Automated Software Engineering manuscript No.
(will be inserted by the editor)

Exploring Output-Based Coverage for Testing PHP Web Applications

Hung Viet Nguyen · Hung Dang Phan · Christian Kästner · Tien N. Nguyen

the date of receipt and acceptance should be inserted later

Abstract In software testing, different testers focus on different aspects of the software such as functionality, performance, design, and other attributes. While many tools and coverage metrics exist to support testers at the code level, not much support is targeted for testers who want to inspect the *output* of a program such as a dynamic web application. To support this category of testers, we propose a family of *output-coverage* metrics (similar to statement, branch, and path coverage metrics on code) that measure how much of the possible output has been produced by a test suite and what parts of the output are still uncovered. To do that, we first approximate the *output universe* using our existing symbolic execution technique. Then, given a set of test cases, we map the produced outputs onto the output universe to identify the covered and uncovered parts and compute output-coverage metrics. In our empirical evaluation on seven real-world PHP web applications, we show that selecting test cases by output coverage is more effective at identifying presentation faults such as HTML validation errors and spelling errors than selecting test cases by traditional code coverage. In addition, to help testers understand output coverage and augment test cases, we also develop a tool called WebTest that displays the output universe in one single web page and allows testers to visually explore covered and uncovered parts of the output.

H. Nguyen
Google LLC, USA

H. Phan
ECpE Department, Iowa State University, USA

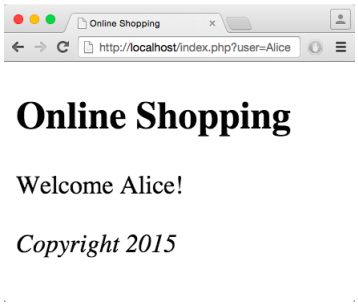
C. Kästner
School of Computer Science, Carnegie Mellon University, USA

T. Nguyen
School of Engineering & Computer Science, University of Texas at Dallas

(a) Example PHP program

```
1 echo "<h2>Online Shopping</h2>";
2 if (isAdmin()) logAdminAccess();
3 if (isLoggedIn())
4    echo "<div>Welcome " . $user . "!</div>";
5 else
6    echo "<div>Please log in first.</div>";
7 echo "<em>Copyright 2015</em>";
```

(b) Output of a test



(c) Output coverage visualization

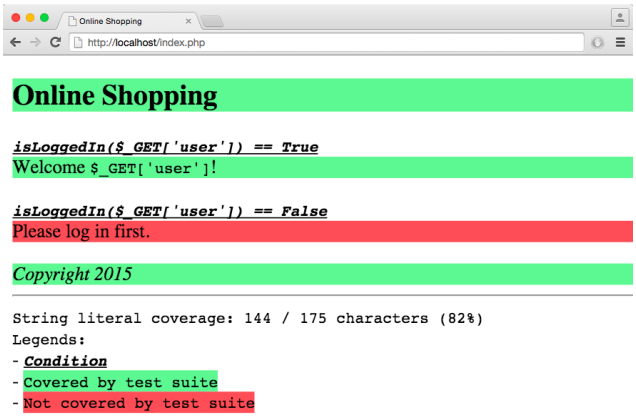


Fig. 1 An example web application and the WebTest tool showing output coverage

1 Introduction

In software development, different testers have different focuses and priorities. Testing may focus on functional correctness, but also on design, performance, security, usability, and other aspects. While code-level testing is well supported by tools and coverage metrics, we argue that testers inspecting the *output* of a program are much less supported, even though the output is the main product of many software systems, including dynamic web applications. For example, a UI tester tasked with checking a web application for layout issues, typographical errors, design standards, or browser compatibility, accessibility support, and user friendliness would likely examine the final rendered web page and would not need to know about how the web page is generated, what languages the server-side code is written in, and what technologies are used underneath. For such testers, code measures such as line or branch coverage represent an improper abstraction, refer to artifacts not directly related to the testing tasks and possibly unknown to the testers, and incentivize testing priorities inefficiently as we will show. Instead, we propose coverage measures specifically for the program's output, to support *output-oriented* testing.

For testers that focus primarily on the output of an application (UI testers and others), we propose coverage metrics on the output of an application, which indicate how much of the possible output has been produced by a set of tests and what output is still uncovered. *Output coverage metrics* measure coverage in terms of the output, not in terms of the code that produces the output, because code that does not produce any output is irrelevant to a UI tester. For example, to inspect all possible outputs of our example in Figure 1a for layout issues, a UI tester would not care about the implementation logic beyond understanding which different pages can be produced and would not care about whether admin access is logged (Line 2). The UI tester would care about seeing all texts *in the context* to *investigate different parts of the output and the consistency among those parts*, for example, ensuring that font sizes are consistent in all outputs. An *output coverage metric* allows the tester to identify parts of the output that have not yet been covered by existing test cases, without having to manually inspect all string literals and how they are combined through data and control flow in the program. With that knowledge, the tester can then create new tests that reveal UI bugs in those missing parts.

The key to providing useful output-coverage metrics is to determine *uncovered* output by comparing tests against the *output universe*, that is, the set of all possible outputs produced by an application. If the output depends on user input or the environment, such output universe can trivially be infinite; instead, we focus on those parts of the output that are produced from *string literals* within the program. We address the output universe problem by approximating all possible outputs using a symbolic-execution engine for PHP programs [Nguyen et al., 2011]. Our key insight is that although the number of concrete outputs may be infinite, there are usually a finite number of structures on a web page that we can derive. The symbolic-execution engine explores feasible paths in a program, considers unknown data such as user inputs as symbolic values, computes possible outputs, and stores them compactly. Although not including user inputs, these concrete outputs often make up the main output of many web pages (75–96 % of all output [Nguyen et al., 2014]), and typically include the elements that are repeated on many pages. Note that Alshahwan and Harman [2014] recently proposed an output-uniqueness criteria for counting different outputs produced by a test suite, but their approach cannot determine uncovered output or distinguish string literals in a program from user input.

We propose *a family of output coverage metrics* similar, in spirit, to traditional statement, branch, and path coverage on source code:

1. *Coverage of string literals (Cov_{str}):* As with statement coverage, we determine which string literals have been output by a test suite. *Relative Cov_{str}* is measured as the ratio between the number of characters in string literals in the output universe that are covered by the test suite and the total number of all the characters in the output universe.
2. *Coverage of output decisions (Cov_{dec}):* As with branch coverage, we look at decisions in the program that can affect the output, namely *output decisions*

(each conditional statement represents two decisions that can be taken, but not all decisions are output decisions). *Relative* Cov_{dec} is measured as the ratio between the number of output decisions covered by a test suite and the total number of all output decisions in the output universe.

3. *Coverage of contexts (Cov_{ctx}):* As with path coverage, we consider all *combinations of output decisions* (all *contexts*) that can be produced by the program. *Relative* Cov_{ctx} is measured as ratio between the contexts covered by a test suite and the contexts in the output universe.

In an experimental evaluation on seven PHP web applications, we show that selecting test cases by output coverage is more effective in identifying output-related bugs—such as HTML validation errors and spelling errors—than traditional test case selection by code coverage, whereas the latter is more suitable for detecting traditional coding bugs. That is, different classes of bugs can benefit from different coverage measures and output coverage is an effective complementary metric for certain output-oriented use cases in testing.

In addition, we illustrate the benefit of output coverage indirectly by demonstrating how output coverage in an output universe can be visualized to help developers create test cases that optimize output coverage and to help navigate and inspect the compact representation of the output directly to detect certain classes of presentation faults. We introduce a prototype visualization tool *WebTest* that compactly represents the output universe, highlighting uncovered parts, as exemplified in Figure 1c.

Our key contributions in this paper include the following:

- We suggest that even though UI testers may have access to source code, functional testing metrics are not well suited for UI testing. That is, different types of testers concerning different aspects of the software including its output are likely to demand different types of coverage metrics.
- We propose a family of output coverage metrics that is complementary to but neither synonymous with nor meant to be substitutable for traditional code-based coverage. As we will explain in Section 2.2, our output coverage metrics provide a tailored view on the test suite of a web application for testers who are focused on the output.
- In contrast to existing work (Section 6) which is limited to *absolute* output coverage and is often inadequate for real-world testing scenarios, we provide *relative* output coverage and information about what has *not* been covered by an existing test suite. It is this kind of information that can help testers augment their test cases.
- We describe a solution to the output universe problem using our existing symbolic execution engine for PHP code and algorithms to compute both absolute and relative output coverage metrics. While several components of the approach (such as the symbolic execution and the core of the output mapping algorithm) existed in our prior work [Nguyen et al., 2011], we provide several enhancements, such as using dynamic instrumentation and location information for better mapping accuracy, and combine them into a holistic approach that solves a new problem (Section 3).

