Locating Need-to-Translate Constant Strings in Web Applications

Xiaoyin Wang[1,2], Lu Zhang[1,2]*, Tao Xie[3]*, Hong Mei[1,2], Jiasu Sun[1,2]
[1]Institute of Software, School of Electronics Engineering and Computer Science
[2]Key Laboratory of High Confidence Software Technologies, Ministry of Education,
Peking University, Beijing, 100871, China
{wangxy06, zhanglu, meih, sjs}@sei.pku.edu.cn
[3]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
xie@csc.ncsu.edu

ABSTRACT

Software internationalization aims to make software accessible and usable by users all over the world. For a Java application that does not consider internationalization at the beginning of its development stage, our previous work proposed an approach to locating need-to-translate constant strings in the Java code. However, when being applied on web applications, it can identify only constant strings that may go to the generated HTML texts, but cannot further distinguish constant strings visible at the browser side (need-to-translate) from other constant strings (not need-to-translate). In this paper, to address significant challenges in internationalizing web applications, we propose a novel approach to locating need-to-translate constant strings in web applications. Among those constant strings that may go to the generated HTML texts, our approach further distinguishes strings visible at the browser side from non-visible strings via a novel technique called flag propagation. We evaluated our approach on three real-world open source PHP-based web applications (in total near 17 KLOC): Squirrel Mail, Lime Survey, and Mrbs. The empirical results demonstrate that our approach accurately distinguishes visible strings from non-visible strings among all the constant strings that may go to the generated HTML texts, and is effective for locating need-to-translate constant strings in web applications.

Categories and Subject Descriptors

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

General Terms

Management, Standardization

Keywords

Software internationalization, Flag propagation, Web application

1. INTRODUCTION

Software applications usually require different local versions for users from different regions of the world. Researchers and practitioners usually refer to techniques for acquiring and managing local versions of software as software internationalization and localization. A typical way to perform software internationalization and localization is to put language-specific elements (such as user-visible constant strings and number/date formats, etc.) to property files (i.e., internationalization), and translate the property files without changing the code (i.e., localization). For various reasons, developers may not internationalize a software application at the very beginning of the development process, but have to perform internationalization on an existing version of the software application [17]. Therefore, the developers need to locate all the hard-coded language-specific elements and externalize them to property files. When locating these language-specific elements, the most time-consuming task is to locate need-to-translate constant strings due to their large number and scattered distribution in the code.

In our previous work [17], we proposed an approach to locating need-to-translate constant strings in Java applications. Our previous approach includes three major steps. First, we locate all of the library-method invocations that may output a string-type argument to the Graphical User Interface (GUI). Then, for each string-type argument that is output to the GUI, we use string analysis [2] to obtain a context-free grammar (CFG), which is built from the Static Single Assignment (SSA) [3] form of the code. The start non-terminal of the CFG is the string-type argument, and the language of the CFG is all the possible values of the string-type argument. Finally, we add a unique annotation to each terminal (constant string) in the CFG to indicate its exact location in the code, and propagate the annotations in the CFG based on string-taint analysis [18] to get the locations of all the constant strings whose values may be passed to the string-type argument. Therefore, our previous approach is able to locate all the constant strings whose values may go to the GUI.

In recent years, web applications are increasingly becoming popular due to the convenience of Internet access. Challenges in the development of web applications are attracting more and more attention from various researchers [13, 7, 1]. Web applications especially require internationalization partly because users from all over the world can access them after they are put online. When applying our previous approach on web applications, we face significant challenges. For a web application, each dynamically generated web page is actually a text in the HTML format displayed at the browser side. Thus, our previous approach is able to locate the constant strings that may go to the browser side for display. However, not all parts of the HTML text sent to the browser side are visible to users. In fact, strings output to the GUI in traditional applications (such as Java GUI applications) are usually plain texts and are directly displayed on the GUI; but strings output to the browser side in a web application may contain HTML tags. According to the syntax of HTML, only some parts of an HTML text are displayed on the

*Corresponding author

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA

screen. The remaining parts of the HTML text are tags used for format description and element identification, etc. So, an approach for web applications needs to further distinguish those user-visible constant strings from other constant strings in each web page.

Typically, there are two types of user-visible strings in an HTML text: strings displayed as texts visible on web pages and strings displayed as labels or default values of input fields on web pages. For static web pages, it is easy to identify the user-visible constant strings with an HTML parser. However, it is challenging to identify the user-visible constant strings for dynamic web pages. For a dynamic web page, user-visible strings appear as constant strings in the code for generating the dynamic page. Therefore, it is infeasible to identify them in the code with an HTML parser. Furthermore, in a typical web application, the set of possible HTML texts generated by the application can be infinite. Thus, it is also infeasible to enumerate all the possible HTML texts generated by the application and check them with an HTML parser.

In this paper, to address the significant challenges in internationalizing web applications, we propose a new approach to locating need-to-translate strings in web applications. Our new approach currently works for only PHP applications, but the idea is general and it is relatively easy to adapt our approach for other types of web applications. Our new approach significantly extends and improves our previous approach [17]. Given a dynamic web page, we use our previous approach to locate constant strings that may go to the browser side for display. Among these constant strings, we propose two novel techniques to identify the preceding two types of user-visible constant strings. The basic idea of the two techniques is as follows. We add location information (such as whether the string is inside a tag) to each terminal (constant string) and each non-terminal. The location information of some constant strings can be determined based on their values. We further propagate the location information to all the other terminals and non-terminals, and use the propagated location information to identify the preceding two types of user-visible constant strings. In this paper, we refer to the added location information as flags and propagation of location information as flag propagation.

This paper makes the following main contributions:

- A demonstration of significant challenges in automatically locating need-to-translate constant strings in web applications;
- Novel algorithms for flag propagation to automatically determine whether a constant string in the code can appear in user-visible parts of the generated HTML texts;
- An empirical evaluation of applying our approach on three real-world open source web applications (in total near 17 KLOC) to evaluate the effectiveness of our approach.

We organize the rest of this paper as follows. Section 2 presents our motivation using some sample PHP code. Section 3 introduces some background knowledge. Section 4 presents our approach in detail. Section 5 presents the implementation of our approach. Section 6 reports an empirical evaluation of our approach. Section 7 discusses related issues. Section 8 discusses related work. Section 9 concludes the paper with future work.

2. MOTIVATION

In this section, we present four code portions in three real-world web applications to motivate our approach. The first code portion is from "question.php" of Lime Survey version 0.97[1].

```
...
echo "and pressed 'refresh' on your browser.
    In this case, your responses have<br />\n";
echo "already been saved.";
echo "</font></center><br /><br />";
...
```

In the preceding code portion, we can see that, to determine that the constant string "already been saved." can be displayed on the screen at the browser side, we need to consider its context. In fact, we can decide that this string is user-visible at the browser side because the strings before it ("<br />") and after it ("</font>") indicate that it is not inside an HTML tag. That is to say, our approach needs to check the strings, which may be either constants or variables, before and after each constant string to determine its context.

