WAFA: Fine-grained Dynamic Analysis of Web Applications

Manar H. Alalfi James R. Cordy Thomas R. Dean

School of Computing, Queen's University, Kingston, Canada
{alalfi, cordy, dean}@cs.queensu.ca

Abstract

Database interactions are a vital source of information in the analysis of highly dynamic systems such as web applications. Most web application security vulnerabilities, such as SQL injection and broken access control, can be traced to problems in database interactions. which are implemented as a set of embedded or constructed SQL statements. The identification and analysis of these embedded statements as an integral component of the host application requires complex analysis including robust parsing, pattern matching, control flow and data flow analysis.

In this paper, we propose an approach to this problem using source transformation technology. A rich model of fine-grained information is extracted from dynamic web applications, allowing us to reason not only about the SQL embedded system, but also about page access, server environment variables, cookies and session management functions. We evaluate our system on the popular bulletin board web application PhpBB, a PHP / MySQL-based dynamic web application.

1 Introduction

Web applications are one of many kinds of systems with multiple components that dynamically interact to deliver a specific business process. Sophisticated static and dynamic analysis is needed when reverse engineering web applications to extract all relevant information from the various components, and to correlate the extracted information to model the actual application behavior.

Several approaches for reverse engineering web applications have been proposed, most of which have focused on extracting the structural levels of the application, such as pages, frames, forms, and hyperlinks [13, 16]. Others have addressed particular aspects of application behavior such as interaction with the browser [9], and still others have aimed at extracting a higher level abstract behavioral model which describes the basic application elements, but does not combine the results or extract the details of database interaction [11, 5, 10].

Most current dynamic web application business processes depend on the support of a database back end. A great deal of information is stored in the database, including critical knowledge such as session management and access permissions. Dynamically identifying and extracting database interactions alone can be misleading, as they do not reflect the actual intended behavior of the application business process as a whole. Analyzing database interactions in the context of the entire web application may clarify aspects of the business process hidden behind the web application presentation level. Since many web application vulnerabilities rely on modifying the database interaction statements at runtime, there is a need to analyze not only the values of the SQL statements constructed at runtime, but also the original source of the host language statements used to construct the SQL statements.

Database interactions are often implemented in web applications using a combination of string concatenation expressions and host language statements that work together to construct an SQL statement. These expressions and statements are composed of constant strings and application variables. Identifying, extracting, and analyzing these dynamically constructed statements in the context of the overall system is not a trivial process.

In this paper we propose an approach to analyze dynamic web applications, extracting a fine-grained model aimed at understanding the interaction between the web application and the database. Our approach is different from other methods in the following aspects:

- An automated instrumentation methodology that handles mixed languages, and can be easily extended to other technologies and host languages.
- Extraction of a model that relates information about pages, server environment variables, database interactions and host language source statements. This model is stored in a database to facilitate accessibility and future analysis.
- Extraction of a web application's embedded SQL components, which is comprised of the source of the original SQL statements, and the corresponding execution

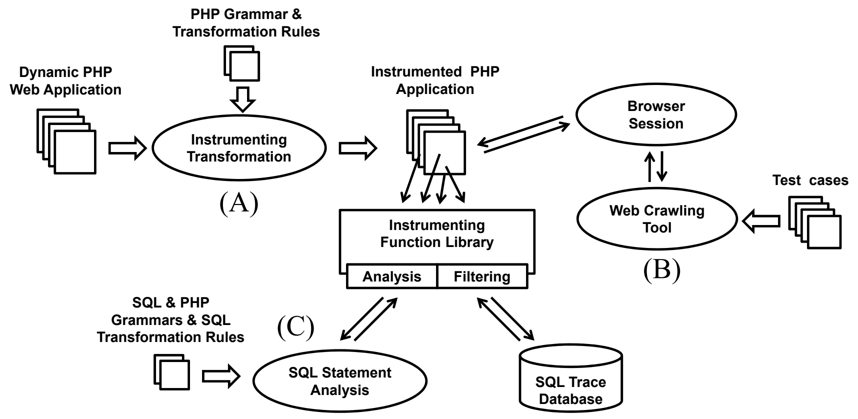
PHP Grammar & Transformation Rules
Dynamic PHP Web Application
Instrumenting Transformation
(A)
Instrumented PHP Application
Browser Session
Web Crawling Tool
Test cases
(B)
Instrumenting Function Library
Analysis
Filtering
SQL & PHP Grammars & SQL Transformation Rules
(C)
SQL Statement Analysis
SQL Trace Database

Figure 1. WAFA Architecture

instances. The SQL components also include both static host application variables and dynamic server environment variables.

This work takes place in the context of a larger project on web application security, specifically the analysis of role based access control [1]. In the class of web applications we are analyzing, user roles and access permissions are stored in the application database with the other application data. Providing the context of the database interactions allows us to better analyze these interactions and recover the details of user roles and permissions. In previous work [2] we recovered sequence diagrams for test sessions at the page level. We have used the work described in this paper to extend the recovered sequence diagrams to include interactions with the database. The extended sequence diagrams are then used as part of a model-based analysis of access control.

