

Automating Coverage Metrics For Dynamic 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—Building comprehensive test suites for web applications poses new challenges in software testing. Coverage criteria used for traditional systems to assess the quality of test cases are simply not sufficient for complex dynamic applications. As a result, faults in web applications can often be traced to insufficient testing coverage of the complex interactions between the components. This paper presents a new set of coverage criteria for web applications, based on page access, use of server variables, and interactions with the database. Following an instrumentation transformation to insert dynamic tracking of these aspects, a static analysis is used to automatically create a coverage database by extracting and executing only the instrumentation statements of the program. The database is then updated dynamically during execution by the instrumentation calls themselves. We demonstrate the usefulness of our coverage criteria and the precision of our approach on the analysis of the popular internet bulletin board system PhpBB 2.0.

Keywords—Testing; Reverse Engineering; Maintenance and Enhancement; Web Applications.

I. INTRODUCTION

Testing is one of the most essential yet complex activities in web application development and maintenance. The dynamic distributed structure of web applications poses new challenges to building comprehensive test suites. Users interact with application pages, providing various inputs that are used to instantiate the server environment variables. These variables are then used to interact with the database backend, retrieving information used to dynamically construct new client pages to be sent back to the users.

Database interaction is the most critical part of this cycle, and often requires extensive testing. For this reason, several approaches have been proposed to assess the correctness of database interactions in standard database systems, for example Cabal and Tuya [2] and Wilmor and Embury [14].

Coverage metrics have been proposed on different levels of granularity for SQL statements either as an isolated component or as an embedded component in the whole system. However, most of these approaches either do not provide automation for coverage assessment, or do not consider other kinds of interactions. Our approach is specialized for web applications, handling similar issues to those tailored for conventional database applications while at the same time addressing the new challenges related to the distributed and dynamic structure of web applications. We have implemented our approach in an extendable and precise tool.

The contributions of this paper are:

- A set of coverage criteria for the testing of dynamic web applications. This can be used to assess thoroughness and the adequacy of test suites applied on different levels, such as the page access level, the server environment variable level, and the database level, as well as interactions between the levels.
- An extendable, automated approach and tool to instrument, collect and analyze the coverage information. The tool also statically extracts and analyzes the embedded SQL subsystem from dynamic web applications.

Web Application Testing

Our tool, called DWASTIC (Dynamic Web ApplicationS Testing Instrumentation Coverage), can be used to support many kinds of testing activities for web applications. It focuses the testing efforts on the vulnerable parts of the code, which are most likely the source of web application faults and attacks such as SQL injection. It also provides a direct way to identify the parts of the code that are not covered by test cases.

DWASTIC is an essential part of a framework aimed at testing the conformance of dynamic web applications with role-based access control security policies [1]. In the framework, a role-based access control (RBAC) security model is recovered from the dynamic web application using a combination of static and dynamic analysis techniques. This paper explains how DWASTIC is used to augment the dynamic analysis with instrumentation for code coverage. This helps to decrease the number of false positives due to an analysis that yields a model that only partially covers the code (leading to verification of properties that may in fact not hold).

The rest of this paper is structured as follows. Section II presents our proposed coverage metrics. Section III presents the details of our approach, and Section IV presents an example that demonstrates our method on a real system. Section V relates our efforts to previous work. Finally, Section VI outlines our conclusions and plans for future work.

II. WEB APPLICATION COVERAGE METRICS

In this section we propose three test coverage dimensions that are specifically tailored to web applications: web application pages, server environment variables and database

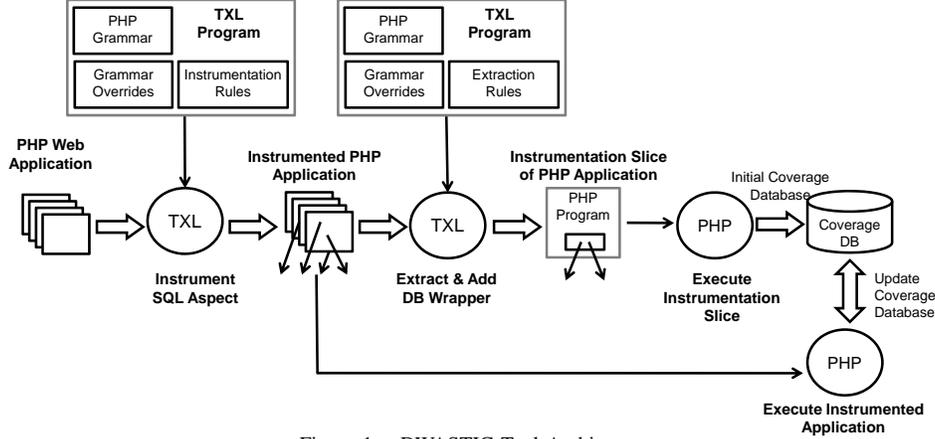


Figure 1. DWASTIC Tool Architecture

interactions. These are not meant to replace traditional code coverage metrics, rather to augment them with specific coverage measures for the client and database interaction aspects of dynamic web applications as well. Our criteria subsume the criteria proposed for database systems [14], [2] and include new measures that are specialized for web applications. The concern in web application testing is whether the application as a whole behaves as specified, and this cannot be determined without thorough testing of all three levels of interaction.