The second code portion is from "day.php3" of Mrbs version 0.6[2]. As described below, it demonstrates some constant strings that can be determined as not user-visible only through their context.

```
...
if ($t == $timetohighlight) {
    $c = "red";
} else {
    $c = "white";
}
...
echo "<td bgcolor=$c>";
...
```

In the preceding code portion, we can see that the constant strings "red" and "white" appearing in the code portion are not user-visible at the browser side because they are inside a tag corresponding to a table column, quoted by "<td" and ">", and they indicate the background color of the table column.

Furthermore, some constant strings that are output to the tag part of the generated HTML texts are still user-visible at the browser side. Such constant strings are the value attributes of some input tags. The following code portion is from "compose.php" of Squirrel Mail version 0.2.1[3].

```
...
if (substr(strtolower($reply_subj), 0, 3) != "re:")
    $reply_subj = "Re: $reply_subj";
echo "         <INPUT TYPE=TEXT NAME=passed_subject
        SIZE=60 VALUE=$̈reply_subj¨>";
...
```

In the preceding code portion, we can see that the italicized constant string (i.e., "Re: ") becomes the value attribute of an input tag. According to the HTML syntax, this constant string is displayed as the default value of an input text field and is thus user-visible. However, not all the value attributes in all the input tags are user-visible at the browser side. According to the HTML syntax, there are 10 types of input tags. Only when the type of an input tag is "text", "button", "submit", or "reset", the value attribute of the input tag is user-visible at the browser side. For example, the italicized text "Y" in the following code portion (from "qanda.php" of Lime Survey version 0.97) is not user-visible at the browser side, because the type of the input tag is "checkbox".

```
...
$answer .= "\t\t\t\t\t\t\t\t\t<input class='checkbox'
type='checkbox' name='$myfname' value='Y'";
if ($_SESSION[$myfname] == "Y")
    $answer .= " checked";

$answer .= " onClick='checkconditions(this.value,
        this.name, this.type)' />$ansrow['answer']\n";
...
```

The preceding four code portions demonstrate that the challenges for accurately determining whether each constant string in the code is user-visible at the browser side lie in the need to know which part of the generated HTML texts each constant string eventually appears in. To address these challenges, we need to search in the code to obtain the context of each constant string, and use the context to determine whether the constant string is user-visible.

[1] http://sourceforge.net/projects/limesurvey/.

[2] http://sourceforge.net/projects/mrbs/. ".php3" is the old extension for PHP.

[3] http://sourceforge.net/projects/squirrelmail/.

3. BACKGROUND

3.1 String Analysis and String-Taint Analysis

For the ease of understanding our approach presented in Section 4, we next introduce string analysis and string-taint analysis.

The general idea of string analysis is as follows. First, the program under analysis is transformed to the Static Single Assignment (SSA) [3] form. Second, string assignments and operations that the string variable under analysis depends on are abstracted as an extended Context Free Grammar (CFG) with string operations (i.e., library string methods such as `String.subString(int,int)` in Java) on the right hand side of productions. Then these string operations are simulated with finite-state transducers (FSTs) [15]. Finally, the language of the generated CFG represents all the possible values of the string variable under analysis.

String-taint analysis is a technique proposed by Wessermann and Su [18] for tracing possible malicious user inputs to check cross-site scripting or SQL injection. String-taint analysis is based on the CFG generated by string analysis. Initially, the terminals corresponding to possible malicious user inputs are annotated with a taint. Then, for each production in the CFG, if there is a terminal or a non-terminal with a taint on its right hand side, the non-terminal on the left hand side is also annotated with the taint. Finally, vulnerabilities are checked on certain non-terminals with taints.

Our previous work [17] adapts string-taint analysis by adding exact locations in the code as annotations to the constant strings (terminals) that are involved in string analysis and propagating these annotations in the generated CFG. During the propagation, the annotation of the non-terminal at the left hand side of each production becomes the union of the annotations of all the terminals and non-terminals at the right hand side and its current annotation. Finally, the annotation of the start non-terminal indicates the exact locations of all the constant strings whose values may be passed to the string variable under analysis (i.e., the start non-terminal in the CFG).

3.2 User-Visible Parts in HTML Texts

We next briefly summarize which parts of a standard HTML text are displayed to users at the browser side. There are two types of user-visible strings in an HTML text. Strings of the first type are those that are outside any HTML tags. These strings are displayed as ordinary texts in web pages. The first code portion in Section 2 is an example of strings of this type. Strings of the second type are those that serve as the value attributes of certain types of input tags[4]. These strings are displayed as labels or default values of input fields corresponding to input tags in the generated HTML texts. Such types of input tags include "button", "text", "submit", and "reset". However, the value attributes of the other types of input tags (e.g., "radio" and "checkbox") are not user-visible. For a checkbox, its value attribute actually indicates whether it is checked in the web page (i.e., "Y" for checked and "N" for unchecked).

4. APPROACH

In this section, we first formalize the target problem in Section 4.1, and then we present two new techniques to identify the two types of user-visible constant strings in Sections 4.2 and 4.3, respectively.

4.1 Problem

According to string analysis, we can represent a dynamic web page as a CFG denoted as a 4-tuple G=(N, T, S, P) (where N is the set of non-terminals, T is the set of terminals, S is the start non-terminal, and P is the set of productions). Non-terminals can be traced to string variables in the code, and terminals can be traced to constant strings in the code. Based on G, $L(G)$=$\{\omega | S \Rightarrow \omega\}$ (where $S \Rightarrow \omega$ denotes that S can deduce ω) is the set of all possible HTML texts generated in the dynamic web page. The nature of deduction in CFGs ensures that any $\omega \in L(G)$ corresponds to at least a deduction tree. The leaf nodes of the deduction tree are terminals, and ω is the concatenation of the leaf nodes. In this paper, we say that S can deduce ω with t (denoted as $S \overset{t}{\Rightarrow} \omega$, where $t \in T$), if $S \Rightarrow \omega$ and t appears as a leaf node in at least one deduction tree of ω.

Thus, given a dynamic web page, the problem of determining constant strings that may go to the HTML texts generated in the dynamic web page can be represented as the calculation of the following set: S_1=$\{t | t \in T \land \exists \omega \in L(G)(S \overset{t}{\Rightarrow} \omega)\}$. Our previous approach [17] uses an adapted string-taint analysis to calculate this set. However, calculating S_1 is not sufficient for locating need-to-translate constant strings in a dynamic web page.

Given $\omega \in L(G)$ and $t \in T$, an HTML parser can determine whether t appears in a user-visible part of ω. In this paper, we denote that t appears in a user-visible part of ω as $Vis(\omega, t)$. Thus, the problem of locating need-to-translate constant strings in a dynamic web page can be represented as the calculation of the following set: S_2=$\{t | t \in T \land \exists \omega \in L(G)(S \overset{t}{\Rightarrow} \omega \land Vis(\omega, t))\}$. S_2 is a subset of S_1.