Another possible application of our approach is the analysis of SQL injection attacks in a way similar to Halfond and Orso [12]. Differences between the structure of the runtime query and the source version of the query indicate input that may have changed the meaning of the query.

2 Approach

Figure 1 shows the architecture of our approach, called WAFA (Web Application Fine-grained Analysis). An instrumentation transformation is used to analyze the source code of the application, inserting appropriate calls to an instrumentation library written in PHP, Figure 1(A). The instrumented application is deployed in a testing environment, where a web crawling tool uses predefined test cases to exercise the web application. Since test case design and coverage are important issues, they are discussed in detail in another recently submitted paper [4]. We focus our discussion here on recovering the interaction behavior between the web application components.

As the tests run, Figure 1(B), the instrumentation library inserts information about each query into a separate instrumentation database. A source transformation is performed on each SQL statement at runtime to analyze its structure. This structural information is inserted in the database along with the query, Figure 1(C).

The schema for our database is shown in Figure 2. It is comprised of five tables and one view. The `Server Pages` table is used to keep track of access to individual pages, while the other four tables contain information about the HTTP variables, environment variables, cookies and database statements associated with each page, linked using the `Page_Ins_ID` field. We combine the information from the various tables into a single unified trace view in the `Dynamic Analysis` view. The following section elaborates the approach in more detail.

3 Instrumentation Methodology

We automatically analyze and add source code instrumentation to web application source using TXL [8], a programming language designed for manipulating and experimenting with programming language notations and features. TXL is a powerful source transformation system that has been used in industrial applications involving millions of lines of source code. The TXL processor takes as input a context-free grammar for the language to be manipulated, parses the source program into a parse tree, and then recursively applies a set of transformation rules, beginning with a main rule, until there are no remaining matches in the parse tree. The transformation is completed by unparsing the transformed tree to the new target source program.

Our implementation presently instruments web applications written in PHP(3,4,5) and MySQL(5.x). However, our TXL-based approach is easily adapted to deal with other scripting languages and database engines. Documents that include a mixture of languages and technologies are easily handled by employing island grammars [17]. In our implementation, the islands are PHP code, while the HTML

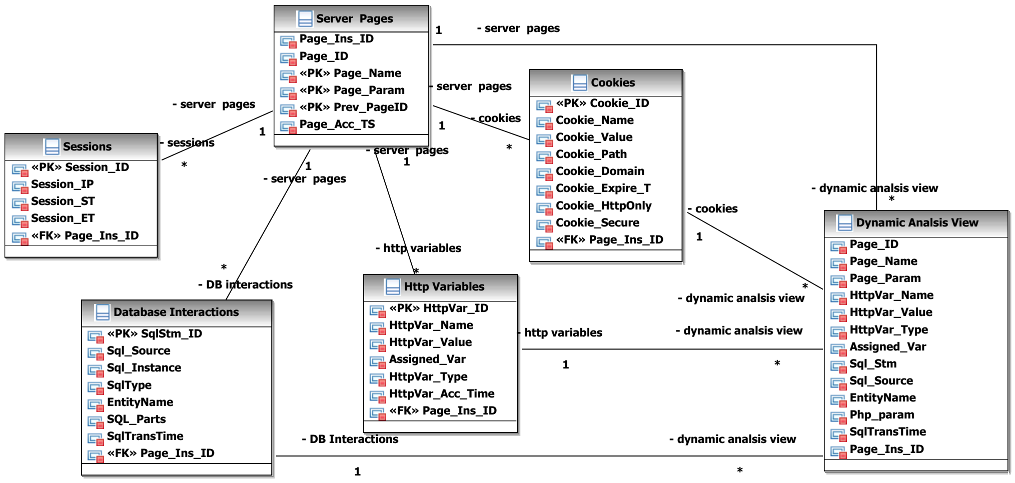

Figure 2. The WAFA Dynamic Analysis database model

source and document text are considered water. Island grammars simplify the transformation as interesting elements can be identified and analyzed without parsing the entire document.

3.1 Instrumenting and Collecting Page Information

In our process, page access is tracked by querying the server execution environment information created by the web server when the user interacts with the web application. Pages are instrumented by inserting an instrumenting function call at the head of each PHP dynamic page that performs this query to retrieve the page URL address along with any parameters passed to the page as well as the page access time, and inserts a trace element of this information in the page access table of our instrumenting database.

As part of the insertion of each page access trace element in the database, an analysis is performed to insure that the inserted trace element is unique. We recognize unique trace elements as those that lead to the generation of a new client page, or that generate a previously visited client page using a different path. Each server page can generate one or more client pages depending on the parameters passed to the page. We consider a generated client page to be the same if the same server page is re-executed without any parameters, or with the same parameters. In such cases we do not insert the new page into the database unless a different path is followed in its generation. Our database is constructed to reject any insertion that violates these conditions.