– Via an empirical study, we provide evidence that detecting different kinds of issues actually benefits from optimizing test suites for different kinds of coverage metrics and show the effectiveness of using output coverage as a criterion for detecting output-related bugs.
– Finally, we demonstrate a prototype visualization of output coverage, demonstrating the potential to assist developers in detecting output-related bugs and augmenting their test suites.

All our tools are publicly available at `https://github.com/git1997/VarAnalysis`.

2 Output Coverage Metrics

A key step in determining output coverage is to compute the *output universe*, that is, the set of all possible outputs from a web application. Based on the output universe, we then define a suite of *output coverage metrics*.

2.1 Representing the Output Universe

Unlike traditional code coverage metrics, such as statement coverage or branch coverage, where the set of all statements and branches in a program is well defined and finite, the set of all possible outputs is usually infinite due to unknown data coming from user inputs and databases. Such unknown data can lead to different outputs with the same structure (e.g., only the user's name changes while the layout of the web page and the welcome message remain the same), or different outputs with different structures (e.g., a different set of functionality is presented depending on whether the user has the administrator role).

We approximate all outputs through symbolic execution of the PHP code. We reuse our symbolic execution engine for PHP called *PhpSync* [Nguyen et al., 2011]. During symbolic execution, we track unknown values (such as user input) as symbolic values, while tracking location information on string literals concretely. This way, we approximate all possible structures of the output with all statically determinable content, leaving symbolic values as 'placeholders' for unresolved data. The result is a representation of the output that can contain the *alternatives from branching decisions* and *symbolic values* representing unknown values. For branching decisions, we explore both branches and track the (symbolic) path conditions, such that some output is only produced under some conditions, illustrated with `#if` directives in an extended example in Figure 2b.

2.2 Family of Output Coverage Metrics

Just as a family of coverage metrics such as statement, branch, and path coverage has been defined on code [Ammann and Offutt, 2008; Miller and

(a) A PHP program (extended from Figure 1) to illustrate different output coverage metrics

```
1  echo "<h2>Online Shopping</h2>";
2  if (isAdmin($_GET["user"]))
3     logAdminAccess();
4  if (isLoggedIn($_GET["user"]))
5     $message = "<div>Welcome " . $_GET["user"] . "!</div>";
6  else
7     $message = "<div>Please log in first.</div>";
8  echo $message;
9  if ($showCoupons)
10    echo "<div>Coupons are available!</div>";
11 echo "<div><em>Copyright 2015</em></div>";
```

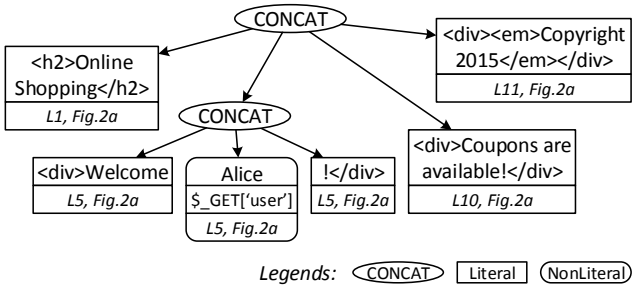
(b) Representation of the output universe with CPP #if directives and symbolic values in Greek letters; origin locations of strings literals in server code are shown in italics

```
1  <h2>Online Shopping</h2> L1
2  #if α // isLoggedIn($_GET['user']) L4
3     <div>Welcome β!</div> // β represents $_GET['user'] L5
4  #else
5     <div>Please log in first.</div> L7
6  #endif
7  #if γ // $showCoupons L9
8     <div>Coupons are available!</div> L10
9  #endif
10 <div><em>Copyright 2015</em></div> L11
```

(c) A concrete output of a test case in which the user (Alice) is logged in as administrator and the $showCoupons option is enabled

```
1  <h2>Online Shopping</h2>
2  <div>Welcome Alice!</div>
3  <div>Coupons are available!</div>
4  <div><em>Copyright 2015</em></div>
```

(d) The S-Model for a concrete output



(e) Output coverage and code coverage for the example

Output coverage			Code coverage		
Cov_{str}	111/142	(78%)	Statement cov.	9/10	(90%)
Cov_{dec}	2/4	(50%)	Branch cov.	3/6	(50%)
Cov_{ctx}	1/4	(25%)	Path cov.	1/8	(13%)

Fig. 2 An example PHP program and its output-universe representation, as well as a concrete execution with corresponding S-Model and coverage results.

Maloney, 1963], we propose *a family of coverage metrics that are applied to the output* of a dynamic web application. We define output coverage for a given test suite as follows:

Coverage of string literals. Coverage for string literals (Cov_{str}) is similar to statement coverage in that we measure how much of the content contained in string literals in the program (that are relevant for the output universe) have been produced in at least one test case. The rationale for this type of coverage is that a string that *could appear* in the output *needs to be tested* in a concrete output generated by at least one test case. Note that not all string literals in the PHP program contribute to the output: For example, the literal "user" in our example is an array access but is not part of the output universe and as such not measured by this metric. To measure the coverage for string literals, we compute their covered length, or equivalently, the number of covered characters:

Definition 1 (Cov_{str}) *The ratio between the number of characters in string literals in the output universe that are covered by the test suite and the total number of all characters in the output universe.*

We define Cov_{str} based on the lengths of the literals instead of the number of literals since a web application often has long literals containing large portions of HTML code. Counting characters aligns better with the chance of bugs since long literals are more likely to contain presentation bugs (e.g., spelling errors and HTML validation errors) than short ones.

Compared to a simpler approach of investigating all string literals in a PHP program individually, testing with output coverage ensures that each literal is produced in the context of at least one full page. Such context allows to investigate presentation issues depending on context as font sizes, colors, surrounding texts, or ordering that cross multiple string literals. For example, a tester may want to assure that all `<div>` tags are correctly nested, which is difficult to assess from looking only at individual string literals. Analogous to statement coverage, to achieve Cov_{str} coverage each literal has to appear in at least one context, not in all possible contexts.

Coverage of output decisions. In addition to covering all string literals, testers might also want to investigate the composition of the page when certain parts are not displayed based on some decision. For instance, they might want to check that the layout of the web page is still correct without the coupon output in Figure 2c. We consider every control-flow decision in the program that affects the output as an *output decision*. A control-flow decision that does not produce output in either branch is not considered as an output decision. We define coverage on output decisions (Cov_{dec}) analogous to branch coverage on code:

Definition 2 (Cov_{dec}) *The ratio between the number of output decisions covered by the test suite and the total number of all output decisions in the output universe.*

Coverage of contexts. The previous measures consider only some contexts in which string literals may appear, while certain bugs in the output may appear in some contexts and not others. (Each context is a specific *combination of output decisions*, which can be listed by traversing the output decisions on the output universe.) Therefore, we also consider all possible contexts and define a corresponding output-coverage metric Cov_{ctx} analogous to path coverage on code:

Definition 3 (Cov_{ctx}) *The ratio between the combinations of output decisions covered by the test suite and the total number of all possible combinations of output decisions in the output universe.*

In our example, there are three control-flow decisions (*if* statements) of which two affect the output; these two output decisions can be combined to provide four different contexts (in contrast to eight paths). Note that, as with path coverage, the number of contexts grows exponentially, even though the domain is technically finite due to our abstractions (e.g., we unroll loops exactly once). Therefore, achieving full Cov_{ctx} coverage is not a realistic goal and is likely of limited practical use. We include Cov_{ctx} and path coverage only for completeness and theoretical purposes. In Figure 2e, we exemplify output coverage metrics and their code coverage counterparts for a single test case.

Relations with code coverage metrics. Our three output coverage metrics are inspired by but do not necessarily correlate with statement coverage, branch coverage, and path coverage, respectively. Specifically, not all statements and decisions in a web application contribute to the output (e.g., the logging functionality in our example), and a single statement can produce multiple parts of the output (e.g., the `echo` statement on line 8 of Figure 2a generates the two alternatives on lines 2–6 in Figure 2b). Our output coverage metrics provide a tailored view on the test suite of a dynamic web application for testers who are interested in the output.

In practice, some testers might be interested in both code coverage and output coverage and build test suites that target both, whereas others focus on specific classes of tests and are more interested in one or the other form of coverage. We do not expect that there is any single combination that would balance both for all use cases. In fact, in some contexts, emphasis may shift during the testing process. Therefore, the proposed output coverage metrics are by no means intended to be a replacement for traditional code coverage metrics. As we will show, both coverage metrics measure distinct aspects and might be useful in different contexts in which developers can select or combine the right metrics as required.

