT_orig, ACT_repair)
12 }
```

Figure 4.5: Repairing vulnerable code and validating the repair

The algorithm then adapts ACT to CC_{tgt} by inlining the remaining statements—those from the methods deeper than m_{inline} in ACT—into m_{inline}. The second for loop in *AdaptACT* from Figure 4.4 performs this process and produces $ACT_{adapted}$. While matching methods and inlining statements, FixMeUp records all matching method pairs (m_i, m'_i), including m_{inline}, in $MethodMap$. In the simplest case, the entry $m'_0 \in CC_{tgt}$ is the only method matching $m_{inline} = m_0$. In this case, FixMeUp inlines every statement in

ACT below m_0 and produces a flattened $ACT_{adapted}$.

Otherwise, consider the longest matching method sequence $(m_0 \dots m_{inline})$ and $(m'_0 \dots m'_{inline})$ in CC_{src} and CC_{tgt}, respectively. For $1 \leq i \leq inline - 1$, m_i and m'_i are exactly the same; only m_0 and m_{inline} are distinct from m'_0 and m'_{inline}, respectively. $AdaptACT$ stores the (m_0, m'_0) and (m_{inline},m'_{inline}) mappings in $MethodMap$.

FixMeUp uses the resulting template $ACT_{adapted}$ to repair the target context using the $ApplyACT$ algorithm in Figure 4.6. This algorithm respects the statement order, control dependences, and data dependences in the template. Furthermore, it avoids duplicating statements that are already present in the target methods.

The algorithm iterates m_{src} over m_0 and m_{inline} in $ACT_{adapted}$ because, by construction, these are the only methods that differ between the template and the target. It first initializes the insertion point ip_{tgt} in m_{tgt} corresponding to m_{src} in $MethodMap$. The algorithm only inserts statements between the beginning of m_{tgt} and the sensitive operation sso, or—if m_{tgt} calls other methods to reach sso—the call site of the next method in the calling context of sso. Intuitively, the algorithm only considers potential insertion points and matching statements that precede sso.

Before FixMeUp inserts a statement s, it checks if there already exists a matching statement $s' \in m_{tgt}$. If so, FixMeUp adds s and s' to $StatementMap$, sets the current insertion point ip_{tgt} to s', and moves on to

```
ApplyACT (ACT, CC_tgt) {
1   // Insert statements only in entry and/or last method of CC_tgt that matches a method from adapted
        ACT. Other methods match ACT exactly (see AdaptACT).
2   // INPUT
3       ACT: access−control template
4       CC_tgt = {(cs'_1, m'_0), (cs'_2, m'_1) ... (sso, m'_n)}: calling context of the vulnerable sensitive operation sso
5   // OUTPUT
6       RepairedAST: AST of the repaired code
7       InsertedCheck: inserted access−control check
8       MatchCount: number of ACT statements found in the target
9
10  MatchCount ← 0
11  InsertedCheck ← null
12  m_0 ← the entry of ACT
13  m_inline ← the method containing check in ACT
14  for ( m_src ∈ {m_0, m_inline} ) {
15    ip_tgt ← null
16    m_tgt ← MethodMap(m_src)
17    for ( s ∈ ACT(m_src) in order ) {
18      // Is there a statement after ip_tgt in m_tgt that matches s?
19      s' ← FindMatchingStmt(s, ip_tgt, m_tgt)
20      if (s' ≠ null) { // target method already contains s
21        ip_tgt ← s'
22        MatchCount++
23      } else { // no match, insert s into target
24       (t, d) ← a pair of statement t and direction d s.t. s is immediately control−dependent on t in d
25       s' ← RenameVars(s, m_tgt) // rename variables in s for m_tgt
26       if (s' is a conditional statement ) { // add two branches
27          add true and false branches to s' with empty statements
28          if (s is the access control check)
29             InsertedCheck ← s'
30       }
31       if (ip_tgt == null) {
32          insert s' at the first statement in m_tgt
33       } else if (t ≠ null) { // s is immediately control−dependent on t
34          // insert on the corresponding conditional branch
35          t' ← StatementMap(t)
36          insert s' at the last statement on branch d of t'
37       } else { insert s' immediately after ip_tgt in m_tgt }
38       ip_tgt ← s'
39       StatementMap ← StatementMap ∪ {(s, s')}
40  } } }
41  RepairedASTs ← all modified ASTs of m_tgt ∈ MethodMap
42  return (RepairedASTs, InsertedCheck, MatchCount)
43 }
```

Figure 4.6: Applying an access-control template

the next statement. Otherwise, it inserts s as follows:

1. Transform s into s' by renaming variables.

2. If s is a conditional, insert empty statements on the true and false

branches of s'.

3. If ip_{tgt} has not been set yet, insert s' at the top of m_{tgt}.
4. Otherwise, if s is immediately control-dependent on some conditional statement t, insert s' as the last statement on the statement list of the matching branch of the corresponding conditional $t' \in m_{tgt}$.
5. Otherwise, insert s' after ip_{tgt}, i.e., as the next statement on the statement list containing ip_{tgt}. For example, if ip_{tgt} is an assignment, insert s' as the next statement. If ip_{tgt} is a conditional, insert s' after the true and false clauses, at tremediathe same nesting level as ip_{tgt}.
6. Add (s, s') to $StatementMap$ and set ip_{tgt} to s'.

ApplyACT returns the repaired AST, the inserted check, and the number of reused statements.

Variable renaming. When FixMeUp inserts statements into a method, it must create new variable names that do not conflict with those that already exist in the target method. Furthermore, because FixMeUp, when possible, reuses existing statements that match statements from the ACT semantically (rather than syntactically), it must rename variables. Lastly, as the algorithm establishes new names and matches, it must rewrite subsequent dependent statements to use the new names. The *isMatching* function in Figure 4.3 establishes a mapping between a variable name from the template and a variable name from the target method when it matches assignment statements.

As FixMeUp inserts subsequent statements, it uses the variable map to replace the names from the template (see Figure 4.7). Before *ApplyACT*

inserts a statement, it calls *RenameVars* to remap all variable names to the names used by the target method. For unmapped variables, *RenameVars* creates fresh names that do not conflict with the existing names.

Dealing with multiple matching statements. In theory, there may exist multiple statements in m_{tgt} that match s and thus multiple ways to insert $ACT_{adapted}$ into the target context. Should this happen, FixMeUp is designed to exhaustively explore all possible matches, generate the corresponding candidate repairs, and validate each candidate. FixMeUp picks the validated candidate that reuses the most statements already present in the target and suggests it to the developer.

4.2.2.4 Validating Repairs

As mentioned previously, FixMeUp can potentially introduce two types of semantic errors into the repaired program: (1) *unintended changes to the inserted policy*, and (2) *unintended changes to the program*. Unintended changes to the inserted policy may occur when existing statements change the semantics of the inserted code. Unintended changes to the program may occur when the inserted code changes the semantics of existing statements.

To detect type (1) errors, FixMeUp computes afresh an ACT from the repaired code and compares it—using *ValidateRepair* from Figure 4.5—with the ACT on which the repair was based. An ACT captures all control and data dependences. Any interference from the existing statements that affects the inserted code must change the dependences of the inserted statements.

