Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis

Xiaoyin Wang[1], Lu Zhang[1]*, Tao Xie[2], Yingfei Xiong[1], Hong Mei[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MOE, China
[2]Department of Computer Science, North Carolina State University, USA
{wangxy06,zhanglu,xiongyf04,meih}@sei.pku.edu.cn, xie@csc.ncsu.edu

ABSTRACT

Web applications are becoming increasingly popular nowadays. During the development and evolution of a web application, a typical type of tasks is to change the presentation of the web application, such as correcting display errors, adding user-interface controls, or changing appearance styles. To change the presentation of a static web page, developers are able to modify the HTML text of the web page using a graphical web-page editor. However, to change the presentation of a dynamic web application, instead of using a graphical web-page editor to directly modify generated web pages, developers need to modify the code that generates the web pages. As manually performing presentation changes in dynamic web applications is tedious and error-prone, we propose a novel approach based on *c llab a i e h b id anal i* that combines static analysis and dynamic analysis to facilitate developers to perform presentation changes in dynamic web applications. Our approach includes two parts. The first part takes as input the presentation change to be performed on a generated web page (with proper runtime information), and uses *d namic ing- igin anal i* to locate the source-code segment that generates the changed part of the web page. The second part checks unexpected impact of directly performing the change on the source-code segment, and asks for human intervention when unexpected impact exists. We implemented our approach for the PHP language and carried out an empirical study on 39 presentation-change tasks identified from 600 bug reports of three real-world dynamic web applications (in total more than 148 KLOC). Among the 39 tasks, our approach is able to correctly locate the place to modify in each presentation-change task and correctly perform the presentation change on the source code in more than half of the tasks.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: [Distribution, Maintenance, Enhancement]

General Terms

Reliability

Keywords

presentation change; web application; dynamic string-origin analysis

*Corresponding Author

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

1. INTRODUCTION

Recently, web applications are becoming increasingly popular due to easier access to the Internet. Various researchers have developed techniques to facilitate the development and evolution of web applications, such as testing web applications [4, 3, 19], static checking for bugs in web applications [10, 33], and refactoring web applications [31, 18]. A typical type of daily tasks during the development and evolution of web applications is presentation changes, which are modifications made to change the appearance of web pages. Typical presentation changes in web applications include correction of display errors, adding user-interface controls, etc. According to our investigation of 600 bug reports from three real-world web applications, about 7% of the bug reports are presentation changes, and these presentation-change bug reports primarily occur in early evolution stages of the applications.

On a static web page, it is straightforward to perform a presentation change, because a developer can directly modify the graphical presentation of each static web page with the help of a graphical web-page editor. However, most web applications nowadays contain a large number of web pages that are generated dynamically at the server side through code written in a programming language (e.g., PHP). In such a case, it is difficult to perform presentation changes, because the developer should not directly modify the generated web pages but needs to modify the code that generates the web pages requiring presentation changes.

Manually performing a presentation change in the code that generates a web page is often tedious and error-prone for two main reasons. First, to locate the piece of code for modification, the developer needs to be familiar with both the structure of the generated web page and the structure of the code generating the web page. Typically, the structure of the web-page-generation code is different from the structure of the generated web page itself. For example, the web-page-generation code may use complex loop structures to generate repetitive fragments in the generated web page. Second, the developer also needs to ensure that the modification of the located code realizes only the presentation change without inducing unexpected impact. Due to the dynamic nature, modifying one place in the web-page-generation code may impact many places in the generated web page and/or other web pages that can be generated by the code. Without knowing the potential impact, the developer may change the code in an unexpected way.

Text search may be helpful in some occasions where the developer uses the need-to-change presentation fragment in the generated web page as a query to search in the web-page-generation code. However, it is common that the text serving as the query appears many times in the web-page-generation code and the developer thus faces a large number of false positives. For example,

if the developer wants to delete a certain tag "<tr>"[1] in a dynamically generated web page, he or she may get hundreds or thousands of irrelevant "<tr>" tags when searching for the code that generates the web page with "<tr>". Furthermore, when the need-to-change presentation fragment is not generated from one entire constant string, text search can hardly locate the piece of code corresponding to the need-to-change presentation fragment. In the preceding example, if the developer searches using "<tr>" together with its surrounding texts to try to filter out some irrelevant results, text search may return nothing, because "<tr>" and its surrounding texts may be in separate constant strings in the source code, and are concatenated at runtime. Fault localization [2] is another approach to supporting locating the code segment to change. However, existing fault-localization techniques usually require to run the web application with many test cases including passing and failing ones, incurring much extra overhead. Furthermore, these techniques typically require developers to inspect a non-trivial number of suspected locations without providing support on changing these locations.

To facilitate developers to perform presentation changes, we propose a novel approach to automating presentation changes in dynamic web applications. Our approach aims to help a developer perform presentation changes in a dynamic web application in a way similar to performing presentation changes in static web pages. To use our approach, a developer needs to run an instrumented version of the web application to generate the need-to-change web page with runtime information produced by the instrumentation. Then the developer can annotate a presentation change on the generated need-to-change web page by selecting a need-to-change HTML segment and providing the new value[2]. With the annotated change and the runtime information as input, our approach automatically performs a change on the source code to fulfil the presentation change if no unexpected impact is found, or provide the developer with the need-to-change location in the source code together with the information of detected unexpected impacts.

Our approach basically consists of two parts. The first part is *d namic ing- igin anal i* , which takes as input the runtime information of a web-page-generating execution, and a given part p in the generated web page. The output of dynamic string-origin analysis is the source-code segment that generates p during the execution. Therefore, our dynamic string-origin analyzer can map a presentation change back into the web-page-generation code. The second part is *ne ec ed-im ac de ec i n*, which takes as inputs the source code of a web application, an annotated presentation change PC on the web page generated by the web application, and a change SC on the source code. The output of our unexpected-impact detector is all the possible places affected by SC except PC.

The general idea of our approach is to generate code changes by mapping changes from the program output to its source code using dynamic analysis and check the safety of the changes through static analysis and dynamic analysis. In this paper, we refer to this two-part analysis for automating code changes as *c llab a i e h - b id anal i* , since both static analysis and dynamic analysis are involved and collaboration between the two types of analysis is essential. Note that the general idea of collaborative hybrid analysis may not be limited to the particular problem discussed in this paper, but be potentially applicable to automate other types of code changes, such as fixing SQL queries or changing file outputs.

[1]In HTML, "<tr>" causes to print a new line in the web page, similar to "\n" in some programming languages.

[2]For insertion, the developer needs actually to choose a point between two characters in the HTML document as the insertion point.

As we implemented and evaluated our approach for dynamic web applications written in the PHP language, we use examples in PHP throughout this paper for illustration. However, the idea of our approach is general and in principle applicable for dynamic web applications written in other languages. The presentation changes that this paper aims to handle are changes in the presentation part of web pages, including the display style, the layout of controls, and the format and labels of controls. Handling changes about the displayed data and computation results is out of the scope of this paper. To evaluate our approach, we carried out an empirical study on 39 presentation changes identified from 600 bug reports in three real-world dynamic web applications (in total up to 148 KLOC). Note that, in our evaluation, we extracted presentation changes from bug reports because we had the recorded developer changes as the ground truth. Our approach is not specific to fixing presentation-change bugs, but can also be applied to evolution-related presentation changes. Our empirical results show that our approach is able to correctly locate the place to modify to realize a presentation change in each presentation-change task. Furthermore, our approach can correctly propagate the presentation change to the source code in more than half of the tasks, and correctly detect unexpected impacts in most of the remaining tasks.