3 Computing Output Coverage

Key Idea. We compute output coverage metrics in three steps illustrated in Figure 3.

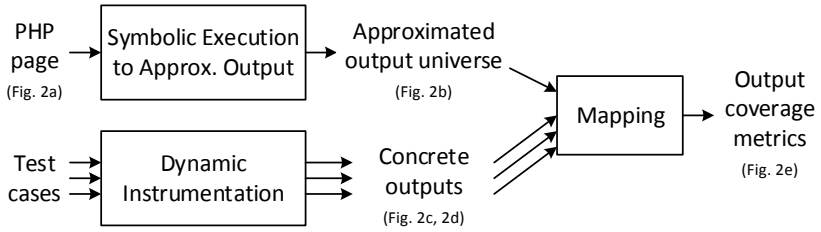


Fig. 3 Computing output coverage

First, we use our symbolic execution engine for PHP [Nguyen et al., 2011] to approximate the output universe of a PHP web application. The symbolic executor considers unknown data such as user inputs and data from databases as symbolic values, explores all feasible paths in a program, and computes all possible alternatives of the output. For scalability, it executes only one iteration of a loop and aborts on recursive function calls. Throughout symbolic execution, we track location information for every character in the output. Details on symbolic execution of PHP can be found elsewhere [Nguyen et al., 2011, 2014].

Second, we instrument a regular PHP interpreter to record the execution of a test case and the generation of its output. We again track location information about every character in string values (either to string literals or to outside sources such as user input).

Third, we match concrete outputs from test executions against the output universe to measure coverage. The location information produced as part of every test execution is used to avoid ambiguity in identifying parts in the output universe that have been covered by the test suite. Since the number of string literals, output decisions, and contexts in the output universe representation is finite, we can compute the three coverage metrics as percentages.

Next, we explain steps 2 and 3 in more detail.

3.1 Dynamic Instrumentation of Test Cases

To enable more precise mapping of test outputs on the output universe, we track *location information* of string values that are output by concrete test executions. (Location information is the position of a string literal in the source code and is used for distinguishing string literals that have the same value but are produced from different locations in the server-side program.) String values in PHP are either introduced in literals or read from variables or functions that represent environment values, database results, or user input (e.g., `$_GET['user']`). Once created, we track location information also through assignments, concatenation, and function calls until it is finally used as part of an `echo` or `print` statement (or inline HTML) to produce output.

We track location by attaching position information to string values. Technically, we attach a tree-based representation for string values called *S-Model*.

```
1  // Evaluating a Literal Expression
2  function Value eval(LiteralExpr literalExpr)
3     value ← evalLiteral(literalExpr)
4     value.SModel ← new Literal(literalExpr)
5     return value
6  end
7
8  // Evaluating a Concat Expression
9  function Value eval(ConcatExpr concatExpr)
10    leftValue ← eval(concatExpr.LeftExpr)
11    rightValue ← eval(concatExpr.RightExpr)
12    value ← evalConcat(leftValue, rightValue)
13    value.SModel← new Concat(leftValue.SModel, rightValue.SModel)
14    return value
15 end
16
17 // Evaluating a Non-Literal Expression
18 function Value eval(NonLiteralExpr nonLiteralExpr)
19    value ← evalNonLiteral(nonLiteralExpr)
20    if value.SModel is not set
21       value.SModel ← new NonLiteralNode(nonLiteralExpr)
22 end
```

Fig. 4 Algorithm to create S-Model (added instrumentation highlighted in italics)

An S-Model may contain three kinds of nodes: Literal and NonLiteral nodes with location information and Concat containing nested S-Model nodes. In Figure 2d, we illustrate the S-Model corresponding to the output of Figure 2c of our running example (for clarity, we show only line information).

We compute location information with an instrumented PHP interpreter, based on the open-source interpreter Quercus.[1] We track the attached S-Model information during the evaluation of the test as shown in Figure 4. Specifically, we handle three kinds of expressions:

1. For a PHP literal expression (lines 2–6), we attach a new Literal node to the string value, pointing to the expression node with its location information.
2. For a PHP concatenation expression (lines 9-15), we create a Concat node to represent the returned string value with its children being the corresponding S-Models of the constituent sub-strings.
3. For all other PHP expressions, such as a PHP variables or a function calls (lines 18-22), we reuse the original interpreter's code to evaluate the expression and obtain its value (line 19). We attach a NonLiteral node referring to the corresponding non-literal expression that creates the value. For example, when the string literals are involved in string operations other than concatenation (such as str_replace), the approach currently tracks the location to the string operation.

The output of the test is collected as a single large S-Model from echo and print statements, collecting all individual string outputs with Concat nodes.

[1] http://quercus.caucho.com/

3.2 Mapping Outputs to Output Universe

A dynamic web application often contains multiple entry points (pages), each of which may be rendered by executing multiple server (PHP) files. We symbolically execute each entry point separately and merge the results in one single model to create the output universe. In the same way that the output representation of one single entry point usually contains alternatives of that page's contents, the output universe contains alternatives of the entry points' outputs, using the same underlying representation of alternatives (#if directives as in Figure 2). This is a standard technique used in our existing work [Nguyen et al., 2015a]. The S-Model, on the other hand, is a *concrete* output representation of one single entry point.

To compute output coverage, we identify which string literals in the output have been covered by outputs generated in a test case. We map Literal nodes of the S-Model for test executions against the output universe to identify covered string literals and output decisions. An S-Model can be considered as a concatenation of string literals (with location information), whereas the output universe can be considered as a concatenation of string literals *and* symbolic values, with some parts being alternatives to one another. Note that we only map the S-Model to the part of the output universe that represents the corresponding entry point's output.

In most cases, the mapping is straightforward, because we simply can match string literals by location information. Matching is more difficult when concrete values of an execution are matched against symbolic values in the output universe, because there might be multiple possible matches. In addition, location information is not always available—there are cases where we might lose location information during processing when strings are processed through library functions. Therefore, for those cases where location information is not available or not sufficient, we use heuristic strategies to determine the best mapping among possible alternatives.

To illustrate our mapping challenge for cases involving symbolic values, consider mapping a string value 'Welcome guest" with the following output universe, which could both be matched against 'Welcome β' if α is true or against the literals 'Welcome guest' if α is false:

```
1 Welcome
2 #if α
3    β // $_GET['user']
4 #else
5    guest
6 #endif
```

To perform mapping in those cases, we use the following heuristic strategies:

– **Pivot mapping:** We first identify *pivots*—the string literals in the S-Model that can be mapped exactly to the string literals in the output universe. The remaining unmapped string literals will correspond to symbolic values or parts with alternatives. For instance, in the above example, we first map the string 'Welcome' to the corresponding string on the output universe. The

Table 1 Subject systems

Name	Version	Files	LOC	Test pool
AddressBook (AB)	6.2.12	100	18,874	75
SchoolMate (SM)	1.5.4	63	8,183	67
TimeClock (TC)	1.04	69	23,403	63
UPB	2.2.7	395	104,640	67
WebChess (WC)	1.0.0	39	8,589	52
OsCommerce (OC)	2.3.4	787	91,482	92
WordPress (WP)	4.3.1	793	342,097	65

unmapped string 'guest' will then be matched against the #if block in the output universe.

- **Best local mapping:** To identify which one of the alternatives should be mapped to a given string value, we recursively map the string value with each alternative and select the one with the best mapping result (the highest number of mapped characters). Note that we perform the mapping locally for (possibly nested) alternatives after pivots are identified in the previous step; we do not consider globally optimal mapping of alternatives. In this example, the string 'guest' will be matched with both values in the true branch (β) and false branch ('guest') of the #if block for comparison.
- **Location mapping:** Since mapping can be ambiguous without location information (e.g., the string value "guest" can be mapped to either the symbolic value β or the string literal "guest"), we use the location information provided by the S-Model of a string value to compare with the location of a string value or symbolic value in the output universe to select a correct mapping. In this example, considering the location information, the string 'guest' can be mapped to only one of the two branches.

Discussion. Since our output mapping algorithm works heuristically in some cases, it is important that there are sufficient pivots to guarantee that the local best mappings are correct. As there are often large chunks of texts that remain unchanged across different executions, these pivots make our mapping algorithm fast and highly precise. Note that our output mapping algorithm is extended from the CSMap algorithm described in our existing work [Nguyen et al., 2011]. While CSMap maps the output universe with a string in the output (without location information), our extended algorithm maps the output universe with an S-Model of a string value with its location information to avoid any possible ambiguous mappings.