While the coverage criteria we propose can be used to support many different testing activities, our specific aim is to provide a completeness measure for extracting an access control security model from a web application under test. This requires that we ensure coverage of all client pages that can be generated from the application, all database interactions applied on application entities, and all user inputs passed to server pages or SQL statements that can influence the dynamically constructed client pages. In the following subsections we elaborate on our proposed criteria and how they can serve our aim.

A. Page Access Coverage

Page coverage measures the adequacy of test cases for ensuring that all server pages are executed at least once and are running properly. The measure can be expressed as:

$$\text{Page Coverage} = \frac{\#ofcov.Pages}{total\#applicationpages}$$

The equation measures the ratio of executed server pages to the total number of the application server pages.

B. SQL Statement Coverage

SQL statement coverage measures the adequacy of test cases to insure that all possible SQL statements, including dynamically constructed ones, are tested at least once. These statements are different from those which perform API calls to issue commands to the database, such as `mysql_query($SQL_Statement, db_connect_id)` in PHP. Using the terminology introduced by the database

community, these API calls are called database interaction points. Coverage based on database interaction points is not sufficient, as each specific database interaction point can issue multiple forms of dynamically constructed SQL statement.

Our SQL statement coverage measure is done both on the level of the whole web application and on the level of each individual server page. The measure based on the application level can be expressed as:

$$\text{SQL_Stm Coverage} = \frac{\#ofcov.SQL_Stm}{total\#ofapplicationSQL_Stms}$$

The measure on the page level can be expressed as:

$$\text{Page_SQL_Stm Coverage} = \frac{\#ofcov.SQL_StmsinaPage}{total\#ofSQL_StmsinthePage}$$

C. Server Environment Variable Coverage

Server environment variables are variables returned by HTTP forms on generated pages using GET or POST. Server environment variable coverage measures the adequacy of test cases to insure the coverage of all server environment variables at the level of the web application, the individual server page level, and the SQL statement level. The measure based on the application level can be expressed as:

$$\text{Server_Env_Var Cov.} = \frac{\#ofpopulatedServer_Env_Var}{total\#oftheapplicationServer_Env_Var}$$

The measure for the page level can be expressed as:

$$\text{Page_Server_Env_Var Cov.} = \frac{\#ofcov.Server_Env_Varincov.Pages}{total\#ofServer_Env_Varinapage}$$

And the measure for the SQL statement level can be expressed as:

$$\text{SQL_Server_Env_Var Cov.} = \frac{\#ofcov.SQL_StmwithServer_Env_Var}{total\#ofSQL_StmwithServer_Env_Var}$$

In each case the metric measures the ratio of the number of server environment variables covered to the total number of variables used at the different levels.

III. CONSTRUCTING THE COVERAGE DATABASE

Figure 1 shows the architecture of our approach. An instrumentation transformation is used to analyze the source code of the application, identifying and globally marking

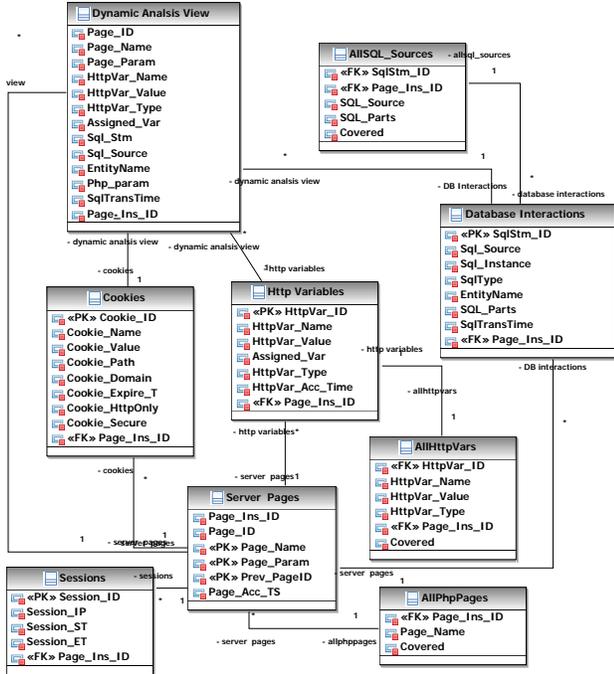


Figure 2. Dynamic Analysis database model

instances of the coverage criteria and inserting appropriate calls to an instrumentation coverage library developed in PHP. These instrumentation calls will update the coverage database as the program is executed (Figure 1).

In order to insure that we have an accurate table of all of the instances to be covered, the initial coverage database itself is automatically derived from the instrumented application. This is done by slicing the instrumentation statements from the instrumented PHP code into a separate PHP program augmented with database calls (Figure 1). As the augmented slice is executed, the program builds the initial coverage database by adding a coverage table entry to a global array as each instrumentation statement in the slice is executed. During insertion, another transformation also analyzes the type of the SQL statement to identify and insert its components into the database. The resulting arrays then become the initial coverage tables in the database.

This slicing method is necessary in order to capture all instances of SQL statements that will be dynamically constructed from string fragments, concatenations and function calls before being passed to the database interface. If we were not to handle these cases, a large fraction of the SQL database interactions would be missed, invalidating our database interaction coverage metrics, and it would not be possible to accurately attach server environment variables to the SQL statements that use them, invalidating our server environment variable coverage metrics. The construction of the SQL SELECT statement from fragments in a dynamic page from PhpBB 2.0 shown in Figure 4 is a typical example.

