Static Approximation of Dynamically Generated Web Pages

Yasuhiko Minamide
Department of Computer Science
University of Tsukuba
Tsukuba 305-8573, Japan
minamide@cs.tsukuba.ac.jp

ABSTRACT

Server-side programming is one of the key technologies that support today's WWW environment. It makes it possible to generate Web pages dynamically according to a user's request and to customize pages for each user. However, the flexibility obtained by server-side programming makes it much harder to guarantee validity and security of dynamically generated pages.

To check statically the properties of Web pages generated dynamically by a server-side program, we develop a static program analysis that approximates the string output of a program with a context-free grammar. The approximation obtained by the analyzer can be used to check various properties of a server-side program and the pages it generates.

To demonstrate the effectiveness of the analysis, we have implemented a string analyzer for the server-side scripting language PHP. The analyzer is successfully applied to publicly available PHP programs to detect cross-site scripting vulnerabilities and to validate pages they generate dynamically.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Formal methods, Validation; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Languages, Verification, Security

Keywords

Server-side scripting, static analysis, context-free grammars, cross-site scripting, HTML validation

1. INTRODUCTION

Server-side programming is one of the key technologies that support today's WWW environment. It makes it possible to generate Web pages dynamically according to a user's request and to customize pages for each user. However, the flexibility obtained by server-side programming makes it much harder to guarantee validity and security of dynamically generated pages. For example, it is well known that inappropriate treatment of input data causes vulnerabilities called cross-site scripting, which may cause leakage of critical information such as HTTP cookies. It is critical for a server-side program to prevent this kind of vulnerability and to guarantee security.

We have applied static program analysis to static checking of properties of Web pages generated dynamically by a server-side program. Our analysis approximates the string output of a program with a context-free grammar. The approximation obtained by the analysis is conservative in the sense that it contains any possible output string that could be generated by the program. By applying the analysis to a server-side program, its dynamically generated Web pages can also be approximated with a context-free grammar. Then, the approximation obtained by the analysis has many applications in checking the validity and security of a server-side program.

We have applied the analysis to detect cross-site scripting vulnerabilities in a server-side program. The vulnerabilities can be detected by checking the approximation against the specifications of safe or unsafe strings. For example, Web pages that do not include code executed at the client side are considered to be safe strings.

The second application of our analysis is HTML validation. We present two approaches to validating pages dynamically generated by a server-side program. The first approach is to generate sample Web pages from the grammar and validate them with a standard HTML validation tool. The second approach is to check the approximation against the HTML specification. This requires to solve an undecidable problem whether a context-free language is a sublanguage of another context-free language. However, by taking the characteristics of the pages generated by a server-side script into consideration, we show that the Web pages generated by the majority of server-side programs can be validated.

Our analysis is based on the Java string analyzer of Christensen, Møller and Schwartzbach [7]. Their analyzer approximates the value of a string expression in a Java program with a regular language instead of a context-free language. Both their and our analyzers first extract a grammar with string operations from a program. After this phase, our string analyzer directly eliminates string operations in the grammar and transforms it into a context-free grammar. In this phase, a string operation is modeled with an automaton with output called a transducer. The theory of transducers has crucial roles both in this phase and in applications of our analysis.

To demonstrate the effectiveness of our string analysis on

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

server-side programs, we have implemented a string analyzer for PHP. The analyzer takes two inputs: a PHP program and an input specification that describes the set of possible input to the program. It then generates a context-free grammar approximating the Web pages generated from the input. The analyzer is successfully applied to publicly available PHP programs to detect cross-site scripting vulnerabilities and to validate pages they generate dynamically.

Huang et al. also developed a static program analyzer for PHP [12]. Their analyzer was based on trust analysis (information flow analysis) and can be used to detect vulnerabilities such as cross-site scripting and SQL injection. Although both their analyzer and ours can be applied to detect vulnerabilities in PHP programs, they differ in principle and we believe that neither analyzer has a theoretically better ability to detect cross-site scripting vulnerabilities than the other.

2. PHP STRING ANALYZER

PHP is one of the most popular server-side scripting languages used to generate Web pages dynamically [1]. We have developed a string analyzer for PHP that approximates the string output of a program as a context-free grammar. The analyzer takes two inputs: a PHP program and an input specification. The input specification is given as a regular expression and describes the set of possible inputs to the PHP program. In this section, we illustrate our string analyzer by examples. The internal structure of the analysis is discussed in Section 5.