As it is infeasible to enumerate each $\omega \in L(G)$ to check whether $Vis(\omega, t)$ holds, we need to analyze the productions in G to identify user-visible constant strings in T. In our approach, we represent $Vis(\omega, t)$ as location characteristics of t, and check which terminals satisfy these location characteristics on the basis of the productions.

4.2 Locating Outside-Tag Constant Strings

If a constant string contains '<' or '>', we can use '<' or '>' to determine which parts of the constant string are outside tags. For example, we know that, in "<abc>def", "abc" appears inside a tag and "def" appears outside tags. However, if a constant string does not contain either '<' or '>', we need to resort to constant strings that may be concatenated to the constant string in the generated HTML texts. For example, if we know that "<abc>" may be concatenated to "def" to form a string like "<abc>def" in the generated HTML texts, we can determine that "def" is outside tags using the '>' in "<abc>". In fact, when two strings are concatenated together, the right-most point of the first string and the left-most point of the second string become the same point. Thus, if one such point is outside tags, the other point is also outside tags. As a result, if we already know some constant strings are outside or inside tags, we can also know whether other constant strings are outside or inside tags by iteratively checking neighboring strings.

Following this idea, this sub-section presents a technique for locating user-visible constant strings appearing outside tags, consisting of four phases: adding flags to variables[5], initializing flags of variables, propagating flags, and identifying outside-tag constant strings.

4.2.1 Adding Flags to Variables

For each variable in the CFG, we add two flags: the left flag and the right flag. The left flag represents whether the left-most point of the variable is inside an HTML tag and the right flag represents whether the right-most point of the variable is inside an HTML tag.

[4] Attributes of other tags (e.g., the alt attribute of the image tag) can also be user-visible. The situation is similar to the value attribute of the input tag so that they can be addressed by the same approach.

[5] Here we use variables to denote either terminals or non-terminals for the ease of presentation. Note that a variable in a CFG is different from a variable in its corresponding code.

4.2.2 Initializing Flags of Variables

We define four different values for flags of variables in the CFG:

- *unknown*: denoting that it is not known whether the left-most point or the right-most point is inside an HTML tag;
- *inside*: denoting that the left-most point or the right-most point is inside an HTML tag;
- *outside*: denoting that the left-most point or the right-most point is outside any HTML tag;
- *conflict*: denoting that the left-most point or the right-most point may be either inside or outside HTML tags.

We initialize the flags of each variable in the CFG as follows.

- If the variable is a non-terminal, set both its left flag and right flag as *unknown*.
- If the variable is a terminal and it contains neither '$<$' nor '$>$', set both its left flag and right flag as *unknown*.
- Otherwise, if the left-most '$>$' or '$<$' is actually a '$<$', set the left flag as *outside*; otherwise, set the left flag as *inside*; and if the right-most '$>$' or '$<$' is actually a '$<$', set the right flag as *inside*; otherwise, set the right flag as *outside*.

4.2.3 Propagating Flags

After initialization, we propagate the flag values between flags on the basis of productions. Here, the propagation is bidirectional and the propagation between flag A and flag B will result in setting both A and B with the same flag value, which we denote as $BiPropagate(A, B)$. We repeatedly do the preceding propagation until the propagation cannot change the value of any flag. The rules for calculating $BiPropagate(A, B)$ are as follows.

- $BiPropagate(A, B)$ = *unknown*, if values of both A and B are *unknown*.
- $BiPropagate(A, B)$ = *inside*, if values of both A and B are *inside*, or one value is *inside* and the other is *unknown*.
- $BiPropagate(A, B)$ = *outside*, if values of both A and B are *outside*, or one value is *outside* and the other is *unknown*.
- $BiPropagate(A, B)$ = *conflict*, if either the value of A or the value of B is *conflict*, or one value is *inside* and the other is *outside*.

Given a production, we propagate the flags of variables in the production with the following rules.

1. If two variables are neighboring at the right hand side of a production, we perform a propagation between the right flag of the first variable and the left flag of the second variable.
2. If a terminal contains neither '$<$' nor '$>$', we perform a propagation between the left flag and right flag of the terminal.
3. We perform a propagation between the right flag of the production's left-hand-side variable and the right flag of the last variable in the production's right hand side.
4. We perform a propagation between the left flag of the production's left-hand-side variable and the left flag of the first variable in the production's right hand side.

4.2.4 Identifying Outside-Tag Constant Strings

After we acquire the final values of the left flags and right flags of all the variables, we consider each terminal (constant string) that contains at least a '<' or a '>' and that has at least one of its two flags as *outside* or *conflict* to be a possible need-to-translate constant string. Note that, as such a constant string may contain tags, we need to further check which parts of the string need translation.

Table 1: Flag Initialization

Variable	left flag	right flag
"<input "	*outside*	*inside*
"Student ID"	*unknown*	*unknown*
"name="	*unknown*	*unknown*
"student"	*unknown*	*unknown*
">"	*inside*	*outside*
all non-terminals	*unknown*	*unknown*

A terminal can have its flag value as *conflict* due to one of the following three factors. First, the code has some bugs and may generate incomplete tags (e.g., `"<input id= </br>"`). Second, there exists some imprecision in string analysis. The reason lies in that string analysis tries to use a CFG to express the possible strings generated by the code, but the set of possible strings generated by the PHP code may go beyond the expressiveness power of CFGs (e.g., PHP code can generate $0^n1^n2^n$, which cannot be expressed exactly by any CFG). Thus string analysis actually approximates the possible strings generated by PHP code using a CFG that expresses the superset of the possible strings generated by the PHP code (e.g., using $0^*1^*2^*$ to approximate $0^n1^n2^n$), and may contain some strings that actually cannot be generated by the PHP code. Third, the constant string corresponding to the terminal may appear both inside tags and outside tags. The following code portion is an example of such a situation.

```
$abc = "dog";
echo <div id = $abc>$abc</div>
```

From the above code portion, as constant string "dog" may appear outside any tag, it should be a need-to-translate constant string. However, the constant string also appears inside an HTML tag. For such a case, the developer can refactor the PHP code by splitting the variable `$abc` into two variables; otherwise, translation of "dog" may impact the structure of the generated HTML. As our approach cannot differentiate which reason causes the *conflict* value currently, we consider all terminals (constant strings) with the *conflict* flag value as need-to-translate to reduce false negatives.

4.2.5 Illustration

To further illustrate the flag-propagation process, we use the following example CFG.

```
S -> A"<input "BCD                                    (1)
A -> "Student ID"                                     (2)
B -> "name="                                          (3)
C -> "student"                                        (4)
D -> ">"                                              (5)
```

In the example productions, non-terminals are represented by capital alphabets and terminals are surrounded by quotation marks. The PHP code corresponding to the CFG tries to generate a string of "Student ID" followed by an input tag. The initialization of the two flags of each variable is shown in Table 1.

Assume that we divide flag propagation into a series of iterations, and in each iteration, from the first production to the last production, we propagate flags for the production by applying the four rules (described in Section 4.2.3) one by one.