PageIndex	PageName	Covered
324	C:\WAMP\WWW\PHPBB2\search.php	0
365	C:\WAMP\WWW\PHPBB2\viewforum.php	0
388	C:\WAMP\WWW\PHPBB2\viewonline.php	0
391	C:\WAMP\WWW\PHPBB2\viewtopic.php	0
421	C:\WAMP\WWW\PHPBB2\adminadmin_board.php	0
450	C:\WAMP\WWW\PHPBB2\adminadmin_disallow.php	0

Table I

Example *AllPhpPages* coverage database table, tracking page coverage

HttpIndex	PageIndex	Assigned_Var	HttpName	HttpType	Covered
376	365	\$forum_id	f	GET	0
377	365	\$forum_id	f	POST	0
378	365	\$forum_id	forum	GET	0
379	365	\$start	start	GET	0
381	365	\$mark_read	mark	POST	0
380	365	\$mark_read	mark	GET	0
382	365	\$tracking_forums	f	COOKIE	0
383	365	\$tracking_topics	t	COOKIE	0
384	365	\$tracking_topics	t	COOKIE	0
385	365	\$tracking_forums	f	COOKIE	0
387	365	\$topic_days	topicdays	POST	0
386	365	\$topic_days	topicdays	GET	0

Table II

Example *AllHttpVars* coverage database table, tracking server environment variable coverage

The complete schema for our dynamic analysis database, including the coverage database, is shown in Table 2. It is comprised of eight tables and one view. Three of the tables, *AllPHPPages*, *AllHttpVars* and *AllSQL_Sources*, are constructed to hold coverage information and are initialized statically by our approach. During execution of the web application, these tables are updated at each instrumentation call to track coverage of page access, server environment variable access, and database SQL statement forms respectively. Tables I, II and III show examples of these coverage tables as initialized by our static instrumentation slice.

The *Server Pages* table (Figure 2) is used to keep track of access to individual pages, and is associated with the *AllPHPPages* table, which contains information about all of the application pages, while the other 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 *AllHttpVars* table is associated with the *HttpVars* table and holds coverage information related to server environment variables. The *AllSQL_Sources* is associated with the *Database Interactions* table and holds coverage information related to database interactions.

A. Instrumenting Web and SQL Aspects

We automatically analyze and add source code instrumentation to web application source using TXL [4], a programming language designed for manipulating and experimenting with programming language notations and features. TXL is a powerful source transformation system that has been

SQL_index	PageIndex	SQL_Source	Covered
366	365	SELECT * FROM phpbb_forums WHERE forum_id = <i>\$forum_id</i>	0
367	365	SELECT MAX(post_time) AS last_post FROM phpbb_posts WHERE forum_id = <i>\$forum_id</i>	0
372	365	SELECT g.group_id, g.group_name FROM phpbb_auth_access aa, phpbb_user_group ug, phpbb_groups g WHERE aa.forum_id = <i>\$forum_id</i> AND aa.auth_mod = 1 AND g.group_single_user = 0 AND g.group_type <> 2 AND ug.group_id = aa.group_id AND g.group_id = aa.group_id GROUP BY g.group_id, g.group_name ORDER BY g.group_id	0
373	365	SELECT COUNT(t.topic_id) AS forum_topics FROM phpbb_topics t, phpbb_posts p WHERE t.forum_id = <i>\$forum_id</i> AND p.post_id = t.topic_last_post_id AND p.post_time >= <i>\$min_topic_time</i>	0
374	365	SELECT t.*, u.username, u.user_id, u2.username as user2, u2.user_id as id2, p.post_time, p.post_username FROM phpbb_topics t, phpbb_users u, phpbb_posts p, phpbb_users u2 WHERE t.forum_id = <i>\$forum_id</i> AND t.topic_poster = u.user_id AND p.post_id = t.topic_last_post_id AND p.poster_id = u2.user_id AND t.topic_type = 2 ORDER BY t.topic_last_post_id DESC	0

Table III
Example *AllSqlsources* coverage database table, tracking SQL statement coverage

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. While our process is presently targeted at PHP and MySQL, this lightweight TXL-based process is adaptable in plug-and-play fashion to deal with other database engines and server page scripting languages such as JSP or ASP. Documents that include a mixture of languages and technologies can be handled using island grammars [8][11], where the interesting elements, PHP code in our case, are considered islands, and uninteresting elements, HTML code and other text in our case, are considered water. Using island grammars simplifies the parsing process as interesting elements can be identified and analyzed without parsing the whole document.

The instrumenting transformation process is used to serve two major purposes. The first is to globally mark, for further static extraction and processing, coverage information on the level of page access, server environment variable access and database interactions, and the second is to insert appropriate calls to an instrumentation coverage library developed in PHP to dynamically update the coverage database as the application under test is executed. The following subsections provide the details of this process.