This paper makes the following main contributions:

- An approach based on collaborative hybrid analysis that automatically performs presentation changes in a dynamic web application.
- Dynamic string-origin analysis, which locates the source-code segment in the web application that generates a certain part of a generated web page.
- Unexpected-impact detection, which determines whether the web-page-generation code can be directly changed to realize the presentation change without unexpected impact.
- An empirical study on a total of 600 bug reports of three real-world dynamic web applications (i.e., $SquirrelMail$, $WebCalendar$, and $OrangeHR$) to demonstrate the effectiveness of our approach.

2. MOTIVATING EXAMPLE

Consider the example from file "mailbox_display.php" in the version of May-23-2001 of the $SquirrelMail$ project. The presentation change is made to fix bug report No. 417165. The textual description of the bug report is "Missing </form> in mailbox_display.php". From the bug report, we know that a bug in the code results in a missing "</form>" in an HTML text generated by "mailbox_display.php". Therefore, the developer needs to change the code in "mailbox_display.php" to make the string "</form>" inserted into an appropriate position in the generated web page. The code related to the change is depicted below.

```
1 function displayMessageArray(...) {
    ...
2   mail_message_listing_beginning(...);
    ...
3   echo '</table></FORM>';
4   echo '</td></tr>';
    ...
6 }
7 function mail_message_listing_beginning(...){
    ...
8   echo '\ n\ n\ n<FORM name=messageList method=post
        action=\"$moveURL\">\ n';
    ...
9 }
```

The form begins in `mail_message_listing_beginning`, but needs to be closed in `displayMessageArray`. Thus, the developer must check the code very carefully to insert the string "</form>" into an appropriate position. Actually, earlier comments to the bug

report suggest to insert `echo '</FORM>'` at the end of function `mail_message_listing_beginning`. However, the correct fix, which was done a month later, actually inserts `echo '</FORM>'` in the middle of `displayMessageArray`, as shown in the italicized Line 3. From the example, we can see that, due to the complexity of the web-page-generation code, it is tedious and error-prone for the developer to perform presentation changes manually.

The HTML segment generated by the preceding code is depicted below. The italicized `</FORM>` is where the missing "</form>" should be inserted into the generated HTML text.

```
<FORM name=messageList method=post
action="move_messages.php?msg=&mailbox=INBOX&startMessage=1">;
</table></FORM></td></tr>;
```

With our approach, the developer can run the instrumented source code of $SquirrelMail$ to record the required execution information. After the developer specifies that the "</FORM>" should be inserted between "</table>" and "</td>" in the generated HTML text, our approach maps all the substrings in the generated HTML text to their origins in "mailbox_display.php" and thus identifies the place in "mailbox_display.php" where the `echo '</FORM>'` should be inserted. In particular, our approach actually appends "</FORM>" to one constant string in "mailbox_display.php". Our approach further checks that this change to the PHP file would not have any unexpected impacts.

3. RELATED WORK

For convenience, we classify related research into seven categories: string-taint analysis, dynamic taint analysis, impact analysis, hybrid analysis, automated support for presentation changes, bidirectional transformation, and automated bug fixing.

String-Taint Analysis. Our dynamic string-origin analysis is closely related to static string-taint analysis, which is developed by Wassermann and Su [32] on the basis of string analysis [6, 20, 28], in order to determine whether the data origins of a given string variable are from an unsafe source. Yu et al. [38] recently proposed an automaton-based string-taint analyzer for PHP with stronger handling of string operations. In our previous research [30, 31], we adapted string-taint analysis to statically locate all the constant strings that are the data origins of a given string variable. Our dynamic string-origin analysis differs from existing research on string-taint analysis as follows. First, we apply string-taint analysis dynamically on one execution trace, while all existing variants of string-taint analysis statically analyze all possible executions. Second, our dynamic string-origin analysis uses execution-specific information (such as actual values of unanalyzable strings and execution-specific finite state transducers) to improve the precision of analysis. Third, the taints used in our string-origin analysis record locations of a runtime string value's data origins, which are different from the taints used in any existing variants of string-taint analysis. Finally, our unexpected-impact detection is able to check the impacts of changing a certain constant string in the code; such checking cannot be handled by existing string analysis and string-taint analysis.

Dynamic Taint Analysis. Dynamic taint analysis [22] is a category of techniques that add taints on a runtime value (e.g., a byte or an object), propagate taints during the flow of the value, and trace where the value goes to by checking the taints. Typically, dynamic taint analysis [12, 24] traces where the user input goes to at runtime, since user input is often considered as insecure in many security-related tasks. As a character in a string takes a byte, our dynamic string-origin analysis is similar to byte-level dynamic taint analysis. However, there are two main differences between the two analyses. First, our dynamic string-origin analysis uses complex taints to trace location information, while existing techniques for dynamic taint analysis mainly use boolean taints to trace security information. Second, our dynamic string-origin analysis is based on analyzing productions in the extracted context free grammar, while byte-level dynamic taint analysis is based on analyzing all operations on each byte of each involved variable. Therefore, it would be much more expensive to adapt byte-level dynamic taint analysis for our target problem than using our dynamic string-origin analysis. For example, byte-level dynamic taint analysis needs to instrument at a level finer than the statement level used in our dynamic string-origin analysis; and byte-level dynamic taint analysis needs to instrument all involved library code, but our dynamic string-origin analysis does not. Note that the propagation of complex taints could further worsen the situation.

Impact Analysis. Our unexpected-impact detection is related to research on impact analysis. Impact analysis refers to approaches to inferring how a change to one piece of code affects other places in the same code base. Arnold and Bohner [1] wrote a book on impact analysis for software changes. The book contains the latest approaches in the area of impact analysis at its publication time, such as dependence analysis [13] and static slicing [36], and provides ideas and guidelines for using impact analysis practically. Ren et al. [25] proposed an automatic impact-analysis tool for Java called Chianti. Chianti first decomposes the differences between two versions of software to a list of atomic changes, and then decides which parts of the code and which test cases may be affected by the changes. Law and Rothermal [17] proposed to use dynamic analysis to decide more precise but incomplete impact of a software change. Existing techniques of impact analysis, either static or dynamic, try to locate all affected places of a given change on the source code. In contrast, our unexpected-impact detection takes as inputs both the source code and an execution trace, and tries to locate only the affected places that are "unexpected" (i.e., the affected places that do not correspond to the developer-specified presentation change).

Hybrid Analysis. There have been some research efforts on integrating static analysis and dynamic analysis. Nimmer and Ernst [23] proposed a bug-detection approach that first mines program invariants from a large number of program executions, and then verifies the mined invariants using static analysis. This research uses dynamic analysis and static analysis as two independent steps, while our approach relies on the collaboration of static analysis and dynamic analysis. Furthermore, dynamic analysis in our approach requires only one single execution, while dynamic analysis in their approach requires multiple executions. Dufour et al. [8] proposed the blended analysis, which performs static analysis in the scope of an execution trace. Conceptually, the first part in our approach is similar to this research, but the second part in our approach involves more static and dynamic analyses.