```
FindMatchingStmt(s, ip_tgt, m_tgt) {
1    //INPUT:
2      s: statement in ACT
3      ip_tgt: last inserted statement in m_tgt
4
5    if (m_tgt contains the sensitive operation sso)
6      SL = { statements in m_tgt after ip_tgt that dominate sso}
7    else
8      SL = { statements in m_tgt after ip_tgt that dominate the callsite of next method in CC_tgt }
9    for(t ∈ SL) {
10     if (isMatching(s, t)) {
11       StatementMap ← StatementMap ∪ {(s, t)}
12       return t
13     }
14     // If multiple statements in SL match s, they are handled as described in Section 5.3
15   }
16   return null
17 }
RenameVars (s, m_tgt) {
1    // INPUT: s ∈ ACT, target method m_tgt
2    // OUTPUT: s' with variables remapped, updated VarMap
3    s' ← clone s
4    if (s = "v_ACT = ..." and v_ACT is local) {
5      if (∄t s.t. (v_ACT, t) ∈ VarMap) {
6        VarMap ← VarMap ∪ {(v_ACT, v_new)}
7    }}
8    for (v ∈ s') {
9      if (∃(v, v_new) ∈ VarMap)
10       replace v with v_new in s'
11   }
12   return s'
13 }
```

Figure 4.7: Matching statements and renaming variables

For example, suppose the reused statement has dependent statements already in the program that are not part of the ACT. In this case, the ACTs will not match and FixMeUp will issue a warning. This validation procedure guarantees that reusing an existing statement is always safe. We examined all 38 repairs suggested by FixMeUp for our benchmarks (see Section 4.4) and in only one case did the insertion of the repair code change the ACT semantics. FixMeUp's validation algorithm detected this inconsistency and issued a warning.

With respect to type (2) errors, unintended changes to the program, observe that the actual purpose of the repair is to change the program's semantics by adding access-control logic. FixMeUp therefore cannot guarantee that the repaired program is free from type (2) errors because it cannot know the full intent of the programmer.

The purpose of repair is to introduce a new dependence: all statements after the inserted access-control check become control-dependent on the check, which is a desired semantic change. Because FixMeUp inserts the check along with the statements defining the values used in the check, the inserted access-control logic may change both control and data dependences of statements that appear after the check. Our repair procedure minimizes the risk of unintended dependences by reusing existing statements as much as possible and by renaming all variables defined in the template to fresh names, thus preventing unintended dependences with the variables already present in the program. In just one of the 38 repairs on our benchmarks (see Figure 4.11 in Section 4.4) did an incorrectly annotated role cause FixMeUp to "repair" a context that already implemented a different access-control policy and thus introduce unwanted changes to the program.

4.3 Limitations

Good program analysis and transformation tools help developers produce correct code. They are especially useful for subtle semantic bugs such as inconsistent enforcement of access-control policies, but developers must still

be intimately involved in the process. The rest of this section discusses the general limitations of any automated repair tool and the specific limitations of our implementation.

Programmer burden. Suggesting a repair, or any program change, to developers requires some specification of correct behavior. We rely on developers to annotate access-control checks and security-sensitive operations in their applications and tag them with the corresponding user role. We believe that this specification burden is relatively light and, furthermore, it can be supported by policy inference tools, such as the technique presented in chapter 3. We require that the specifications be consistent for all security-sensitive operations in a given role. If the programmer wants different checks in different contexts for the same operation, the specification won't be consistent and our approach will attempt to conservatively over-protect the operation. For example, Figure 4.8 shows that FixMeUp inserts a credential check performed in one context into a different context that already performs a CAPTCHA check, in this case introducing an unwanted duplicate check. Developers should always examine suggested repairs for correctness.

We believe that the consequences of access-control errors are sufficiently dire to motivate the developers to bear this burden in exchange for suggested code repairs, since it is easier to reject or manually fix a suggested change than it is to find the error and write the entire repair by hand. The latter requires systematic, tedious, error-prone examination of the entire program and its call graph. Language features of PHP, such as the absence of a proper

module system, dynamic typing, and *eval*, further complicate this process for PHP developers. The number of errors found by FixMeUp in real-world PHP applications attests to the difficulty of correctly programming access control in PHP.

Static analysis. FixMeUp uses a standard static interprocedural data- and control-dependence analysis to extract the program slice representing the access-control logic. Because this analysis is conservative, the slice could contain extraneous statements and therefore would be hard to apply as a transformation. Program slicing for more general debugging purposes often produces large slices [70]. Fortunately, access-control policies are typically self-contained and much more constrained. They consist of retrieving stored values into local variables, checks on these variables, and code that exits or restarts the program after the check fails. Consequently, access-control templates tend to be short (see Table 4.2).

Our FixMeUp prototype does not handle all of the dynamic language features of PHP, nor does it precisely model all system calls with external side effects. In particular, the analysis resolves dynamic types conservatively to build the call graph, but does not model eval or dynamic class loading, which is unsound in general. In practice, only myBB uses eval and we manually verified that there are no call chains or def-use chains involving eval that lead to security-sensitive operations, thus eval does not affect the computed ACTs.

Static analysis can only analyze code that is present at analysis time. PHP supports dynamic class loading and thus potentially loads classes our

code does not analyze. However, our benchmarks use dynamic class loading in only a few cases, and we did analyze the classes they load. To handle these cases, we annotated 18 method invocations with the corresponding dynamic methods to generate a sound call graph that includes all possible call edges.

Our analysis models database connections such as open, close, and write, file operations that return file descriptors., but it does not perform symbolic string analysis on the arguments. This is a possible source of imprecision. For example, consider two statements: writeData("a.txt",$data) and $newdata = readData($b). If $b is "a.txt", the second statement is data-dependent on the first. A more precise algorithm would perform symbolic analysis to determine if the two statements may depend on each other and conservatively insert a dependence edge. Although this omissions makes our analysis unsound in general, in practice, we never observed these types of dependences. Therefore, even a more conservative and precise analysis would have produced the same results on our benchmarks.

Statement matching is weaker than semantic equivalence. For example, our matching algorithm does not capture that statements $a = b + c$ and $a = add(b, c)$ are equivalent. Another minor limitation of our matching algorithm is the use of coarse-grained statement dependences instead of variable def-use chains on the remapped variable names. A more precise algorithm would enforce consistency between the def-use information for each variable name var_x used in s_x and var_y used in s_y, even if the names are not the same given the variable mapping produced thus far. The current algorithm may yield a

match with an inconsistent variable mapping in distinct statements and thus change the def-use dependences at the statement level. We never encountered this problem in practice and, in any case, our validation procedure catches errors of this type.

4.4 Evaluation

We evaluate FixMeUp on ten open-source interactive PHP Web applications, listed in Table 4.2. We chose SCARF, YaPiG, AWCM, minibloggie, and DNscript because they were analyzed in prior work on detecting access-control vulnerabilities [66, 74]. Unlike FixMeUp, none of the previous techniques repair the bugs they find. In addition to repairing known vulnerabilities, FixMeUp found four new vulnerabilities in AWCM 2.2 and one new vulnerability in YaPiG that prior analysis [74] missed. We added Newsscript and phpCommunityCal to our benchmarks because they have known access-control vulnerabilities, all of which FixMeUp repaired successfully. To test the scalability of FixMeUp, we included two relatively large applications, GRBoard and myBB. Table 4.2 lists the lines of code (LoC) and total analysis time for each application, measured on a Linux workstation with Intel dual core 2.66GHz CPU with 2 GB of RAM. Analysis time scales well with the number of lines in the program.

Our benchmarks are typical of server-side PHP applications: they store information in a database or local file and manage it based on requests from Web users.

Web applications	LoC	Analysis time (s)	Role tag	ACT instances	ACT LoC	missing checks	alternative policies	inserted policies partial	inserted policies full	inserted policies warn	unwanted side effects
minibloggie 1.1	2,287	26	admin	2	6	1	0	0	0	1	0
DNscript	3,150	22	admin	14	4	3	0	0	3	0	0
			normal	8	4	1	1	1	1	0	0
Events Lister 2.03	2,571	24	admin	9	4	2	1	0	3	0	0
Newsscript 1.3	2,635	65	admin	1	8	1	0	1	0	0	0
SCARF (before patch)	1,490	40	admin	4	4	1	0	1	0	0	0
			normal	1	4	0	0	0	0	0	0
YaPiG 0.95	7,194	250	admin	3	5	0	0	0	0	0	0
			normal	3	11	1	1	2	0	0	1
phpCommunityCal 4.0.3	12,298	367	admin	5	8	12	0	12	0	0	0
AWCM 2.2	11,877	1221	admin	38	8	0	0	0	0	0	0
			normal	8	4	4	3	6	1	0	0
GRBoard 1.8.6.5	50,491	1742	admin	14	4	2	0	1	1	0	0
			normal	9	4	3	1	4	0	0	0
myBB 1.6.7	107,515	5133	admin	38	2	0	0	0	0	0	0
			normal	31	8	0	0	0	0	0	0
totals						31	7	28	9	1	1

Table 4.2: PHP benchmarks, analysis time in seconds, ACT characterization, and repair characterization

Table 4.2 shows that four applications have a single access-control policy that applies throughout the program. The other six have two user roles each and thus two role-specific policies. Policies were specified by manual annotation. They are universal, i.e., they prescribe an access-control check that must be performed in all contexts associated with the given role.

FixMeUp finds 38 access-control bugs, correctly repairs 30 instances, and issues one warning. Nine of the ten benchmarks contained bugs. Seven bugs were previously unknown. As mentioned above, five of the previously unknown bugs appeared in applications that had been analyzed in prior work which missed the bugs. Five of the ten applications implement seven correct, but alternative policies in some of their contexts (i.e., these policies differ from the policy in the template).

The fourth and fifth columns in Table 4.2 characterize the access-control templates; the third column lists the user role to which each policy applies. Six applications have two policies, *admin* or *normal*. The fourth column shows the total instances of the template in the code, showing that developers often implement the same access-control logic in multiple places in the program. For example, the DNscript application has two roles and thus two role-specific access-control policies. Out of the 22 templates in DNscript, only 2 are unique. The "LoC" column shows the size of each template (in AST statements). The templates are relatively small, between 2 and 11 statements each.

The "missing checks" and "alternative policies" columns in Table 4.2 show that FixMeUp finds a total of 38 missing checks. The "alternative

policies" column shows that in seven cases FixMeUp inserts an access-control policy, but that the target code already has a *different* check.

AddDn.php

```
1 <?
2 session_start();
3 if (!$_SESSION['member']) {
4     header('Location: login.php');
5     exit;
6 } ...
7 ?>
```

Process.php

```
1 <?
2 session_start(); // existing statement
3 if (!$_SESSION['member']) { // [FixMeUp repair]
4     header('Location: login.php'); // [FixMeUp repair]
5     exit; // [FixMeUp repair]
6 }
7 ...
8 $number = $_POST['image'];
9 if(md5($number) != $_SESSION['image_random_value']) {
10   echo 'Verification does not match.. Go back and refresh your browser and
         then retype your verification';
11   exit();
12 }
13 \?>
```

Figure 4.8: *DNscript*: Different access-control checks within the same user role

The "inserted polices" columns shows that FixMeUp made 37 validated repairs with one warning, 30 of which fixed actual vulnerabilities. For the other 7, the program already contained alternative logic for the same role (e.g., CAPTCHA vs. login). The case that generated the warning is shown in Figure 4.9. FixMeUp only inserts statements that are missing from the target. In minibloggie, the statements session_start() and dbConnect() are both in the template and in *Del.php*, thus FixMeUp does not insert them. It only inserts the missing statement if (!verifyuser()) {header ('Location: ./login.php');}. The access-control check at line 10, however, now depends on the conditional

at line 7. This dependence did not exist in the original ACT and does not pass FixMeUp validation.

Attempted repair of Del.php

```
1  <? ...
2  session_start(); // existing statement
3  ...
4  if ($confirm=="") {
5     notice("Confirmation", "Warning : Do you want to delete this post ? <a
          href=del.php?post_id=".$post_id."&confirm=yes>Yes</a>");
6  }
7  elseif ($confirm=="yes") {
8     dbConnect(); // existing statement
9
10    if (!verifyuser()) { // [FixMeUp repair]
11       header('Location: ./login.php'); // [FixMeUp repair]
12     die; // [FixMeUp repair]
13    }
14
15    $sql = "DELETE FROM blogdata WHERE post_id=$post_id";
16    $query = mysql_query($sql) or die("Cannot query the database.<br>" .
          mysql_error());
17    $confirm ="";
18    notice("Del Post", "Data Deleted");
19 }
20 ?>
```

Access-control template of minibloggie

```
1  <?
2  1. ProgramEntry
3  include "conf.php";
4  include_once "includes.php";
5  session_start();
6  dbConnect();
7  if (!verifyuser()) {
8    header('Location: ./login.php');
9  }
10 ?>
```

Figure 4.9: minibloggie: Attempted repair

The "partial" and "full" columns shows that, in 28 of 38 attempted repairs, FixMeUp reused some of the existing statements in the target when performing the repair, and only in 9 cases did it insert the entire template. This reuse demonstrates that repairs performed by FixMeUp are not simple clone-and-patch insertions, and adapting the template for each target is critical

to successful repair.

Figure 4.10 shows repairs to GRBoard in *remove_multi_file.php* and *swfupload_ok.php*. These two files implement different access-control logic to protect role-specific sensitive operations. Observe that $GR variable in *swfupload_ok.php* is not renamed and the existing variable is used instead, i.e., $GR = new COMMON() at line 4. On the other hand, in *remove_multi_file.php*, FixMeUp defines a new variable $GR_newone to avoid unwanted dependences when it inserts this statement.

Correct repair of remove_multi_file.php