The first-iteration propagation goes as follows. First, we take the first production and perform a propagation with the first rule between the right flag of `A` and the left flag of `"<input "`. Since the left flag of `"<input "` is *outside*, the right flag of `A` becomes *outside*. Similarly, the left flag of `B` becomes *inside*. As there are no terminals without containing either '$<$' or '$>$', there is no propagation with the second rule for the first production in this iteration. The third rule changes nothing, because both the right flag of `D` and the right flag of `S` are *unknown*. The fourth rule also changes nothing, because both the left flag of `A` and the left flag of `S` are *unknown*. Second, we then turn to the second production. As the right flag of `A` is *outside* now, the right flag of `"Student ID"` becomes *outside* using the third rule. As we consider the second rule before the third rule, we cannot change the left flag of `"Student ID"` to *outside* in the first iteration. Third, the left flag of `"name="` becomes *inside* using the fourth rule, since the left flag of `B` is inside. As we consider the second rule before the fourth rule, we cannot change the right flag of `"name="` to *inside* in the first iteration. Fourth, we change nothing with the fourth production in the first iteration. Finally, the left flag of `D` becomes *inside*, and the right flag of `D` becomes *outside*.

After the first iteration, the CFG with flag values is as below. For simplicity, we use (O) to denote *outside*, (I) to denote *inside*, (U) to denote *unknown*, and (C) to denote *conflict*.

```
(U)S(U) -> (U)A(O)(O)"<input "(I)(I)B(U)(U)C(U)(I)D(O)  (1)
(U)A(O) -> (U)"Student ID"(O)                          (2)
(I)B(U) -> (I)"name="(U)                               (3)
(U)C(U) -> (U)"student"(U)                             (4)
(I)D(O) -> (I)">"(O)                                   (5)
```

Then we perform propagation for the second iteration, and the CFG with flag values is as follow. The flags in the bold font indicate the changes from the first iteration.

```
(U)S(O) -> (O)A(O)(O)"<input "(I)(I)B(I)(U)C(I)(I)D(O)  (1)
(O)A(O) -> (O)"Student ID"(O)                          (2)
(I)B(I) -> (I)"name="(I)                               (3)
(U)C(I) -> (U)"student"(I)                             (4)
(I)D(O) -> (I)">"(O)                                   (5)
```

Then after several iterations, the final CFG with flag values becomes as follow. The flags in the bold font indicate the changes from the second iteration.

```
(O)S(O) -> (O)A(O)(O)"<input "(I)(I)B(I)(I)C(I)(I)D(O)  (1)
(O)A(O) -> (O)"Student ID"(O)                          (2)
(I)B(I) -> (I)"name="(I)                               (3)
(I)C(I) -> (I)"student"(I)                             (4)
(I)D(O) -> (I)">"(O)                                   (5)
```

Finally, from the CFG with flag values, we can distinguish that the constant string "Student ID" in the second production appears outside tags, since both its left and right flags are *outside*.

4.3 Locating Need-to-Translate Constant Strings Inside Input Tags

To check whether a constant string is a need-to-translate constant string inside an input tag, we need to check whether and where the constant string may appear in an input tag. To achieve this purpose, we extend flag propagation described in Section 4.2. In particular, the technique for locating need-to-translate constant strings inside input tags consists of four phases. The first phase is to locate all the occurrences of "input" (in the CFG) that correspond to an input tag. The second phase is to identify the scope of each input tag. The third phase is to identify the value and the type attributes in the input-tag scope. The fourth phase is to determine whether a constant string as a value attribute is need-to-translate according to the content of its corresponding type attribute.

To simplify the situation, we make an assumption that all the HTML keywords (e.g., "input", "type", "value", and possible type values like "button" and "text") in the generated HTML text appears in the code as a whole. In other words, the keywords are not split into more than one constant string in the PHP code but there can be constant strings in which one or more keywords appear as substrings. Our algorithm described below works only under this assumption. So the algorithm cannot deal with the code below.

```
echo <inp
echo ut id="id1">
```

We expect that this assumption holds in most cases in real PHP code, because there is no need to use string variables to generate keywords and few programmers would write such PHP code, which is quite difficult to understand and exists with no good reasons.

4.3.1 Locating Input Tags

In this phase, we first collect a list of candidate input tags by collecting all the terminals that both have "input" as a substring and are decided as inside-tag in identification of outside-tag constant strings. Some of the collected terminals containing "input" may not correspond to an input tag in the generated HTML text, because "input" in an inside-tag terminal may serve as an attribute of an HTML tag (e.g., "input" in `<div name="input">`). According to the HTML syntax, only an "input" that appears at the beginning of a tag corresponds to an input tag. In other words, each such "input" has its nearest previous non-space character as '<'. So we then check whether the non-space character appearing nearest before "input" is a '<'.

In particular, we determine input tags as follows. If the terminal containing an "input" has a non-space character immediately before the "input" and the non-space character is a '<', then we know that the "input" corresponds to an input tag. If the terminal containing an "input" has a non-space character immediately before the "input" but the non-space character is not a '<', then we know that the "input" does not correspond to an input tag. If the terminal containing such an "input" has no non-space character immediately before the "input", we put the terminal into an undecided list and use the following flag propagation to further check terminals in this list.

Similar to the technique described in Section 4.2, we also give each variable in the CFG two flags: the left flag and the right flag. This time a flag has only two values: *unknown* and *before_input*. The value *before_input* represents that the corresponding point of the flag is before a string with "input" value but after any non-space character. Then, for each terminal in the undecided list, we set the value of its left flag as *before_input* and the value of its right flag as *unknown*. For each other terminal, we set the values of both flags as *unknown*. After that, we propagate the flag values in the CFG in a way similar to the flag propagation for locating outside-tag constant strings in Section 4.2.3 using the following propagation rules.

1. If two variables are neighboring at the right hand side of a production, we perform a propagation from the left flag of the second variable to the right flag of the first variable.
2. If a terminal contains only spaces, we perform a propagation from the right flag of the terminal to the left flag of the terminal.
3. We perform a propagation from the right flag of the production's left-hand-side variable to the right flag of the last variable in the production's right hand side.
4. We perform a propagation from the left flag of the first variable in the production's right hand side to the flag of the production's left-hand-side variable.

Here, the propagation is unidirectional and the direction is backward in possible HTML texts. Therefore, we denote the propagation from flag A to flag B as assigning $BackPropagate(A, B)$ to B without changing the value of A. We define $BackPropagate(A, B)$ as follows.

- $BackPropagate(A, B)$ = *unknown*, if values of both A and B are *unknown*.
- $BackPropagate(A, B)$ = *before_input*, if either the value of A or the value of B is *before_input*.

The preceding definition of $BackPropagate(A, B)$ ensures that a flag of value *unknown* cannot be propagated to a flag of value *before_input*. Furthermore, as we are looking for the previous instead of the following character of a given terminal, the preceding definition of $BackPropagate(A, B)$ ensures that we search in the correct direction.