1) *Instrumenting Page Access*: In DWASTIC, application sources are processed statically, one page a time. A set of rooted transformation rules is applied on each page to mark, extract and analyze coverage information. When a page is sent to DWASTIC for processing, the TXL main rule imports the page's full path name and generates a unique page identifier in the TXL global variable `uniquePageid`, which will be used for any further analysis associated with this page. The transformation rule `instrumentPage` (Figure 3) is then called to add dynamic instrumentation at the top of the page. `instrumentPage` constructs a block of statements in the `PageIns` variable, which includes adding a new entry

```

rule instrumentPage InputFile [stringlit]
% Get global unique page id generated by the main rule
import uniquePageid [id]

% Transform the entire page exactly once
replace $ [Document]

PHPO [PHPOpenTag]
TopS [TopStatement*]
PHPC [PHPCloseTag]

% Insert page instrumentation and coverage code
by
PHPO
`{
'global $$Sql_index;
'global $PhpFileName;
'global $PhpFileIndex;
'global $Http_Source;
'$GLOBALS['PhpFileName']=[uniquePageid] = InputFile;
'$GLOBALS['PhpFileIndex'] = uniquePageid;
'include_once('sensfuncDBJan92009.php');
'IsCovered '('$GLOBALS['PhpFileIndex']');
`}
TopS
PHPC
end rule

```

Figure 3. The TXL `instrumentPage` rule adds page coverage instrumentation to the top of each processed page

to the PHP global array `$GLOBALS["PhpFileName"]` to represent the current processed page name and unique ID. A call to the coverage function `IsCovered` is added to track the page access at run time and to update the entry for this page ID in the coverage database. Other statements added to the `PageIns` block are used to assist in globally defining other PHP arrays associated with other coverage information related to database interactions and server environment variables, as well as a call to our PHP instrumentation coverage library `sensfuncDBJan92009.php`. The first few lines of Figure 4 show the result of adding page coverage instrumentation to the `search.php` page.

2) *Instrumenting Server Environment Variables*: Each page is also transformed by a specialized TXL rule to identify server environment variables, replacing them with an instrumentation function which collects the server environment variable's names, values, and the PHP variables which receive the values. This information is passed as parameters to the instrumentation function `HttpVar_track()`. The server environment variables are also added to the PHP user defined global array `$GLOBAL["Http_Source"]` and passed as a

```

<?php
{
    global $Sql_index;
    global $PhpFileName;
    global $PhpFileIndex;
    global $Http_Source;
    $GLOBALS ["PhpFileName"] [324] = "C:\WAMP\WWW\PHPBB2\search.php";
    $GLOBALS ["PhpFileIndex"] = 324;
    include_once ('sensfuncDBJan92009.php');
    IsCovered ($GLOBALS ["PhpFileIndex"]);
}
. . .
$search_id = (isset ($HTTP_GET_VARS ['search_id'])) ? HttpVar_track ('O_CVar_$search_id', 'search_id',
    $HTTP_GET_VARS ['search_id'], $GLOBALS ["Http_Source"] [350] = array ('search_id', "GET",
    324, '$search_id'), 350, GET) : '';
. . .
for ($i = 0; $i < count ($search_id_chunks); $i ++)
{
    {
        $where_sql = ($search_author == '' && $auth_sql == '') ? 'post_id IN ('.implode ('', $search_id_chunks [$i]).')'
            : 'p.post_id IN ('.implode ('', $search_id_chunks [$i]).')';
        $GLOBALS ["SqlParts"] ['where_sql'] = (((('$search_author == \'\' && $.GLOBALS ["SqlParts"] ['auth_sql'] .' == \'\'')) ?
            ('post_id IN ('.implode (String, $search_id_chunks[$i]).(')')
            : ('p.post_id IN ('.implode (String, $search_id_chunks [$i]).(')'));
    }
    {
        $select_sql = ($search_author == '' && $auth_sql == '') ? 'post_id : 'p.post_id';
        $GLOBALS ["SqlParts"] ['select_sql'] = (((('$search_author == \'\' && $.GLOBALS ["SqlParts"] ['auth_sql'] .' == \'\'')) ?
            ('post_id') : ('p.post_id'));
    }
    {
        $from_sql = ($search_author == '' && $auth_sql == '') ? POSTS_TABLE : POSTS_TABLE.' p';
        $GLOBALS ["SqlParts"] ['from_sql'] = (((('$search_author == \'\' && $.GLOBALS ["SqlParts"] ['auth_sql'] .' == \'\'')) ?
            (POSTS_TABLE) : (POSTS_TABLE).' p'));
    }
    if ($search_time)
    {
        {
            $where_sql.= ($search_author == '' && $auth_sql == '') ? " AND post_time >= $search_time " :
                " AND p.post_time >= $search_time";
            $GLOBALS ["SqlParts"] ['where_sql'].= (((('$search_author == \'\' && $.GLOBALS ["SqlParts"] ['auth_sql'] .'
                == \'\'')) ? (' AND post_time >= $search_time ') : (' AND p.post_time >= $search_time'));
        }
    }
    $sql = "SELECT ".$select_sql." FROM $from_sql WHERE $where_sql";
    $GLOBALS ["Sql_Source"] [334] [0] = (('SELECT ').(('$GLOBALS ["SqlParts"] ['select_sql'] .')).(('
        FROM $.GLOBALS ["SqlParts"] ['from_sql'] .('
        WHERE $.GLOBALS ["SqlParts"] ['where_sql'] .')));
    $GLOBALS ["Sql_Source"] [334] [1] = 324;
    $GLOBALS ["Sql_index"] = 334;
}
. . .
if (!$result = $db -> sql_query ($sql)){
    message_die (GENERAL_ERROR, 'Could not obtain post ids', '', __LINE__, __FILE__, $sql);
}
. . .
?>

```

Figure 4. Coverage instrumentation added by DWASTIC to the *search.php* dynamic page of the PhpBB 2.0 application.