```
1  <?
2  include('class/common.php'); // [FixMeUp repair]
3  $GR_newone = new COMMON(); // [FixMeUp repair]
4  if (($_SESSION['no'] != 1)) { // [FixMeUp repair]
5    $GR_newone->error('Require admin priviledge', 1, 'CLOSE'); // [FixMeUp
         repair]
6  }
7
8  if(!$_POST['id'] || !$_POST['filename']) exit();
9  $_POST['id'] = str_replace(array('../', '.php'), '', $_POST['id']);
10 $_POST['filename'] = str_replace(array('../', '.php'), '', $_POST['
       filename']);
11 //@SSO('admin')
12 @unlink('data/'.$_POST['id'].'/'.$_POST['filename']);
13 ...
14 ?>
```

Correct repair of swfupload_ok.php

```
1  if (isset($_POST["PHPSESSID"])) session_id($_POST["PHPSESSID"]);
2
3  include 'class/common.php'; // existing statement
4  $GR = new COMMON(); // existing statement
5  if (!$_SESSION['no']) { // [FixMeUp repair]
6    $GR->error('Require login procedure'); // [FixMeUp repair]
7  }
8  ...
9  if(time() > 600+@filemtime($tmp)) $tmpFS = @fopen($tmp, 'w'); else $tmpFS
       = @fopen($tmp, 'a');
10 //@SSO('member')
11 @fwrite($tmpFS, $saveResult);
12 @fclose($tmpFS);
```

Figure 4.10: GRBoard: Same ACT in different contexts

slidesshow.php

```
1  ...
2  $gid=$_GET['gid']; //existing statements
3  $form_pw_newone = $_POST['form_pw']; // [FixMeUp repair]
4  ....
5  if (!check_admin_login()) {// [FixMeUp repair]
6    if ((strlen($gid_info['gallery_password']) > 0)) { // [FixMeUp repair]
7      // @ACC('guest')
8      if (!check_gallery_password( $gid_info['gallery_password'],
           $form_pw_newone)) { // [FixMeUp repair]
9       include( $TEMPLATE_DIR . 'face_begin.php.mphp'); // [FixMeUp repair]
10      error(_y('Password incorrect.')); // [FixMeUp repair]
11 }  }  }
12 ....
13 if (!check_gallery_password($gid_info['gallery_password'],$form_pw)){
14     include( $TEMPLATE_DIR . 'face_begin.php.mphp');
15     error(_y("Password incorrect."));
16 }
```

Figure 4.11: YaPiG: Attempted repair

Figure 4.8 also shows how FixMeUp leaves line 2 intact in *process.php* when applying the template based on *AddDn.php*. This reuse is crucial for correctness. Had FixMeUp naively inserted this statement from the template rather than reuse the existing statement, the redundant, duplicated statement would have introduced an unwanted dependence because this function call has a side effect on the $\$\_SESSION$ variable. Because of statement reuse, however, this dependence remains exactly the same in the repaired code as in the original.

The last column demonstrates that the inserted statements in 37 repair instances introduce no unwanted dependences that affect the rest of the program. Figure 4.11 shows one instance where a repair had a side effect because of an already present alternative policy. Line 13 shows an access-control check already present in *slidesshow.php*. Because the policy implemented by the existing check does not match the ACT that prescribes additional checks for

the administrator role, FixMeUp inserts Line 3-11. However, the function call on Line 8 has a side effect on $\$\_SESSION$ and $\$\_COOKIE$ which are used in the function call at Line 13. This side effect is easy to detect with standard dependence analysis, but the reason it occurred is a faulty annotation: the access-control policy represented by the ACT should not have been applied to this context.

We reported the new vulnerabilities found by FixMeUp and they were assigned CVE candidate numbers: CVE-2012-2443, 2444, 2445, 2437 and 2438. We confirmed the correctness of our repairs by testing each program and verifying that it is no longer vulnerable. When an unauthorized user invokes the repaired applications through either an intended or unintended entry point and attempts to execute the sensitive operation, every repaired application rejects the attempt and executes the code corresponding to the failed check from the original ACT.

4.5 Conclusion

We presented FixMeUp, the first static analysis tool for automatically finding and repairing access-control bugs in Web applications. FixMeUp starts with an access-control policy that maps security-sensitive operations—such as database query sites and privileged file operations—to access-control checks that protect them from unauthorized execution. FixMeUp then automatically extracts the code responsible for access-control enforcement, uses it to create an access-control template, finds calling contexts where the check is

missing or is implemented incorrectly, and repairs the vulnerability by applying the template. FixMeUp successfully repaired 34 access-control bugs in 9 real-world PHP applications, demonstrating its practical utility.

Chapter 5

Detecting Code Injection Attacks on Web Applications

The previous two chapters addressed access-control vulnerabilities in which Web attackers cause unintended control flows to security-sensitive operations. This chapter addresses code injection vulnerabilities. A *code injection attack* occurs when a Web adversary manages to inject his/her own code into a program generated by the Web application. This attack is an unintended data flow. Injected code may steal data, compromise database integrity, and/or bypass authentication and access control, violating system correctness, security, and privacy properties.

Database queries generated by server-side Web applications are the classic target of code injection. However, the recent trend towards using NoSQL databases [46] instead of relational SQL databases is not reducing code injection threats. Many NoSQL databases, including MongoDB, CouchDB, and DynamoDB, use JSON and/or JavaScript as query languages, but this does not help protect NoSQL-based applications from code injection attacks. In 2010, Diaspora reported a serious NoSQL code injection vulnerability in its social community framework [47]. Code injection attacks on JavaScript

queries for MongoDB were also demonstrated at Black Hat 2011 [73].

By definition, a code injection attack on a Web application involves **tainted code**: the application generates a string that is interpreted as an executable program (e.g., an SQL or NoSQL query), and the string contains user input that is interpreted as code when the program executes. Preventing code injection attacks requires *precisely* determining (1) which parts of the generated string are *code*, and (2) which parts of the generated string are *tainted* by user input.

All prior approaches to runtime detection of code injection attacks suffer from two types of problems. They either fail to precisely define what constitutes code, or their taint analysis algorithm does not identify exactly which characters in the application-generated SQL string originate from user input and which originate from the application itself. Errors of both types lead to false positives (benign queries rejected) and false negatives (missing code injection attacks).

This chapter introduces Diglossia, a new runtime tool that precisely and efficiently detects code injection attacks. The key idea behind our approach is to transform the problem of detecting injected code into a string propagation and parsing problem.

To identify taints efficiently, Diglossia dynamically creates a shadow string for each query issued by the application P. In the shadow query, all application-generated parts use shadow characters, while all tainted parts—i.e.,

substrings originating from user input—use original characters. When P is invoked, DIGLOSSIA dynamically generates a set of shadow characters that occur in neither user input, nor the original query language. DIGLOSSIA then creates a one-to-one map from each character used by the query language to a unique shadow character. As P executes, DIGLOSSIA adds a shadow execution that computes *shadow values* for all strings computed by P that depend on user input. The shadow execution follows the control flow of P's execution and performs shadow operations only on input-dependent string and character array operations. In a shadow string, all characters c originating from P are remapped to shadow characters sc where $sc = map(c)$, while all characters originating from user input remain intact. Value shadowing is a precise, lightweight way to propagate character-level taint information, because it performs the same string and array operation as the program, but only for a subject of operation. We implement this functionality as a PHP interpreter extension that dynamically remaps characters and computes shadow values in tandem with the string and character array operations performed by P.

When P issues a query, DIGLOSSIA examines the original query and its shadow using a *dual parser*. Dual parsing is the key technical innovation in DIGLOSSIA. For any string accepted by the original query language, the dual parser accepts the same string, as well as strings in which the original characters are replaced with their corresponding shadow characters. DIGLOSSIA examines the parse trees of the actual query and its shadow and establishes the following two conditions:

1. There is a one-to-one mapping between the parse tree of the actual query and the parse tree of the shadow query. In particular, all code in the actual query maps exactly to equivalent code in the shadow query.

2. The shadow query does not contain any code in the original language L.

If either condition does not hold, DIGLOSSIA reports a code injection attack. Intuitively, the presence of any original-language code in the shadow query and/or any syntactic difference between the actual query and its shadow indicate a code injection attack.

We demonstrate the precision and efficiency of DIGLOSSIA on 10 open-source PHP Web applications that issue queries to relational MySQL and MongoDB NoSQL backend databases. DIGLOSSIA detects all 25 code injection attacks we attempted against these applications with no perceptible performance overhead.

By recasting the problem of detecting code injection attacks as a string propagation and parsing problem, we gain substantial improvements in efficiency and precision over prior work. DIGLOSSIA uses shadow values only to detect injected code and does not actually submit shadow queries to the database. Therefore, in contrast to SQL keyword randomization [6] and complementary encoding [40], DIGLOSSIA does not require any changes to Web applications, databases, query parsers, Web servers, or Web browsers.

		1	2	3	4	5	6	7	8	9	10	11
	Ray and Ligatti's definition of code injection [59]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No
Tools	Halfond et al. [22], Nguyen-Tuong et al. [45]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
	Xu et al. [81]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
	SQLCHECK [72]	Yes	No	No	Yes	No	No	No	No	No	No	No
	CANDID [3]	Yes	Yes	Yes	No	No	No	Yes	No	No	No	Yes
	DIGLOSSIA	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No

1 SELECT bal FROM acct WHERE pwd='' OR 1=1 - -'

2 SELECT balance FROM acct WHERE pin= exit()

3 ...WHERE flag=1000 > GLOBAL

4 SELECT * FROM properties WHERE filename='f.e'

5 ...pin=exit()

6 ...pin=aaaa()

7 SELECT * FROM t WHERE flag=TRUE

8 SELECT * FROM t WHERE flag=aaaa

9 SELECT * FROM t WHERE flag=password

10 CREATE TABLE t (name CHAR(40))

11 SELECT * FROM t WHERE name='x'

Table 5.1: Canonical code injection attacks and non-attacks misclassified by prior methods. Underlined terms are user input.

5.1 Motivation

This section starts with a definition of code in target query languages. It then explains common mistakes in detecting SQL injection attacks, which are originated from the absence of a strict "*code*" definition. It also introduces a syntax mimicry code-injection attack that is capable of bypassing prior syntax-based methods.

5.1.1 Pitfalls of Detecting SQL Injection Attacks

We illustrate SQL injection attacks using 11 canonical examples described by Ray and Ligatti [59]. Table 5.1 shows how five prior tools and DIGLOSSIA classify these cases. Underlined terms are user input. Below, we review each attack and non-attack on this list and explain how DIGLOSSIA improves over prior work.

1. SELECT bal FROM acct WHERE pwd='<u>' OR 1=1 - -</u>'

   This case is the classic SQL injection attack with a backquote that ends a string and injects user input as code into the query. All tools detect this code injection. DIGLOSSIA detects it because the injected code "OR", "=", and "-" appears in original characters in the shadow query.

2. SELECT balance FROM acct WHERE pin= <u>exit()</u>

   User input injects <u>exit()</u>, which is a built-in function call. SQLCHECK misclassifies this case because the function call is an ancestor of complete leaf nodes (injected) in the query's parse tree. DIGLOSSIA detects this

injection because *exit* is a bound variable (and, therefore, code), yet appears in original characters in the shadow query.

3. ...WHERE flag=1000>GLOBAL

   The injected ">" is code that SQLCHECK misses because, again, this input is correctly positioned in the parse tree. DIGLOSSIA detects it because > is code, yet appears in original characters in the shadow query.

4. SELECT * FROM properties WHERE filename='f.e'

   Even if *f.e* is an object reference, the quotes enforce its interpretation as a string. SQLCHECK strips off quotes and misclassifies *f.e* as a reference, generating a false positive. All other tools, including DIGLOSSIA, correctly classify this input as a string literal and not an injection.

5. ...pin=exit()

   All tools except DIGLOSSIA miss the injection of the exit identifier because they do not reason about bound names at all. DIGLOSSIA detects code injection because *exit* is bound (and, therefore, code), yet appears in original characters in the shadow query.

6. ...pin=aaaa()

   When the identifier is undefined, only DIGLOSSIA correctly detects code injection.

7. SELECT * FROM t WHERE flag=TRUE

   Since the injected TRUE is a literal value, this case is not an attack. CANDID incorrectly classifies this input as code injection because the TRUE

literal is parsed to a different terminal than the benign input "aaaa", which is parsed to an identifier. DIGLOSSIA correctly parses this input as a literal in both the actual query and its shadow, and does not report an attack.

8. SELECT * FROM t WHERE flag=aaaa

   This attack injects a bound identifier (equal to the benign input used by CANDID) into the query. It is missed by all prior methods. DIGLOSSIA detects code injection because *aaaa* is bound (and, therefore, code), yet appears in original characters in the shadow query.

9. SELECT * FROM t WHERE flag=password

   This attack injects a bound identifier into the query and is missed by all prior methods. DIGLOSSIA detects code injection because *password* is bound (and, therefore, code), yet appears in original characters in the shadow query.

10. CREATE TABLE t (name CHAR(40))

    DIGLOSSIA does not detect this case as code injection. Unlike Ray and Ligatti, we consider integer literals, even in SQL type definitions, to be values, thus this case is not an injection attack from our viewpoint.

11. SELECT * FROM t WHERE name='x'

    Since the injected 'x' is a string literal, this case is not an attack. CANDID uses 'aaa' instead of 'x' in the shadow execution; they are different terminals and CANDID incorrectly reports a code injection attack. Xu et al. classify this case as an attack because tainted meta-characters (quotes) appear in

the query. Halfond et al. also classify this case as an attack because quotes do not come from a trusted source. DIGLOSSIA, on the other hand, parses ‘x’ into a literal in both the actual query and its shadow, and correctly does not report an attack.

5.1.2 Syntax Mimicry Attacks

The query containing injected code need not be syntactically different from a benign query (we call such injections *syntax mimicry* attacks). Consequently, detection tools such as CANDID that look for syntactic discrepancies between the actual query and the query on a benign input will miss some attacks.

vulnerable.php