Like the bidirectional flag propagation for locating outside-tag constant strings, the backward propagation ends when no flag value changes. Then, we check whether there is a terminal whose right flag has the value of *before_input* and whose left flag has the value of *unknown*. For such a terminal (denoted as t), if the last non-space character in t is '<', we know that it is possible that a '<' appears as the nearest non-space character before an "input" in the undecided list. Thus, we determine the "input" as an input tag. Otherwise, we filter out the "input". Note that the preceding backward flag propagation is for checking only one “input” in the undecided list, and we use the same way to check every other “input” in the undecided list one by one.

Furthermore, it is possible that an “input” has more than one possible nearest previous non-space character and not all of them are ‘<’. In such situations, the “input” has other usages than being a tag mark[6]. In these situations, our approach may produce some false positives. The empirical study in Section 6 shows that such false positives are of only a small number in practice.

4.3.2 Identifying Scopes of Input Tags

In the second phase, we find out all the terminals that may appear inside the scope of each input tag. We start from the input tags located in the first phase. For each terminal corresponding to an input tag, if the terminal contains a ‘>’ following the “input”, the scope of the input tag can be easily determined. Otherwise, if the terminal does not contain such a following ‘>’, we perform flag propagation to identify all the terminals that are possibly in the scope of the input tag.

Here the flag propagation is also unidirectional and the direction direction is forward. We give each variable a left flag and a right flag. The value of a flag is *unknown* or *after_input*. The value *after_input* represents that the corresponding point of the flag is after a tag mark “input” but before a ‘>’ character, which indicates the end of the input tag. We initialize the right flag of the terminal as *after_input* and all the other flags as *unknown*.

Like bidirectional and backward propagation, the forward propagation also has four rules.

1. If two variables are neighboring at the right hand side of a production, we perform a propagation from the right flag of the first variable to the left flag of the second variable.
2. If a terminal contains neither ‘<’ nor ‘>’, we perform a propagation from the left flag of the terminal to the right flag of the terminal.
3. We perform a propagation from the left flag of the production’s left-hand-side variable to the left flag of the first variable in the production’s right hand side.
4. We perform a propagation from the right flag of the last variable in the production’s right hand side to the right flag of the production’s left-hand-side variable.

If we denote the forward propagation from flag A to flag B as assigning $ForPropagate(A, B)$ to B without changing the value of A, the definition of $ForPropagate(A, B)$ is as follows.

- $ForPropagate(A, B)$ = *unknown*, if values of both A and B are *unknown*.
- $ForPropagate(A, B)$ = *after_input*, if either the value of A or the value of B is *after_input*.

The forward propagation ends when no flag value changes. Then, there are three types of terminals that may contain substrings in the scope of the input tag: the terminal corresponding to the input tag, terminals whose left flag and right flag both have the value of *after_input*, and terminals whose left flag has the value of *after_input* and whose right flag has the value of *unknown*. For the terminal corresponding to the input tag, the characters after the tag mark “input” are in the scope of the input tag. For each terminal whose left flag and right flag both have the value of *after_input*, all the characters are in the scope of the input tag. For each terminal whose left flag has the value of *after_input* and whose right flag has the value of *unknown*, the characters before the first ‘>’ is in the scope of the input tag. For the ease of presentation, we also use the scope of an input tag to denote the characters determined to be in the scope of the input tag.

4.3.3 Identifying Value and Type Attributes of Input Tags

After determining the scope of a given input tag, the third phase further determines the parts that indicate the value attribute and type attribute of the input tag in its scope. Here, we need to handle only each input tag whose scope is not contained as a whole in a terminal. Otherwise, the type and the value attributes can be easily acquired. The key technique in this phase is the forward flag propagation described in Section 4.3.2.

As identifying the type attribute of an input tag is very similar to identifying the value attribute of an input tag, we present only the identification of the value attribute of an input tag. Let us consider the HTML syntax about the value attribute of an input tag. For an input tag, there are two ways to indicate its value attribute. The first way is to use a pair of single or double quotations. For example, an input tag with value “abc” can be defined as `<input value='abc'>` or `<input value="abc">`. The second way is not to use quotations. For example, an input tag with value “abc” can also be defined as `<input value=abc>`.

The identification of the value attribute of an input tag consists of four steps. First, we locate all the keyword “value” in the scope of the given input tag. Second, for each keyword “value”, we find all the possible nearest following non-space characters of “value”[7]. Third, for each character found in the second step, if the character is ‘=’, the “value” before ‘=’ indicates that the value attribute is after ‘=’. In this case, we find all the possible nearest following non-space characters of ‘=’. Then the acquired characters should be the first character in the value attribute. Fourth, for each character acquired in the third step, if the character is a ‘"’ or ‘'’, we find all the terminals that may appear in the scope after this character and before another ‘"’ or ‘'’; otherwise, we find all the terminals after this character and before a space or ‘>’.

In the identification of the value attribute of an input tag, we adapt the second propagation rule in forward flag propagation. The essence of the adaptation is as follows. If we are looking for a character (denoted as C), we perform a propagation from the left flag of a terminal to the right flag of the terminal only when the terminal does not contain C. For example, in the second and third steps, when we are looking for the nearest non-space character following “value” or ‘=’ , C becomes any non-space character (Σ/space), and the second propagation rule allows propagation through terminals containing only spaces.

4.3.4 Identifying Need-to-Translate Constant Strings Inside Input Tags

After we acquire the terminals in the type and the value attributes of an input tag, we determine whether the terminals in the value attribute is need-to-translate as follows. We check all the terminals that possibly serve as the type attribute of the input tag. If there is any keyword (e.g., “text” and “button”) indicating that the value attribute of the input tag is user-visible, we conservatively consider all those terminals possibly serving as the value attribute as need-to-translate. Additionally, if an input tag has no type attribute but only a value attribute, the value attribute of the input tag is need-to-translate, because according to the HTML syntax, the default type attribute for an input tag is “text”.

5. IMPLEMENTATION

As our previous approach [17] works for only Java code, our new approach for PHP applications cannot reuse the implementation of our previous approach. Therefore, we adapted an existing

[6] We use “tag mark” to denote a constant string corresponding to an HTML tag, such as the “input” in “<input type=text name=”.

[7] In a generated HTML text, only one value attribute is active; however, the string forming the value attribute can have multiple origins. So an input tag may have multiple terminals as its possible value attributes. The type attribute has the same characteristic.

PHP string analyzer called PHPSA [15]. PHPSA builds a CFG from the PHP code, and the start non-terminal of the built CFG represents all possible HTML texts generated by the PHP code. We re-implemented the adapted string-taint analysis [17] on top of PHPSA. Thus, each time we run the adapted PHPSA, we obtain a CFG with all terminals and non-terminals annotated with locations in the code. Since the start non-terminal represents all possible HTML texts generated by the PHP code, its annotations (location information) indicate all constant strings in the code that may appear in the generated HTML texts, and these constant strings form a list of candidate constant strings.