3.2 Instrumenting and Collecting Server Information

In PHP, predefined global variables contain information about a script's environment, such as the client's web browser, the HTTP host, and the HTTP connection. All those variables along with any function that manipulates them are instrumented. When the application is executed, a trace element on each piece of information will be inserted in a database for later analysis.

In particular, four types of HTTP variables are identified and instrumented, namely, the `GET`, `POST`, `COOKIE`, and `SESSION` variables. References to these variables are replaced with a call to the instrumenting function `HttpVar_track()`, which takes as parameters the HTTP variable name, value, type, name of the PHP variable to which the HTTP variable is assigned(if any), and the page number in which the variable is located. After adding the information to our database, the function returns the value of the HTTP variable so as to preserve the semantics of the code. An example of this instrumentation is shown in Figure 3. In the Figure, references to the HTTP parameter `mode` are logged in the database along with the fact that it is assigned to the PHP variable `$mode`.

3.3 Cookies and session management functions

In the HTTP Protocol, state is preserved between consecutive requests using cookies, a small identifier that is stored in the client browser and sent back to the server on each subsequent request. This identifier is used by the web application to store and retrieve data specific to that session. The predefined global variables `$HTTP_COOKIE_VAR` is used to access the cookies, and this variable is instrumented as described in the previous section. In addition, we instrument changes to the cookie by adding a call to our instrumentation function, `cookie_track()`, after each call to the PHP function `setcookie()`. Our function takes the same parameters and stores them in the `Cookie` table of our instrumentation database. An example of this transformation is shown in Figure 4. The line in bold is added

```
<?php
...
if ( !empty($HTTP_POST_VARS['mode']) || !empty($HTTP_GET_VARS['mode']) )
  $mode = ( !empty($HTTP_POST_VARS['mode']) ) ?
$HTTP_POST_VARS['mode'] : $HTTP_GET_VARS['mode'];
                                        ⇩ Transformed into . . .

if ( !empty($HTTP_POST_VARS['mode']) || !empty($HTTP_GET_VARS['mode']) )
  $mode = (! empty ($HTTP_POST_VARS ['mode'])) ?
HttpVar_track ('$mode', 'mode', $HTTP_POST_VARS ['mode'],186,192, POST):
HttpVar_track ('$mode', 'mode', $HTTP_GET_VARS ['mode'],186,191, GET);
...
?>
```

Figure 3. Results of instrumenting server environment variables in a snippet of code in PhbBB 2.0 application

Each HTTP reference variable is identified and transformed into an instrumenting function call which is passed the variable name, assigned variable name, variable value, unique ids for the variable name and the page name, and the variable type. The instrumenting function returns the variable value as its return value.

```
<?php
. . .
if ($userdata ['session_logged_in']){
setcookie ($board_config ['cookie_name'].'_f_all', time (), 0,
          $board_config ['cookie_path'], $board_config ['cookie_domain'],
          $board_config ['cookie_secure']);
                                  ⇩ New instrumentation function added . . .

cookie_track ($board_config ['cookie_name'].'_f_all', time (), 0,
              $board_config ['cookie_path'], $board_config ['cookie_domain'],
              $board_config ['cookie_secure']);}
. . .
?>
```

Figure 4. Result of instrumenting cookie management functions in a snippet of code in the PhbBB 2.0 application

Each cookie management function is identified and an additional instrumenting function call that captures all cookie management function parameters is added

to the contents of the if statement directly after the call to `setcookie()`. If cookies have been disabled, then session information is encoded into URLs using PHP utility routines. We instrument the calls to these functions in a similar manner.

3.4 Instrumenting and Collecting Database Interactions

The most complex set of transformations is used to identify and instrument the interactions with the database. We break the transformation into four parts. The first is to identify dynamically constructed SQL statements. Once they are identified, then we can insert code to construct a string value that contains the same SQL statement, but with the names of the application variables that are used instead of the variables. Together the two strings are inserted into the instrumentation database. An analysis of each of the SQL statements is done to identify the type of the statement, and the key elements present in the statement. This information is added to the database entry for the statement.

3.4.1 Identifying Dynamically Constructed SQL Statements

Our approach uses a separate lexical token class (`SQLCommandString`) to distinguish string literals that begin with the SQL keywords `Select`, `Insert`, `Update`, `Delete`, `Create`, `Alter` and `Drop` from other strings in the PHP source text. This allows us to use the parser to recognize concatenation expressions built from these strings (Figure 5). The transformation then targets assignment statements that use these strings to build larger strings. Once an assignment is found that uses one of these SQL keyword strings, other assignments using the same PHP variable are also checked and instrumented. The code is normalized prior to the transformations, moving string expressions to separate assignment statements and replacing them with a temporary PHP variable.