```
1  <?
2  // Build a JavaScript query that checks whether pwd field is the same as
       user input $_GET['id']
3
4  $query = "function q() { ";
5  $query .= "var secret_number = this.pwd;";
6  $query .= "var user_try = ' . $_GET['id'] . ';";
7  $query .= "if (secret_number!=user_try) return false;";
8  $query .= "return true;";
9  $query .= "}";
10
11 $collection->find(array('$where' => $query));
12 ?>
```

Attack URL

```
http://victimHost/vulnerable.php?id=secret_number
```

Figure 5.1: JavaScript syntax mimicry attack.

Figure 5.1 shows sample PHP code that builds a JavaScript query for a MongoDB. User input in $\$\_GET[\text{‘}id\text{’}]$ is supposed to be a numeric lit-

eral. If the attacker inputs *secret_number* instead of a number, the query will return "true", sabotaging the intended semantics. CANDID will use "aaaaaaaaaaaa" as the benign input for *secret_number* in its shadow execution and miss the attack, but DIGLOSSIA will detect it.

login.php in *minibill*

```
1 <?
2 $Q = "SELECT * FROM users
3       WHERE email='{$_REQUEST['email']}'
4       AND password='{$_REQUEST['password']}'
5       LIMIT 1";
6 $res = mysql_query($Q);
7 ?>
```

Attack URL

http://victimHost/login.php?email=no\&password=AND others='any

Actual query

SELECT * FROM users WHERE email='no\' AND password=' AND others='any' LIMIT 1

Query on a benign input

SELECT * FROM users WHERE email='aaa' AND password='aaaaaaaaaaaaaaa' LIMIT 1

Figure 5.2: SQL syntax mimicry attack on *minibill*.

Figure 5.2 shows *login.php* in *minibill*, an actual PHP program vulnerable to syntax mimicry attacks. The attack URL makes the syntactic structures of the actual and shadow queries equivalent. Observe, however, that the attack query refers to the *others* field instead of the intended *password* field. This particular attack may not seem damaging, but if the actual query had used *OR* instead of *AND*, the attack would have been much more serious.

Figure 5.3 shows another PHP program with an injection vulnerability (CVE-2013-0135). The attack URL results in this query resetting the passwords of users whose ZIP code is 77051. DIGLOSSIA can detect syntax

reset_password_save.php in *phpAddressBook* 8.2.5

```
1 <?
2 $password_hint = $_REQUEST['password_hint'];
3 $email=$_REQUEST['email'];
4 ...
5 // Assume that $cleanpw is "arbitrary"
6 $query = "UPDATE users SET password='$cleanpw', password_hint='
      $password_hint' WHERE email='$email'";
7
8 $res = mysql_query($query);
9 ?>
```

Attack URL exploiting CVE-2013-0135

```
http://victimHost/login.php?password_hint=no\&email=WHERE zip='77051
```

Actual query

```
UPDATE users SET password='arbitrary', password_hint='no\' WHERE email=' WHERE zip='77051'
```

Query on a benign input

