Diglossia: Detecting Code Injection Attacks with Precision and Efficiency

Sooel Son
The University of Texas at Austin
samuel@cs.utexas.edu

Kathryn S. McKinley
Microsoft Research
The University of Texas at Austin
mckinley@cs.utexas.edu

Vitaly Shmatikov
The University of Texas at Austin
shmat@cs.utexas.edu

ABSTRACT

Code injection attacks continue to plague applications that incorporate user input into executable programs. For example, SQL injection vulnerabilities rank fourth among all bugs reported in CVE, yet all previously proposed methods for detecting SQL injection attacks suffer from false positives and false negatives.

This paper describes the design and implementation of DIGLOSSIA, a new tool that precisely and efficiently detects code injection attacks on server-side Web applications generating SQL and NoSQL queries. The main problems 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 recognize code, DIGLOSSIA relies on the precise definition due to Ray and Ligatti. 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 thus exactly the taint from user input.

Our key technical innovation is *dual parsing*. 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 demonstrate that DIGLOSSIA accurately detects both SQL and NoSQL code injection attacks while avoiding the false positives and false negatives of prior methods. By recasting the problem of detecting injected code as a string propagation and parsing problem, we gain substantial improvements in efficiency and precision over prior work. Our approach does not require any changes to the databases, Web servers, or Web browsers, adds virtually unnoticeable performance overhead, and is deployable today.

Categories and Subject Descriptors Security and privacy — Intrusion/anomaly detection and malware mitigation — Intrusion detection systems

Keywords Web application security; Dynamic analysis; Code injection; SQL injection; NoSQL injection; Taint tracking

CCS'13 November 04 - 08 2013, Berlin, Germany

http://dx.doi.org/10.1145/2508859.2516696.

1. INTRODUCTION

> Diglossia */dīˈ glôsēə/ noun*
> A situation in which two languages (or two varieties of the same language) are used under different conditions within a community, often by the same speakers.
>
> *Oxford Dictionaries*

Modern Web applications accept input from users and incorporate it into dynamically generated code. For example, a Web application may invite the user to fill a form, post a comment, or submit a username and password for authentication. The application then takes this user-provided input and inserts it into a dynamically generated program in another language—for example, a new client-side script, or an SQL or JavaScript query to a back-end database.

A *code injection attack* occurs when a malicious user manages to inject his own code into the program generated by the application. 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 a classic target of code injection. For example, SQL injection attacks on retail stores owned by TJX Companies compromised more than 45 million credit and debit numbers in 2005-2007 [23]. SQL injection vulnerabilities still rank fourth among all reported CVE bugs [4], and, according to the 2012 WhiteHat security report, SQL injection attacks are the eighth most prevalent attack type [25].

The recent trend towards NoSQL databases [14] is not improving the situation. 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 injection vulnerability in its social community framework [15]. Code injection attacks on JavaScript queries for MongoDB were demonstrated at Black Hat 2011 [21].

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 therefore 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 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 (code injection attacks accepted as valid queries).

Our contributions. We design, implement, and evaluate 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.

In tandem with the application computing its output string, DIGLOSSIA computes a *shadow* of this string. The purpose of the shadow is to identify user input in the application-generated string. In the shadow string, all characters introduced by the application itself are remapped to a shadow character set, which is disjoint from all characters in user input; the shadow value thus precisely encodes which characters came from user input and which came from the application. To precisely identify code, DIGLOSSIA relies on the definitions by Ray and Ligatti [17]. DIGLOSSIA uses a novel *dual parsing* technique to compare the shadow string with the actual string generated by the application and ensure that the actual string does not contain any code tainted by user input.

Our basic approach is language-agnostic and does not depend on the details of the language used to implement the application, nor the target language of the generated string. For concreteness, our DIGLOSSIA prototype works with server-side PHP applications generating database queries in SQL, JSON, or JavaScript. Consequently, we use the term *query* for the generated string.

Defining code injection attacks. Ray and Ligatti show that 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 [17].

Following Ray and Ligatti, only values (numeric and string literals) and reserved values (NULL, TRUE, etc.) are *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.