Sections in boldface have been added by our instrumenting transformation to instrument coverage for pages, server environment variables and SQL statements.

```

function sql_query ($query = "", $transaction = FALSE){
    . . .
    $this->query_result=mysql_query($query, $this->db_connect_id);
     Transformed into
    $this -> query_result = mysql_query_track($query, $this -> db_connect_id,
    $GLOBALS["Sql_Source"][$GLOBALS ["Sql_index"]], $GLOBALS ["Sql_index"]);
    . . .}

```

Figure 5. DWASTIC instrumentation for the database interaction points of the *mysql4.php* function of PhpBB 2.0

parameter to the same function as well. When the application under test is executed and the `HttpVar_track()` function is called, the server environment variable access information is inserted in the `HttpVar` table, and the coverage count

for the accessed variable is updated in the `AllHttpVar` coverage table. The `$search_id` assignment statement in Figure 4 shows an example in which `$Http_Get_Vars ['search_id']` has been identified, instrumented, and

```

% Begin with PHP grammar
include "php.grm"

% Override to isolate coverage instrumentation parts
redefine Expr
  [CoverageAspect]
  |...
end redefine

% Custom grammar to identify coverage instrumentation parts
define CoverageAspect
  '$GLOBALS'["Sql_Source"]'[ [Expr?] ']' ['0'] '= [SqlPartExpr]';
  '$GLOBALS'["Sql_Source"]'[ [Expr?] ']' ['1'] '= [Expr]';
  '$GLOBALS'["Sql_Source"]'[ [Expr?] ']' ['0'] ' .= [SqlPartExpr]';
  '$GLOBALS'["PhpFileName"]'[ [Expr?] ']' '= [Expr]';
  '$GLOBALS'["Http_Source"]'[ [Expr?] ']' '= [Expr][NL]';
end define

% Allow for output of coverage aspect only
redefine program
  ...
  |[CoverageAspect*]
end redefine

% Transform instrumented PHP program to its coverage aspect
function main
  replace * [program]
    P [program]

  % Use TXL grammatical type extraction to gather aspect fragments
  Construct CoverageInstrumentationAspect [CoverageAspect*]
    _ [^ P]

  by
    CoverageInstrumentationAspect
end function

```

Figure 6. TXL program to identify and extract the coverage aspect of an instrumented PHP program

added to the `$GLOBAL ["Http_Source"]` coverage array.

3) *Instrumenting SQL Statement Sources*: Identifying, extracting, and analyzing the dynamically constructed SQL statements in the context of the overall web system is not a trivial process, and often requires a great deal of complicated analysis using robust parsing, pattern matching, and control and data flow analysis. SQL statements are often constructed inter-procedurally, using a combination of string concatenation statements and host language statements that work together to construct the text of the SQL statement. These combinations are not only constant strings, but also include SQL statement fragments and host application variables.

In our approach, the most complex set of transformation rules is used to handle this task, and is mainly composed of three parts: The first part is to identify the beginning of dynamically constructed SQL statements. We do that by distinguishing string literals that begin with the SQL keywords `Select`, `Insert`, `Update`, `Delete`, `Create`, `Alter` and `Drop` using a separate TXL token class, and then use the parser to recognize concatenations built from these strings. Our transformation targets assignment statements that use these strings to build larger strings. Prior to the transformations, the code is normalized, replacing string expressions in other statements with a temporary PHP variable and inserting an assignment before the statement.

Once an assignment is found that uses one of the SQL keyword strings, other assignments using the same PHP variable are also checked and instrumented. Each identified statement is followed by a newly constructed assignment

statement that updates the SQL substrings in the corresponding entry of a PHP global array specifically created by our transformation approach to hold a copy of the dynamically constructed SQL source strings. Our transformation process generates a unique identifier for each newly identified SQL statement, which is used as the index for the newly constructed string in the global array.

In the second part, while constructing the SQL statement source from string fragments and concatenation statements, special care is given to the kind of the concatenated fragment, so that we can retain the original PHP variable names rather than their run-time values in the SQL statement text. This retains in our database the link between dynamically generated SQL statements and the variables they use. Our approach distinguishes four SQL fragment types: PHP constant variables, PHP variables, string expressions, and SQL fragment variables. The final SQL statement is constructed by single quoting all fragment types other than constant variables and SQL fragments, which are kept unquoted for later substitution in the execution phase (Section III-C).