```
UPDATE users SET password='arbitrary', password_hint='aaa' WHERE email='aaaaaaaaaaaaaaaa'
```

Figure 5.3: SQL syntax mimicry attack on *phpAddressBook*.

mimicry attacks such as this one because, unlike CANDID, it creates shadow queries from the *same input* as the actual execution. The syntactic structures of the actual and shadow queries are equivalent, but the shadow contains the code "WHERE" in original characters (since it originated from user input). Therefore, DIGLOSSIA reports an attack.

5.1.3 Defining code

As Section 5.1.1 shows, defining code simply as pre-specified keywords and operators does not provide a clear distinction between code and non-code. Instead, precisely identifying code and non-code requires parsing the query [59].

DIGLOSSIA accepts only values (numeric and string literals) and reserved values (NULL, TRUE, etc.) as *non-code*. *Code* comprises all reserved keywords, operators, and method calls, as well as all uses of bound identifiers (variables, types, and method names). Note that this definition forbids the dangerous programming practice where certain user inputs are intended by a developer to be interpreted as code in the query. In the absence of strict access control on database operations, this practice may lead to arbitrary code execution and should be deprecated.

5.2 Implementation

DIGLOSSIA is as an extension to the PHP interpreter. It is implemented in C using PECL (PHP Extension Community Library). The Web server invokes the interpreter automatically when the URL hosting a PHP application is accessed. DIGLOSSIA has three phases, as depicted in Figure 5.4 and described below.

Phase I creates a shadow character map and the dual parser.

Phase II computes a shadow value for each string that depends on user input.

Phase III detects injected code by examining and comparing the actual query string and its shadow.

Phase I creates a map from all characters c in the query language L to a disjoint set of shadow characters $SC = \{map(c)\}$. Phase I also creates

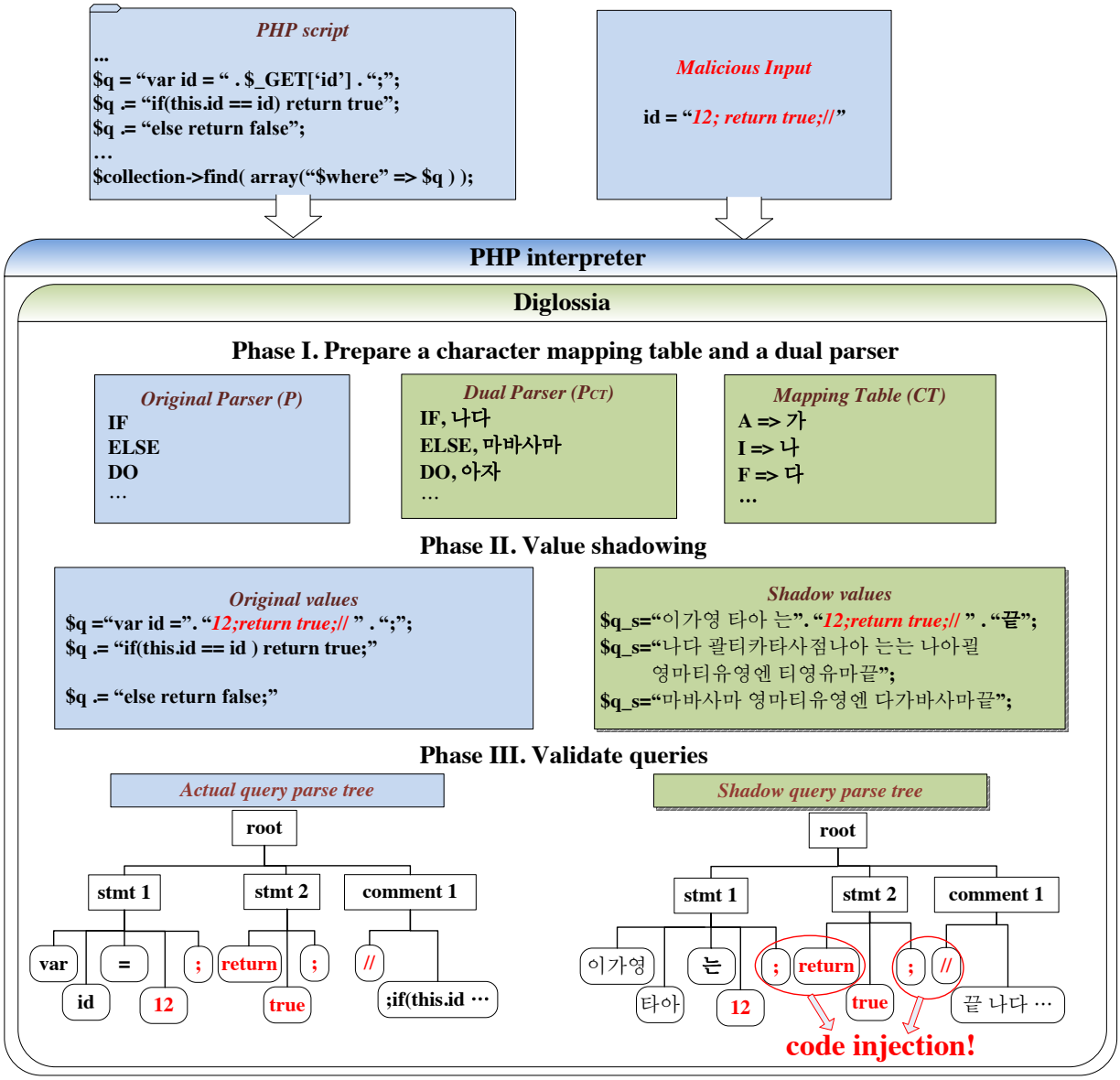

Figure 5.4: Overview of DIGLOSSIA.

the dual parser for the shadow language SL, which is a superset of L and described in more detail in Section 5.2.3.

In tandem with the execution of the application, Phase II creates and computes shadow values for all strings and array operations that depend on user input. When the Web server invokes a PHP application, DIGLOSSIA creates a shadow string value for each input string, exactly equal to that string. Therefore, at the beginning of the execution, all shadow values consist only of original characters. For every subsequent string or character array computation where one or both operands already have shadow values, DIGLOSSIA computes the shadow value for the result of the operation. If an operand does not have a shadow value, DIGLOSSIA creates a shadow value for it by remapping each character to the corresponding shadow character.

When the PHP application issues a query q, Phase III intervenes and checks whether the query includes injected code. To this end, DIGLOSSIA parses q and its shadow q' with the dual parser and checks the following two conditions.

First, there must exist a one-to-one mapping between the nodes in the respective parse trees of q and q'. Furthermore, each parse tree node in q' must be a shadow of the corresponding node in q, as defined in Section 5.2.3. For instance, a string literal node in q must map to a string literal node in q', except that the string in q only uses characters in C, whereas the string in q' may use characters in $C \cup SC$. This isomorphism condition ensures that *shadow characters in the shadow query correspond exactly to the untainted*

characters in the actual query.

Second, all code in the shadow query q' must use only the characters in SC, because all characters in C come from user input.

If both conditions are satisfied, DIGLOSSIA passes the original query q to the backend database. Otherwise, DIGLOSSIA stops the application and reports a code injection attack.

5.2.1 Character Remapping

We implemented character remapping and dual parsing for SQL, JSON, and JavaScript query languages. These languages use ASCII characters, found on standard English keyboards, for all keywords, numerals, identifiers (variables, types, method names, etc.) and special values (NULL, TRUE, etc.). Although the languages are different and DIGLOSSIA has a separate parser for each, we use the term "query language L" generically to simplify the exposition.

Let C be the subset of ASCII characters consisting of the lower- and upper-case English alphabet and special characters (DIGLOSSIA does not remap digits). Formally, C includes characters whose decimal ASCII codes are from 33 to 47 and from 58 to 126. DIGLOSSIA dynamically creates a one-to-one mapping from each character in C to a shadow UTF-8 character that occurs in neither C, nor user input. Observe that since L uses only characters from C, no shadow characters appear in code written in L.

UTF-8 is a variable-byte representation that uses one to four 8-bit bytes

to encode characters. The total number of UTF-8 characters is 1,112,064 and it is easy to find 84 characters among them that do not occur in user input. In our current implementation, every webpage request (i.e., every invocation of a PHP application) results in a different random map. To create this map, DIGLOSSIA (1) randomly selects two-byte shadow characters from among 1,112,064 possible UTF-8 characters, and (2) examines all variables holding user input (e.g., $POST$, GET, and $COOKIE$) to ensure that shadow characters do not occur in them.

It is also possible to pre-compute a set of random mappings offline to reduce runtime overhead.

5.2.2 Value Shadowing

As the application executes, DIGLOSSIA computes shadow values for the results of all string and character array operations that depend on user input. Because DIGLOSSIA is implemented using PECL, a part of a PHP interpreter, it can directly manage memory and monitor program statements during the application's execution.

DIGLOSSIA allocates shadow values on the heap and stores their addresses in a *shadow value table* indexed by the address of the memory location for the original value. For operations that do not involve user input, including all non-string, non-array operations, conditionals, branches, arithmetic operations, etc., DIGLOSSIA performs no computations or allocations. Therefore, the control flow of value shadowing follows the control flow of the application.

When a Web server invokes the PHP application, it passes in user inputs as strings. DIGLOSSIA allocates a shadow value for each input string, equal to the string itself, and adds this value to the shadow value table. If the application reads in additional user input, DIGLOSSIA repeats this process. These initial shadow values contain only characters from the original character set C.

Whenever the application performs a string or character array operation *lhs = operation(op1, op2)* where one or both operands (*op1* and *op2*) already have shadow values—and, therefore, the operation is data-dependent on user input—DIGLOSSIA computes the shadow value $shadow_{lhs}$ for the result as follows.

If one operand *op* does not already have a shadow value, DIGLOSSIA allocates a new shadow value and remaps each character in *op* to the corresponding shadow character, creating $shadow_{op}$. Given individual characters $c_i \in op$, $shadow_{op} = map(c_0) \,||\, \ldots \,||\, map(c_{n-1})$ where n is the length of *op*. This remapping guarantees that all characters introduced by the application itself are in the shadow character set, regardless of whether they appear in the application as explicit string literal constants, come from libraries, or are generated dynamically. DIGLOSSIA then computes

$$shadow_{lhs} = operation(shadow_{op1}, shadow_{op2}).$$

If *lhs* does not have an entry in the shadow value table, DIGLOSSIA allocates a shadow value and enters it in the table. DIGLOSSIA shadows

built-in PHP string and array operations. Built-in PHP string operations include string trim, string assignment, substring, concatenation, and replacement. Built-in PHP array operations include array merge, push, pop, and assignment.

Memory for shadow values is proportional to memory tainted by user input, and shadow computations are proportional to the number of program statements that depend on user input. The number of lookups for taint information is significantly smaller than in byte-level taint tracking methods. In value shadowing, the number of lookups is the same as the number of involved values; in contrast, the number of lookups in precise byte- and character-level taint tracking methods is proportional to the byte or character length of every value. Furthermore, fine-grained taint tracking methods require heavy augmentation of built-in operations on strings and bytes to precisely propagate taint information. In contrast, value shadowing performs only the same string and array operations on shadow values as the application performs on the actual values, which is lighter and more efficient.

Figure 5.4 shows an overview of our approach, in which we remap ASCII characters into Korean characters and use the latter to compute shadow values. In Figure 5.4, the assignment $\$q = \text{“}var\ id\ =\ \text{”}.\underline{\text{“}12; return\ true; //\text{”}}.\text{“;”};$ concatenates string constants with user input. We compute the shadow value as $\$q_s = map(\text{“}var\ id\ =\ \text{”}).\underline{\text{“}12; return\ true; //\text{”}}.map(\text{“;”});$. Observe that computing the shadow value involves the same concatenation operation on the shadow values as done in the original application. All strings originating from

user input remain the same, but string constants introduced by the application have been remapped to (in this case) Korean UTF-8 characters. DIGLOSSIA stores the resulting $\$q_s$ as the shadow of q and uses it for subsequent shadow operations.

```
// boxes show the shadow operations
1 $input = $_GET['input'];
$input_s = $_GET[‘input’];
2 $amount = $_GET['amount'];
$amount_s = $_GET[‘amount’];
3 $SQL = 'CCS13SELECT * FROM ';
$SQL_s = map(‘CCS13SELECT * FROM ’);
4 if ($input < 100) {
5  $SQL   = $SQL . 'small_numbers WHERE count < ' . $amount ;
$SQL_s = $SQL_s . map(‘small_numbers WHERE count < ’) . $amount_s;
6 } else if ($input > 200) {
7   $SQL = $SQL . 'large_numbers WHERE count > ' . $amount ;
$SQL_s = $SQL_s . map(‘large_numbers WHERE count > ’) . $amount_s;
8 } else {
9   $SQL = $SQL . 'middle_numbers WHERE count < ' . $amount ;
$SQL_s = $SQL_s . map(‘middle_numbers WHERE count < ’) . $amount_s;
10 }
11 $SQL = substr($SQL, 6); // trim five characters from the start
$SQL_s = substr( $SQL_s, 6 );