With their definition of code and non-code in hand, Ray and Ligatti show that all prior approaches for detecting code injection suffer from false negatives and false positives. They illustrate these inaccuracies with 11 SQL injection attacks and non-attacks, which are explained in detail in Section 3.1 and summarized in Table 1. For example, the attack in Case 5 injects a call to a bound method. All prior approaches miss this attack, but DIGLOSSIA detects it. On the other hand, Case 7 shows how a use of a reserved literal (TRUE) causes at least one prior tool to report a false positive whereas DIGLOSSIA correctly classifies this case.

Value shadowing. 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 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 computes *shadow values* for all strings computed by P that depend on user input. DIGLOSSIA 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. 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.

Dual parsing. To guarantee that the query issued by the application P does not contain injected code, it is sufficient to ensure the following two properties. First, the shadow query must not contain injected code (technically, the code in the shadow query must not be tainted by user input). Second, the actual query must be syntactically isomorphic to the shadow query.

When P issues a query, DIGLOSSIA examines this 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.

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

Unlike prior tools, DIGLOSSIA correctly classifies 10 out of 11 challenging cases described by Ray and Ligatti [17]—see Table 1. The sole exception is the case where user input is a number specifying how many objects of a certain type to allocate. Even though Ray and Ligatti classify this case as code injection, this is a matter of opinion. We instead classify all integer literals in SQL type definitions as values—for example, 64 in *CHAR(64)* [22]. Therefore, DIGLOSSIA does not report Case 10 in Table 1 as a code injection.

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 [2] and complementary encoding [11], DIGLOSSIA does not require any changes to Web applications, databases, query parsers, Web servers, or Web browsers. Unlike these tools, DIGLOSSIA can be deployed *today*.

2. RELATED WORK

This section compares our approach with prior techniques for statically detecting SQL injection vulnerabilities in Web applications

		1	2	3	4	5	6	7	8	9	10	11
	Ray and Ligatti's definition of code injection [17]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No
Tools	Halfond et al. [5], Nguyen-Tuong et al. [13]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
	Xu et al. [26]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
	SQLCHECK [20]	Yes	No	No	Yes	No	No	No	No	No	No	No
	CANDID [1]	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 1: Canonical code injection attacks and non-attacks misclassified by prior methods. Underlined terms are user input.

and dynamically detecting attacks exploiting these vulnerabilities. In Section 3, we explain Ray and Ligatti's examples shown in Table 1 and why prior methods misclassify between 5 and 8 of these 11 examples.

Static methods. Static methods use pointer and taint analysis to find unsanitized data flows from user input to database queries [6,9, 19]. 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 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 [24]. Static dataflow analysis must be conservative, thus static methods inevitably suffer from false positives.

Dynamic methods. Most dynamic methods aim to precisely track the source of every byte and thus determine which parts of the query come from tainted user input and which come from the application itself [3, 5, 13, 16, 26]. All of these tools use a simple, imprecise definition of "code" and consequently suffer from false positives and false negatives (see Table 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 [20]. 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 [18], while Liu et al. compare the query to previous queries generated from benign inputs [8]. 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 [1]. As Ray and Ligatti point out, this analysis is insufficient to differentiate code from non-code [17]. Furthermore, 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 3.3.

Randomization and complementary encoding. To prevent injection of SQL commands, SQLrand remaps SQL keywords to secret, hard-to-guess values [2]. 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 [11]. 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.

In contrast, DIGLOSSIA is a simple PHP extension that does not require any modifications to applications or databases.

3. EXAMPLES OF CODE INJECTION

This section gives examples of SQL, NoSQL, and syntax mimicry code injection attacks.

3.1 SQL injection attacks

We illustrate SQL injection attacks using 11 canonical examples described by Ray and Ligatti [17]. Table 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='' OR 1=1 - -'
   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= exit()
   User input injects exit(), 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.

3.2 NoSQL injection attacks

Modern Web applications are increasingly using NoSQL backend data stores instead of relational SQL databases. NoSQL is a new class of distributed, scalable databases [14] that store data in

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   global $mongo;
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 1: JSON injection vulnerability.

key-value pairs. NoSQL databases include Amazon's DynamoDB, MongoDB, CouchDB, and several others. For example, MongoDB is an open-source, document-oriented NoSQL database that supports JSON and JavaScript as query languages. It has been adopted by Craigslist, Foursquare, and other popular Web services [10].