Automated Support for Presentation Changes. Whyline [16] is a debugging tool that allows developers to ask "why" and "why not" questions about presentation bugs. It provides some support for presentation changes. Similar to our approach, Whyline also uses program instrumentation to build mappings between GUI components and the source code. Therefore, Whyline is able to answer questions about the GUI components such as "why is the color of the button red?", and guide developers to the related places in the code to fix presentation bugs. However, there are two main differences between Whyline and our approach. First, as Whyline is developed for traditional GUI structures, it is not applicable for dynamic web applications in which the whole GUI is built by concatenations and operations of strings. In contrast, our approach uses dynamic string-origin analysis to map GUI components in web applications to their origins in the source code. Second, beside mapping

between GUI components and the source code, our approach further provides support for developers to propagate changes in generated web pages back to the source code.

Nguyen et al. [21] proposed an approach that automatically propagates fixes of syntactical errors in HTML texts back to the server-side code. Their basic idea is to first statically build from the source code a symbolic model that estimates all the possible HTML pages that the source code may generate, and then they match the fixed HTML page to the model to locate the source-code segment that corresponds to the fixed part of the HTML page. Our work differs from this approach in three main aspects. First, our approach is able to handle presentation changes besides syntactical fixes. Second, we use dynamic analysis, which is able to precisely handle all statically uncertain issues such as user inputs and library functions without source code. Thus, our approach is able to construct a correct mapping from an arbitrary part of the HTML page back to the source code. Third, our approach further detects the unexpected impacts of propagating a change on the HTML page back to the source code.

There is concurrently conducted work by Samimi et al. [26] that proposed an approach to automatically fixing HTML-generation errors. Their approach is based on multiple test cases, and generates a string constraint on the printed constant strings (constant strings that are echoed to the HTML page) to make all the test cases pass. Then a fix is automatically generated through solving the constraint. Compared to their approach, our approach relies on only one test case. Furthermore, by further tracing back to the data origins of printed variables, our approach is able to generate fixes beyond changing printed constant strings.

Bidirectional Transformation. Bidirectional transformation aims to tackle the problem of maintaining data consistency between two related data sources [9, 14]. When one of the two related data sources changes, the other data source must change accordingly. Bidirectionalization [37, 11], which aims to add the capability of backward transformation to one-directional transformation programs, is a state-of-the-art solution for bidirectional transformation. Thus, maintaining the consistency between the presentation and the source code can also be viewed as a bidirectionalization problem in general. There are two main differences between our approach and bidirectional transformation. First, bidirectional transformation propagates changes in output back to input, while our approach propagates changes in output back to the program. Second, existing bidirectionalization techniques are able to handle only data transformation programs that contain no complex operations (e.g., string operations, arithmetic operations), while our approach is able to handle string operations that are popular in our target problem.

Automated Bug Fixing. Automatically performing presentation changes can be viewed as a case of automated bug fixing, such as Wei et al. [34], Weimer et al. [35], Carzaniga et al. [5], and Dallmeier et al. [7]. Typically, these approaches search for good fixes using specifications or test oracles as the criterion. Furthermore, these approaches are concerned with bugs that result in erroneous program states instead of the presentation. Different from these approaches, our approach actually propagates the fixes written by developers in the generated web pages to the source code, and our approach is concerned with presentation bugs that may not be related to program states.

4. APPROACH

To solve the problem of automating presentation changes in dynamic web applications, we propose a general idea named *c l-lab a i e h b id anal i* , which contains two main parts: (1) a change-mapping part that involves dynamic analysis to map the change on the generated HTML text to the code generating the HTML text and (2) a checking part that involves both static analysis and dynamic analysis to make sure that the change in the code would not bring unexpected impacts. Note that this idea is general and may be applicable to various code-change scenarios other than presentation changes in dynamic web applications.

In the change-mapping part, a developer needs to (re-)generate the need-to-change web page through executing an instrumented version of the web application. Along with the generation of the need-to-change web page, our approach uses the instrumented code to record some runtime information. With the help of the runtime information, our approach uses *d namic ing- igin anal i* to locate in the web-page-generation code the data origins of generated parts. Thus, our approach is able to map the change (i.e., insertion, deletion, or replacement) made by the developer on the generated web page to the data origins in the web-page-generation code. In the checking part (which uses both static analysis and dynamic analysis), our approach further checks whether the change can be applied without further revisions of the code. If so, our approach recommends direct propagation of the change to the web-page-generation code. Otherwise, our approach highlights the need-to-change place in the code and provides the reason for not being able to perform the change directly. In this paper, we refer to this checking as *ne ec ed-im ac de ec i n*.

In the preceding approach to automating presentation changes in dynamic web applications, there are two main technical challenges to overcome. The first challenge is what dynamic information to record and how to use the collected information to achieve required precision for mapping the change from the generated web page to the code. To overcome this challenge, we propose dynamic string-origin analysis based on static string-taint analysis (which typically does not meet our requirement when handling user inputs, string operations, etc.) to deal with our dynamic information. The second challenge is how to check against unexpected impacts. To overcome this challenge, we propose unexpected-impact detection. We describe the details of how we overcome the two challenges in the next two subsections.

4.1 Dynamic String-Origin Analysis

As mentioned previously, our dynamic string-origin analysis is based on string-taint analysis [33, 30]. The process of string-taint analysis is as below. The analysis transforms the web-page-generation code to an extended Context Free Grammar (CFG) containing Finite State Transducers (FSTs). Then, the analysis propagates taints from the terminals to the nonterminals through productions and FSTs in the CFG, and determines the taints of the nonterminals.

However, static string-taint analysis may not be suitable for the target problem without adaptation. First, it may generate false positives due to its approximation for statically unknown elements (e.g., user input, possible numbers of loop iterations). Second, static string-taint analysis considers all possible executions and therefore can hardly achieve mappings between code elements and the exact positions in the generated HTML text.

Therefore, we propose *d namic ing- igin anal i* . One concern of dynamic analysis is the runtime overhead imposed by the instrumentation, but such concern is not a problem in our case: we could have two versions of the program, an instrumented one for performing the presentation change and the original one for normal execution and testing. The basic idea of dynamic string-origin analysis is to perform string-taint analysis on an execution trace instead of the source code. As dynamic string-origin analysis makes use of the recorded runtime information of the execution, there would be no approximation. Moreover, instead of boolean taints, our dynamic string-origin analysis uses a taint that records the source-

code-location information, so that the taints of a generated HTML text represent all of its data origins.

We next present how we record runtime information for our dynamic string-origin analysis (Section 4.1.1). We then present how we adapt string-taint analysis for our purpose (Section 4.1.2).

4.1.1 Rec ding f R n ime Inf ma i n

In the instrumented web application, the instrumented code records two kinds of information. The first kind of information is the executed statements in their execution order. This kind of information provides the basis for our dynamic string-origin analysis. The second kind of information is the values of expressions that existing techniques of string-taint analysis cannot handle precisely. Typically, those expressions include user inputs (e.g., "POST" and "GET" operations), the invocations of library functions (except string operations), array elements, and arithmetic or boolean expressions used as strings. For such an expression, existing techniques of string-taint analysis use the closure of the alphabet to estimate its value. For example, the statement `$x = $_Post['name']` is transformed to a production "X $\rightarrow \sigma$*". As we use our dynamic string-origin analysis on one execution trace, we are able to use the actual values of these expressions, and thus further reduce the imprecision of string-taint analysis.