4 Empirical Evaluation

We hypothesize that *output coverage is a preferable metric to assess bugs that manifest in the output of a web application*, such as HTML validation errors, layout issues, and spelling errors, whereas traditional code coverage metrics perform better with regard to bugs within the PHP code, such as undefined

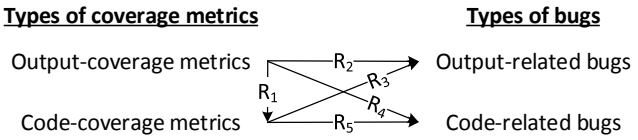


Fig. 5 Relationships among output-coverage metrics, code-coverage metrics and the two types of bugs

variables and missing parameters. That is, bugs in the output have different characteristics than bugs in the source code and are better addressed with a different form of coverage.

In the following, we distinguish between *output-related bugs* and *code-related bugs* and compare the effectiveness of *output-coverage metrics* and *code-coverage metrics* to assess coverage and prioritize testing effort. Specifically, as visualized in Figure 5, we test (1) whether output coverage and code coverage measure different aspects of a test suite (correlation $R_1 \ll 1$), (2) whether output coverage is a better predictor for output-related bugs than code coverage ($R_2 > R_3$), and (3) whether code coverage is a better predictor for code-related bugs than output coverage ($R_5 > R_4$). In addition, we check (4) whether a test selection strategy that prioritizes output coverage will find more output-related bugs than one that prioritizes code coverage, and (5) vice versa (i.e., whether a test selection strategy that prioritizes code coverage will find more code-related bugs than one that prioritizes output coverage). Note the complementary nature of those questions: The first three questions assess the predictive power of coverage metrics, whereas the last two questions assess the effect of using a coverage metric as an optimization criterion, as we will explain.

In addition, we indirectly demonstrate how output coverage could be *used in praxis* by testers who focus on output quality, illustrating a prototype visualization tool to visually guide testers in augmenting or prioritizing their test suites regarding output coverage.

4.1 Experiment Setup

To compare output coverage and code coverage, we execute various test suites of different size on existing PHP web applications. For each test suite, we assess coverage regarding both coverage metrics and use objective measures for three classes of bugs on the concrete output: HTML validation errors as reported by a static validator, spelling errors as reported by a spell checker, and PHP errors or warnings as reported by the PHP interpreter. We subsequently use that data to identify to what degree coverage metrics would have been suitable to select test suites that maximize the number of detected bugs of a certain class.

Subject systems. We assembled a corpus of seven open-source PHP web applications for our experiment, as characterized in Table 1. The first five are commonly used for evaluating research on PHP analysis [Artzi et al., 2010;

Nguyen et al., 2014, 2015a; Samimi et al., 2012a]. In addition, we selected two popular systems, *OsCommerce* and *WordPress*, that are newer, larger, and more frequently used.

Test case generation. As common for open-source PHP applications, none of our subject system comes with a comprehensive test suite. To avoid biases, we use an external tool to automatically generate a large pool of test cases from which we can derive many different test suites. We use a state-of-the-art black-box test generation framework, the web crawler Crawljax,[2] which systematically follows links and forms on a web application. We configured Crawljax to run with a depth of 5–10 depending on the sizes of subject systems. Among thousands of generated tests, we remove redundant test cases producing the exact same textual output, resulting in 52 to 92 tests per application (see *Test pool* in Table 1).

Computing output coverage and code coverage. For two of our seven subject systems (*OsCommerce* and *WordPress*), we are not able to compute the output universe due to technical (but not conceptual) limitations in the used symbolic-execution engine (i.e., engineering limitations due to an incomplete support of PHP's object system). Whereas we can compute S-models for concrete executions and assess absolute coverage of string literals (e.g., 100 characters of string literals are covered), without knowing the output universe, we cannot compute relative output coverage metrics. We decided to include the subject systems nonetheless because, for our technical evaluation, we can substitute relative Cov_{str} with absolute Cov_{str}, as the output universe is constant independent of the test suite. In addition, including the two larger and more popular systems improves the external validity of our results.

Technically, we use our dynamic instrumentation during a test run (Section 3.1) to record the number of statements, branches, and string literals that have been covered by a test case to compute (absolute) statement coverage, branch coverage, and coverage of string literals (Cov_{str}) for a given test suite. For the first five subject systems, we additionally compute coverage of output decisions (Cov_{dec}) based on the output universe. (We do not attempt to measure Cov_{ctx} since, similar to path coverage, it is not practical to compute, as we also explained earlier in Section 2.2).

Identifying bugs. Given a test case or test suite, we count the number of bugs it reveals. We initially considered *seeding bugs*, but decided that, assuming a uniform distribution of bugs, we could just assess the effectiveness of two test suites by comparing their coverage: A test suite with a higher coverage should, by definition, find more randomly seeded bugs on average. We use this insight and evaluate coverage, but additionally also analyze *reported issues* (bugs and warnings), as found by off-the-shelf bug detection tools in our subject systems.

For *output-related bugs*, we detect two kinds of issues: (1) *HTML validation errors*, as detected by the validation tool JTidy[3] and (2) *spelling errors*, as

[2] `http://crawljax.com/`

[3] `http://jtidy.sourceforge.net/`

detected by the open-source spell checker Jazzy.[4] Jazzy may flag false positives, such as unknown tool names, but we count those as spelling errors nonetheless, as a tester would have to investigate them as well. We chose these types of issues since they are common issues that can be objectively detected by external tools, without relying on human subjects to manually find, seed, or classify bugs. While semantic UI errors such as layout, font, or color issues are also a type of output-related bugs, for the purpose of our evaluation, it does not make a difference whether the errors are syntactic or semantic. Semantics UI errors are subjective and do not have a readily available ground truth (most of the subject systems do not have a long enough history of bug reports), and randomly seeding artificial ones does not provide any more insight.

For *code-related bugs*, we enable all levels of error reporting in PHP (setting `error_reporting(E_ALL)`) and collect PHP issues that are reported during a test run since testers might need to be aware of all such errors, warnings, or notices.

For bugs found within a test suite, we only count unique bugs as identified by a unique location in the server-side code; for output-related bugs, we map the bug locations back to PHP server-side code using our origin location tracking.

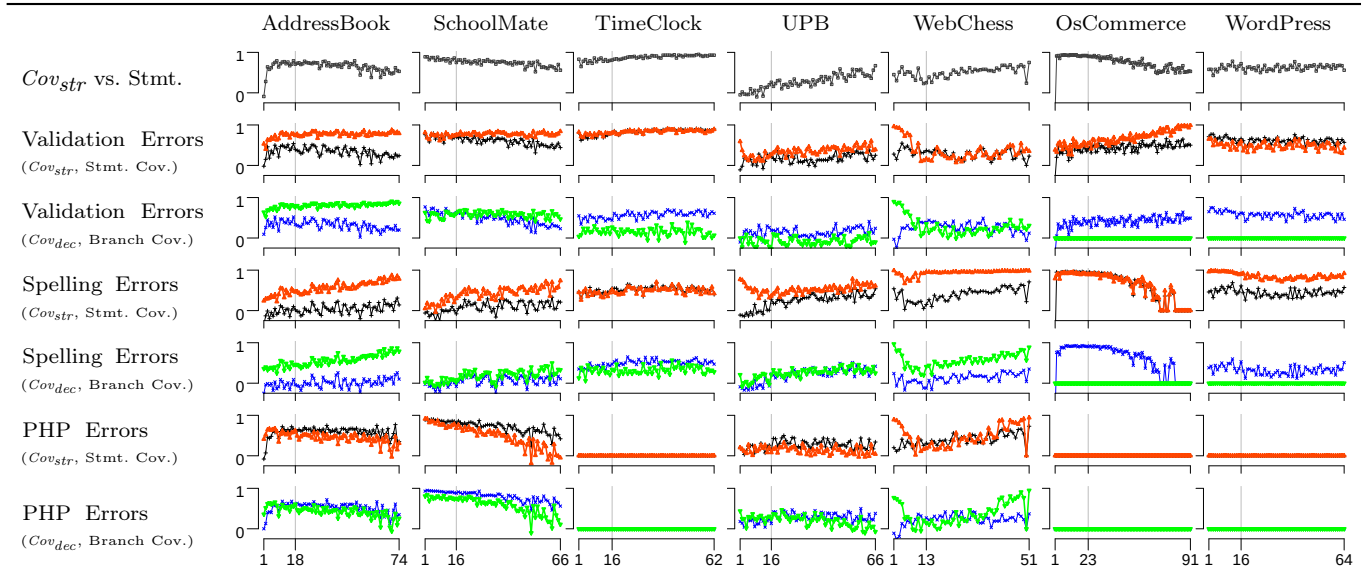
Note how we triangulate results with two different distributions of bugs: uniform distribution by assessing coverage and distribution of reported issues by external tools, which are likely more representative of real bugs that developers investigate and possibly fix.