To illustrate the string analysis, let us consider the following program.

```
<?php
for ($i = 0; $i < $n; $i++)
  $x = "0".$x."1";
echo $x;
?>
```

In PHP, the infix operator dot represents string concatenation. This program concatenates the same number of `"0"`s to the left and `"1"`s to the right of `$x`: the number depends on the value of `$n`.

The input specification is given by specifying the initial values of global variables in our analyzer. The initial values of `$x` and `$n` are described in the following specification.

```
$x : /abc|xyz/
$n : int
```

The specification `/abc|xyz/` is a regular expression representing the set of strings $\{abc, xyz\}$. Only the type is specified for the variable `$n`.

The idea of string analysis is to consider assignments as production rules of a context-free grammar. By considering assignments as production rules and translating the input specification into production rules, we can obtain the following grammar approximating the output of the program.

$$\begin{array}{lcl} X & \rightarrow & abc \\ X & \rightarrow & xyz \\ X & \rightarrow & 0X1 \end{array}$$

This grammar represents the set of strings, $\{0^n abc1^n \mid n \geq 0\} \cup \{0^n xyz1^n \mid n \geq 0\}$, as we expect. Based on this idea, we have developed a string analyzer for PHP.

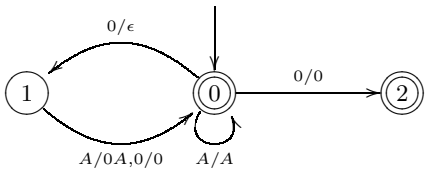
This analysis works even if a program contains string operations other than concatenation. Let us consider the following revised program:

```
<?php
for ($i = 0; $i < $n; $i++)
  $x = "0".$x."1";
echo str_replace("00","0",$x);
?>
```

where `str_replace("00","0",$x)` replaces the string `00` in `$x` with `0`. The set of strings the variable `$x` may contain after the `for` loop is represented by the context-free grammar above. Therefore, we can obtain the context-free grammar for the output of the revised program if we know how a grammar is transformed by `str_replace("00","0",$x)`.

A finite automaton with output called a transducer plays a crucial rule here. A transducer has the key property that the image of a context-free language under a transducer is context-free [4]. Furthermore, many string operations can be realized by transducers.

Let us consider `str_replace("00","0",$x)` in the example: this operation can be realized by the following transducer:



where A is any character except 0. There are three states 0, 1 and 2 in this transducer: the state 0 is the start state, and 0 and 2 are the final states. The transitions labeled with $0/\epsilon$ and $A/0A$ mean that the transducer produces ϵ and $0A$ for the inputs 0 and A, respectively. For example, this transducer outputs $0abc11$ for the input $00abc11$.

There is an algorithm to compute the image of a context-free language under a transducer. Then, we can compute the following context-free grammar with the start symbol S by computing the image of the previous context-free grammar.

$$\begin{array}{lcl} S & \rightarrow & abc \mid xyz \mid X1 \\ X & \rightarrow & 0abc \mid 0xyz \mid 0S1 \end{array}$$

This is the approximation obtained by our analyzer. This grammar represents the following set of strings.

$$\begin{array}{ll} & \{0^n abc1^{2n} \mid n \geq 0\} \cup \{0^n xyz1^{2n} \mid n \geq 0\} \\ \cup & \{0^{n+1} abc1^{2n+1} \mid n \geq 0\} \cup \{0^{n+1} xyz1^{2n+1} \mid n \geq 0\} \end{array}$$

As we can see in this example, the output of a program can often be precisely approximated even if it contains string operations.

3. DETECTING CROSS-SITE SCRIPTING VULNERABILITIES

We consider detection of cross-site scripting vulnerabilities in PHP programs as the first application of our analyzer. A cross-site scripting vulnerability results from inappropriate treatment of input data and may cause leakage of critical information such as HTTP cookies. To prevent the vulnerability, a string coming from a user input must be sanitized before embedding it in a Web page. Sanitization is achieved

by escaping the special characters such as < and & in HTML. Let us consider the following PHP program.

```
<html><body>
<?php
$x = $_POST['name'];
//$x = htmlspecialchars($x);
for ($i = 0;  $i < 2; $i++)
  echo $x;
?>
</body></html>
```

This program receives a potentially unsafe input string from a Web browser in the associative array `$_POST`. The value corresponding to the key `name` is assigned to the variable `$x` and output in the `for` loop. Thus it may generate a Web page with unsafe strings from the input and is therefore vulnerable. This vulnerability is called a cross-site scripting vulnerability. If the sanitization code that is commented out in the code is enabled, the code is safe: `htmlspecialchars` converts special characters such as < and & into the corresponding HTML entities: `<` and `&`. The resulting string is safe to embed in a Web page.