12 Interpose and  validate( $SQL, $SQL_s );
13 mysql_query( $SQL );
?>
```

Figure 5.5: An example of value shadowing.

Figure 5.5 illustrates how DIGLOSSIA computes shadow values. Given that $\$input$ is 150, this PHP application computes the $\$SQL$ string to be used as the query. $\$SQL_s$ is the shadow value of $\$SQL$. Let SO_i be the shadow operation corresponding to the ith line of the application (it is shown in the

gray box underneath the corresponding line). The full execution sequence comprises lines 1, SO_1, 2, SO_2, 3, SO_3, 4, 6, 9, SO_9, 11, SO_{11}, 12, and 13 in order. Observe that non-string, non-character-array operations are not shadowed.

Line 13 makes the database call with the query stored in string $\$SQL$. In this case, $\$SQL$ has a shadow value $\$SQL_s$ because the query depends on user input.

5.2.3 Detecting Injected Code

When the application issues a query q using calls such as *mysql_query*, *MongoCollection::find*, or *MongoCollection::remove*, DIGLOSSIA intervenes and compares q with its shadow q'. DIGLOSSIA checks that (1) q and q' are syntactically isomorphic, and (2) the code in the shadow query q' is not tainted. If either condition fails, it reports an attack. DIGLOSSIA performs both checks at the same time, using a *dual parser*.

Intuitively, the purpose of the dual parser is to analyze the shadow query using the grammar of the query language L, but taking into account the fact that the shadow query contains a mix of original and shadow characters. Value shadowing guarantees that all characters in q' that were introduced by the application are in the shadow character set, and all characters in q' that originate from user input are in the original character set.

We first formally define a new shadow language SL that is a superset of the original query language L. We then describe how we optimize our

implementation by re-using the parser for L to parse the shadow language SL.

Query language and grammar. Let $G = (N, \Sigma, R, S)$ be the context-free grammar of the query language L. N is the set of non-terminal states, representing operations, conditionals, expressions, etc. Σ is the set of terminal states, disjoint from N. We will use the symbol ϵ to refer to individual terminal states in Σ. R is the set of production rules that express the finite relation from N to $(N \cup \Sigma)^*$. $S \in N$ is the unique start symbol.

When the parser uses this grammar G to accept a program P, it produces a parse tree that maps every character in P to a terminal. Each terminal is either *code* or *non-code*. Code terminals include operations (e.g., "+" and "-"), keywords, bound identifiers, and method calls. Non-code terminals include constant literals, string literals, and reserved symbols (NULL, TRUE, etc.).

Shadow language and grammar. Given a query language L and its grammar G, DIGLOSSIA defines a corresponding shadow language SL and shadow grammar SG. As described in Section 5.2.1, every character c used in L has a corresponding shadow character sc. Characters in SL are drawn from $C \cup SC$, where C is the original character set and SC is the shadow character set.

We define $SG = (N, \Sigma_s, R_s, S)$ to be the grammar of the shadow language SL. N and S are the same as in G. For every terminal $\epsilon \in \Sigma$, there exists exactly one corresponding shadow terminal $\epsilon_s \in \Sigma_s$, defined as follows.

Let σ_ϵ be any string accepted by ϵ. If ϵ is an identifier or string literal, then, for each legal character c occurring in σ_ϵ, the shadow terminal ϵ_s accepts c or $map(c)$. In other words, any identifier or string literal from the original language L can be expressed in an arbitrary mixture of original and shadow characters in the shadow language SL. For these terminals, ϵ_s accepts a superset of ϵ.

For any other terminal ϵ in G, the corresponding shadow terminal ϵ_s accepts only σ_ϵ or $map(\sigma_\epsilon)$. In other words, any non-identifier, non-string-literal terminal in the shadow language must be expressed entirely in original characters, or else entirely in shadow characters. For instance, if the query language L contains a "SELECT" terminal, the shadow grammar will accept "map(SELECT) * FROM table", but not "SELEmap(CT) * FROM table". This restriction immediately rules out some injection attacks even before the security checks described below. For example, keywords that contain both original and shadow characters will not even parse.

For each production $rule \in R$, SG has a corresponding $rule_s \in R_s$. Formally, $rule$ has the form: $rule : n \rightarrow v_1 v_2 ... v_l$ where $n \in N, v \in N \cup \Sigma$. In $rule_s$, all non-terminals are the same as in $rule$, while the terminals ϵ_s are defined as above.

Consider the following example, where $rules$ are the rules from the original grammar, and $rules_s$ are the corresponding rules from the shadow grammar.

$$
\begin{array}{lll}
rules: & select\_stmt & \rightarrow SELECT\_term\ list\_exp\ table\_exp \\
 & SELECT\_term & \rightarrow SELECT \\
 & identifier & \rightarrow \{a|b|\ldots\} \\
 & \ldots & \\
rules_s: & select\_stmt & \rightarrow SELECT\_term\ list\_exp\ table\_exp \\
 & SELECT\_term & \rightarrow SELECT \mid map(SELECT) \\
 & identifier & \rightarrow \{a|b|\ldots|map(a)|map(b)|\ldots\} \\
 & \ldots &
\end{array}
$$

The example shows one non-terminal rule and two terminal rules. Since $select\_stmt \in G$ is a non-terminal rule, it is exactly the same in both grammars. The terminal rule for $SELECT\_term \in SG$ accepts both $SELECT$ and $map(SELECT)$, a superset of the original language, since $SELECT$ is a keyword. The terminal rule for $identifier \in SG$ accepts strings with an arbitrary mix of original characters c and the corresponding shadow characters $map(c)$.

Applying these simple transformations to the original language and parser, we create a shadow language and parser. Shadow production rules defined in this fashion do not add conflicts, thus the parser for SG produces a deterministic parse tree.

Each character map requires its own shadow grammar. Since a fresh map is dynamically generated for each page request (i.e., each invocation of a PHP application), automatically building a new parser for each execution would be expensive. Instead, DIGLOSSIA takes advantage of the fact that the non-terminals are the same in G and SG, and there is a one-to-one correspondence between the terminals. This enables DIGLOSSIA to re-use the parser for

G when parsing SG.

A parser is a function that chooses the next parsing state based on the current state and the input token. If a particular token t triggers a production rule in G (e.g., $SELECT\_term \in G$ in the example above), then the remapped token t_s triggers the corresponding rule in SG (e.g., $SELECT\_term \in SG$ in the example above). This feature enables DIGLOSSIA to use the same internal handle for both t and t_s, while extending the set of accepted characters. With this optimization, DIGLOSSIA can use the same parsing tables for all dynamically generated shadows of a given query language.

Using the dual parser to detect injected code. Let DP be the dual parser that can parse query strings according to either the original grammar G, or the shadow grammar SG defined above.

Given the actual query q issued by the application, DP parses it using G and generates a parse tree T. DP then parses the corresponding shadow query q' and generates a parse tree T'. If DP cannot produce a parse tree for either q or q', it rejects the query and reports a code injection attack.

Otherwise, DP compares the terminal nodes in the two parse trees, T and T', and checks the following two conditions:

1. For each node $t_i \in T$, there exists a one-to-one mapping to $t'_i \in T'$ and, furthermore, t'_i is the shadow of t_i. For example, if t_i is a particular code operator, then t'_i is the same code operator.

2. If t_i parses to a code terminal, then for every character $t_{ij} \in C$, there exists a one-to-one mapping from t_{ij} to the correct shadow character $t'_{ij} \in SC$ such that $t'_{ij} = map(t_{ij})$, where map is the shadow character map.

If either condition is violated, DIGLOSSIA reports a code injection attack.

The actual query q may only use the original characters $c \in C$ for code, whereas its shadow q' may only use the shadow characters $sc \in SC$ for code. For example, if an identifier terminal $\epsilon \in q$ is generated by merging a string constant with user input, the identifier terminal $\epsilon_s \in q'$ will contain original characters. This case is an instance of code injection because the code of the query depends on user input. DIGLOSSIA makes sure that all code terminals in q come *entirely* from the application itself and not a single character comes from user input.

On the other hand, the non-code in q' may use any combination of original and shadow characters, reflecting the fact that non-code may be derived from strings originating from user input or the application itself. For example, if the query q contains a string literal "ab", then "$map(a)map(b)$", "a$map(b)$" or '$map(a)$b" can all occur in the shadow query q'.