3.4.2 Constructing SQL Statement Sources

As the SQL assignment statements identified in the previous section are encountered, our transformation inserts additional assignments into the code to accumulate the source representation of the SQL statement as shown in bold in

```
% SqlCommandString is any string or character literal
%   beginning with an SQL query verb
tokens

   SqlCommandString   "'SELECT #'*'"
                    | "'INSERT #'*'"
                    | "'DELETE #'*'"
                    | "'UPDATE #'*'"
                    | "\"SELECT #\"*\""
                    | "\"INSERT #\"*\""
                    | "\"DELETE #\"*\""
                    | "\"UPDATE #\"*\""
        % and any others needed
end tokens

% SqlExpr is any string concatenation expression
%   that begins with one of the above magic words

define SqlExpr
   [SqlCommandString] [CatAddExpr*]
end define
```

Figure 5. PHP grammar extension to recognize guest patterns (SQL statements)

Figure 6. The SQL command that is constructed by the code in the Figure is the string:

"INSERT INTO phpbb_themes(themes_id, template_name, style_name) VALUES('8', 'test', 'test_style');"

At the same time, the assignment statements inserted by our instrumentation construct a second string:

"INSERT INTO phpbb_themes($db_fields[$i], $db_fields[$i], $db_fields[$i]) VALUES ($db_values[$i], $db_values[$i], $db_values[$i]);"

Just before the SQL command string is sent to the database, both strings are inserted into the instrumentation database. This gives us a runtime snapshot that contains not only the values used in the actual SQL query, but also the PHP variables from which the query was constructed. Figure 8 shows the TXL transformation rule `instrumentQueriesSource` that identifies an assignment containing an SQL Command String and adds the instrumentation assignments.

The instrumented SQL statement is constructed in the TXL variable `SQLE_Source`. The parts of the SQL expression are passed as a parameter to the `collectparameters` function which processes them one part a time. It classifies each SQL part it receives into one of three categories:

1. *Constant variables*. Constant variables are the same for all SQL statement instances, so the function concatenates them to the result without quotes , adding the actual values at run time. The first assignment shown in Figure 6 references the constant variable, `THEMES_TABLE`. This variable, which contains the name of the table, is concatenated without quotes in the instrumentation assignment on the next line, shown in bold. Note that at this point in time, the assignment statements have yet to be generated. The function is constructing the expressions that will occur on the right hand side of the instrumentation assignment statements.

```
<?php
...
{
$sql = "INSERT INTO ".THEMES_TABLE." (";
$GLOBALS ["Sql_Source"] [594] [0] = 'INSERT INTO '.THEMES_TABLE.' (';
$GLOBALS ["Sql_Source"] [594] [1] = 516;
$GLOBALS ["Sql_index"] = 594;
}
for ($i = 0; $i < count ($db_fields); $i ++)
{
    {
          $sql.= $db_fields [$i];
          $GLOBALS ["Sql_Source"] [594] [0].= '$db_fields [$i]';
    }
    if ($i != (count ($db_fields) - 1))
    {
          {
                $sql.= ", ";
                $GLOBALS ["Sql_Source"] [594] [0].= ', ';
          }
    }
}
{
    $sql.= ") VALUES (";
    $GLOBALS ["Sql_Source"] [594] [0].= ''.') VALUES (';
}
for ($i = 0; $i < count ($db_values); $i ++)
{
    {
          $sql.= "'".$db_values [$i]."'";
          $GLOBALS ["Sql_Source"] [594] [0].= '$db_values [$i]';
    }
    if ($i != (count ($db_values) - 1))
    {
          {
                $sql.= ", ";
                $GLOBALS ["Sql_Source"] [594] [0].= ', ';
          }
    }
}
{
    $sql.= ")";
    $GLOBALS ["Sql_Source"] [594] [0].= ')';
}

...

?>
```

Figure 6. Instrumented snippet of code for PhpBB2.0 application - 1

Sections in boldface have been added by our instrumenting transformation. This example demonstrates how the dynamic SQL statement is constructed from its fragments using forward flow analysis.

2. *PHP variables*. These variables are bound to different values at run time, generating different versions of the SQL statement. The function concatenates a quoted version of the variables to the result, protecting the variable from runtime substitution. In Figure 6, the assignments of the PHP variables, $db_fields[$i] and $db_values[$i], are quoted when concatenated to the instrumentation variable `$GLOBAL["Sql_source"][549][0]`.