We can detect cross-site scripting vulnerabilities by checking the approximation obtained by the string analyzer against the specification of the set of safe or unsafe strings. The approximation must be constructed for the input specification corresponding to the fact that any string might be sent by a Web browser. The following is the specification for the variable `$_POST`.

```
$_POST : [/.*/]
```

This represents an array which contains any string. A specification of an array is given in the form [*reg*].

Let us consider detecting a cross-site scripting vulnerability by checking whether a Web page generated with a PHP program may contain the `<script>` tag. A Web page containing this tag can be considered as an unsafe output for a program that is not expected to generate such a page. To conduct this security check, we specify the set of unsafe strings with the following regular expression.

```
.*<script>.*
```

This regular expression denotes the set of strings that contain the `<script>` tag. Then, we can check whether the context-free language obtained by the analyzer is disjoint with this set. If they are disjoint, it is guaranteed that the output cannot contain a `<script>` tag. Checking the previous program against the specification above by our analyzer produces the following counter example.

```
<html><body>
<script>
</body></html>
```

That means that the program may produce the above output and thus may be unsafe. On the other hand, if the sanitization code is enabled, the analyzer says that the approximation of the output and the specification are disjoint, and therefore the code is safe. This check is done by an algorithm checking whether a context-free language is disjoint with a regular language. The algorithm is based on the context-free graph reachability algorithm proposed for interprocedural dataflow analysis [21, 15].

Notice that the counter-example above cannot actually be an output of the program because there is a `for` loop that repeats twice in the program. This kind of impreciseness is inevitable in program analysis.

There is another mode of checking in our analyzer to specify the set of safe strings instead of unsafe strings. The following regular expression specifies the set of strings that contain only the `html` and `body` tags:

```
(<html>|</html>|<body>|</body>|[^<])*
```

where `[^<]` represents any character except <. This can be considered the specification of the set of safe strings for the program above. Without the sanitization code, our analyzer shows a counter example as before. With the sanitization code, it says that the PHP program is safe. This check is conducted by computing the complement of the language of the regular expression. Because the complement of a regular language is regular, we can use the algorithm checking disjointness between a regular language and a context-free language as before.

Dynamically generated Web pages often contain code to be executed at the client side identified by the `script` tag. Then, it is not clear how to specify safe or unsafe strings. Still, we can check whether the output of a PHP program may contain an invalid tag such as `<xyz>`. If an output of the program contains such a tag, the program is likely to have cross-site scripting vulnerabilities or at least some bugs.

Furthermore, our string analyzer can be used to check what tags may appear in output, by applying a transducer that extracts tags from a string. Our string analyzer has a mode to specify a regular expression to extract strings matching it from the approximation of the output. The following regular expression specifies the set of strings with < and > as the first and last characters, respectively, with no < and > between them.

```
<[^<>]*>
```

With sanitization, the analyzer shows the following results for the program above.

```
</body>
</html>
<body>
<html>
```

On the other hand, it says that the set of tags is not finite without sanitization.

4. HTML VALIDATION

As the second application of our analyzer, we describe HTML validation of the Web pages dynamically generated by a server-side program. There are several HTML validation tools that can validate a static Web page against the HTML specification. However, to check that a server-side program always generates valid Web pages, it must be extensively tested to cover all the possible execution path of the program. This requires knowledge of the internal structure of the program and is very time consuming.

On the other hand, we can automatically test or check the validity of the pages based on the context-free grammar obtained by our analyzer. There are two approaches to HTML validation in our analyzer: the automated validation test and matching validation.

4.1 Automated Validation Test

The first approach is to generate sample Web pages from the context-free grammar and check the pages with a standard HTML validator such as the W3C Markup Validation Service [25] and WDG HTML Validator [20]. This approach will be quite effective if we can obtain a set of sample pages which cover all the possible execution path.