To identify the need-to-translate constant strings from candidates, we implemented the techniques introduced in Section 4. Our implementation reads the CFG (with annotations) generated by the adapted PHPSA, performs the flag-propagation algorithms (in which our implementation adds and propagates flags for only candidate constant strings), and acquires a list of user-visible terminals that may appear in the user-visible parts of the generated HTML texts.

6. EVALUATION

6.1 Research Questions

Our evaluation aims to answer the following research questions:

- **RQ1**: How effective is our approach for locating need-to-translate constant strings in web applications?
- **RQ2**: How effective are our two new techniques for locating the two types of need-to-translate constant strings in web applications compared with a related previous approach [17]?

The first research question is concerned with evaluating our approach as a whole. The second research question is concerned with evaluating our two new techniques proposed in this paper.

6.2 Approaches Under Comparison

To answer the two preceding research questions, we compared the following three approaches in our evaluation. First, we considered our approach proposed in Section 4. Second, since our approach extends our previous approach [17], we also considered our previous approach without extension as a baseline in the evaluation. Third, as the extension over our previous approach consists of two techniques, we also considered an approach with only one technique (i.e., the technique for locating outside-tag constant strings). For the ease of presentation, we use "BS" to denote our previous approach, which serves as the baseline; "BS+O" to denote the approach with only the technique for locating outside-tag constant strings, indicating that this approach is the baseline approach plus the technique for locating outside-tag constant strings; "All" to denote our new approach proposed in Section 4, indicating that our approach is the baseline approach plus the two techniques.

6.3 Subject Programs

We used three real-world open source web applications as subjects: Lime Survey, Squirrel Mail, and Mrbs. All three applications, which are accessible from SourceForge[8], are written in PHP. The primary reason for choosing these three applications as subjects is that they belong to different web application domains and they have different web page structures. Lime Survey, started in March 2003, is a popular web application to do surveys on the Internet. Squirrel Mail is a well-known web-based email client started since November 1999. Mrbs is a web-based meeting room reservation system started since early May 2000. The information of the subjects is presented in Table 2. For each subject, Columns 1 to 6 show the name and the version number of the application, the starting month of the application, the number of developers involved in the development of the application, the number of lines of code (LOC) of the application, the number of files of the application, and the number of constant strings of the application, respectively.

The developers of all the three subjects did not consider internationalization at the beginning, and they used many hard-coded constant strings in English in early versions of theses subjects. In June 2003, the developers of Lime Survey internationalized Lime Survey and updated the application from Version 0.97 to Version 0.98. The primary aim of the internationalization was to create a version for Spanish users. In January 2000, the developers of Squirrel Mail internationalized Squirrel Mail and updated the application from Version 0.2.1 to Version 0.3prel. In May 2000, the developers of Mrbs began to internationalize Mrbs and updated the application from version 0.6 to 0.7. For all the three subjects, we applied the three approaches on versions before internationalization.

6.4 Metrics

To evaluate the three approaches, we obtained the exact need-to-translate constant strings in the three subjects using a procedure similar to the evaluation in our previous work [17]. First, we deemed constant strings in the version before internationalization as need-to-translate constant strings, if the developers externalized them in the subsequent internationalized version. Second, as the approaches under comparison did find a number of need-to-translate constant strings that were not externalized in the subsequent internationalized version for each subject, we also deemed as need-to-translate the constant strings that at least one approach located and we manually verified to be need-to-translate.

In particular, when one approach located a constant string not externalized in the subsequent internationalized version, we further checked the versions later than the subsequent internationalized version. We deemed the constant string as need-to-translate, if it was externalized in a later version. If not, we manually generated some input data to execute the subsequent internationalized version. If we could see the string on some web page and found it not understandable to a user not familiar with English, we deemed it as need-to-translate; otherwise, we deemed it as not requiring translation. The numbers of the need-to-translate constant strings in the subjects are shown in the last column in Table 2.

All the numbers in our statistics and evaluation are the numbers of constant string locations in the code rather than constant string values. For example, constant strings of the same value but appearing in two different locations in the code are counted as two. Furthermore, the numbers are for the versions before internationalization. For example, if three constant strings in the previous versions are externalized as one string in the internationalized version, the constant strings are counted as three.

Based on the obtained need-to-translate constant strings, for each approach and each subject, we calculated the number of constant strings that need translation but are not located by the approach (denoted as false negatives), and the number of constant strings that are located by the approach but actually do not need translation (denoted as false positives). In the brackets after the numbers shown in the last two columns of Table 3, the percentages of false negatives and false positives are presented.

6.5 Results and Analysis

6.5.1 RQ1: Overall Effectiveness

The rows where subject names are marked with "All" in Table 3 show the results of using our approach on the three subjects. In the table, we use FN to denote false negatives and FP to denote false positives. We use Lime and Squirrel as abbreviations of Lime Survey and Squirrel Mail, respectively. The "Need-to-

[8] http://www.sourceforge.net/

Table 2: Basic information of the subjects

Application /Version	Starting Month	#Developers	#LOC	#Files	#Constant Strings	#Need-to-Translate (#Not externalized in the subsequent version)
Lime Survey 0.97	March 2003	3	11.3k	25	6493	290(5)
Squirrel Mail 0.2.1	Nov. 1999	1	4.0k	39	2457	184(74)
Mrbs 0.6	May 2000	2	1.4k	19	704	57(20)

Table 3: Results of the three experimented approaches

Subject (Approach)	Need-to-Translate	Located	FN	FP
Lime (ALL)	**290**	**219**	**89(31%)**	**18(6%)**
Lime (BS+O)	290	198	110(38%)	18(6%)
Lime (BS)	290	599	89(31%)	398(137%)
Squirrel (ALL)	**184**	**192**	**0(0%)**	**8(4%)**
Squirrel (BS+O)	184	180	12(7%)	8(4%)
Squirrel (BS)	184	718	0(0%)	534(290%)
Mrbs (ALL)	**57**	**42**	**17(30%)**	**2(4%)**
Mrbs (BS+O)	57	42	17(30%)	2(4%)
Mrbs (BS)	57	140	17(30%)	100(175%)

Translate" and "Located" columns show the number of exact need-to-translate constant strings and the number of need-to-translate constant strings located by each approach, respectively. From the results, we have three main observations.

First, our approach to locating need-to-translate constant strings in web applications (using both new techniques) is able to locate most of the need-to-translate strings. In Lime Survey, our approach locates 201 of 290 exact need-to-translate constant strings. In Squirrel Mail, our approach locates all the 184 need-to-translate constant strings. In Mrbs, our approach locates 40 of 57. All the 89 false negatives in Lime Survey are due to the analysis inaccuracy of the PHP string analyzer used in our implementation. In particular, in Lime Survey, there are constant strings that are compared with the default values of input tags in the generated web pages that are fetched by `$POST['name of the input tag']`. Since the default values of the input tags are need-to-translate (when the input tags are of certain types), the constant strings that are compared to the default values are also need-to-translate. The PHP string analyzer used in our implementation cannot catch the assignment relation between the default value of an input tag and the variable `$POST['name of the input tag']`. Since the CFG itself does not contain this relation, our approach currently cannot handle this situation. We believe that our approach should be able to locate these false negatives if the underlying string analyzer is implemented in a more advanced way. The 17 false negatives in Mrbs are due to that Mrbs used some static HTML texts together with PHP code, and our approach cannot locate need-to-translate constant strings in static HTML texts. However, such need-to-translate constant strings can be easily found using an HTML parser.