We next illustrate how we record the runtime information with the following example code portion:

```
1  $i = 0;
2  $uid = str_replace("'", "&#39;", $_Post['id']);
3  $sql = "select * from utbl where uid = ".$uid;
4  $query = mysql_query($sql);
5  $num = mysql_num_rows($query);
6  while($i <$num) {
7    $result = mysql_fetch_array($query);
8    $str.= "<td>".$i." ".$result["title"]."</td>";
9    $i++;
10 }
11 echo $str;
12 echo "<tr>".$uid;
```

The execution trace of the code portion is as below, where the value of $_Post['id'] is "wxy", and there are two titles in the database for user "wxy": "T1" and "T2". For brevity, we present only the line number of a statement and the recorded values associated with the statement. For Line 2, we record the value of a user input. For Lines 4, 5, and 7, we record the return values of library functions, where "Resource id#1" is the handler of the SQL query, and "#Array1#" and "#Array2#" represent two arrays of values read from the database. For Line 8, we record the value of an array element, and for Line 1 or 9, we record the value of an arithmetic expression.

```
1:$i=0, 2:$_Post['id']="wxy", 3,
4:$query= Resource id#1, 5:$num = 2,
7:$result=#Array1#,
8:$result["title"]="T1", 9:$i=1,
7:$result=#Array2#,
8:$result["title"]="T2", 9:$i=2,
11, 12
```

4.1.2 Anal i i h R n ime Inf ma i n

When placing the executed statements in their execution order, we get a new program, which we refer to as the trace program in this paper. From the trace program, we construct an extended CFG in a way similar to the construction of an extended CFG in string-taint analysis. The only difference lies in that we use the recorded actual values for places where existing techniques of string-taint analysis use the closure of the alphabet for estimation. Note that the trace program contains neither branches nor loops. Thus, the imprecision caused by the control dependencies on user inputs has already been removed naturally.

For the code portion and the runtime information presented in Section 4.1.1, we construct the following extended CFG, in which, instead of using the alphabet closure to estimate the values of user inputs and unanalyzable variables (i.e., `$_Post['id']`, `$num`, `$result["title"]`, and `$i`), we use their actual values:

```
$i1 → 0
$Post_id → wxy
$uid → preg_replace(', &#39;, $Post_id)
$str1 → <td>.0. .T1.</td>
$str2 → <td>.1. .T2.</td>
$page → $str2.$expr
$expr → <tr>.
```

As our extended CFG is constructed from an execution trace, either a terminal or a nonterminal represents just one string. Therefore, we are able to use the taint of a terminal or a nonterminal to represent the origins of different parts in the string represented by the terminal or the nonterminal. Initially, we give each terminal a location flag as the taint. In particular, if the terminal represents a string constant, the location flag records the exact location of the string constant in the code. If the terminal represents a string value read from files, databases, the network, or the user input, the location flag records the location of the reading statement. If the terminal represents a non-string-type variable or constant concatenated with strings, the location flag records the location of the concatenation. The taint of a terminal also contains information to distinguish these different types of terminals. We give no taint to any nonterminal initially, and we propagate taints of the terminals to the nonterminals. The taint of a nonterminal is a list of index ranges and their corresponding location flags[3]. For example, a nonterminal whose value is "abcde15" may have the following taint: 1-3 (file1.php, Line 1, constant), 4-5 (file2.php, Line 2, database), 6-7 (file2.php, Line 5, number). The taint indicates that the origin of substring "abc" is the string constant in Line 1 of file1.php, the origin of substring "de" is from a database in Line 2 of file2.php, and the origin of the substring "15" is from a number in Line 5 of file2.php. As our taint of a terminal/nonterminal indicates the origins of its substring, we also refer to our taint as the $origin\ signature$ below.

To ensure that we have calculated the origin signatures of all the nonterminals at the right-hand side of a production before we calculate the origin signature of the nonterminal at the left-hand side of the production, we propagate the origin signatures in the same order as the execution order. To present how we propagate origin signatures in a production whose right-hand side is simple concatenation, let us consider a production in the form of "Y $\rightarrow X_1X_2X_3...X_n$", in which Y is the nonterminal at the left-hand side, and each X_i is a terminal or a nonterminal whose origin signature is $sig(X_i)$ and whose value is $value(X_i)$. The origin signature of Y after propagation is basically the concatenation of $sig(X_i)$ with adjustment of the index ranges. For example, if the length of $value(X_1)$ is $length(X_1)$, we need to increase the indices in $sig(X_2)$ by $length(X_1)$ to form the corresponding part of indices in the origin signature of Y. As each nonterminal in our CFG represents just one string, our propagation of the origin signature is able to calculate the exact origin of any substring of the string represented by the nonterminal. If the right-hand side of a production is not just concatenation of nonterminals and/or terminals, but is a more complex string operation (such as $str\_replace$ and $trim$), there is a Finite State Transducer (FST) in the extended CFG to simulate the string operation. In existing techniques of string-taint analysis [32, 30], an FST can read a string str with its origin signature $sig(st)$ and output a string $str1$ and its origin signature $sig(str1)$. If we use exactly the same FSTs used in existing techniques of string-taint analysis in our dynamic string-origin analysis, the FSTs can guarantee that (1) $str1$ is the same as the

[3]Note that our previous research [30, 31] uses location flags but not index ranges as taints.

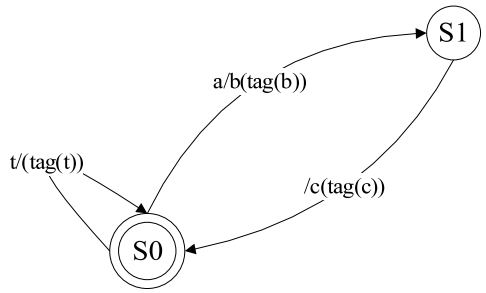

Figure 1: An example of location-FST (t=Σ/a)

output of the string operation with input str, and (2) $sig(str1)$ indicates the origin of each substring in $str1$ that comes from str. However, if the output of a string operation has substrings coming from more than one parameter of the string operation, an FST used in existing techniques of string-taint analysis can calculate only a partial origin signature. To overcome this limitation, we extend the existing FSTs by adding origin signatures to output strings on edges of the FSTs. As any real argument of a string operation with multiple parameters can be a string variable, one origin signature added on one edge of the FSTs may contain multiple origins. Note that, as FSTs can simulate string operations defined in library code, using FSTs naturally avoids instrumenting library code.

To illustrate our adaptation on FSTs, we present an example in Figure 1, which shows an adapted FST for `str_replace('a', $y, $x)`, where the value of $y is "bc". In the figure, we use "tag(ch)" to represent the location flag of a character "ch". Consider the case where the value of $x is "abcda", the location flag of "abc" in $x is tag1, the location flag of "da" in $x is tag2, the location flags of the "b" and "c" in $y are tag3 and tag4, respectively. The output of the FST would be "b(tag3)c(tag4)b(tag1)c(tag1)d(tag2) b(tag3)c(tag4)", in which the tag after each character indicates the location flag of the character. In contrast, using the standard FST, the output would be "bcb(tag1)c(tag1)d(tag2)bc", and the location information of the characters from the argument $y would get lost.