Let us consider validation of the Web pages generated by the following PHP program.

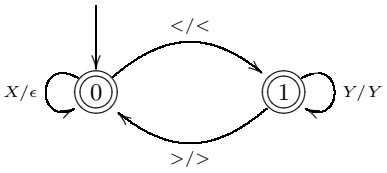
```
<head><title>test</title></head>
<body>
<?php
if ($_POST["abc"] == "abc")
    echo "abc";
else
    echo htmlspecialchars($_POST["xyz"]);
?>
</body>
```

To simplify discussion, we check this program for the input specification `$_POST : [/[a-z<]*/]`, representing arrays with strings consisting of lower case characters and the character `<`. Then, our string analyzer extracts the following sample pages if we ask the analyzer to extract three sample pages.

```
<head><title>test</title></head>
<body>
abc
</body>
--------------------------------
<head><title>test</title></head>
<body>
a
</body>
--------------------------------
<head><title>test</title></head>
<body>
&lt;
</body>
```

The validity of these pages can be checked with the standard validation tools.

Furthermore, the test cases can be reduced with the following transducer, which removes the parts of a Web page not related tags:



where X is any character except $<$ and Y is any character except $>$. If we apply this transducer to the previous example, the approximation obtained by the analyzer is reduced to a singleton set and thus can be completely checked by a standard validation tool.

4.2 Matching Validation

Because the validity checking described above cannot cover all cases in general, it is not complete. Therefore, it is desirable to develop a validation algorithm to check whether a PHP program always generates a valid HTML page. The specification of HTML can be described with a context-free grammar. Hence, we had a validation algorithm if we had an algorithm to determine $L_1 \subset L_2$ for two context-free grammars L_1 and L_2. Unfortunately, this problem is known to be undecidable in general. However, taking the characteristics of the pages generated by a server-side program into consideration, we can still validate the majority of server-side programs.

We investigated the depth of nesting of tags in the pages generated by server-side programs. We believe that in majority of cases, the nesting depth of tags in the generated pages is bounded. We think that this holds because the majority of PHP programs are used to generate pages with table-like structures.

If the nesting depth is bounded, the validity check can be done in the following manner. First, the grammar L_i, specifying valid pages with the maximum tag-nesting depth i, is constructed from the context-free grammar L_{HTML} of the HTML specification. With this restriction on the depth, the language of L_i is regular. Then, the context-free language obtained by the analyzer is checked against L_i: if the language is a subset of L_i, the program always generates a valid HTML page with a depth that is equal to or less than i. This is repeated for $i = 1, 2, 3, \ldots$. However, it will be computationally expensive to construct L_i $(i = 1, 2, \ldots)$ directly for L_{HTML} and check the approximation against them.

We have implemented in our analyzer a simplified version of the validation above, which checks that start and end tags always match. We call this validation a matching validation. Let us consider the following context-free grammar with the start symbol U specifying a set of strings that contain matching `head`, `title`, and `body` tags[1].

$$\begin{array}{lcl} T & \rightarrow & \texttt{<head>}\, U\, \texttt{</head>} \mid \texttt{<title>}\, U\, \texttt{</title>} \mid \\ & & \texttt{<body>}\, U\, \texttt{</body>} \\ U & \rightarrow & \epsilon \mid TU \end{array}$$

Then L_i with the start symbol U_i is constructed as follows:

$$\begin{array}{lcl} U_0 & \rightarrow & \epsilon \\ T_1 & \rightarrow & \texttt{<head>}\, U_0\, \texttt{</head>} \mid \texttt{<title>}\, U_0\, \texttt{</title>} \mid \\ & & \texttt{<body>}\, U_0\, \texttt{</body>} \\ U_1 & \rightarrow & \epsilon \mid T_1 U_1 \\ T_2 & \rightarrow & \texttt{<head>}\, U_1\, \texttt{</head>} \mid \texttt{<title>}\, U_1\, \texttt{</title>} \mid \\ & & \texttt{<body>}\, U_1\, \texttt{</body>} \\ U_2 & \rightarrow & \epsilon \mid T_2 U_2 \end{array}$$

Based on the series of grammars like L_i, we have implemented a matching validation that checks that start and end tags always match in our analyzer.