   In some situations, an SQL statement is constructed in fragments using multiple PHP variables before assembling the final SQL statement. Currently, we are using naming conventions (ending in `_sql`) to identify the part variables, which are collected in a TXL global variable. Once identified, we instrument each

```
<?php
  ...
 {
   $join_sql_table = (! $post_id) ? '' : ", ".POSTS_TABLE." p, ".POSTS_TABLE." p2 ";
   $GLOBALS ["SqlParts"] ['join_sql_table'] = (((! $post_id)) ? ('') : (', ').(POSTS_TABLE).(' p, ').(POSTS_TABLE).(' p2 '));
 }
 {
   $join_sql = (! $post_id) ? "t.topic_id = $topic_id" : "p.post_id = $post_id AND t.topic_id = p.topic_id AND
   p2.topic_id = p.topic_id AND p2.post_id <= $post_id";
   $GLOBALS ["SqlParts"] ['join_sql'] = (((! $post_id)) ? ('t.topic_id = $topic_id') : ('p.post_id = $post_id AND
                        t.topic_id = p.topic_id AND p2.topic_id = p.topic_id AND     p2.post_id <= $post_id'));
 }
 {
   $count_sql = (! $post_id) ? '' : ", COUNT(p2.post_id) AS prev_posts";
   $GLOBALS ["SqlParts"] ['count_sql'] = (((! $post_id)) ? ('') : (', COUNT(p2.post_id) AS prev_posts'));
 }
 {
   $order_sql = (! $post_id) ? '' : "GROUP BY p.post_id, t.topic_id  ORDER BY p.post_id ASC";
   $GLOBALS ["SqlParts"] ['order_sql'] = (((! $post_id)) ? ('') : ('GROUP BY p.post_id, t.topic_id  ORDER BY p.post_id ASC'));
 }
 {
   $sql = "SELECT t.topic_id, t.topic_title, t.topic_status, t.topic_replies, t.topic_last_post_id, f.forum_name,
           f.forum_status,f.forum_id, ".$count_sql."FROM ".TOPICS_TABLE." t, ".FORUMS_TABLE. " f".$join_sql_table.
           " WHERE $join_sql AND f.forum_id = t.forum_id  $order_sql";
   $GLOBALS ["Sql_Source"] [394] [0] = (('SELECT t.topic_id, t.topic_title, t.topic_status, t.topic_replies,
           t.topic_last_post_id, f.forum_name, f.forum_status, f.forum_id,).((''.$GLOBALS ["SqlParts"] ['count_sql'].'')).
           ('FROM  ').(TOPICS_TABLE).(' t, ').(FORUMS_TABLE).(' f').((''.$GLOBALS ["SqlParts"] ['join_sql_table'].'')).
           (('WHERE '.$GLOBALS ["SqlParts"] ['join_sql'].('AND f.forum_id = t.forum_id'.$GLOBALS ["SqlParts"] ['order_sql'].''))));
   $GLOBALS ["Sql_Source"] [394] [1] = 391;
   $GLOBALS ["Sql_index"] = 394;
 }
  ...
?>
```

Figure 7. Instrumented snippet of code for PhpBB2.0 application - 2

Sections in boldface have been added by our instrumenting transformation. This example shows how the dynamic SQL statement is expanded from its fragments using backward flow analysis

of the variables in a similar manner. In Figure 7, the SQL statement assigned to `$sql` is constructed from parts contained in the $\$count\_sql$, $\$join\_sql\_table$, $\$join\_sql$, and $\$order\_sql$ variables. The Figure shows how each of the part variables are instrumented and collected in the global array `SqlParts`. The final assembly of the SQL statement is shown at the bottom of the Figure. The instrumentation string is constructed by concatenating the parts that were previously stored in the `SqlParts` array.

3. *String expressions.* In PHP, string expressions can have two forms: double quoted and single quoted strings. The difference between the two forms is that embedded PHP variables will be substituted in double quoted strings but not in single quoted strings. Thus the `collectparameters` function transforms double quoted strings into single quoted ones to protect embedded variables, unless the variables are recognized as SQL fragment variables. In Figure 7 the string literals in the SQL fragment assigned to the PHP variable, $join_sql have been changed to single quotes when assigned to the instrumentation array.

Once the expressions have been collected, a unique identifier for the SQL statement is generated. The TXL rule in Figure 8 generates the assignment statements to collect the SQL instrumentation strings in the PHP global array `Sql_Source` using the unique identifier. The unique identifier for the page (`uniquePageid`) is also stored in the array so that it can be stored in the database to link the SQL statement to the page from which it was generated.

The TXL rule `instrumSqlStmParts` is called on the remaining source code to search for and mark any concatenation statement that may contribute to the construction of the SQL statement identified in the rule. It takes as parameters the PHP variable from the left hand side of the assignment, and the statement's unique identifier. Figure 6 shows how the SQL statement with the unique id 594 is constructed from it's fragments that are scattered in conditional statements and loops.

3.4.3 Binding SQL Statement Source with its Runtime Instances

In the previous subsection, we identified and globally instrumented SQL statement construction. We now identify and instrument the actual SQL execution points, combining the instantiated SQL statements with the SQL statement source. The TXL rule `instrumentSQL`, shown in Figure 9, identifies calls to the `mysql_query` PHP function and replaces them with the instrumentation function `mysql_query_track()`. This function takes as parameters the the original function's SQL statement Q and database connection R, as well as two additional instrumentation parameters: the SQL statement source, from the global array `"Sql_Source"` (mapped by the SQL statement unique identifier in the global variable `"Sql_index"`), and the SQL statement's unique identifier.

```
rule instrumentQueriesSource
   % Get the page ID for the page containing the SQL query source construction
   import uniqePageid [id]

   % Find the first statement of any SQL query source construction
   replace [TopStatement*]
      OCV [ObjectCVar] AsOp [AssignOp] SqlE [SqlExpr] ;
      Rest [TopStatement*]
   . . .
   % Collect and expand SQL statement source
   construct SQLE_Source [Expr]
      SqlE [collectparameters]

   % Create a unique id for the constructed SQL statement source
   construct uniqeid [id]
      _ [!]

   % Replace the statement with an instrumented version
   by
      {
         % Original statement
         OCV AsOp SqlE;
         % Added instrumentation statements
         '$GLOBALS '[ "Sql_Source" '] '[ uniqeid '] '[ 0 '] = SQLE_Source ;
         '$GLOBALS '[ "Sql_Source" '] '[ uniqeid '] '[ 1 '] = uniqePageid ;
         '$GLOBALS '[ "Sql_index" '] = uniqeid ;
      }
      % And instrument any following SQL query source construction fragments
      Rest [instrumSqlStmParts OCV uniqeid]
end rule
```