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.

Figure 1 shows a PHP application with a JSON injection vulnerability. Line 13 of *mongodbadmin.php* in Figure 1 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 $\$\_REQUEST[\text{'}search\text{'}]$. 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 1 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 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, which always returns "true". Note that user-injected symbols ;, $return$, }, and // are parsed into code in the JavaScript query:

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

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: JavaScript injection vulnerability.

3.3 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

```
<?
// Build a JavaScript query that checks whether pwd
     field is the same as user input $_GET['id']

$query = "function q() { ";
$query .= "var secret_number = this.pwd;";
$query .= "var user_try = ' . $_GET['id'] . ';";
$query .= "if (secret_number!=user_try) return false;";
$query .= "return true;";
$query .= "}";

$collection->find(array('$where' => $query));
?>
```

Attack URL

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

Figure 3: JavaScript syntax mimicry attack.

Figure 3 shows sample PHP code that builds a JavaScript query for a MongoDB. User input in $\$\_GET['id']$ is supposed to be a numeric literal. If the attacker inputs *secret_number* instead of a number, the query will return "true", sabotaging the intended semantics. CANDID will use "aaaaaaaaaaa" as the benign input for *secret_number* in its shadow execution and miss the attack, but DIGLOSSIA will detect it.

Figure 4 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.

login.php in *minibill*

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

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 4: SQL syntax mimicry attack on *minibill*.

reset_password_save.php in *phpAddressBook* 8.2.5

```
<?
$password_hint = $_REQUEST['password_hint'];
$email=$_REQUEST['email'];
...
// Assume that $cleanpw is "arbitrary"
$query = "UPDATE users SET password='$cleanpw',
      password_hint='$password_hint' WHERE email='$email
      '";

$res = mysql_query($query);
?>
```

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='aaaaaaaaaaaaaaa'

Figure 5: SQL syntax mimicry attack on *phpAddressBook*.

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 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 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.

4. DESIGN AND 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 has three phases, as depicted in Figure 6 and described below.