The third part identifies and instruments the application's database interaction points. At those points, the database interface call statement `mysql_query()` is replaced with a call to our instrumenting function `mysql_query_track()` as shown in Figure 5. The instrumenting function call takes both of the two versions of the SQL statement, the uninstantiated source statement collected in the previous step and available globally at this point, and the instantiated execution instance of the statement, both of which

```

<?php
. . .
$GLOBALS["PhpFileName"][467] = "C:\WAMP\WWW\PHPBB2\admin\admin_forums.php";
$GLOBALS["Http_Source"][512] = array ('mode', "POST", 467, '$mode');
$GLOBALS["Http_Source"][511] = array ('mode', "GET", 467, '$mode');
. . .

$GLOBALS["SqlParts"]['table'] = ((FORUMS_TABLE));
$GLOBALS["Sql_Source"][471][0] = (('SELECT * FROM '
    . $GLOBALS["SqlParts"]['table']));
$GLOBALS["Sql_Source"][471][1] = 467;
$GLOBALS["Sql_Source"][471][0] .= ((' WHERE $catfield = $cat'));
$GLOBALS["Sql_Source"][471][0] .= ((' ORDER BY $orderfield ASC'));
$GLOBALS["Sql_Source"][472][0] = (('UPDATE ' . $GLOBALS["SqlParts"]['table'] .
    ' SET $orderfield = $i WHERE $idfield = ')).
    ('$row [$idfield]');
$GLOBALS["Sql_Source"][472][1] = 467;
$GLOBALS["Http_Source"][513] = array ('addforum', "POST", 467, '');
$GLOBALS["Http_Source"][514] = array (POST_FORUM_URL, "GET", 467, '$forum_id');
$GLOBALS["Sql_Source"][473][0] = (('SELECT * FROM ').(PRUNE_TABLE).
    (' WHERE forum_id = $forum_id'));
$GLOBALS["Sql_Source"][473][1] = 467;
. . .

BuildAllPHPPagesTable ($GLOBALS ["PhpFileName"]);
BuildAllHttpTable ($GLOBALS ["Http_Source"], $GLOBALS ["$PhpFileIndex"]);
BuildAllSQLSourcesTable ($GLOBALS ["Sql_Source"]);

?>

```

Figure 7. Part of the extracted instrumentation slice for PhpBB 2.0 augmented with database insertion code

are stored in our instrumentation database table `Database Interactions`. The instrumenting function then updates the coverage database by incrementing the coverage count of the executed SQL source statement in the `ALLSQL_Sources` coverage table. Finally, it executes the original database interaction statement. Figure 4 shows examples of database statement source fragments identified, instrumented, and added to the `$GLOBAL["SqlParts"]` coverage array.

Our methodology can capture, instrument, and correlate SQL source statements and the database interaction points even when they are spread over separate source files. There is no need to combine the source files into a single processing unit, since the relation is done using global arrays.

B. Extracting the Instrumentation Slice

Once the web application has been instrumented for page access, server environment variables and databases interactions as described above, the DWASTIC tool extracts the instrumentation slice from the application based on the grammatical patterns defined in Figure 6. The five *CoverageAspect* patterns identify three kinds of assignment statements: instrumentation statements generated for collecting SQL statement sources from their fragments using assignments and concatenation statements (the first three patterns), instrumentation statements tracking page access (the fourth pattern), and instrumentation statements tracking server environment variables (the fifth pattern).

Based on these grammatical patterns, the main transformation rule of Figure 6 extracts all instances of these instrumentation statements from the application source code, and groups them into a single new file. This file is then automatically transformed into a PHP program by enclosing

it in PHP opening and closing tags, including a reference to the application constants, and inserting call statements to PHP functions that insert the coverage information collected in the global arrays into the coverage database to form the initial coverage tables. An elided view of the generated slice as a PHP program is shown in Figure 7.

C. Executing and Analyzing the Instrumentation slice

The extracted instrumentation slice PHP program constructed in the previous step uses three global arrays, one for `SQL_statement` sources, one for server environment variables, and one for page access. When the slice program is executed, it populates the global arrays with one instance of every generated SQL source statement, every server environment variable access, and every page access that is instrumented in the application. This effectively builds the coverage tables for each, which are then inserted as the initial tables of the coverage database described in Figure 2. The generated SQL statement sources are analyzed during insertion to identify the basic query components, and to find any server environment variables and application variables embedded in the statement. These details are added to the database as well.

IV. AN EXAMPLE APPLICATION

We have assessed our approach by analyzing two production dynamic web applications, *PhpBB 2.0*, with millions of installations the world's most popular internet forum system, and *Moodle*, a popular open source course management system. To automate the collection of usage traces, we have used WATIR (Web Application Testing In Ruby) [5] [13], a scriptable library to drive web browsers by clicking links, pressing buttons, and filling in forms.

Role	SQL Statements Coverage			Page Access Coverage			Server Environment Variables Coverage		
	Covered	Total	%	Covered	Total	%	Covered	Total	%
Anonymous User	41	440	9.3%	28	69	40.0%	21	374	5.6%
Admin	123	440	28 %	56	69	81 %	68	374	18.2 %

Table IV
Coverage metrics results for pages, server environment variables and SQL statements at the application level for a sample test case

Role	Page Name	SQL Statements Coverage			Server Environment Variables Coverage		
		Covered	Total	%	Covered	Total	%
Anonymous User	Viewforum.php	5	7	71.42%	1	12	8.33%
	Viewtopic.php	4	13	30.76%	1	16	6.25%
Admin	Viewforum.php	6	7	85.7%	5	12	41.6%
	Viewtopic.php	6	13	46.2%	5	16	31.2%

Table V
Coverage metrics results for server environment variables and SQL statements at the page level for a sample test case

Role	Page Name	Server Environment Variables Coverage in SQL Statements		
		Covered	Total	%
Anonymous User	Viewforum.php	5	7	71.42%
	Viewtopic.php	3	12	25%
Admin	Viewforum.php	6	7	85.7%
	Viewtopic.php	5	12	41.6%