In summary, given the parse tree for the actual query q and the parse tree for the shadow query q', DIGLOSSIA checks whether the two queries agree on code and non-code. Since all code in q' that comes from the application itself is in shadow characters and all code in q' that comes from user input

is in original characters, DIGLOSSIA checks whether q' contains any code in original characters and, if so, reports a code injection attack.

5.3 Limitations

DIGLOSSIA follows Ray and Ligatti's strict definition of code and non-code [59] which does not permit any user input to be used as part of code in the query. If the application developer *intentionally* incorporates user input into the query code (a dangerous and ill-advised programming practice), DIGLOSSIA will report a code injection attack when the application is executed.

The ability to recognize and separate code and non-code in the query string generated by the application critically depends on using the correct grammar for the query language. If the language accepted by the database's query parser differs from the language accepted by DIGLOSSIA's parser during its analysis, DIGLOSSIA may generate both false positives (mistakenly parse a tainted part of the query as code, even though it will not be parsed as code by the database) and false negatives (mistakenly parse a tainted part of the query as non-code, even though it will be parsed as code by the database).

If the application passes an input-tainted string to a third-party PECL extension or some other built-in function that is not implemented in PHP, value shadowing can be incomplete because DIGLOSSIA cannot observe the string operations inside these functions. Incomplete value shadowing may lead to false negatives (missed attacks). Fortunately, unlike Java and C applications, PHP applications do not use third-party libraries heavily.

DIGLOSSIA propagates taint information by performing shadow operations on shadow values, mixtures of original and shadow characters. Thus, the semantic of particular shadow operations may be different to their corresponding original operations. If the application generates a tainted string query by changing characters with forceful typecasting operations to numbers and numeric operations, the corresponding shadow operations cannot preserve the same semantic due to randomly chosen shadow characters. When performing such shadow operations on tainted queries, DIGLOSSIA may generate false positives. However, building database queries mostly involve a series of string built-in operations. We did not observe any string operation that causes false positives.

5.4 Evaluation

To evaluate DIGLOSSIA, we created a test suite of ten Web applications implemented in PHP (see Table 5.2). Four of our benchmark applications use MongoDB and contain NoSQL injection vulnerabilities, which we found by manual inspection of the applications' source code: *mongodb_php_basic*, *mongodb-admin*, $MongoTinyURL$, and *simple-user-auth*. Two, $MongoPress$ and *rockmongo*, were chosen to demonstrate the performance of DIGLOSSIA on relatively large applications. The remaining four applications were chosen because they contain known SQL injection vulnerabilities [30, 41].

We implemented concrete attacks exploiting the known vulnerabilities in the benchmark applications. We also implemented concrete instances for

Applications	Database	LoC	Attacks	Detected
MongoPress	MongoDB	35,231	0	0
mongodb-admin		555	2	2
mongodb_php_basic		209	1	1
rockmongo		11,218	0	0
MongoTinyURL		60	1	1
simple-user-auth		236	1	1
faqforge	MySQL	1,520	1	1
schoolmate		7,024	6	6
webchess		5,780	12	12
MyBB with MyYoutube(1.0)		108,267	1	1

Table 5.2: Benchmark PHP applications.

all of Ray and Ligatti's canonical cases listed in Table 5.1. All experiments were performed on an Intel(R) dual core 3.30 GHz machine with 8G of RAM.

Table 5.2 summarizes the results of our evaluation on the ten benchmark Web applications. The first column lists the applications, the second column shows the backend database each application uses, the third column shows the size of the application. The fourth column shows the number of different code injection attacks we attempted against the application, while the last column demonstrates that DIGLOSSIA successfully detected all attacks.

Figure 5.6 shows the time it took to build the front page of each application, measured as the average of 50 runs with the database cache disabled. Range bars represent 95% confidence intervals. Most interval ranges overlap, thus the performance overhead of DIGLOSSIA is unnoticeable to the users of the application.

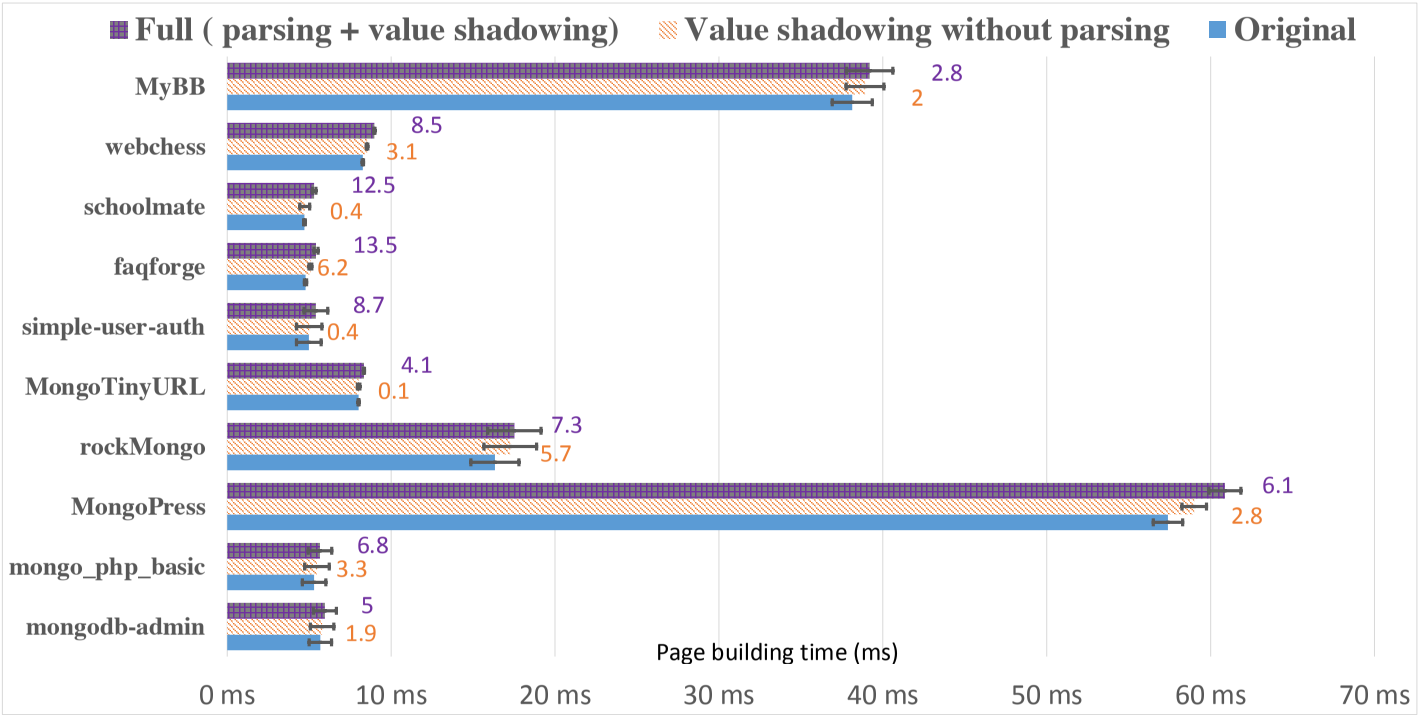

Figure 5.6: Performance overhead of DIGLOSSIA with the database cache disabled.

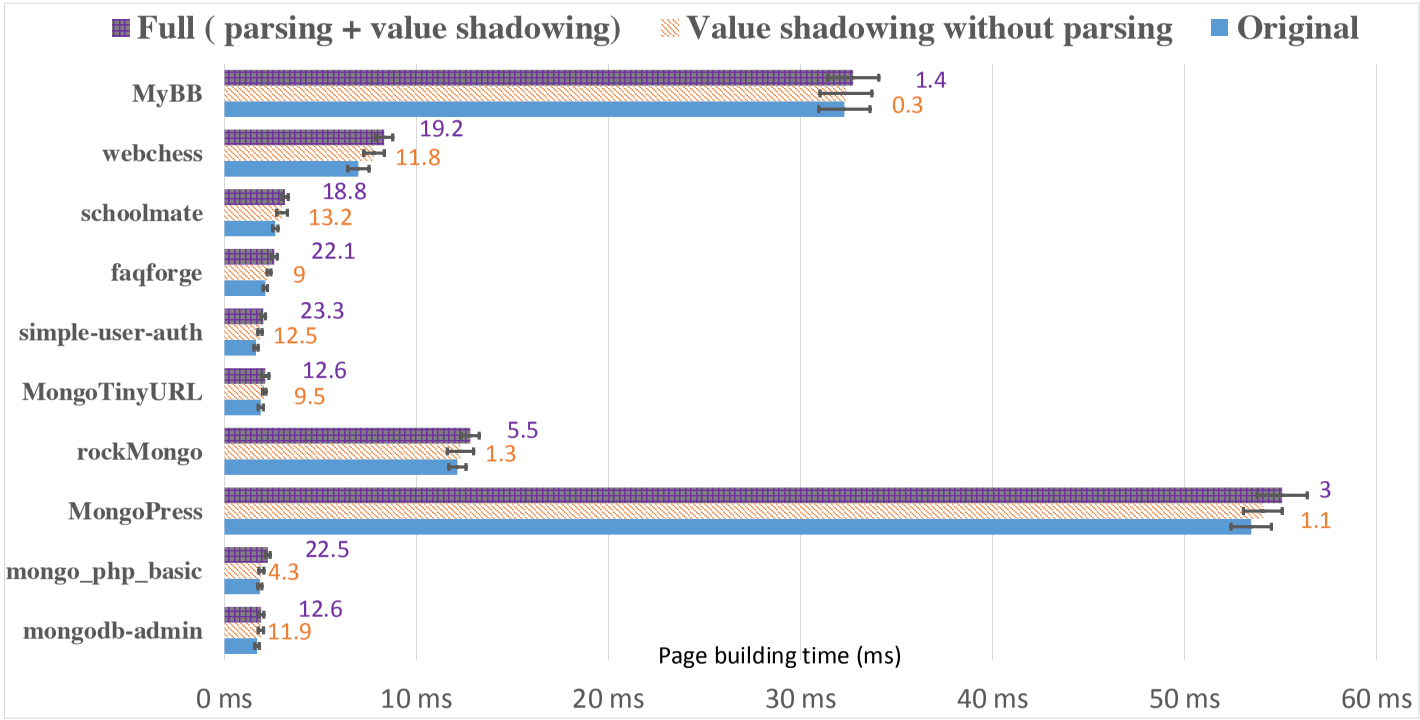

Figure 5.7: Performance overhead of DIGLOSSIA with the database cache enabled.

The numbers at the top of each bar represent overhead percentages, computed by taking the time it took to build the page with DIGLOSSIA deployed and dividing it by the original page-building time. The maximum overhead is 13%, but the actual time difference is less than 2 ms, within the variance of the original page-building time.

Figure 5.7 shows the performance overhead of DIGLOSSIA with the database cache enabled. The overall response times are lower, thus the overhead percentages are bigger than those in Figure 5.6. However, most overhead is not statistically significant compared to the variation in the page-loading time. To measure the overhead of validating a query, we implemented five simple PHP applications. Each of them performs only one of the following five query operations on an empty database. We selected the following queries that are used in the benchmark applications and MongoDB PHP code samples [39].

1. SELECT gameID FROM mainT WHERE lastMove < 2
2. UPDATE user_names SET access_f = TRUE WHERE uid='arbitrary' AND pwd ='guessme'
3. SELECT ids.name AS iname, files.name AS fname, COUNT(*) AS C FROM tokens INNER JOIN ids ON ids.eid = tokens.eid WHERE ids.eid = 'arbitrary' GROUP BY ids.eid
4. function nosql_query() { var id = 'any_id'; if(id == this.id) return true; else return false; }
5. {'key' : {'$gt' : 10 } }

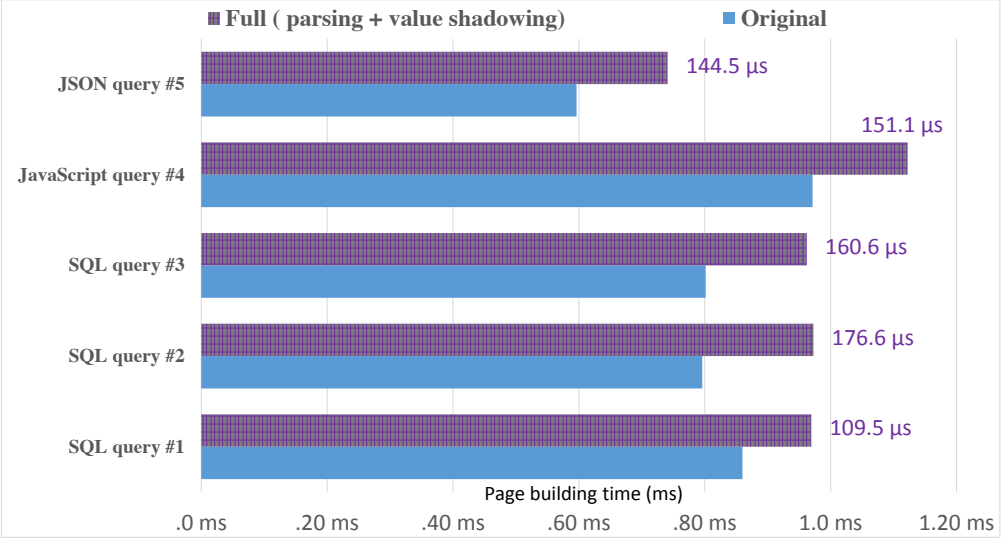

Figure 5.8: Performance overhead of dual parsing in DIGLOSSIA.

Figure 5.8 shows the time DIGLOSSIA took to execute each application. We measured the average execution time of 50 runs. The number at the top of each bar represents the execution time for each application. We measured the time it took to build each application with DIGLOSSIA deployed and subtract from it the original page-building time. These experimental results show that DIGLOSSIA accurately detects SQL and NoSQL code injection attacks with virtually unnoticeable performance overhead.

5.5 Conclusion

To the best of our knowledge, DIGLOSSIA is the first tool capable of *accurately* detecting both SQL and NoSQL injection attacks on server-side PHP applications at runtime, without any modifications to applications or backend databases.

DIGLOSSIA follows Ray and Ligatti's definition of code and non-code, combined with very precise character-level taint tracking, and thus avoids the false positives and false negatives of prior tools for detecting code injection attacks. In tandem with the execution of the application, DIGLOSSIA remaps all characters introduced into the query by the application itself into a shadow character set, while leaving the characters that originate from user input intact. The resulting query and its shadow are then analyzed using a *dual parser* that can parse both the original and shadow query languages. Dual parsing is the main technical innovation of this work. Any discrepancy between the parse trees of the query and its shadow, or the presence of any original characters in the code of the shadow query indicate that the code of the actual query is tainted by user input and thus a code injection attack has occurred.

DIGLOSSIA imposes negligible performance overhead and does not require any changes to the existing applications, databases, Web servers, or Web browsers. It can be easily added to the PHP environment and is ready to deploy today.

Chapter 6

Conclusion

Web security has become a principal factor in building reliable Web services. Web attacks take advantage of application vulnerabilities caused by developers' mistakes or their ignorance of security. Identifying and repairing these vulnerabilities demand tedious and error-prone code auditing, which motivates the search for alternatives.

We argue that automated security tools help identify and remediate Web vulnerabilities, resulting in better Web security. For access-control vulnerabilities in server-side Web applications, we presented RoleCast and FixMeUp that find and repair access-control vulnerabilities, respectively. RoleCast infers a role-specific access-control policy without *a priori* access-control specification by exploiting common software-engineering patterns. Furthermore, FixMeUp suggests candidate repairs to developers in order to ease the patching of access-control vulnerabilities. We demonstrate that our methods are effective in enforcing consistent role-specific and context-specific access-control logic.

To defeat code-injection threats against SQL and NoSQL database queries, we present Diglossia. It is a dynamic detection tool that identifies