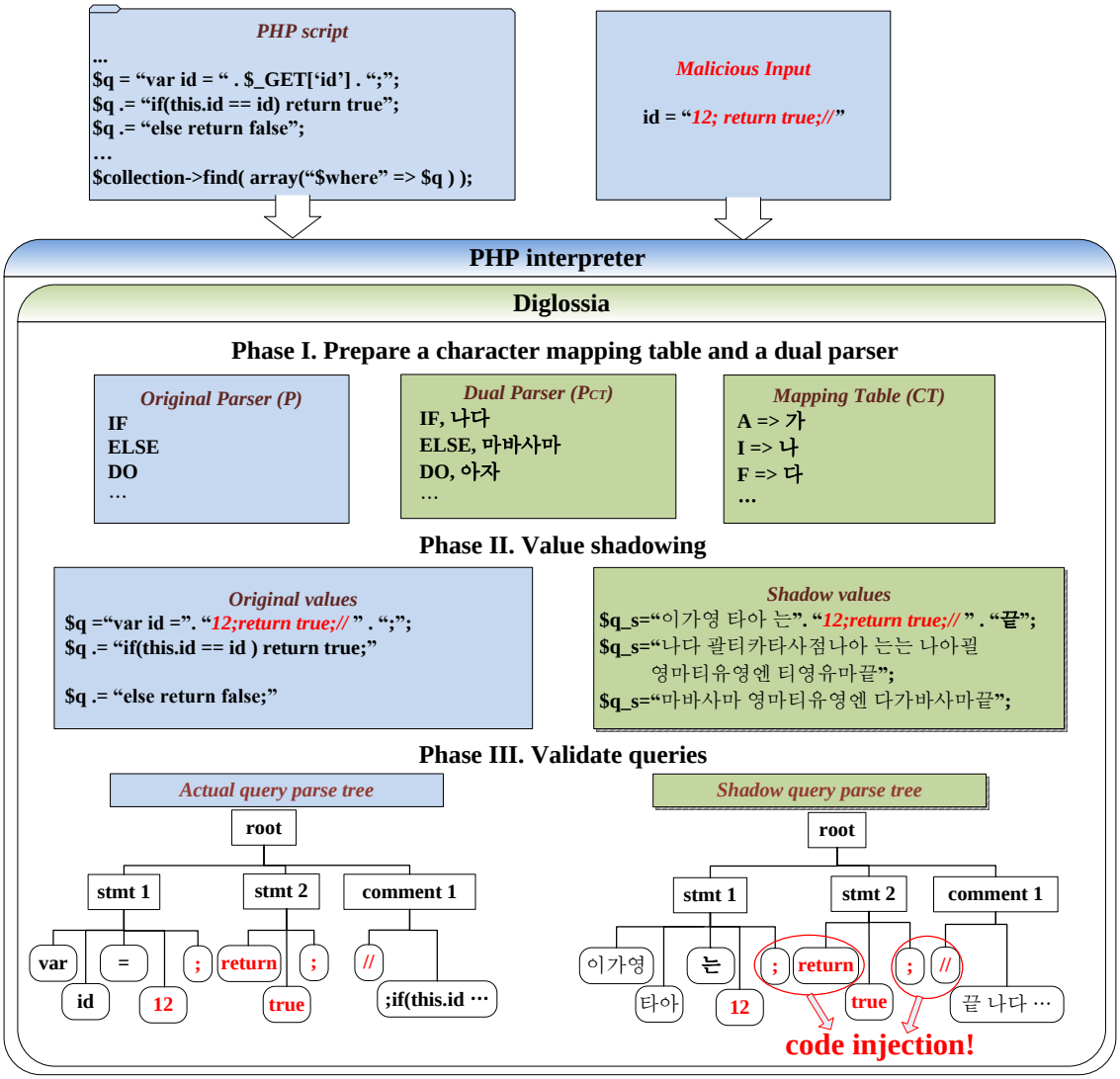

Figure 6: Overview of DIGLOSSIA.

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 the dual parser for the shadow language SL, which is a superset of L and described in more detail in Section 4.3.

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. 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 constants, come from a library, or are generated dynamically.

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 4.3. 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 back-end database. Otherwise, DIGLOSSIA stops the application and reports a code injection attack.

4.1 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.

4.2 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, 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 the *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.

Figure 6 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 6, the assignment $\$q = \text{“}var\ id\ =\ \text{”}.\text{“}12; return\ true; //\text{”}.\text{“;”}$; concatenates string constants with user input. We compute the shadow value as $\$q_s = map(\text{“}var\ id\ =\ \text{”}).\text{“}12; return\ true; //\text{”}.map(\text{“;”});$. 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.

Figure 7 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.