Table VI
Coverage metrics results for pages and server environment variables at the SQL statement level for a sample test case

Page Name	Uninstrumented Exec. Time (sec.)	Instrumented Exec. Time (sec.)	Performance degradation (percent)
http://localhost/phpBB2/index.php	2.172	3.11	30.2%
http://localhost/phpBB2/faq.php	1.328	1.328	0.0%
http://localhost/phpBB2/search.php	1	1.515	34.0%
http://localhost/phpBB2/memberlist.php	0.953	1.485	35.8%
http://localhost/phpBB2/groupcp.php	1.016	1.5	32.3%
http://localhost/phpBB2/profile.php?mode=register	2.14	3.156	32.2%
http://localhost/phpBB2/profile.php?mode=editprofile	2.031	3	32.3%
http://localhost/phpBB2/privmsg.php?folder=inbox	2	2.484	19.5%
http://localhost/phpBB2/login.php	2	2.984	33.0%
http://localhost/phpBB2/index.php	1.344	1.828	26.5%
http://localhost/phpBB2/search.php?search_id=unanswered	2.188	3.188	31.4%
http://localhost/phpBB2/index.php?c=1	1.328	1.796	26.1%
http://localhost/phpBB2/viewforum.php?f=1	1.453	2	27.4%
http://localhost/phpBB2/profile.php?mode=viewprofile&u=2	1.016	1.984	48.8%
http://localhost/phpBB2/viewtopic.php?p=10#10	1.031	1.985	48.1%
http://localhost/phpBB2/viewonline.php	0.953	1.484	35.8%
http://localhost/phpBB2/profile.php?mode=viewprofile&u=2	1.016	1.984	48.8%
		Average:	31.9%

Table VII
Performance penalty of DWASTIC instrumentation on dynamic pages of PhpBB 2.0

We show here the detailed results for two specific test cases: an anonymous user interacting with a PhpBB forum, and an Admin user visit. These two test cases are simple interactions using only the hyperlinks, not including the population of forms. The complete results are too large to include in this paper, but the example data in the initial coverage databases shown in Tables I, II, and III refer to a subset of the instrumentation points available to be covered in these tests. Using the results from these two tests, we

calculate values for the metrics proposed in Section II-B.

Table IV shows the overall results for the Page Cov., the SQL_Stm Cov., and the Server_Env_Var Cov. metrics for each test. In the table, the Total is the total number of instrumentation points in each of the categories. For example, the anonymous user test covered 28 of the 69 total pages, while the admin user test covered 56 of the 69 total pages. Since the two example test cases do not include forms, neither all SQL statements nor all environment

variables are covered. We have manually confirmed that the Total number for each coverage aspect extracted by our approach is equal to the actual number of aspects in the code.

Table V shows the results for the `Page_Server_Env_Var Cov.` and the `Page_SQL_Stm Cov.` metrics for two pages, `Viewforum.php` and `Viewtopic.php`. Even without form filling, the Admin user covers more SQL statements and more variables because he can access more links. Some examples are the new topic and post reply links.

Table VI shows the results for the `SQL_Server_Env_Var Cov.` metric for the same two pages. This metric measures the number of SQL statements that reference server environment variables. Comparing Table V and Table VI, we find that the page `Viewtopic.php` has 12 out of 13 SQL statement that reference server variables. The admin user test cases covers 5 of them. This measure is essential for evaluating test cases for SQL statements that use user input such as test cases for SQL injection.

Since the run-time instrumentation must update the coverage database, it imposes a runtime penalty on the web application. Table VII shows the runtime measure for 17 pages. The second column (Non Instrumented Exec Time) indicates the time needed to execute the original, uninstrumented page. The third column (Instrumented Exec Time) shows the time needed to execute the instrumented page. The performance penalty of the instrumentation ranges from non-existent (`faq.php`) up to about 50%. The average penalty is 32%, which is acceptable for a testing environment.

V. RELATED WORK

Several approaches and coverage metrics have been proposed to assess the quality of test cases aimed at ensuring the correctness of database interactions in standard database systems. For example, Suárez-Cabal and Tuya [2] propose a coverage metric and a tool specialized for a subset of SQL SELECT statements designed to help improve test suites to detect faults at the level of SELECT statements in a database application. A coverage tree is built from each SELECT statement encoding the conditions specified by the where and join clauses of the query. The approach is specifically aimed at analyzing static SQL SELECT statements, while ours handles all SQL statements, including dynamically constructed ones.

Willmor and Embury [14] propose two test adequacy criteria for database applications. The first criterion checks the coverage of the structural aspects of the database application. This includes aspects such as the different types of operations, transaction statements, and the entities represented in the database. The other criterion is a define-use criterion, which measures all of the possible database system operations' define-use pairs. The coverage information collected in our approach is sufficient to compute the coverage metrics proposed by Willmor and Embury.

Halfond and Orso [7] propose a coverage criterion which measures the coverage of all the possible SQL command forms that can be issued at each database interaction point. They describe the prototype tool DITTO which statically analyzes the application source code, identifying database interaction points and the string variables containing the SQL commands. Java String Analysis (JVS) is used to build a character-based NDFA for each string variable, which is then converted into an SQL-level NDFA which represents a static model of all the SQL queries that can be generated. The construction of SQL static models is adapted from the method of Gould et al. [6]. Their metric compares the number of forms covered by the test cases to the total number of forms possible at each interaction point. Our approach constructs exact versions of the SQL statement templates constructed between define-use paths for a web application. It is easily extendable to web applications implemented in any technology, while Halfond and Orso's is limited to Java applications and has limitations when collecting SQL statement fragments from external sources.