After taint propagation, the origin signature of the start nonterminal (i.e., S) describes the origins of all the substrings in the generated web page. Using this origin signature, when the developer makes a change in the generated web page, our approach is able to map the change to its data origins in the web-page-generation code.

4.2 Unexpected-Impact Detection

As mentioned previously, propagating a presentation change from a web page back to the web-page-generation code may have unexpected impacts on other places in the generated web page and/or on other web pages that can be generated by the code. Specifically, we deem an impact of a code fix as expected, if the impact matches the user-specified presentation change in the generated web page. Other impacts of the code fix are deemed unexpected.

When the unexpected impact is on other places in the generated web page, we refer to it as inner-page impact. Existing bidirectionalization techniques have already considered this category of unexpected impacts, and refer to them as conflicts. In particular, conflict detection in bidirectionalization checks whether multiple elements in the output of the transformation are mapped to a single element in the input, and whether these output elements are changed inconsistently. Similar to existing bidirectionalization techniques, our approach checks whether two substrings in the generated web page (corresponding to the start variable in the CFG) sharing the same location flag change consistently or not. Note that, in nature, the checking of this category of unexpected impacts is based on dynamic analysis, which analyzes the CFG constructed from one execution trace.

When the unexpected impact is on other web pages, we refer to it as inter-page impact. Bidirectionalization techniques do not consider this category of unexpected impacts, because a transformation program in bidirectionalization always transforms one input to the same output. When a change propagation (denoted as C) has an inter-page impact, the changed code must have some impact to multiple web pages (denoted as $P-Set$). Among these pages, only one (denoted as p) is shown in the execution and contains the user-specified presentation change. In each page in $P - Set$ other than p, there must be some part (denoted as pa) that is data-dependent or control-dependent to C, and part of the code that generates pa is not included in the execution trace because pa is not generated. Therefore, the clue of inter-page impact should be the existence of some code that is data-dependent or control-dependent on the changed code, but not included in the execution trace.

Therefore, we detect inter-page impacts by first searching the execution trace for code elements (variables, expressions, and return values) that may contain the changed part as a substring, and then statically checking in the source code whether these code elements are used in branch predicates in the execution trace or are used in statements not in the execution trace. Specifically, to detect inter-page impacts, we use a static-analysis technique that consists of four steps. First, in the origin signature of the start nonterminal, we find the location flags corresponding to the changed part on the changed web page, and use these flags to form a flag set (denoted as *Flag*). Second, for each nonterminal in the CFG, if its origin signature contains any location flag in *Flag* , we put the nonterminal into a set called *RefNT*. Note that the origin signature of a nonterminal is the sequence of the location flags of its each substring. Third, we map each nonterminal in *RefNT* back to the code elements that the nonterminal represents. If a code element is represented by a nonterminal in *RefNT*, we put it into a set called *RefVa* . Fourth, we check whether there exists a reference of a code element in *RefVa* that (1) is in branch predicates recorded in the input execution trace or (2) is in any statements not recorded in the input execution trace. If so, we deem that we have detected an unexpected impact in the second category, because the change may affect some parts of the code that are not executed in the current execution.

When our approach detects an unexpected impact in either category, our approach highlights the unexpected impact and recommends to the developer that the presentation change cannot be directly performed to the source code.

4.3 Practical Issues

Beside the preceding two categories of unexpected impacts, our approach also considers two practical issues before performing a presentation change. First, the presentation change made in a generated web page may not always map to constant strings in the code. When a presentation change maps to strings read from files, databases, the network, or the user input, we would recommend the developer to do some manual refactoring to realize the presentation change, since the developer needs to adjust the processing logic to change the value of data read from outside the web-page-generation code. Second, if the made presentation change is an insertion and the inserted string is between two concatenated constant strings, the string can be inserted either to the right of the first string or to the left of the second string. In such a case, our approach would also ask the developer to choose one insertion point.

5. EMPIRICAL STUDY

To evaluate our approach, we implemented our approach for the PHP language and conducted an empirical study on our approach using three PHP projects as subjects.

Table 1: Subject web applications used in our study

Subject	#Dev.	Start	End	KLOC
SquirrelMail	10	Apr-15-2000	Dec-23-2001	8 to 26
WebCalendar	7	Jun-06-2000	Dec-11-2002	6 to 17
OrangeHRM	33	Mar-02-2006	Oct-27-2006	96 to 105

5.1 Research Questions

Our empirical study tries to answer the following two research questions.

- **RQ1**: How effective is our approach on locating the source code that generates the changed part of the HTML page?
- **RQ2**: How effective is our approach on detecting unexpected impacts and discovering practical issues?

The first research question is mainly concerned with the effectiveness of our technique for dynamic string-origin analysis. The second research question is mainly concerned with the effectiveness of our techniques to detect unexpected impacts and to deal with practical issues.

5.2 Study Design

We used three popular open source dynamic web applications as subjects in our empirical study: $SquirrelMail$, $WebCalendar$ and $OrangeHRM$. $SquirrelMail$ is one of the most popular web-based email clients, $WebCalendar$ is one of the most popular web-based calendars and memorandums, and $OrangeHRM$ is a web-based human-resource management systems. All of the three web applications are dynamic web applications written in PHP and their source code is accessible from SourceForge[4]. We choose these three applications because they are from different domains and their presentation styles are different from each other, so that our empirical results on them would probably be generalizable to different presentation styles. Furthermore, all three subjects have more than 2000 bug reports so that we have enough data sets to construct presentation changes for our empirical study.

To perform our study, we manually studied 200 bug reports for each web application. In particular, we chose the earliest 200 bug reports marked as fixed in the bug repository of each web application. The reason is that we need bug reports marked as fixed to compare the results of our approach with the actual results and most of early bug reports were already fixed no matter whether they were easy or difficult to fix. However, among recent bug reports, the bug reports marked as fixed may be more likely to be easy to fix and thus may not be representative. Table 1 depicts the detailed information of the three web applications.

In Table 1, Columns 3 and 4 present the submission date of the first fixed bug report and the submission date of the 200th fixed bug report in the form of "mmm-dd-yyyy", respectively. Column 5 presents the size of each web application in Kilo Lines Of Code (KLOC). Since the size of the source code would change from the start date to the end date, we give two numbers for the size of the source code on the start date and that on the end date, respectively. For example, "8 to 26" indicates that the size of the source code is 8 KLOC on the start date and 26 KLOC on the end date.

Among the 600 studied bug reports, we manually identified 43 bug reports[5] corresponding to presentation changes from the version histories and the bug repositories of the three subjects. Table 2 depicts the result of our manual investigation of the 600 bug reports. In Table 2, Columns 2 to 6 present the number of studied bug reports, the number of presentation-change bug reports, the percentage of presentation-change bug reports among all studied bug reports, the number of duplicate presentation-change bug reports, and the number of identified presentation-change tasks, respectively.