Figure 8. The `instrumentQueriesSource` transformation rule

The `instrumentQueriesSource` rule captures each SQL statement and transforms it into its source by collecting the statement fragments and manipulating them to keep any embedded variable unsubstituted

```
rule instrumentSQL
   replace [Expr]
      'mysql_query ( Q [Expr], R [Expr] )
   by
      'mysql_query_track ( 'mysql_query, Q, R,
         '$GLOBALS '[ "Sql_Source" '] '[ '$GLOBALS '[ "Sql_index" '] '],
         '$GLOBALS '[ "Sql_index" '] )
end rule
```

Figure 9. The `instrumentSQL` transformation rule

The `instrumentSQL` rule captures each SQL execution statement and transforms it into a call to an instrumenting function that combines the globally constructed SQL statement source with it's execution instance

3.4.4 Analyzing SQL Statement Sources

During runtime execution of the instrumented PHP application, the instrumentation function `mysql_query_track()` is called to insert trace elements into the `Database Interaction` database table. This function invokes a TXL program that parses the SQL source statement, identifying the statement type, the application variables and the database tables used in the statement. It also identifies any PHP variables embedded in the SQL statement and associates them with the syntactic part in which they are used. For example, the program will identify whether a PHP variable is used in a `WHERE` clause or an `ORDER BY` clause. The results of this analysis are also stored in the `Database Interaction` table so that PHP variables can be directly linked to the run-time values and database interactions they control.

4 Evaluation

We have evaluated our approach by analyzing several production dynamic web applications, two of which are *PhpBB 2.0* (a popular internet forum system), and *Moodle* (a popular open source course management system). We use Web Application Testing In Ruby (WATIR) [6, 19], a library used to script web browsers, to help automate the collection of usage traces.

Tables 1, 2, and 3 show a subset of the results of one test case (visits to `Page_ID` 15 and 17) for an anonymous user interacting with a PhpBB forum. The tables also show some of the HTTP variables and database interactions generated by the visits. Based on the `Page_ID` values, these three tables are joined into a single view for ease of analysis.

In the tables we can see that the `viewforum` page, with `Page_ID` 15, passes values to the HTTP Get vari-

Page_ID	Page_Name	Page_Param	Prev _ID	Page Type	Page_Acc_Ts
15	http://phpBB2/viewforum.php	?f=1&sid=01e7ff1f13225be3cb129f8	14	PHP	1239318861
17	http://phpBB2/viewtopic.php	?t=1&sid=01e7ff1f13225be3cb129f8	16	PHP	1239318876

Table 1. Sample trace elements for the *Server Pages* database table

Var_ID	Page ID	HttpVar Name	HttpVar Value	HttpVar Type	Assigned Var	HttpVar_Acc Time
34	15	f	1	GET	$forum_id	1239318862
35	15	phpbb2mysql_data	a:2:{s:11:"autologinid";s:0:"";s:6:"userid";i:-1;}	COOKIE	$sessiondata	1239318863
36	15	phpbb2mysql_sid	01e7ff1f13225be3cb12b89857d3f9f8	COOKIE	$session_id	1239318863

Table 2. Sample trace elements for the *Http Variables* database table

Traces related to PageID 15, `viewforum.php`

SqlStm _ID	Page ID	Sql_Instance	EntityName	SqlTransTime	Sql_Source	Sql_Parts
2787	15	SELECT * FROM phpbb_config	phpbb_config	1239318862	SELECT * FROM phpbb_config	SelectStm SelectExpr *
2788	15	SELECT * FROM phpbb_forums WHERE forum_id = 1	phpbb_forums	1239318863	SELECT * FROM phpbb_forums WHERE forum_id = $forum_id	SelectStm WHERECLAUSEVAR$forum_id WhereExpr forum_id = $forum_id SelectExpr *
2789	15	SELECT * FROM phpbb_themes WHERE themes_id = 1	phpbb_themes	1239318864	SELECT * FROM phpbb_themes WHERE themes_id = 1	SelectStm WhereExpr themes_id = 0 SelectExpr *
2813	17	UPDATE phpbb_topics SET topic_views = topic_views + 1 WHERE topic_id = 1	phpbb_topics	1239318882	UPDATE phpbb_topics SET topic_views = topic_views + 1 WHERE topic_id = $topic_id	UpdateStm WHERECLAUSEVAR$topic_id WhereExpr topic_id = $topic_id SetList topic_views = topic_views + 1