Smith et al.[10] proposes two coverage metrics for SQL injection vulnerability testing. The first metric measures the percentage of database interaction points (API calls) that are tested at least once to the total number of identified database interaction points. The second coverage metric measures the percentage of input variables tested at least once to the total number of variables found in any target SQL statement. The database interaction points and input variables are counted manually, and the instrumentation process is also done manually. While their first metric does not consider all dynamically constructed SQL statements, our proposed coverage criteria not only handle input variables used in SQL statements on the application level, but also at the page level, and our instrumentation is automated using source transformations.

There are other approaches that are similar to ours in identifying, extracting and analyzing database interactions. Cleve and Hainaut [3] use aspect-based tracing to relate and extract the basic components of prepared statements. While the tracing approach used does not modify the source code, it does not deal with the dynamically constructed SQL statements using string concatenations scattered throughout the code that our slicing resolves. Their approach is also yet to be evaluated on a production system.

Brink et al. [12] propose a tool for assessing the quality of database interactions in standard applications. They extract 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. The purpose is to extract the queries for quality assessment, while our purpose is to determine the coverage of test cases.

Ngo and Tan [9] 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 then slices them out as an interaction Control Flow Graph (ICFG). Each interaction path is then symbolically executed, and all possible interaction types are derived from the generated symbolic expression using inference rules. In a case study the approach was able to extract 80% of the database interactions. The complexity of the extraction process is high, as it is composed of five stages, and is affected by factors such as number of interaction paths (i-paths), and the length and complexity of each i-path. The authors also do not specify how to handle SQL statements constructed from sequences of string fragments and concatenations, which is handled by our instrumentation slice technique.

VI. CONCLUSION

In this paper, an original approach to automate coverage metrics for dynamic web applications has been proposed, implemented and demonstrated in practice on a production web application. First, we proposed a set of coverage criteria specialized for web applications which takes into account the complex and distributed structure of these applications. We have demonstrated how a dynamic web application written in PHP can be automatically instrumented using source transformations, and that database SQL statements dynamically constructed from string fragments can be handled using a source slicing technique to identify and build a coverage database. The proposed coverage criteria and the automatic tool helps improve the quality of test cases and focusses testing efforts on the application component interactions, which are often the source of web applications vulnerabilities. The approach is being used to provide a completeness measure for extracting an access control security model from web applications under test. The accuracy of the results is both hand verified and robust since it is automatically back-checked at run time.

As future work, we plan to extend the approach to handle other web technologies and database engines and to evaluate it on a wide range of applications of different sizes. We also plan to use DWASTIC to support other testing activities for web applications, such as SQL injection and cross-site scripting analysis.

ACKNOWLEDGEMENTS

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

- [1] M. H. Alalfi, J. R. Cordy, and T. R. Dean. A Verification Framework for Access Control in Dynamic Web Applications. In *Canadian Conf. on Computer Science and Software Eng.*, Montreal, Canada, May 2009, pp. 109-113.
- [2] M. J. S. Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *12th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering*, Newport Beach, California, October-November 2004, pp. 253-262.
- [3] A. Cleve and J.-L. Hainaut. Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering. In *15th Working Conf. on Reverse Eng.*, Antwerp, Belgium, October 2008, pp. 192-196.
- [4] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming* 61(3), August 2006, pp. 190-210.
- [5] Canoo Engineering AG. Canoo WebTest, <http://webtest.canoo.com>, accessed 20 August 2009.
- [6] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *26th Intl. Conf. on Software Engineering*, Edinburgh, UK, May 2004, pp. 645-654.
- [7] W. G. J. Halfond and A. Orso. Command-Form Coverage for Testing Database Applications. In *21st Intl. Conf. on Automated Software Engineering*, Tokyo, Japan, Sept. 2006, pp. 69-80.
- [8] L. Moonen. Generating robust parsers using island grammars. In *8th Working Conf. on Reverse Eng.*, Stuttgart, Germany, October 2001, pp. 13-22.
- [9] 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), February 2008, pp. 160-175.
- [10] B. Smith, Y. Shin, and L. Williams. Proposing SQL statement coverage metrics. In *4th Intl. Workshop on Software Engineering for Secure Systems*, Leipzig, Germany, May 2008, pp. 49-56.
- [11] N. Snytnyky, J. R. Cordy, and T. R. Dean. Robust multi-lingual parsing using island grammars. In *CASCON 2003*, Toronto, October 2003, pp. 266-278.
- [12] H. van den Brink, R. van der Leek, and J. Visser. Quality Assessment for Embedded SQL. In *7th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, Paris, France, Sept. 2007, pp. 163-170.
- [13] WatirCraft. WATIR, <http://wtr.rubyforge.org>, accessed 2 March 2009.
- [14] D. Willmor and S. M. Embury. Exploring Test Adequacy for Database Systems. In *3rd UK Software Testing Research Workshop*, York, UK, Sept. 2005, pp. 123-133.