5. INSIDE THE STRING ANALYZER

We describe the algorithm to approximate the string output of a program with a context-free grammar. The analysis consists of two phases: the first phase extracts a grammar extended with string operations and the second phase transforms the grammar into a context-free grammar. Transducers have crucial roles in the second phase.

[1]For simplicity, we ignore the text except for the tags.

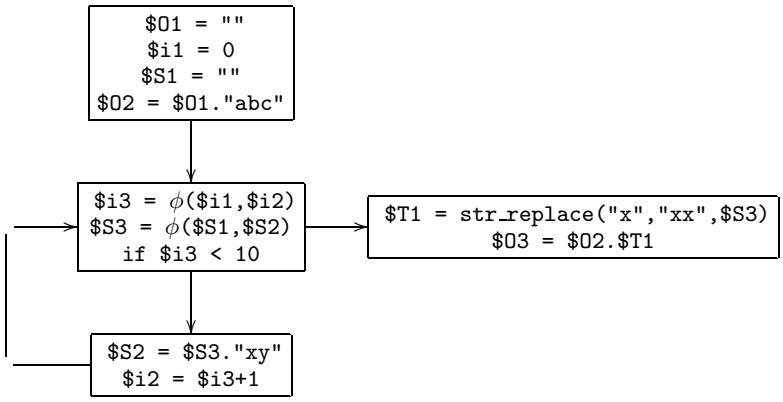


Figure 1: A PHP program in static single assignment form

5.1 Extracting a Grammar

The first phase of the analysis extracts a grammar from a program by considering assignments as production rules, as we described in Section 2. However, we must translate output functions into assignments and divide live ranges of variables to obtain precise approximations.

Let us consider the following program to illustrate how to extract a grammar.

```
$i = 0; $S = "";
echo "abc";
while ($i < 10) {
   $S = $S."xy";
   $i = $i + 1;
}
$T = str_replace("x","xx",$S);
echo $T;
```

The first step is to translate the output functions in the program into assignments to treat them uniformly with other assignments. For the translation, we introduce a global variable `$O` representing the current output. The previous example is translated as follows.

```
$O = "";
$i = 0; $S = "";
$O = $O."abc";
while ($i < 10) {
   $S = $S."xy";
   $i = $i + 1;
}
$T = str_replace("x","xx",$S);
$O = $O.$T;
```

The value of `$O` is initialized to the empty string at the beginning of execution. The function `echo` is translated into a combination of an assignment and a concatenation.

Then, we translate the program into static single assignment form [2, 22] to represent constraints between variables in greater detail. The result of the translation is shown in Figure 1. In static single assignment form, each assignment introduces a new variable and ϕ-functions are introduced where control flows merge. For example, `$i3 =` ϕ`($i1,$i2)` means that `$i1` is assigned to `$i3` if its execution comes from the block above and `$i2` is assigned to `$i3` if its execution comes from the block below.

Finally, we extract a grammar from a program in static single assignment form. An assignment of the form $x = \phi(y, z)$ is translated into two productions: $x \to y$ and $x \to z$ and the other assignments are directly translated into productions.

$$
\begin{array}{lcl}
O_1 & \to & \epsilon \\
S_1 & \to & \epsilon \\
O_2 & \to & O_1 abc \\
S_3 & \to & S_1 \\
S_3 & \to & S_2 \\
S_2 & \to & S_3 xy \\
T_1 & \to & \texttt{str\_replace}(x, xx, S_3) \\
O_3 & \to & O_2 T_1
\end{array}
$$

Notice that productions are generated only for assignments related to string values here. This is done by using information obtained by the program analysis called alias analysis in our implementation. The grammar extracted from a static single assignment form in this way may not be context-free. For example, the grammar above has the string operation `str_replace` in the right-hand side of a production. We call this kind of grammar a context-free grammar with operation productions. We describe in the next subsection how to obtain a context-free grammar from a grammar with operation productions.

5.2 Transforming a Grammar into a Context-Free Grammar

A grammar with operation productions is extracted from a program as described in the previous section. The next phase is to transform it into a context-free grammar. We can transform it into a context-free grammar representing the same set of strings if the following conditions hold.

- Each string operation in the grammar transforms a context-free grammar into another context-free grammar.
- No string operation occurs in a cycle of productions.

The string operations in PHP are classified in Table 1 to discuss the first of above conditions. The functions are classified by considering them as functions with one argument by fixing the other arguments. The second and last rows show the number and examples of the functions in each class, respectively.