code-injection attacks while a Web application is executing. We demonstrate its precision and efficiency in detecting subtle code-injection attacks using dual parsing and value shadowing. We also show that a precise definition of code in the target query language is essential for accurate identification of SQL and NoSQL command injection attacks.

In the thesis, we demonstrate the utility and benefits of static and dynamic tools that help improve server-side Web security. Automated security tools can not only help developers who lack security expertise, but also enforce consistent security properties while minimizing tedious code auditing for better server-side Web security.

Bibliography

[1] J. Andersen and J. Lawall. Generic patch inference. In *ASE*, pages 337–346, 2008.

[2] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of Web-based applications. In *CCS*, pages 25–35, 2007.

[3] S. Bandhakavi, P. Bisht, Madhusudan P, and V. N. Venkatakrishnan. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In *CCS*, pages 12–24, 2007.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, pages 75–88, 2008.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *CCS*, pages 607–618, 2010.

[6] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *ACNS*, pages 292–302, 2004.

[7] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A data-centric Web application security framework. In *WebApps*, pages 1–1, 2011.

[8] D. Canali and D. Balzarotti. Behind the scenes of online attacks: an analysis of exploitation behaviors on the web. In *NDSS*, 2013.

[9] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, pages 39–50, 2008.

[10] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133, 2010.

[11] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, pages 1–1, 2011.

[12] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in Web applications. In *RAID*, pages 63–86, 2007.

[13] CVE detail. http://www.cvedetails.com/vulnerabilities-by-types.php.

[14] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[15] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication and access control vulnerabilities in Web applications. In *USENIX Security*, pages 267–282, 2009.

[16] D. Dede. Attack of wordpress blogs on rackspace. http://blog.sucuri.net/2010/06/mass-attack-of-wordpress-blogs-on-racksp%ace.html, 2010.

[17] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *CCS*, pages 251–262, 2011.

[18] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[19] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in Web applications. In *USENIX Security*, pages 143–160, 2010.

[20] J. Grossman. Whitehat security website statistics report. 2012.

[21] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *WWW*, pages 561–570, 2009.

[22] W. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *FSE*, pages 175–185, 2006.

[23] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in Web applications via model checking and runtime enforcement of navigation state machines. In *ASE*, pages 235–244, 2010.

[24] Internet World Stats Usage and Population Statistics. http://www.internetworldstats.com/stats.htm.

[25] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, 2011.

[26] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy*, pages 258–263, 2006.

[27] N. Jovanovic, C. Kruegel, and E. Kirda. BLUEPRINT: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy*, pages 331–346, 2009.

[28] JSON. http://www.json.org.

[29] JSP. http://java.sun.com/products/jsp.

[30] A. Kieżun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[31] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *OOPSLA*, pages 359–372, 2002.

[32] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou. SQLProb: A proxy-based architecture towards preventing SQL injection attacks. In *SAC*, 2009.

[33] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *ESEC/FSE*, pages 296–305, 2005.

[34] V.B. Livshits and M. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security*, 2005.

[35] N. Meng, M. Kim, and K. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[36] N. Meng, M. Kim, and K. McKinley. LASE: Locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[37] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Security and Privacy*, pages 481–496, 2010.

[38] MongoDB production deployments. http://www.mongodb.org/about/production-deployments/.

[39] MongoDB. http://www.mongodb.org.

[40] R. Mui and P. Frankl. Preventing Web application injections with complementary character coding. In *ESORICS*, pages 80–99, 2011.

[41] MyYoutube MyBB Plugin 1.0 SQL Injection. http://www.exploit-db.com/exploits/23353.

[42] H. Nguyen, T. Nguyen, G. Wilson Jr., A. Nguyen, M. Kim, and T. Nguyen. A graph-based approach to API usage adaptation. In *OOPSLA*, pages 302–321, 2010.

[43] H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*, pages 15–29, 2007.

[44] T.T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T.N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE*, pages 315–324, 2010.

[45] A. Nguyen-tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 372–382, 2005.

[46] NoSQL. http://nosql-database.org/.

[47] NoSQL injection attack on Diaspora. http://www.kalzumeus.com/2010/09/22/security-lessons-learned-from-the-diaspora-launch/.

[48] OWASP top 10 application security risks. https://www.owasp.org/index.php/Top˙10˙2010-Main, 2010.

[49] Owasp attack. https://www.owasp.org/index.php/Category:Attack.

[50] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W-F. Wong, Y. Zibin,

M. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, 2009.

[51] PHC : open source php compiler. http://www.phpcompiler.org.

[52] PHP. http://www.php.net.

[53] PHP advent 2010: Usage statistics. http://phpadvent.org/2010/usage-statistics-by-ilia-alshanetsky.

[54] T. Pietraszek, C. Berghe, C. V, and E. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, pages 124–145, 2005.

[55] M. Pistoia, R. Flynn, L. Koved, and V. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, pages 362–386, 2005.

[56] N. Popper and S. Sengupta. U.S. says ring stole 160 million credit card numbers. http://dealbook.nytimes.com/2013/07/25/arrests-planned-in-hacking-of-fi%nancial-companies/, 2013.

[57] B. Proffitt. Yahoo's 450,000 Account Security Breach. http://readwrite.com/2012/07/12/yahoos-450-000-account-security-breach-%whose-fault-was-it, 2012.

[58] Quercus. http://quercus.caucho.com.

[59] D. Ray and J. Ligatti. Defining code-injection attacks. In *POPL*, pages 179–190, 2012.

[60] R. Richmond. Twitter Is Hacked Tuesday Morning, 2010. http://bits.blogs.nytimes.com/2010/09/21/twitter-hacked-tuesday-morning%/.

[61] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[62] P. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *ESORICS*, pages 100–116, 2011.

[63] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich Web applications. In *NDSS*, 2010.

[64] R. Sekar. An efficient black-box technique for defeating web application attacks *. In *NDSS*, 2009.

[65] A. Sistla, V. Venkatakrishnan, M. Zhou, and H. Branske. CMV: Automatic verification of complete mediation for Java Virtual Machines. In *ASIACCS*, pages 100–111, 2008.

[66] S. Son, K. McKinley, and V. Shmatikov. RoleCast: Finding missing security checks when you do not know what checks are. In *OOPSLA*, pages 1069–1084, 2011.

[67] S. Son and V. Shmatikov. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *PLAS*, 2011.

[68] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[69] Soot: A Java optimization framework. http://www.sable.mcgill.ca/soot/.

[70] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *PLDI*, pages 112–122, 2007.

[71] V. Srivastava, M. Bond, K. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *PLDI*, pages 343–354, 2011.

[72] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *POPL*, 2006.

[73] B. Sullivan. Server-side JavaScript injection. http://media.blackhat.com/bh-us-11/Sullivan/BH˙US˙11˙Sullivan˙Server˙Side˙WP.pdf, 2011.

[74] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in Web applications. In *USENIX Security*, pages 11–11, 2011.

[75] F. Sun, L. Xu, and Z. Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS*, 2014.

[76] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *SS*, pages 379–394, 2008.

[77] Apache Tomcat. http://tomcat.apache.org.

[78] J. Vijayan. TJX data breach: At 45.6M card numbers, it's the biggest ever. http://www.computerworld.com/s/article/9014782/TJX˙data˙breach˙At˙45.6M˙card˙numbers˙it˙s˙the˙biggest˙ever, 2007.

[79] G. Wasserman and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[80] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[81] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[82] A. Yip, X. Wang, N. Zeldovich, and F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.

[83] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI*, pages 415–424, 2007.

[84] M. Zhou, P. Bisht, and V. N. Venkatakrishnan. Strengthening XSRF defenses for legacy web applications using whitebox analysis and transformation. In *ICISS*, pages 96–110, 2010.

Vita

Sooel Son was born in Suwon, South Korea on 1 July 1981. He received the degree of Bachelor of Science from Hanyang University at Ansan in 2008. He entered the Computer Science Ph.D. program at the University of Texas at Austin in 2008.

Permanent address: 11500 Jollyville Rd Apt 1024
Austin, Texas 78759

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.