In Table 2, we have three main observations. First, in all of the web applications under study, there exist some bugs related to presentation changes. This fact indicates that most web applications may require presentation changes during their development and evolution. Second, the average percentage of presentation bugs among all bugs is 7.2%. This percentage number indicates that presentation-change bug reports are an important category of bug reports for web applications and presentation-change tasks are quite common in the early evolution history of web applications. Third, in $SquirrelMail$, presentation-change bug reports account for 3.5% of all the studied bug reports, while in $OrangeHRM$, presentation-change bug reports account for 11.0% of all the studied bug reports. This difference indicates that the frequency of presentation changes may be different for different web applications. A possible explanation is that the GUI structure of $SquirrelMail$ is more stable than that of $OrangeHRM$, because there is a defacto standard GUI structure for web-based email-client applications.

Furthermore, to understand whether some presentation-change tasks take developers non-trivial effort to perform, we studied the processing days of the presentation-change bug reports and other bug reports[6]. For each software project under study, Table 3 compares the processing days of the presentation-change bug reports and the processing days of all the 200 studied bug reports. In Table 3, Column 1 presents the software project under study. Columns 2 and 3 present the average processing days and the processing-day range for the presentation-change bug reports among the 200 studied bug reports of the software project, respectively. Columns 4 and 5 present the average processing days and the processing-day range for all the 200 studied bug reports of the software project, respectively. All the processing-day values are calculated as the difference between a bug report's closing date and its submission date. If a bug report is closed on the day it is submitted, we deem the processing days as zero day.

From Table 3, we have two main observations. First, the average processing-day values of presentation-change bug reports vary from 20.1 days to 59.3 days in the three software projects. This observation indicates that presentation-change tasks are generally not trivial and developers do take some time to perform them. Second, the average processing-day values of presentation-change bug reports are typically comparable to that of all bug reports: presentation-change bug reports take longer time in two subjects, and shorter time in one subject. This observation indicates that presentation-change tasks are not significantly simpler than other bug-fixing tasks. Third, the maximal processing days of all bug reports are much more (3-5 times) than those of the presentation-change bug reports. This result shows that the presentation-change bug reports are usually not the most time-consuming ones to process among all the bug reports, but require average processing days of all the bug reports.

Among the 200 studied bug reports for each subject, there are duplicate bug reports[7]. As the developers of the three web applications did not record duplication relationships between bug re-

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

[5] The information of the bug reports are available on the project web site: `http://research.csc.ncsu.edu/ase/projects/apc/`

[6] Note that the processing days of a bug report may not exactly reflect the difficulty of bug-fixing tasks for various reasons (e.g., developers' different schedules).

[7] In fact, automated detection of duplicate bug reports in bug repositories is a recent research focus [29].

Table 2: Result of manual investigation of the studied bug reports

Subject	Bug Reports Studied	Presentation-Change Reports	% of Presentation-Change Reports	Duplicate Presentation-Change reports	Presentation-Change Tasks
SquirrelMail	200	7	3.5%	1	6
WebCalendar	200	14	7.0%	2	12
OrangeHRM	200	22	11.0%	1	21
Total	600	43	7.2%	4	39

Table 3: The Processing Days of the Bug Reports

Subject	Presentation Change Bug Reports		All Bug Reports	
	Avg. Processing Days	Processing-Day Range	Avg. Processing Days	Processing-Day Range
SquirrelMail	59.3	0-248	38.8	0-645
WebCalendar	44.3	0-230	116.5	0-1119
OrangeHRM	20.1	1-51	18.4	0-260

ports, we manually picked out the duplicate bug reports from the 43 presentation-change bug reports. As shown in Table 2, we identified 4 duplicate presentation-change bug reports in total. Since it is difficult for us to remove all duplicate bug reports in the 600 studied bug reports without detailed analysis of each bug report, we used all the bug reports (including the 4 duplicate presentation-change bug reports) when calculating the numbers for the percentage.

For each of the 39 unique presentation-change bug reports, we built a presentation-change task according to the following procedure. First, we checked the description of the presentation-change bug report in the bug-tracking system and the comments of the code commits. From this information, we figured out which code commit contains code changes related to the presentation-change bug report. Second, according to the description in the presentation-change bug report, we used our approach to instrument the web application and executed the instrumented web application to generate the need-to-change web page. Third, we identified the required change in the generated web page and used our approach to map the required change to the web-page-generation code. Finally, we checked the code changes in the corresponding code commit to discover how the developers changed the code to realize the presentation change. We used these code changes as the ground truth of the corresponding presentation-change task.

For each of the 39 tasks, if our approach can directly propagate the change in the generated web page to the source code, we checked whether the change suggested by our approach matches the actual change; otherwise, we checked whether our approach can correctly detect the unexpected impacts and practical issues by checking with developers' actual changes. To check the correctness of a mapping, we compared the places suggested by our approach with the places changed by the developers. In the case of unexpected impacts and mapping to places other than constant strings, the fix made by the developers is typically more complex. In such case, we checked whether the statements that contain the places suggested by our approach are involved in the fix made by the developers. In the case of the insertion issue, we checked whether the two suggested places match the actual change in the fix made by the developers. If one suggested place matches it, we deem that our approach correctly locates the need-to-change place.

5.3 Overall Results

Table 4 depicts the overall results of our approach on preforming the 39 presentation-change tasks in the three web applications. For each subject, Columns 2 to 9 in Table 4 present the number of presentation-change tasks under study, the number of tasks for which our approach correctly maps the changed substrings in the web page to their corresponding places in the source code, the number of tasks for which our approach directly propagates the change to the source code in a way matching the actual change, the number of tasks for which our approach directly propagates the change to the source code without matching the actual change, the number of tasks for which our approach does not recommend direct propagation due to inner-page impact, the number of tasks for which our approach does not recommend direct propagation due to inter-page impact, the number of tasks for which our approach does not recommend direct propagation due to an insertion issue, and the number of tasks for which our approach does not recommend direct propagation due to mapping the change to code elements other than constant strings, respectively.

From Table 4, we have four main observations. First, for all of the 39 changing tasks, our approach is able to correctly map the presentation change to the corresponding places in the source code. This observation indicates that dynamic string-origin analysis on the execution trace is very precise in providing developers with places to modify for realizing presentation changes.

Second, for 20 out of the 39 tasks, our approach is able to directly propagate the presentation change to the source code correctly. This observation indicates that developers can perform more than half of the presentation changes in dynamic web applications as in static web pages with the help of our approach.

Third, there are 3 tasks for each of which our approach correctly locates the corresponding source code and recommends direct propagation, but the actual change is not a direct change. In each of the 3 tasks, the developers actually try to make a static part in the generated web page dynamic. For example, in Bug Report No. 1510677 of the $OrangeHRM$ subject, the users complained that the feedback information of an operation is always in red no matter whether the operation fails or not, and demanded that the information should be in green when the operation succeeds. Our approach is able to locate the corresponding constant string about the color in the source code and confirm that changing the substring from "#FF0000" (red) to "#005500" (green) would not have unexpected impacts. However, in the actual fix, the developers add some condition checking instead of directly changing the substring "#FF0000". In such a case, there is no wonder that our approach cannot produce the correct fix. The reason is that the intention of the change is not to always have the feedback information in red. Since the developers know the intention of the change, they should know that the direct propagation is incorrect beforehand.

Finally, for each of the other 16 tasks, our approach is able to correctly decide that the presentation change in the generated web page cannot be directly propagated to the source code. Among these 16 tasks, 3 tasks are due to unexpected impacts on other possible generated web pages, 6 tasks are due to ambiguity of insertion, 6 tasks are due to mapping the presentation change in the generated web page to code elements other than constant strings. This observation indicates that the proposed techniques to detect unexpected impacts and to address the practical issues are helpful to prevent our approach from producing incorrect changes.