Classification	# Ops	Operations
homomorphism	11	htmlspecialchars, strtolower, addslashes
transducer	20	str_replace, trim, ucfirst, stripslashes
pushdown transducer	1	strip_tags
others	6	strrev, str_shuffle, str_repeat, crypt, md5, sha1

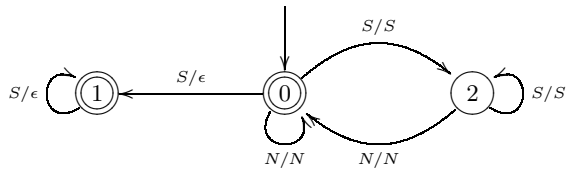
Table 1: Classification of the string functions in the String Functions Section of the PHP manual

- A homomorphism is a mapping from characters to strings. The image of a context-free language under a homomorphism is also context-free and it is straightforward to compute the grammar representing the image. For example, `htmlspecialchars` is a homomorphism as follows:

  ```
  <  ↦  &lt;
  >  ↦  &gt;
  &  ↦  &amp;
  "  ↦  &quot;
  a  ↦  a
  b  ↦  b
  ⋮     ⋮
  ```

- A transducer is a finite automaton with output, as described in Section 2. The image of a context-free grammar under a transducer is also context-free. We have developed an algorithm to compute the image based on the context-free graph reachability algorithm [21, 15]. The algorithm is expensive in the sense that the image has $O(n^3m)$ productions for a context-free grammar with m productions and a transducer with n states.

  A transducer can be nondeterministic. The following is a nondeterministic transducer realizing the function `rtrim`, which strips whitespace from the end of a string:

  where S and N are space and nonspace characters, respectively.

- A pushdown transducer is a pushdown automaton with output. The image of a context-free grammar under a pushdown transducer may not be context-free. We found only one function in this class: the function `strip_tags` strips HTML tags from a string. It leaves the character `>` which does not have the matching `<` as follows:

  ```
  strip_tags("<<abc>>>>") = ">>"
  ```

  A stack is necessary in modeling this function as an automaton to count the number of unmatched `<` symbols.

- There are operations not in the classes listed above: the class of context-free languages is closed for `strrev`, but not closed for the others [2].

The table shows that the class of context-free languages is closed for a large proportion of the functions in PHP and thus they can be eliminated from a grammar. The other kinds of functions are eliminated from a grammar by approximating it.

- Any pushdown transducer is conservatively approximated by a transducer that forgets the stack of the pushdown transducer.

- The output of some string operations is reasonably approximated by a regular expression. For example, the output of the function `md5` is approximated with the regular expression, `[0-9a-f]{32}`, representing 32-character hexadecimal numbers.

The second condition "No string operation occurs in a cycle of productions," is also crucial in obtaining a context-free grammar precisely approximating a grammar with operation productions. If we have a string operation in a cycle of productions, the language it generates may not be context-free. Let us consider the following grammar with the start symbol S.

$$\begin{array}{lcl} S & \rightarrow & 01 \\ S & \rightarrow & \texttt{str\_replace}(0, x0y, T) \\ T & \rightarrow & \texttt{str\_replace}(1, 1z, S) \end{array}$$

The language it generates is $\{x^n0y^n1z^n \mid n \geq 0\}$, which is not context-free. Therefore, we must approximate the language to obtain a context-free grammar for it. We adopted the same approximation as the Java string analyzer [7]: computing the set of characters Σ the language may contain and approximating the language with Σ^*. For the example above, we obtain the context-free language corresponding to $\{x, y, z, 0, 1\}^*$. This results in a very rough approximation, but this situation rarely occurs in a grammar extracted from a real PHP program.

5.3 Regular Expression Functions

Scripting languages such as Perl and PHP offer advantages in string manipulation by providing powerful string manipulation functions based on regular expressions. They are so powerful that the class of context-free languages is not closed for them. However, they can often be approximated with a combination of transducers with sufficient accuracy.

Let us consider how to model the following string operation.

[2]The functions `md5` and `sha1` can theoretically be represented with transducers. However, they require a huge number of states.

```
preg_replace("/a([0-9]*)b/",
             "x\\1y",
             $x)
```

This function replaces substrings matching `a([0-9]*)b` in `$x` with `x\\1y` where `\\1` is replaced with the string matching the first grouped subexpression `([0-9]*)`. The following example clarifies the operation.