Comparing output coverage and code coverage. Finally, we compare the effectiveness of output coverage and code coverage regarding finding the different classes of bugs across multiple test suites for each subject system. We use the two complementary strategies discussed above: First we *assess the predictive power of a coverage metric through correlations with random test suites*; then we *assess the effect of using a coverage metric as an optimization criterion* when creating a test suite.

- *Random test suites:* We generate 100 random test suites of size k; each test suite is composed of k randomly selected tests from the subject system's test pool. Despite their same size, the test suites contain different tests and cover different amounts of code and output and may find different numbers of bugs. By correlating the values of the different coverage measures with the number of detected bugs (using Pearson correlation), we can assess the effectiveness of a metric. We expect that output-coverage metrics correlate stronger with output-related bugs than code-coverage metrics, and vice versa (i.e., $R_2 > R_3$ and $R_5 > R_4$ in Figure 5). As a sanity check, we also compare the correlation of output coverage against code coverage across all test suites, expecting that they do measure different aspects and do not correlate perfectly (i.e., $R_1 \ll 1$ in Figure 5).
  We repeat the entire experiment for different sizes of the test suite k, from 1 to the size of the test pool minus 1 (since there is only one test suite with size equal to the size of the test pool, and correlation is not defined for such a set).

[4] `http://sourceforge.net/projects/jazzy/`

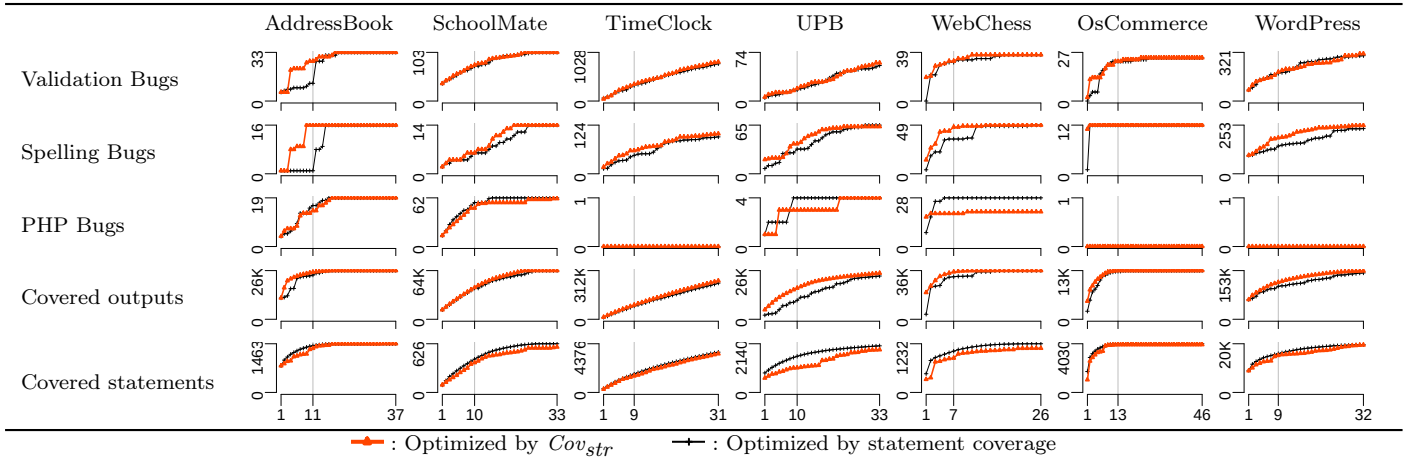
– *Optimized test suites:* To simulate a developer optimizing a test suite toward a specific metric, we optimize the test suite with k tests to maximize a coverage metric. Specifically, we implemented a (deterministic) greedy algorithm that incrementally adds test cases to a test suite such that each added test case makes the resulting test suite achieve the highest coverage. We create such an optimized test suite for all possible sizes k and for both Cov_{str} and statement coverage.
We expect that a test suite optimized for output coverage is more effective at finding output-related bugs than a test suite optimized for code coverage, and vice versa. (Note that the experiment does not aim to simulate optimizing for both types of coverage simultaneously as it would be a matter of trade-offs that depend on context-dependent factors.)



Sys.	Cov_{str} vs. Stmt Cov (p_1)	HTML Validation Errors: Stmt Cov_{str} (p_2)	HTML Validation Errors: Stmt Cov (p_3)	HTML Validation Errors: Branch Cov_{dec} (p_2)	HTML Validation Errors: Branch Cov (p_3)	Spelling Errors: Stmt Cov_{str} (p_2)	Spelling Errors: Stmt Cov (p_3)	Spelling Errors: Branch Cov_{dec} (p_2)	Spelling Errors: Branch Cov (p_3)	PHP Errors: Stmt Cov_{str} (p_4)	PHP Errors: Stmt Cov (p_5)	PHP Errors: Branch Cov_{dec} (p_4)	PHP Errors: Branch Cov (p_5)
AB	0.73	0.77	0.37	0.80	0.31	0.47	-0.02	0.46	-0.06	0.49	0.61	0.47	0.63
SM	0.81	0.76	0.68	0.66	0.61	0.35	0.05	0.09	0.03	0.76	0.86	0.73	0.91
TC	0.81	0.80	0.80	0.27	0.44	0.51	0.55	0.34	0.47	No PHP errors reported			
UPB	0.16	0.31	0.17	0.10	0.19	0.41	0.18	0.33	0.20	0.16	0.43	0.37	0.42
WC	0.25	0.16	0.37	0.35	0.39	0.95	0.15	0.41	-0.12	0.18	0.31	0.15	0.27
OC	0.90	0.54	0.41	na*	0.41	0.90	0.93	na*	0.90	No PHP errors reported			
WP	0.72	0.59	0.66	na*	0.62	0.90	0.61	na*	0.53	No PHP errors reported			

*: Cov_{dec} cannot be computed without output universe, see Section 4.1.

Fig. 6 Results of experiment 1. The plots are organized into seven rows: The first row shows the correlation between Cov_{str} and statement coverage across different sizes of the random test suites (up to the size of the test pool minus 1, as explained in the text). Each plot in the second row has two lines: The first line shows the correlation between Cov_{str} and coverage of validation errors, whereas the second line shows the correlation between statement coverage and coverage of validation errors. The remaining rows of plots are organized in similar manner as the second row but for other metrics and types of bugs. To make the data more readable, we record all the correlations where the size of the random test suites is at 25 % the size of the test pool (highlighted with gray vertical lines in the plots). For example, for *AddressBook*, we record the correlations where the size of the test suites is 18.) We present the data in the table below the plots and highlight the results that support or contradict our hypotheses.



	Validation Errors		Spelling Errors		PHP Errors		Output covered		Stmt. covered	
System	Cov_{str}	Stmt Cov	Cov_{str}	Stmt Cov	Cov_{str}	Stmt Cov	Cov_{str}	Stmt Cov	Cov_{str}	Stmt Cov
AB	27	12	16	1	14	16	25,593	24,000	1,325	1,403
SM	76	74	6	6	49	56	41,840	41,050	383	428
TC	355	328	59	47	0	0	91,286	84,928	1,481	1,653
UPB	19	17	40	33	3	4	16,940	10,325	1,081	1,590
WC	34	33	47	35	19	28	35,266	31,643	973	1,081
OC	22	22	12	12	0	0	13,403	13,215	3,955	4,025
WP	202	188	190	149	0	0	119,598	105,300	15,517	16,297

Fig. 7 Results of experiment 2. The plots are organized into five rows: The first three rows show the number of bugs detected when the test suites are optimized by Cov_{str} or statement coverage over different sizes. The last two rows show the output coverage and statement coverage of those test suites, respectively. Note that we draw the plots up to half the size of the test pool since they typically cover all the bugs at that point (e.g., in *AddressBook*, the plots show the maximum size at 37 instead of 75, which is the size of the test pool for this system). Similarly to Figure 6, the table below the plots shows the results for a fixed size of the test suites (again marked with gray vertical lines in the plots), where we highlight the results that support or contradict our hypotheses. We select the fixed size at 15 % because optimized test suites are expected to achieve maximum coverage at a smaller size than that of random test suites.