Table 4: Overall results of our approach

Subject	Tasks	Correct Map	Correct Direct Propagation	Incorrect Direct Propagation	Inners-Page Impact	Inter-Page Impact	Issue of Insertion	Issue of None-Constant Strings
SquirrelMail	6	6	2	0	0	0	2	2
WebCalendar	12	12	7	2	1	1	1	0
OrangeHRM	21	21	11	1	0	2	3	4
Total	39	39	20	3	1	3	6	6

5.4 Example Tasks

In this subsection, we use two example tasks to further illustrate how our approach works in different situations.

Example 1. The first example is about bug report No. 601006 of $WebCalendar$, which takes the developers 13 days to process. The bug is for the version on Aug. 8th, 2000. The summary of the bug report is "Rejected e-mail link missing a quote". The description of the bug report states that a quotation mark is missing for the email links in the rejected list. This missing quotation mark causes the page to be displayed incorrectly. Therefore, the presentation-change task is to add the quotation mark to a proper place in the source code. The erroneous HTML fragment is as below. The underlined part of the HTML should be "\">".

```
...
1 <BR><STRIKE><A HREF="mailto:mymail@gmail.com?
2 subject=WebCalendar:mycal\>
3 Xiao</a></STRIKE>Rejected";
...
```

The source code that generates the preceding HTML fragment is as below. Our approach is able to locate the underlined constant string in the code and recommend a direct insertion of a quotation mark between the two characters of the constant string. It is not easy to manually locate the fixing place for this bug because the whole HTML tag is broken into concatenated fragments and string variables in the PHP source code, while our approach is able to locate the place and fix it automatically. Note that text search can hardly help much in this case, because the strings immediately before and after the underlined ">" are from the database and do not appear in the PHP file at all. Therefore, using text search, the developers would find nothing by searching for the texts around the underlined ">". If searching for the string "subject" in the PHP file, there would be 77 places for further inspection.

```
  ...
4 echo "<BR><STRIKE><A HREF=\"mailto:" . $tempemail .
5   "?subject=$subject\>" . $tempfullname .
6   "</a></STRIKE> (" . translate("Rejected") . ")\ n";
  ...
```

Example 2. The second example is about bug report No. 1530219 of $OrangeHRM$, which takes the developers 14 days to process. This bug is for the version on Jul. 27th, 2006. The summary of the bug report is "PIM - Emergency Contact - There has to be a space after fields". The bug report states that the developers should insert a space after the fields in the page about the emergency-contact information of employees. The need-to-change HTML fragment is as below. The underlined space is what should be added.

```
  ...
1 <tr>
2 <td>Home Telephone__</td>
  ...
3 <td><strong>Home Telephone</strong></td>
  ...
```

The source code that generates the preceding HTML fragment is as below. Our approach is able to locate "</td>" in Line 5 as the right side of the insertion, and "Home Telephone" in Line 4 as the left side of the insertion. Furthermore, our approach is able to identify that changing "Home Telephone" would have inner-page impact because of another appearance of "Home Telephone" generated by Line 6. With the preceding information, a developer should be able to choose the correct insertion point without further inspection of the code.

```
  ...
4 $hmtele = "Home Telephone";
  ...
5 <td><?=$hmtele?>(insertion)</td>
  ...
6 <td><strong><?=$hmtele?></strong></td>
  ...
```

Using text search, a developer can easily locate the appearance of "Home Telephone" in Line 4, but adding a space after "Home Telephone" would be an incorrect change of the code. As a result, the developer may notice this incorrectness only after some testing, which may be performed by a tester sometime later. Furthermore, even after the developer knows that the space is added to a wrong place, the developer still needs to inspect the code to find the correct place to add the space.

5.5 Threats to Validity

In our study, we applied our approach on 39 presentation-change tasks for three dynamic web applications. This factor may be a threat to the external validity, since it is possible that our empirical results may be specific to the used tasks and web applications, and thus may not be generalizable. To reduce this threat, we used web applications from different domains as subjects. The main threat to construct validity is the way we construct our presentation-change tasks. Since the tasks are recovered from bug reports and version histories, the scenarios in the constructed tasks may not be exactly the same as the scenarios in real-world development. To reduce this threat, we carefully studied the bug reports and the version histories of each subject, and examined the changed code to confirm that we correctly reproduced the need-to-change web pages and identified the required changes. The main threat to the internal validity is the possible faults in implementing our approach. To reduce this threat, we carefully reviewed the source code of our implementation before conducting our empirical study. Another threat to the internal validity is that the developers themselves may make mistakes so that the bug fixes that we used as the ground truth may not all be correct. To reduce this threat, we use the earliest bug reports in the software-development history, so that the developers should have enough time to correct their fixes if they made mistakes.

6. DISCUSSION

Limitations of Our Approach. Our approach has three main limitations. First, our approach is more suitable for being applied to small atomic presentation changes such as changing a certain property of a certain GUI component. For pervasive-style changes and large structure changes, the developers can still trace the whole need-to-change part in the generated web page to the data origins in the web-page-generation code. However, in such a case, the located data origins may include user inputs or computation results, or the data origins may affect a large number of other web pages so that intensive human invention may be required.

Second, since our approach does not record or analyze the execution of JavaScript at the browser side, we cannot handle the presentation changes in the web-page parts generated dynamically at the browser side using AJAX or similar technologies. In such a case, the code that generates the web page is itself dynamically generated, and thus more difficult to instrument and analyze.

Third, although our approach includes a series of techniques to check whether the intended presentation change can be directly propagated to the web-page-generation code, we still cannot guarantee that a directly propagated presentation change is desirable. Our empirical results confirm the existence of such undesirable propagated presentation changes. However, our empirical results also show that undesirable cases are rare and developers' intention for such a presentation change is typically different from ordinary presentation changes. Further investigation may be needed to study the condition for propagating a desirable presentation change.

Using CFGs in Dynamic Analysis. Note that although Context-Free Grammars (CFGs) are essential in static string-taint analysis, it is not necessary for us to use CFGs in our dynamic string-origin analysis. In fact, due to its dynamic nature, the CFG constructed by our dynamic string-origin analysis is a simple regular grammar. Therefore, it is feasible to directly process the execution information without involving CFGs[8]. However, transforming the execution information to a CFG makes it possible for us to take advantage of existing tools for static string-taint analysis, which provides a convenient way to deal with string operations provided by library methods without instrumenting the library code.

Dealing with CSS Properties. CSS (Cascading Style Sheet) has been widely used in web-application development and web-page design. By using CSS, developers can define a format in an external format file (i.e., .css file), and link HTML tags to their formats. For example, a developer can define a format "f1", which indicates the color of the text to be red. Then, she can use "f1" in the "class" property of HTML tags to link the tag with the format. For example, the developer can write the HTML text " abc ", and then "abc" is shown in red. The advantage of using CSS is that it can help developers perform format-related presentation changes just in the CSS file when they want to switch all the tags linked to a certain format to another format. However, there are still quite many presentation changes on dynamically generated HTML pages that are not about format switching. Actually, both examples depicted in this paper are about display errors rather than format switching. Furthermore, it is also possible for developers to change only one tag linked with the format (e.g., changing "abc" to be in black, but keeping other texts with the "f1" tag to be still in red). In these cases, the change should be made in HTML texts instead of CSS files, and our approach is able to handle these cases well. Actually, two of the three subject applications (i.e., $Squirrel$ and $OrangeHRM$) use CSS properties in their presentation, but the presentation changes required in these two applications are similar to those in $WebCalendar$.