Table 3. Sample trace elements for the *Database Interactions* database table

Traces related to PageIDs 15 and 17, `viewforum.php` *and* `viewtopic.php`

able `f` and the HTTP Cookies variable `sid`. Table 2, *HTTP Variables*, shows that the `f` Get variable is assigned to the $\$forum\_id$ PHP variable, and `sid` is assigned to the $\$session\_id$ PHP variable. Table 3, *Database Interactions*, shows some of the SQL statements generated from this interaction. We can see that `SQLStm_ID` 2788 uses the PHP variable $\$forum\_id$ to retrieve the forum information, while the other database interactions shown for this page visit do not depend on user inputs to perform their transactions. The *Database Interactions* table columns are populated as a result of the execution of the instrumented application except the two columns, *EntityName* and *SQl_Parts*, which come from analysis of the SQL source statements using TXL. The column *EntityName* shows the name of database tables that the SQL statements are performed on. Column *Php_Parts* shows three pieces of information, the SQL statement type (`SelectStm`, `UpdateStm`, and so on), the embedded PHP variable and its syntactic location within the SQL statement (such as the `WHERE` clause), and the statement's `WHERE` and `SELECT` expressions. This example illustrates the key benefit of our approach, linking the runtime instance of the query to the elements used to assemble it including PHP variables and HTTP request variables.

Table 4 shows some statistics for the test scenario of an anonymous user visiting a PhpBB forum. It shows the number and type of SQL statements executed during this interaction, the number of SQL statements that depend on user inputs, the location of the embedded PHP variables in the SQL statement, and the number and the type of the HTTP variables used in the interaction. Table 4 shows the total number of pages visited during this interaction as well as the number of filtered client pages stored and analyzed by

Filtered client pages	Utility pages visited	Client pages generated	All visited pages
20	180	50	230

	No. of SQL statements				SQL statements Use of HTTP Variables		No. of Http Variables			
Statement \Variable type	SEL	INS	DEL	UPD	Where clause	Others	ALL	POST	GET	COOKIE
All traces	115	0	0	1	14	3	51	0	15	35
Index.php	22	0	0	0	2	0	8	0	0	8
viewtopic.php	9	0	0	1	3	1	3	0	1	2
viewforum.php	11	0	0	0	6	1	3	0	1	2

Table 4. Trace statistics for anonymous user interactions with a PhpBB 2.0 forum

our tool. In the work described in this paper, both the SQL query source and the runtime SQL instances are collected at run time. In a new experiment we have shown how the SQL source statements can be extracted statically by slicing the instrumentation aspect of an instrumented PHP application into a separate PHP program. When executed offline, this program inserts the original SQL statement sources into the database. The runtime SQL commands are then collected dynamically when the application is executed and related to the original SQL statement sources as described in section 3.4.4. The dynamic approach of this paper adds an average overhead of approximately 70%, while the static optimization reduces the average overhead to 30%.

5 Related Work

Several techniques have been proposed to support understanding and analysis of web applications using reverse engineering. A detailed study of the state of the art in this field can be found in our recent survey [3]. To the best of our knowledge this is the first approach that dynamically analyzes database interactions in combination with other web application basic elements of information, such as pages, server environment variables, application variables and session and cookie management functions. The combined information gained from all of those sources provides a strong infrastructure that can serve many analysis tasks requiring precise fined-grained information, such as web application security analysis.

The problems of identifying and analyzing database interactions have been previously studied for standard systems. For instance, Cleve and Hainaut [7] use aspect-based tracing to relate and extract the basic components of the dynamic SQL query. This includes the basic dynamic query, the variable parts of the query, the query result, in addition to some environment variable such as the class and the line number in which the query called. The authors provide three kinds of post analysis of the trace elements after the program is executed which include constant/variable identification, value-based dependency analysis and static statement restructuring. However their approach works only with prepared statements and does not handle the case where the SQL statement is constructed in string variables and passed as a string to the database API. Their approach is also yet to be evaluated on a production system.

Brink et al. [18] propose a tool for assessing the quality of database interactions in standard applications. They first extracted embedded SQL statements using control and dataflow analysis. The identification of SQL string literals are done using a standard Java program that tokenizes the source program based on predefined SDF grammars. Then they collected the identified parts in a query object which includes information about the reconstructed query, its location, and name and type of variables required for the reconstructed query. A post analysis is done over the extracted queries for quality assessment purposes. This analysis is done for PL/SQL, COBOL, Visual basic, and Java. The authors' aim is to extract the queries for quality assessment, while our aim is to reverse engineer a web application to gain a rich infrastructure that can support different kind of analysis including quality assessment of database interactions. While the identification of SQL queries in this work is similar to ours, using source transformation technology we combine the process of parsing, pattern matching and flow analysis into a single coherent step, yielding a faster and more flexible analysis with more accurate results.