We do not specifically evaluate performance, as this is not a key issue. The symbolic execution ran within seconds for each of the first five systems, and our output coverage computation completed for less than a second per test case.

4.2 Results

Random test suites. We show all computed correlations for four kinds of coverage metrics and three kinds of bugs for different sizes of random test suites in Figure 6. For better readability, we exemplify specific results for test suites of a fixed size as part of that figure.

Output-coverage metrics tend to have low to strong correlation with code metrics, depending on how much code not directly contributing to the output is executed by the test suites. This non-perfect correlation is a first indicator that code coverage provides some indication of how much output has been covered, but is only an imperfect predictor. Both coverage metrics indeed measure different aspects of the test suite and may be suitable for different tasks.

In most systems and for most test suite sizes, we can see, as hypothesized, that output-related bugs correlate well with output-coverage metrics and this correlation is stronger than the correlation with code-coverage metrics. We can also see the opposite effect for code-related bugs. For example, for *AddressBook*, Cov_{str} is a much better predictor for HTML validation errors and spelling errors than statement coverage, whereas statement coverage is a better predictor for PHP errors (for test suites with size larger than 8). In addition, Cov_{str} tends to correlate better with output-related bug coverage than Cov_{dec} since Cov_{str} better measures "the amount of output" that is covered by a test suite. This suggests a test selection strategy which optimizes a test suite based on Cov_{str}.

Since we deal with real bugs that are not evenly distributed and there is randomness from the test suite selection, the results do not uniformly support our hypothesis, but they amount to strong evidence overall. Investigating the cases that contradict our hypothesis, we found that the bugs are clustered in only a few places. For example, *TimeClock* has a small number of files that produce large chunks of output but do not contain any spelling errors, leading to low correlations between Cov_{str} and spelling bugs. In *WordPress*, the skewed distribution of validation errors is caused by large amounts of output coming from JavaScript code, which is not relevant for HTML validation. Similarly, in *WebChess*, PHP bugs are found in only a few source files.

Optimized test suites. Whereas our first experiment compared the predictive power for random test suites, our second experiment explores the effect of creating test suites to optimize specific coverage metrics. That is, we investigate whether a small test suite with k tests is better at finding output-related bugs if tests have been specifically selected to maximize output coverage, compared to a test suite of the same size optimized for code coverage.

We plot the results for different sizes of test suites in Figure 7 and exemplify the specific results for a small test suite.

Again, the results largely confirm our hypothesis. While at a certain size, both test suites optimized for output coverage and code coverage detect all bugs in our subject systems, at smaller sizes test suites optimized for output coverage have an edge at detecting output-related bugs, whereas test suites optimized for code coverage have an edge for detecting code-related bugs. Despite some noise, the results are consistent across all systems.

Comparing output coverage across test suites optimized for output coverage and code coverage and vice versa (last two rows of plots in Figure 7) further allows us to quantify the benefit of optimizing for a coverage metric if we assume a uniform distribution of bugs in the output or the code (e.g., if we used uniformly seeded bugs). We can see that under these idealized (uniform) conditions, test suites optimized by output coverage will cover additional output, unless the test suites are too large that they all achieve maximum output coverage. It also confirms our first experiment in that both coverage metrics measure distinct characteristics.

Qualitatively investigating the cases where output coverage performs better than code coverage in detecting output-related bugs, we found that many systems contain large pieces of code for purposes such as user credentials validation, numerical computations, or filesystem operations, which—when executed—do not contribute significantly to the output. In such cases, a high coverage on code does not always translate into a high coverage on the output, and hence, a high coverage of output-related bugs.

4.3 Threats to Validity and Limitations

We made an explicit decision to not use bugs previously reported and fixed by developers in the subject systems but rather to use two proxies: bugs and warnings reported by neutral off-the-shelf tools and analyzing coverage as a proxy for uniformly seeding bugs, thus triangulating our results over two distributions of bugs for code and output issues each. Note that the former may report issues that are not relevant to developers, but we believe that their distribution will likely mirror those of relevant issues. Our results may not generalize to systems that have bug distributions that are different from those detected by these tools and different from uniform distributions.

We use Pearson correlation in our experiments since (1) assuming a uniform distribution of bugs, there should be a linear correlation of coverage of bugs (covering twice the outputs/statements might discover twice the bugs), and (2) our null hypothesis for Cov_{str} versus statement coverage is that they are linearly correlated and measure effectively the same. One could argue that a rank-based correlation (e.g., Spearman) may be more appropriate for decision and branch coverage. Our results (Figure 6) are stable with regard to both Pearson and Spearman; correlation coefficients are very similar (usually $<$

0.1 difference, often <= 0.02); only three relative close cases change colors in Figure 6 (one from green to red and two from red to green).

As usual, one has to be careful in generalizing results beyond the subject systems studied. However, these subject systems represent two major venues, including five systems commonly used in research and two large systems commonly used in practice.

Regarding the effectiveness of our technique, an evaluation of the symbolic-execution engine was done in a prior work [Nguyen et al., 2014]. Implementation-wise, our symbolic-execution engine is limited to PHP web applications without heavy use of object-oriented code [Nguyen et al., 2015a]. Specifically, we do not yet handle anonymous classes and object iteration (iterating through an object's properties with a `foreach` loop like entries in a map), or advanced features such as traits, object serialization, and cloning.[5] Since writing a full-fledged (and *symbolic*) interpreter for a mature and complex language like PHP is nontrivial, we needed to prioritize our engineering efforts for a research prototype towards parts of the language that are most commonly used such as (named) class declaration, object instantiation, field access, method invocation, and other features that are non-object-oriented. Similarly, we also currently do not handle client-side JavaScript processing as it is outside of this paper's scope and would require symbolic analysis of JavaScript code. Nevertheless, we do not anticipate any issues with symbolic execution that are specific to currently unsupported language features, but expect that they are a matter of engineering investments in supporting more language constructs, which can be improved independently in a separate work. The idea of output coverage is general to other forms of output and user interfaces, and the approach is applicable beyond PHP web applications (e.g., statically recovering possible variations of Java Swing dialogs or Android activities and representing them and their coverage in a visual form).

5 An Application Scenario: Visualizing Output Coverage

As a final indicator for the applicability of output coverage, we outline a prototype visualization tool that can help developers to assess output coverage and to select test cases that optimize output coverage, as done mechanically in our second experiment. This prototype also illustrates how output coverage can simplify many decisions during quality assurance that might otherwise require careful inspection of the program logic to understand how string literals are composed and propagated.

Whereas coverage metrics summarize coverage in a single number, for code coverage, visualizations that highlight *both covered and uncovered* lines of code (e.g., *EclEmma* and *Cobertura*) help developers to understand coverage and create test cases for uncovered parts. For output coverage, highlighting *only* covered string literals or output decisions within the source code is often an

[5] `http://php.net/manual/en/language.oop5.php`

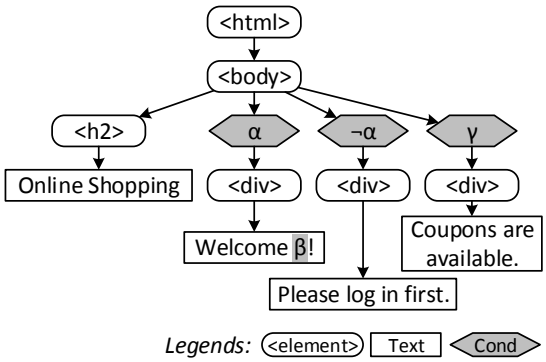


Fig. 8 The VarDOM for the PHP program in Figure 2a

inadequate abstraction for testers who are focused on the output and output-related quality criteria such as consistent font size and colors. Instead, we borrow from coverage visualizations for source code, but apply them to the rendered output of a web page. To that end, we design a prototype tool named *WebTest* that *displays the output universe in one single web page* and allows testers to visually *explore both covered and uncovered parts* of the output universe, as initially exemplified in Figure 1c. Since the output universe can be significant in size and cover many alternative pages, we additionally develop more compact representations in which a user can interactively explore coverage in different parts of the output space. We envision testers using WebTest to augment their test cases or navigate and inspect the output universe directly to detect certain classes of presentation faults.