4.3 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*.

```
// 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 7: An example of value shadowing.

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 different 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 4.1, 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:

$$rule : n \rightarrow v_1 v_2 ... v_l \text{ where } n \in N, v \in N \cup \Sigma$$

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.

$$\begin{aligned}
rules: \quad & \textit{select_stmt} \rightarrow SELECT_term\ list_exp\ table_exp \\
& SELECT_term \rightarrow SELECT \\
& \textit{identifier} \rightarrow \{a|b|\ldots\} \\
& \ldots \\
rules_s: \quad & \textit{select_stmt} \rightarrow SELECT_term\ list_exp\ table_exp \\
& SELECT_term \rightarrow SELECT \mid map(SELECT) \\
& \textit{identifier} \rightarrow \{a|b|\ldots|map(a)|map(b)|\ldots\} \\
& \ldots
\end{aligned}$$

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. LIMITATIONS

DIGLOSSIA follows Ray and Ligatti's strict definition of code and non-code [17] 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).

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 2: Benchmark PHP applications.

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. For example, we did not observe any calls to third-party libraries that perform string or array operations on user inputs in our benchmarks.

6. EVALUATION

To evaluate DIGLOSSIA, we created a test suite of ten Web applications implemented in PHP (see Table 2). Four of our benchmark applications use MongoDB and contain NoSQL injection vulnerabilities, which we found by manual inspection of the applications' source code: $mongodb\_php\_basic$, $mongodb\text{-}admin$, $MongoTinyURL$, and $simple\text{-}user\text{-}auth$. 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 [7, 12].

We implemented concrete attacks exploiting the known vulnerabilities in the benchmark applications. We also implemented concrete instances for all of Ray and Ligatti's canonical cases listed in Table 1. All experiments were performed on an Intel(R) dual core 3.30 GHz machine with 8G of RAM.

Table 2 summarizes the results of our evaluation on the ten benchmark Web applications. The first column lists the applications, the second column shows the back-end 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 8 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. 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 9 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 8. However, most overhead is not statistically significant compared to the variation in the page-loading time.

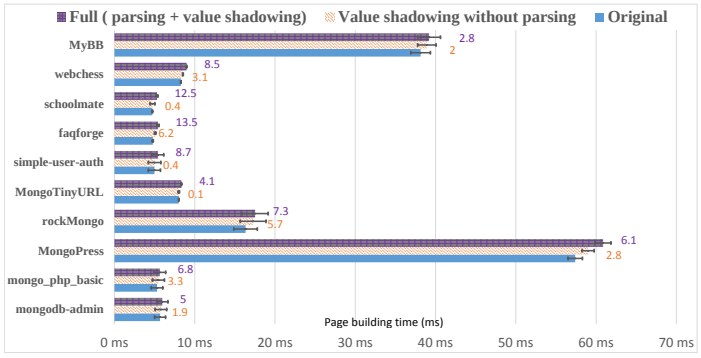

Figure 8: Performance overhead of DIGLOSSIA with the database cache disabled.

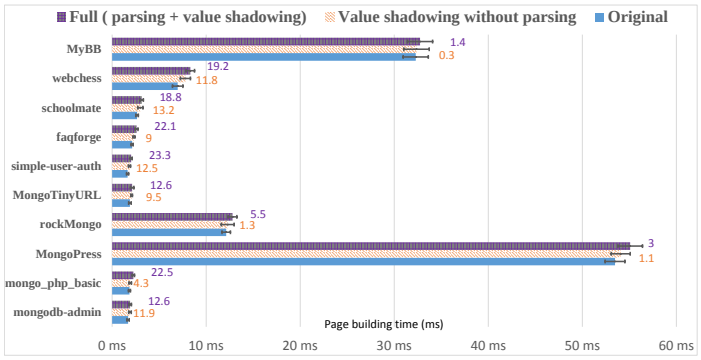

Figure 9: Performance overhead of DIGLOSSIA with the database cache enabled.

These experimental results show that DIGLOSSIA accurately detects SQL and NoSQL code injection attacks with virtually unnoticeable performance overhead.

7. 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 back-end 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.

Acknowledgments. We are very grateful to Venkat Venkatakrishnan for his insightful critique of an early version of this research, and to Jay Ligatti for his comments on a draft of this paper. This work was partially supported by the NSF grants CNS-0746888, CNS-0905602, SHF-0910818, CCF-1018271, and CNS-1223396.

8. REFERENCES

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

[2] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *ACNS*, 2004.

[3] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *SWS*, 2009.

[4] CVE Details. http://www.cvedetails.com/vulnerabilities-by-types.php.

[5] W. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *FSE*, 2006.

[6] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *S&P*, 2006.

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

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

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

[10] mongoDB production deployments. http://www.mongodb.org/about/production-deployments/.

[11] R. Mui and P. Frankl. Preventing web application injections with complementary character coding. In *ESORICS*, 2011.

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

[13] A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically hardening Web applications using precise tainting. In *SEC*, 2005.

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

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

[16] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2006.

[17] D. Ray and J. Ligatti. Defining code-injection attacks. In *POPL*, 2012.

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

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

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

[21] B. Sullivan. Server-side JavaScript injection. http://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf, 2011.

[22] The BNF grammar for SQL-99. http://savage.net.au/SQL/.

[23] 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.

[24] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

[25] WhiteHat website security statistics report. https://www.whitehatsec.com/resource/stats.html, 2012.

[26] 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.