Ngo and Tan [15] propose an automatic static technique to extract database interaction points from web applications. The approach first identifies all program paths that include a database interaction and slice them out as an interaction Control flow Graph (ICFG), then each interaction path is symbolically executed, and all possible interaction types are derived from the generated symbolic expression using inference rules. Evaluating the approach on a case study, the approach is able to extract 80% of the database interactions. The complexity of the extraction process is high as it is composed of 5 stages, and is affected by factors such as number of the interaction paths (i-paths), and the length and complexity of each ipath. The authors also do not specify how to handle SQL statements constructed from sequences of string concatenations. Table 5 summarizes and compares related work.

Approach	Static\ Dynamic Analysis	Host language \Application	Extendable	Client Application Source code	Parsing tech.
Instrumenting transformation (WAFA)	S\D	PHP \Web Applications	Yes	Syntactic modification	Island grammar
Tracing Aspects [7]	D	Java	No	No modification	
Symbolic execution and inference rules [15]	S	PHP\Web Applications	Yes	No modification	Independent parsers
Control and flow analysis [18]	S	PL/SQL, COBOL, Visual Basic	Yes	No modification	JJForester(Parser) ANTLR(tokenizer)

Table 5. Related work comparisons

6 Future Work and Conclusions

We have presented WAFA, an automated reverse engineering approach to recover fine-grained interaction behavior of dynamic web applications. To the best of our knowledge, our approach is the first one to extract the web application's embedded SQL subsystem, which includes both the original SQL statement source as well as corresponding execution instances, and an analysis to attach it to both static host application variables and dynamic server environment variables.

We are currently expanding the set of test cases for PhpBB and Moodle, and plan to extend our evaluation to other PHP-based applications. Our approach is primarily aimed at server side code, since we have been working with traditional PHP-based web applications. AJAX requests can also be, and in many cases are, implemented in PHP. When used with AJAX, our technique can be used to directly link HTTP request variables to the database interactions of the AJAX request. This may help in analysis of AJAX applications as well as traditional applications.

References

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean. A Verification Framework for Access Control in Dynamic Web Applications. In *C3S2E, Canadian Conference on Computer Science and Software Engineering, Montral*, ACM International Conference Proceeding Series, pages 109–113, 2009.

[2] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Automated Reverse Engineering of UML Sequence Diagrams for Dynamic Web Applications. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 295–302, Denver, USA, 2009.

[3] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Modeling methods for web application verification and testing: State of the art. *Software Testing, Verification and Reliability*, 2009 (in press).

[4] M. H. Alalfi, J. R. Cordy, and T. R. Dean. DWASTIC: Automating Coverage Metrics for Dynamic Web Applications. In *SAC, The 2009 ACM Symposium on Applied Computing, Switzerland*, (submitted).

[5] G. Antoniol, M. Di Penta, and M. Zazzara. Understanding Web Applications through Dynamic Analysis. In *IWPC 2004, 12th International Workshop on Program Comprehension*, pages 120–131, 2004.

[6] Canoo Engineering. Canoo WebTest, *http://webtest.canoo.com*, accessed 30 April 2009.

[7] A. Cleve and J.-L. Hainaut. Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering. In *WCRE 2008, 15th Working Conference on Reverse Engineering*, pages 192–196, October 2008.

[8] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[9] G. A. Di Lucca and M. Di Penta. Considering Browser Interaction in Web Application Testing. In *WSE 2003, 5th International Workshop on Web Site Evolution*, pages 74–81. IEEE Computer Society, 2003.

[10] G. A. Di Lucca and M. Di Penta. Integrating Static and Dynamic Analysis to improve the Comprehension of Existing Web Applications. In *WSE 2005, 7th IEEE International Workshop on Web Site Evolution*, pages 87–94, 2005.

[11] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Tramontana. Supporting Web Application Evolution by Dynamic Analysis. In *IWPSE 2005, 8th International Workshop on Principles of Software Evolution*, pages 175–186, 2005.

[12] W. G. J. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE 2006, 28th International Conference on Software Engineering, Shanghai, China, May 20-28*, pages 795–798, 2006.

[13] G. A. D. Lucca, A. R. Fasolino, and P. Tramontana. Reverse engineering Web applications: the WARE approach. *Journal of Software Maintenance*, 16(1-2):71–101, 2004.

[14] L. Moonen. Lightweight Impact Analysis using Island Grammars. In *IWPC 2002, 10th International Workshop on Program Comprehension*, pages 219–228, June 2002.

[15] M. N. Ngo and H. B. K. Tan. Applying static analysis for automated extraction of database interactions in web applications. *Information & Software Technology*, 50(3):160–175, 2008.

[16] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *ICSE 2001, 23rd International Conference on Software Engineering*, pages 25–34, 2001.

[17] N. Synytskyy, J. R. Cordy, and T. R. Dean. Robust multilingual parsing using island grammars. In *CASCON 2003, Conference of the Centre for Advanced Studies on Collaborative Research*, pages 266–278, October 2003.

[18] H. van den Brink, R. van der Leek, and J. Visser. Quality Assessment for Embedded SQL. In *SCAM 2007, 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 163–170, 2007.

[19] WatirCraft. WATIR, *http://wtr.rubyforge.org*, accessed 30 April 2009.