To reason about the structure of the output universe and modify the HTML code to highlight covered and uncovered fragments, we parse the symbolic output (text with conditional segments and symbolic values) into a DOM structure with conditional nodes. We use variability-aware parsers developed initially to parse un-preprocessed C code [Kästner et al., 2011], which we recently adopted also for HTML, JavaScript, and CSS, to build a structure coined *VarDOM* [Nguyen et al., 2014]. In the following, we describe the VarDOM representation and how we visualize the VarDOM on a web page.

5.1 The VarDOM Representation

The VarDOM is a single tree structure that represents the hierarchical DOM structure (HTML Document Object Model) of a web page with additional nodes representing optional elements and symbolic values. Figure 8 shows the VarDOM of the PHP program in Figure 2a, which is the result of parsing its output universe representation in Figure 2b.

Each conditional node tracks the symbolic condition describing when the corresponding subtree is included in the page. For example, in Figure 8, the three tags are displayed under different constraints α, $\neg\alpha$, and γ, respectively

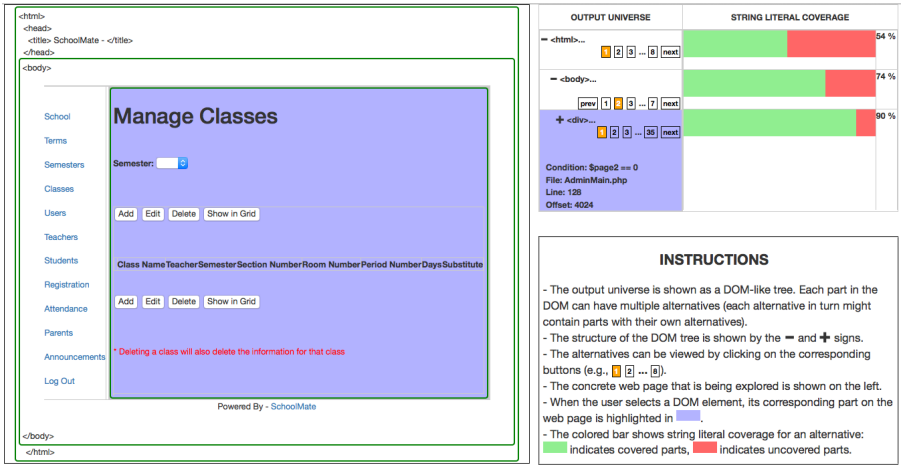


Fig. 9 Screenshot of WebTest on *SchoolMate-1.5.4*

(these symbols have location information in the source code, not shown in Figure 8). The parsing process can handle arbitrary conditions on the input strings, which do not need to align with the structure of the DOM (e.g., a single opening tag can be closed by two alternative closing tags) [Kästner et al., 2011]. In this way, the VarDOM compactly represents the output universe in a single DOM-like structure. By tracking location information, we can identify which structures in the VarDOM have been covered entirely or partially.

5.2 Visualizing Output Coverage with VarDOM

To visualize coverage, we render the VarDOM as an interactive web page as illustrated in Figures 1c and 9. We process VarDOM nodes as follows:

- Regular *DOM nodes* in the VarDOM are rendered as plain HTML. That is, we simply display the corresponding HTML element and recursively displaying its child nodes (if any) in HTML format. We optionally show some key elements as *<html>* and *<title>* verbatim, as done in many WYSIWYG HTML editors to emphasize the structure of the page. When possible, we also inject background colors to highlight covered and uncovered fragments.
- *Symbolic values* are rendered using special placeholders. For instance, in Figure 1c, the symbolic value is shown as `$_GET['user']`.
- For *conditional nodes* (stemming from control-flow decisions in the PHP program and shown as `#if` directives in the output universe as in Figure 1c), we either show all alternatives sequentially as in Figure 1c or provide an interactive mechanism to select as in Figure 9. (We describe more about this mechanism below.)

– *Coverage information* is encoded as background colors, similar to IDE tools for code coverage. We highlight the parts that are entirely covered or uncovered by a given test suite in green and red background colors (injected by manipulating CSS attributes). Information about different coverage metrics is shown at the bottom part of the visualized web page (in Figure 1c, we show Cov_{str} only).

The interactive mechanism is necessary for many web applications, because the output universe can be very large. For example, even in the relatively small *SchoolMate* application, a rendering of all alternatives sequentially would fill 59 printed pages. Therefore, instead of displaying all alternatives at once, we dynamically show and hide the alternatives on the output universe in tabs such that only one alternative is shown for each decision at a time, but such that a user can interactively explore alternatives through buttons in the interface. For example, in Figure 9, a tester has selected the first alternative out of eight possible cases for the top-most decision. As coverage in certain parts may be hidden from view, we additionally indicate relative coverage, including potentially hidden parts of inner decisions, with our coverage metrics in a side bar.

Importantly, our encoding mostly preserves the visual layout of the original page, allowing testers to quickly explore output-related issues such as inconsistent font sizes or spelling issues. Furthermore, it allows testers to assess *how well a test suite covers the output universe* and to *explore uncovered parts* of the web page to create new test cases, all at a fundamentally different abstraction level of outputs rather than at the level of string literals and code instructions within PHP code.

6 Related Work

Output coverage. Supporting testing, especially web testing, from the perspective of a tester interested in the output has received increased attention recently.

First, our work is related to the *output-uniqueness* test selection criteria [Alshahwan and Harman, 2012, 2014], which aimed to generate diverse outputs and show that output uniqueness provides a useful replacement for whitebox testing if the source code in unavailable. They proposed seven syntactic abstractions pertinent to web applications to avoid sensitivity to nondeterministic output. They measure coverage only as the absolute number of distinct observed outputs of a test suite, but have no notion of an output universe that could help identify uncovered outputs nor the notion of decisions in producing that output leading to our Cov_{dec} and Cov_{ctx} metrics. In contrast, our contribution is in enabling output uniqueness and diversity to test adequacy criterion and showing a difference between the measures for different kinds of bugs. In addition, our symbolic approximation can precisely identify nondeterminism from the environment or user input. Finally, our goal is not to substitute code

coverage by output coverage, but to provide a complementary coverage metric that is tailored to the purposes for a specific group of testers. Specifically, our evidence generally tends to align with previous findings and does not contradict them. The correlation between Cov_{str} and statement coverage is high in most systems as in the work of Alshahwan and Harman [2014], but as we showed, the difference is strong enough and has a clear one-directional effect to explain differences in effectiveness for different kinds of bugs. Correlations between decision and branch coverage are similar (0.70, 0.82, 0.35, 0.69, 0.36, N/A, N/A). We did not compare context and path coverage, as the output universe grows exponentially or is in fact infinite.

Second, Zou et al. [2014] introduce a V-DOM coverage for web applications. They convert a PHP program into C code and performs static analysis for control flows and data flows to build a V-DOM tree. Roughly similar to our VarDOM [Nguyen et al., 2014], V-DOM's nodes represent all possible DOM objects that can appear in any possible executions of a page. V-DOM coverage is defined as the ratio of the number of covered DOM objects over the total number of DOM objects. In comparison, their work focuses only on the coverage on DOM objects, whereas we provide a suite of coverage metrics analogous to code coverage metrics, including coverage on *output decisions* for which we additionally track decisions that lead to different outputs. V-DOM does not target the server-side branches or paths relevant for generation of client pages. As a consequence of not tracking decisions, they cannot create the more compact visual representation we introduce in Figure 9. Technically, their V-DOM coverage can be defined in terms of the nodes in VarDOM.

We would have liked to empirically compare with those existing works [Alshahwan and Harman, 2014; Zou et al., 2014]. Unfortunately, neither the tools nor data (test suites) were available for a direct comparison. As they rely on further unavailable tools, reimplementing them was not realistic. We publicly released our tool for future comparisons.

Third, DomCovery [Mirzaaghaei and Mesbah, 2014] is another DOM-based coverage criteria for web applications. The coverage is defined at two levels: (1) the percentage of DOM states and transitions covered in the total state space, and (2) the percentage of elements covered in each particular DOM state. In contrast, we do not focus on the coverage on DOM states in the state space. DomCovery does not account for elements that are shared across different states, which in contrast, we represent explicitly. Finally, they do not aim to track or reveal the alternative parts from output decisions, as we do in WebTest.

More generally, earlier work has proposed output uniqueness for testing. Several researchers introduce the concept of an equivalent class [Goodenough and Gerhart, 1975; Ostrand and Balcer, 1988; Weyuker and Ostrand, 1980] in which an element in a class leads to a correct output if all elements in the class lead to correct output. Subdomain partition methods are proposed for the input space via specification analysis [Ostrand and Balcer, 1988]. Richardson and Clarke [1981] uses symbolic evaluation to partition the set of inputs into

procedure subdomains so that the elements of each subdomain are processed uniformly by testing.