Second, for each subject, our approach produces a few false positives. In Lime Survey, Squirrel Mail, and Mrbs, the numbers of constant strings that are located by our approach but do not need translation are 18, 8, and 2, respectively. Compared to the numbers of need-to-translate strings in the three subjects, the numbers of false positives are quite small. One reason for false positives is our conservative policy on *conflict* flags. This reason accounts for 12 false positives in Lime Survey and 7 false positives in Squirrel Mail. The second reason of false positives is the user-visible constant strings that cannot be translated. This reason accounts for 1 false positive in Squirrel Mail (which is the trademark "Squirrel Mail" itself), and 6 false positives in Lime Survey, which are user-visible SQL queries. The third reason is some small PHP chips inside static HTML texts such as `<div name=<?php echo "div".$i?> >`. Since our approach considers only the content inside the PHP code, we cannot decide whether `"div"` is inside a tag or not. We conservatively consider such constant strings as need-to-translate and result in 2 false positives in Mrbs. In fact, such false positives can be resolved by parsing the static HTML to add the context for each portion of PHP code.

Third, for each subject, our approach is able to locate some constant strings that the developers did not externalize in the subsequent internationalized version but we verified as need-to-translate. The developers may have either missed them or did not externalize them at that time due to time or workload limit. In both cases, locating such strings should be helpful for the developers to obtain a version with better quality of internationalization earlier. In total, our approach locates 99 such strings in the three subjects (5 in Lime Survey, 74 in Squirrel Mail, and 20 in Mrbs shown in the brackets of the last column of Table 2). Among the 99 strings, 62 (5 in Lime Survey, 44 in Squirrel Mail, and 13 in Mrbs) were externalized and translated in a later version and 37 still remained hard-coded in all the later versions or were removed due to modifications other than internationalization.

One example of not-externalized constant strings is presented in the following code portion (in "smtp.php" of Squirrel Mail).

```
...
switch ($err_num) {
   ...
   case 502:$message = "Command not implemented";
         $status = 0;
         break;
   ...
}
...
echo "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
   &nbsp;&nbsp;&nbsp;<B>Reason: </B>$message<BR>";
...
```

This code portion outputs an error message `"Command not implemented"` to the generated web page, and then the error message is shown on the browser. However, the developer of Squirrel Mail did not externalize this string until three years later (Jan 2003).

6.5.2 *RQ2: Effectiveness of the Two New Techniques*

In Table 3, the rows where subject names are marked with "BS" show the results of using the baseline approach (i.e., our previous approach [17]) on the three subjects, and the rows where subject names are marked with "BS+O" show the results of using the baseline approach together with only the first technique (i.e., locating outside-tag constant strings). After a close examination of Table 3, we have the following four main observations concerning the effectiveness of the two new techniques.

First, when used for locating need-to-translate constant strings in web applications, our previous approach produces large numbers of false positives. Typically, the numbers of false positives are much larger than the numbers of need-to-translate constant strings. For Squirrel Mail, the number of false positives is almost three times as large as the number of need-to-translate constant strings. This observation indicates that, without further distinguishing user-visible constant strings from non-visible constant strings, our previous approach is not suitable for web applications.

Second, our first technique (i.e., locating outside-tag constant strings) is able to dramatically reduce the false positives. After comparing the rows where subject names are marked with "BS+O" and "BS" for each subject, we find that our first technique reduces 380 of 398 false positives in Lime Survey, 526 of 534 false positives in Squirrel Mail, and 98 of 100 false positives in Mrbs, respectively. This observation confirms that our first technique is highly effective in locating need-to-translate constant strings in web applications. However, our first technique cannot reduce all the false positives. We have analyzed the reasons of those false positives in Section 6.5.1.

Third, our second technique (i.e., locating need-to-translate constant strings inside input tags) is able to locate some need-to-translate strings without inducing extra false positives. In particular, after comparing the rows where subject names are marked with "All" and "BS+O", we obtain the numbers of new need-to-translate constant strings located by our second technique (i.e., 21 for Lime Survey, 12 for Squirrel Mail, and 0 for Mrbs). This observation indicates that our second technique is also effective.

Fourth, the combination of our two new techniques accurately tackles the challenges that we have faced when adapting our previous approach for web applications. After comparing the rows where subject names are marked with "All" and "BS", we find that none of the false negatives produced by our approach is due to our two new techniques. That is to say, if we consider the evaluation of only the two new techniques, the combination of the two techniques is able to find all the need-to-translate constant strings among the candidate constant strings located by our previous approach with less than 10% false positives.

6.6 Threats to Validity

In our evaluation, the threats to internal validity mainly lie in the way we verified constant strings not externalized in the subsequent internationalized version to be need-to-translate strings. Since the later versions involve various modifications for other purposes, it is error-prone to verify constant strings as need-to-translate in versions later than the version immediately after internationalization. Furthermore, the manual verification of constant strings not externalized in any version as need-to-translate may be prone to accidental mistakes or personal perspectives to the notion of being "need-to-translate". To reduce these threats, for each subject, we examined all these strings in the later versions carefully, executed the internationalized subject to see whether they appear on the browser and decided whether they are not understandable to a user who is not familiar with English.

The main threats to external validity lie in that the three subjects used in our empirical study are all open source web applications in PHP, and all of them are of moderate sizes (i.e., several thousand lines of code). Furthermore, to make use of the manually externalized constant strings as golden results, we use old versions of applications generated years ago. Therefore, it is possible that the findings of our empirical evaluation are specific to old versions of moderate-size open-source web applications, and may not be generalized to other subjects. To reduce these threats, we plan to apply our approach to more web applications, especially those that are commercial, with larger code bases or using new web technologies.

7. DISCUSSION

As discussed in our previous paper [17], the conservative nature of string analysis may induce imprecision and this imprecision may impact the empirical results of locating need-to-translate constant strings in web applications. However, according to our empirical results reported in this paper, we have not found false positives due to the preceding weakness of string analysis for locating need-to-translate constant strings in web applications.

One important situation that our approach currently cannot handle is the case where a web application dynamically generates JavaScript code, and the generated JavaScript code, when executed by the browser, outputs some need-to-translate strings to the generated web page directly or outputs some need-to-translated strings by calling APIs of JavaScript libraries. For example, our approach currently cannot handle Ajax web applications. It is not easy to locate such need-to-translate strings because we need to analyze the syntax of the generated JavaScript part without obtaining all the possible contents of the JavaScript part. Since the JavaScript syntax is much more complex than HTML, we may need to propose more complex adaptations to our flag propagation as well as developing some new techniques to handle dynamically generated JavaScript code.