Dealing with Non-Presentation Changes. Our approach may also be helpful for some non-presentation-change tasks, especially those user-visible non-presentation changes. For a user-visible non-presentation change, our approach is able to provide the starting locations for the developers to further investigate.

Pervasiveness of Presentation Changes. Our empirical results demonstrate that, in the studied web applications, more than 7% of bug reports among the 600 studied bug reports are about presentation changes. Actually, there are also presentation-change tasks triggered by the design decisions of developers (especially before the release of the first version) but not related to bug reports. That is to say, presentation changes are a common type of software changes in the early evolution history of web applications. Indeed, in the later evolution history of a web application, its presentation structures may become stable and the number of presentation-change tasks may decrease accordingly.

[8]Actually, the simplicity of only one execution gives us many alternatives for implementing our dynamic string-origin analysis.

Due to the difficulty of constructing tasks for presentation changes triggered by design decisions, our empirical study uses presentation-change tasks constructed only from bug reports. However, we expect that our approach should still be applicable for such a presentation change because the developers should be able to generate the need-to-change web page and the desired change to the generated web page for such a presentation-change task.

Strength of Collaborative Hybrid Analysis. In our approach, the basic idea of *c llab a i e h b id anal i* plays an important role. Based on this idea, we can use one execution to perform precise analysis without considering other possible executions. On the basis of one execution, it is also natural for developers to provide the desired presentation change, because only with a generated web page can developers know what the presentation change is. However, changing the source code based on only one execution may be too risky. The second part of our collaborative hybrid analysis uses both dynamic and static analyses to reduce the risk. Therefore, we design the unexpected-impact detection in a conservative way.

Therefore, we expect that the paradigm of *c llab a i e h b id anal i* might provide a useful framework to automate various kinds of code changes. First, when looking for the right place to realize a specific code change, dynamic analysis can be a means more precise and/or more reliable than static analysis, because dynamic analysis is able to use the information specific to one execution to avoid interferences between different executions. Second, as it may be risky to perform a code change based on dynamic analysis, static analysis should be indispensable to guard the code change. Indeed, different combinations of dynamic and static analyses may be needed for different code changes.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach based on collaborative hybrid analysis to automating presentation changes in dynamic web applications. In particular, we use dynamic analysis to propagate the change on the generated HTML text to the source code, and use both static analysis and dynamic analysis to ensure the safety of such propagation. We carried out an empirical study on three widely used open source PHP web applications. We identified and constructed 39 presentation-change tasks from their bug reports. Our empirical results demonstrate that our approach is able to correctly locate the place to modify to realize the presentation change in each presentation-change task and correctly propagate the presentation change to the source code in more than half of the tasks. For most of the remaining tasks, our approach is able to correctly detect unexpected impacts.

There are three main ways to further improve and extend our approach in future work. First, as there are several threats to validity in our empirical study, we plan to further reduce these threats in future work. Specifically, we plan to apply our approach on more presentation-change tasks for more dynamic web applications. Furthermore, our evaluation involves a relatively small number of cases related to unexpected-impact detection. We plan to evaluate our approach on more presentation-change tasks involving unexpected-impact detection, so that we can have a better evaluation on this part of our approach. Second, our approach cannot guarantee that a presentation change is desirable. In future work, we plan to investigate new techniques for this purpose, such as considering multiple executions. Third, we currently evaluate our approach with the development histories. We plan to carry out a user study on groups of developers to evaluate our approach on the practice of web-application development and maintenance.

Acknowledgments. The authors from Peking University are sponsored by the National Basic Research Program of China (973)

No. 2009CB320703, the Science Fund for Creative Research Groups of China No. 61121063, and the Natural Science Foundation of China No. 91118004. Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNF-0958235, ARO grant W911NF-08-1-0443, and an NSA Science of Security Lablet Grant.

8. REFERENCES

[1] R. Arnold and S. Bohner. *S f a e Change Im ac Anal i* . IEEE Computer Society Press, 1996.

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, Practical fault localization for dynamic web applications In *P c. ICSE*, pages 265–274, 2010.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *P c. ISSTA*, pages 261–272, 2008.

[4] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *P c. ESEC/FSE*, pages 81–90, 2009.

[5] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *P c. FSE*, pages 237–246, 2010.

[6] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *P c. SAS*, pages 1–18, 2003.

[7] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *P c. ASE*, pages 550–554, 2009.

[8] B. Dufour, B. G. Ryder, and S. Gary. Blended analysis for performance understanding of framework-based applications. In *P c. ISSTA*, pages 118–128, 2007.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *P c. POPL*, pages 233–246, 2005.

[10] W. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *P c. FSE*, pages 181–191, 2008.

[11] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *P c. ICFP*, pages 205–216, 2010.

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

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM T an . n P g. Lang. and S* ., 12(1):26–60, 1990.

[14] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Highe -O de and S mb lic C m a i n*, 21(1-2):89–118, June 2008.

[15] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, January 1976.

[16] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *P c. ICSE*, pages 301–310, 2008.

[17] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *P c. ICSE*, pages 308–318, 2003.

[18] B. Livshits and E. Kiciman. Doloto: Code splitting for network-bound web 2.0 applications. In *P c. FSE*, pages 350–360, 2008.

[19] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *P c. ICSE*, pages 210–220, 2009.

[20] Y. Minamide. Static approximation of dynamically generated web pages. In *P c. WWW*, pages 432–441, 2005.

[21] H. Nguyen, H. Nguyen, T. Nguyen, and T. Nguyen Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *P c. ASE*, pages 13–22, 2011.

[22] T. Nguyen, G. Guarnieri, E. Shirley, and D. Evans Automatically hardening web applications using precise tainting. In *P c. IFIP Sec i C nfe ence*, pages 295–308, 2005.

[23] J. W. Nimmer and M. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *P c. RV*, pages 255–276, 2001.

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

[25] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: A tool for change impact analysis of Java programs. In *P c. OOPSLA*, pages 432–448, 2004.

[26] H. Samirni, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *P c. ICSE*, pages 277–287, 2012.

[27] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *P c. POPL*, pages 372–382, 2006.

[28] T. Tateishi, M. Pistoia, and O. Tripp: Path- and index-sensitive string analysis based on monadic second-order logic. In *P c. ISSTA*, pages, 166–176, 2011.

[29] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information In *P c. ICSE*, pages 461–470, 2008.

[30] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings for software internationalization. In *P c. ICSE*, pages 353–363, 2009.

[31] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In *P c. FSE*, pages 87–96, 2010.

[32] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *P c. PLDI*, pages 32–41, 2007.

[33] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *P c. ICSE*, pages 171–180, 2008.

[34] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *P c. ISSTA*, pages 61–72, 2010.

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

[36] M. Weiser. Program slicing. In *P c. ICSE*, pages 439–449, 1981.

[37] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *P c. ASE*, pages 164–173, 2007.

[38] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *P c. TACAS*, pages 154–157, 2010.