Analyzing web output. Many researchers have investigated the output of web applications and the relationship between code and output for various purposes. Wang et al. [2012] aim to map changes in the client-side code to PHP code using the recorded run-time mappings and static impact analysis. Elbaum et al. [2006] use dynamic analysis to analyze responses to draw inferences about its interface. PHPQuickFix [Samimi et al., 2012b] examines *constant prints*, i.e., the PHP statements that print directly string literals and repairs HTML ill-formed errors. PHPRepair [Samimi et al., 2012b] follows a dynamic approach in which a given test suite is used to generate client-side code with different server-side executions, while tracing the origin of output strings. Minamide [2005] proposed a string analyzer that takes a PHP program and a regular expression describing the input, and validates approximate HTML output via context-free grammar analysis. Several string taint-analysis techniques were built for PHP web programs and software-security problems [Wassermann and Su, 2008; Xie and Aiken, 2006; Yu et al., 2011]. In prior work, we have analyzed both code and all possible outputs to trace data flows from the code to the web page and back [Nguyen et al., 2015a] and used the VarDOM to provide IDE support [Nguyen et al., 2014, 2015b]. In this work, we analyze the output universe for a new purpose to define and visualize output coverage of a test suite.

Web testing. More generally, there is a rich literature on web testing [Doğan et al., 2014; Li et al., 2014] overall. This adopts many quality assurance strategies developed in other contexts to the specifics of web applications, including dynamic symbolic execution [Artzi et al., 2010; Saxena et al., 2010; Wassermann et al., 2008], search-based testing [Ali et al., 2010; Alshahwan and Harman, 2011; McMinn, 2004], mutation testing [Brady, 2016; Praphamontripong and Offutt, 2010], random testing [Artzi et al., 2011; Frantzen et al., 2009; Heidegger and Thiemann, 2010], and model-based testing [Andrews et al., 2005; Ricca and Tonella, 2001]; they are all focused on analyzing the source code of the web application. Also several specialized techniques to generate test cases by crawling the web page [Girardi et al., 2006; Mesbah and van Deursen, 2009; Mesbah et al., 2012; Milani Fard et al., 2014; Raghavan and Garcia-Molina, 2001] and collecting session data [Elbaum et al., 2003] have been explored. As suggested previously [Alshahwan and Harman, 2014], output coverage can be used a post-processing step to select a subset of the test cases generated by these tools in order to focus on output defects. We have used such crawler to create the test suites in our evaluation.

7 Conclusion

We explored output coverage for web testing, but the ideas generalize to testing outputs of other kinds of applications. We use symbolic execution of PHP to

approximate the output universe, identify which parts of the output universe are covered by test cases, and subsequently parse and transform the DOM structure of the resulting page to visualize coverage. As shown in our evaluation, selecting test cases by output coverage is more effective in identifying output-related bugs, such as HTML validation errors and spelling errors, than traditional selection by code coverage. In principle, the same mechanisms could be applied also to other forms of output and user interfaces, for example, statically recover possible variations of Java Swing dialogs or Android activities and represent them and their coverage in a visual form. Although different analysis and visualization techniques will be needed (e.g., to discover a sequence of API calls setting various parameters instead of `echo` statements printing text), the same basic ideas apply.

Acknowledgements. Kästner's work has been supported in part by the National Science Foundation (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Nguyen's work has been supported in part by CCF-1349153, CCF-1320578, and CCF-1413927.

References

Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.*, 36(6):742–762, November 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.52. URL `http://dx.doi.org/10.1109/TSE.2009.52`.

Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4577-1638-6. doi: 10.1109/ASE.2011.6100082. URL `http://dx.doi.org/10.1109/ASE.2011.6100082`.

Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1345–1348. IEEE Press, 2012.

Nadia Alshahwan and Mark Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 181–192, New York, NY, USA, 2014. ACM.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4: 326–345, 2005.

Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.*, 36 (4):474–494, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2010.31. URL http://dx.doi.org/10.1109/TSE.2010.31.

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985871. URL http://doi.acm.org/10.1145/1985793.1985871.

Padraic Brady. Mutation testing framework for php, 2016. URL https://github.com/padraic/humbug.

Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. Web application testing: A systematic literature review. *J. Syst. Softw.*, 91:174–201, May 2014. ISSN 0164-1212. doi: 10.1016/j.jss.2014.01.010. URL http://dx.doi.org/10.1016/j.jss.2014.01.010.

Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL http://dl.acm.org/citation.cfm?id=776816.776823.

Sebastian Elbaum, Kalyan-Ram Chilakamarri, Marc Fisher, II, and Gregg Rothermel. Web application characterization through directed requests. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 49–56, New York, NY, USA, 2006. ACM. ISBN 1-59593-400-6. doi: 10.1145/1138912.1138923. URL http://doi.acm.org/10.1145/1138912.1138923.

Lars Frantzen, Maria Las Nieves Huerta, Zsolt Gere Kiss, and Thomas Wallet. Web services and formal methods. chapter On-The-Fly Model-Based Testing of Web Services with Jambition, pages 143–157. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01363-8. doi: 10.1007/978-3-642-01364-5_9. URL http://dx.doi.org/10.1007/978-3-642-01364-5_9.

Christian Girardi, Filippo Ricca, and Paolo Tonella. Web crawlers compared. *International Journal of Web Information Systems*, 2(2):85–94, 2006.

John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510. ACM, 1975.

Phillip Heidegger and Peter Thiemann. Contract-driven testing of javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13952-3, 978-3-642-13952-9. URL http://dl.acm.org/citation.cfm?id=1894386.1894395.

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of*

the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048128. URL http://doi.acm.org/10.1145/2048066.2048128.

Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. Two decades of web application testing-a survey of recent advances. *Inf. Syst.*, 43(C):20–54, July 2014. ISSN 0306-4379. doi: 10.1016/j.is.2014.02.001. URL http://dx.doi.org/10.1016/j.is.2014.02.001.

Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004. ISSN 0960-0833. doi: 10.1002/stvr.v14:2. URL http://dx.doi.org/10.1002/stvr.v14:2.

Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070522. URL http://dx.doi.org/10.1109/ICSE.2009.5070522.

Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012. ISSN 1559-1131. doi: 10.1145/2109205.2109208. URL http://doi.acm.org/10.1145/2109205.2109208.

Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 67–78, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642991. URL http://doi.acm.org/10.1145/2642937.2642991.

Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963.

Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. doi: 10.1145/1060745.1060809. URL http://doi.acm.org/10.1145/1060745.1060809.

Mehdi Mirzaaghaei and Ali Mesbah. Dom-based test adequacy criteria for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 71–81, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610406. URL http://doi.acm.org/10.1145/2610384.2610406.

Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings*

of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 518–529, New York, NY, USA, 2014. ACM.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 369–380, New York, NY, USA, 2015a. ACM.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Varis: IDE support for embedded client code in PHP web applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 693–696, Piscataway, NJ, USA, 2015b. IEEE Press.

T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988. ISSN 0001-0782. doi: 10.1145/62959.62964. URL http://doi.acm.org/10.1145/62959.62964.

Upsorn Praphamontripong and Jeff Offutt. Applying mutation testing to web applications. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 132–141, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4050-4. doi: 10.1109/ICSTW.2010.38. URL http://dx.doi.org/10.1109/ICSTW.2010.38.

Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 129–138, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4. URL http://dl.acm.org/citation.cfm?id=645927.672025.

Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7. URL http://dl.acm.org/citation.cfm?id=381473.381476.

Debra J. Richardson and Lori A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 244–253, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL http://dl.acm.org/citation.cfm?id=800078.802537.

Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287, Piscataway, NJ, USA, 2012a. IEEE Press.

Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 277–287, Piscataway, NJ, USA, 2012b. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337257.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.38. URL http://dx.doi.org/10.1109/SP.2010.38.

Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 16:1–16:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393614. URL http://doi.acm.org/10.1145/2393596.2393614.

Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368112. URL http://doi.acm.org/10.1145/1368088.1368112.

Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 249–260, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390661. URL http://doi.acm.org/10.1145/1390630.1390661.

E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3):236–246, May 1980. ISSN 0098-5589. doi: 10.1109/TSE.1980.234485. URL http://dx.doi.org/10.1109/TSE.1980.234485.

Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267336.1267349.

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 251–260, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985828. URL http://doi.acm.org/10.1145/1985793.1985828.

Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. Virtual dom coverage for effective testing of dynamic web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 60–70, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610399. URL http://doi.acm.org/10.1145/2610384.2610399.