8. RELATED WORK

To our knowledge, our work is the first reported effort directly focusing on automatically locating need-to-translate constant strings in web applications. Our previous work [17], which locates need-to-translate strings in Java code, is the most related work. However, when used for web applications, our previous work cannot differentiate between constant strings that form HTML tags and constant strings that are displayed on the screen.

Furthermore, there have been a couple of published books on how to internationalize a software application [5, 16]. These books provide guidelines on finding out need-to-translate constant strings and externalizing them, but they do not propose any approach to perform the task automatically. On top of that, some researchers analyzed the process of internationalization and presented issues to be considered during the process, including the problem of locating need-to-translate strings [10, 4, 9]. However, none of them provided any automatic approach to locating need-to-translate strings.

There are also tools (e.g., GNU gettext[9], Java internationalization API[10]) to help developers externalize need-to-translate constant strings after the developers locate them. Other tools like KBabel[11] help developers edit and manage resource files (called PO files) containing externalized constant strings. These tools cannot locate need-to-translate constant strings, but can be used together with our approach to improve the efficiency of internationalization.

String analysis and string-taint analysis are recent improvements of traditional static data-flow analysis [11]. Christensen et al. [2] first proposed string analysis, which is an approach for obtaining possible values of a string variable. Then string analysis is widely used in checking dynamically generated SQL queries. Gould et al. [6] used a string-analysis-based approach to check the correctness of dynamically generated query strings. Halfond and Orso [8] used string analysis to detect and neutralize SQL injection attacks.

Minamide [15] first used string analysis on web applications. He proposed to simulate string operations in the extended CFG with FSTs, and implemented a string analyzer for PHP code to predict dynamically generated web pages. Based on Minamide's work, Xie and Aiken [21] proposed a technique on detecting SQL injection vulnerabilities in scripting languages. Recently, Wassermann and Su developed string-taint analysis [18] based on Minamide's work and applied this analysis on detecting cross-site scripting vulnerabilities [19]. Recently, Wassermann and Su [20] developed an approach to generating test cases for security vulnerabilities and Kieyzun et.al. [12] further improved their approach. Compared to these approaches, we apply string-taint analysis on locating need-to-translate string candidates, and developed flag-propagation algorithms to find need-to-translate strings among the candidates.

Like string-taint anlysis, our flag-propagation algorithms also propagate information on the CFG generated by string analysis. However, our flag propagation differs from taint propagation in three main aspects. First, the purpose of flag propagation is different from that of taint propagation. Taint propagation aims to find all

[9] http://www.gnu.org/software/gettext/manual/gettext.html

[10] http://java.sun.com/docs/books/tutorial/i18n/index.html

[11] http://kbabel.kde.org/

the non-terminals that may contain a taint (i.e., non-terminals that may deduce to a terminal with taint), while flag propagation aims to find the preceding or following terminals of a given terminal with some conditions (e.g., the nearest, non-space, before or after certain character values like '>'). Second, taint propagation requires only one taint (similar to the flags in flag propagation), which may be of two or more taint values, while our algorithms require at least two flags, which represent the left-most point and the right-most point of a variable. Third, taint propagation has only one propagation rule to propagate all the taints at the right hand side of a production to the left hand side of the production, while our algorithms have four propagation rules to make sure that flags can be propagated through neighboring variables, variable deductions, and terminals.

Another area related to our approach is machine translation [14]. Machine translation has been used in translating web pages in some products (e.g., google translation[12], babel[13]). If the technique of machine translation is advanced enough to accurately translate the generated web pages in real time, the internationalization and localization of web applications may become unnecessary. However, it may take a long way to achieve this goal and almost all web applications are using internationalization and localization now. Additionally, the technique of machine translation can also be integrated in our approach to help developers translate the need-to-translate strings after our approach identifies them.

9. CONCLUSION AND FUTURE WORK

In this paper, we identify significant challenges in automatically locating need-to-translate constant strings in web applications. To address the challenges, we propose two techniques to distinguish two types of user-visible constant strings among all the constant strings that may go to the generated HTML texts. We evaluated our approach on three real-world open source web applications: Lime Survey, Squirrel Mail, and Mrbs. The empirical results demonstrate that, as our approach is able to accurately distinguish visible strings from non-visible strings, our approach is effective for locating need-to-translate constant strings in web applications.

We plan to extend our approach in three ways. First, to reduce the previously mentioned threats to the validity of our empirical evaluation, we plan to conduct experiments on larger and newer commercial web applications.

Second, we plan to capture the string relations missed by the current PHP analyzer and reduce the false negatives in future work. As stated in Section 6.5, integrating an HTML parser in our approach enables our approach to handle small pieces of PHP code inside static HTML texts. Therefore, we plan to integrate such a parser in our approach in near future. We also plan to integrate dynamic analysis in our approach to reduce false positives.

Third, our approach currently works only for PHP web applications. We plan to extend it for web applications written in other languages (e.g., JSP). Since JSP web applications contain Java code, we need to integrate our approach with our previous approach for locating need-to-translate constant strings in Java code.

Acknowledgment

The authors from Peking University are sponsored by the National Basic Research Program of China (973) No. 2009CB320703, the High-Tech Research and Development Program of China (863) No. 2007AA010301 and No. 2006AA01Z156, the Science Fund for Creative Research Groups of China No. 60821003, and the National Science Foundation of China No. 90718016. Tao Xie's work is supported in part by NSF grants CNS-0720641, CCF-0725190, and Army Research Office grant W911NF-08-1-0443.

[12] http://translate.google.com/

[13] http://babel.yahoo.com/

10. REFERENCES

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

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

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. and Sys.*, 13(4):451–490, October 1991.

[4] V. Dagiene and R. Laucius. Internationalization of open source software: framework and some issues. In *Intl. Conf. on Info. Tech.: Research and Edu.*, pages 204–207, 2004.

[5] B. Esselink. *A Practical Guide to Software Localization: For Translators, Engineers and Project Managers*. John Benjamins, 2000.

[6] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654, 2004.

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

[8] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *ASE*, pages 174–183, 2005.

[9] P. A. V. Hall. *Decision Support Systems for Sustainable Development*. Springer US, 2002.

[10] J. Hogan, C. Ho-Stuart, and B. Pham. Key challenges in software internationalisation. In *Australian Computer Science Workshops Frontiers*, pages 187–194, 2004.

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

[12] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209, 2009.

[13] A. Koesnandar, S. G. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee. Using assertions to help end-user programmers create dependable web macros. In *FSE*, pages 124–134, 2008.

[14] A. Lopez. Statistical machine translation. *ACM Comput. Surv.*, 40(3):1–49, 2008.

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

[16] E. Uren, R. Howard, and T. Perinotti. *Software Internationalization and Localization: An Introduction*. Van Nostrand Reinhold, 1993.

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

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

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

[20] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.

[21] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, pages 179–192, 2006.