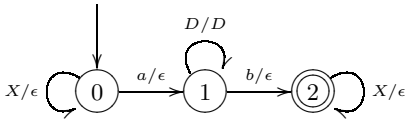
```
preg_replace("/a([0-9]*)b/",
             "x\\1y",
             "a01ba234b") = "x01yx234y"
```

A regular expression replacement function like this is approximated with a combination of transducers. To illustrate how a grammar is transformed by the transducers for the function above, let us consider the following grammar L with the start symbol Y.

$$\begin{array}{rcl} X & \rightarrow & \epsilon \mid X0 \\ Y & \rightarrow & aXb \mid b11a \end{array}$$

The grammar is transformed by transducers as follows:

1. Construct the grammar approximating the set of strings matching the pattern `[0-9]*` and occurring between `a` and `b`. It is done by the following transducer:



   where X is any character and D is any digit. The transducer is applied to L and then we obtain the following grammar with the start symbol X_1.

$$\begin{array}{rcl} X_0 & \rightarrow & X_0 0 \mid 0 \\ X_1 & \rightarrow & \epsilon \mid X_0 \end{array}$$

   This grammar corresponds to the regular expression 0^*.

2. Construct the grammar approximating the set of strings that the pattern is replaced with. It is obtained from the grammar above and the replacement string `x\\1y`.

$$\begin{array}{rcl} X_0 & \rightarrow & X_0 0 \mid 0 \\ X_1 & \rightarrow & \epsilon \mid X_0 \\ Z & \rightarrow & x X_1 y \end{array}$$

3. Construct the grammar representing the set of strings that are obtained by replacing the substrings matching with `a([0-9]*)b` in L with Z. Then we obtain the following grammar.

$$Y \rightarrow Z \mid b11a$$

   This transformation is done with two transducers used by Mohri and Sproat [17] to compile context-sensitive rewrite rules on strings.

Finally, we obtain the following grammar with the start symbol Y by combining the grammars in steps 2 and 3.

$$\begin{array}{rcl} X_0 & \rightarrow & X_0 0 \mid 0 \\ X_1 & \rightarrow & \epsilon \mid X_0 \\ Y & \rightarrow & x X_1 y \mid b11a \end{array}$$

This precisely represents the result of `preg_replace`. However, the context-free grammar obtained in this method is not precise in general, but an approximation.

Our analyzer dynamically constructs the transducers described above for a grammar with regular expression functions and translates it into a context-free grammar. To construct the transducers described above, the pattern strings such as `/a([0-9]*)b/` must be determined at first. However, the pattern strings might be dynamically constructed at runtime and not known statically. Thus, we apply our string analysis to the pattern string and approximate it with a finite set of strings [3]. Then, the method above is applied for each pattern string.

6. IMPLEMENTATION AND EXPERIMENTS

We have implemented a PHP string analyzer, which uses a context-free grammar to approximates Web pages generated by a PHP program. It can be used for the various kinds of checks we have described in this paper. The analyzer is implemented with the Objective Caml programming language [14] and contains about 10,000 lines of code. It consists of a library to manipulate formal languages including automata, transducers and context-free grammars, and the analyzer itself. The analyzer supports basic features of PHP, though not objects and references.

The analyzer first translates a PHP program into a functional intermediate language, which is basically equivalent to the static single assignment form. A grammar with string operations is extracted from it. In this phase, we use alias analysis to treat arrays. Finally, the context-free grammar approximating it is obtained by the methods described in Section 5.

The specification of inputs is given in the following format.

```
//  specification for variables
$x : [int]        // array of integers
$y : /abc|xyz/    // string : regular expression
                  // abc|xyz
$_POST : [/.*/]   // array containing any string

//  specification of the return value of functions
isset : bool
mysql_num_fields : int
mysql_field_name : /.*/
```

The specification consists of two parts: specification of variables and functions. For a variable, we can specify its type or a regular expression representing its value. For a function, we can only specify the return value of the function. No relationship between arguments and the return value can be specified in our current implementation.

We applied our string analyzer to several publicly available PHP programs. Table 6 summarizes our experiments: it shows the number of nonterminals and productions of the context-free grammar obtained by the analyzer, and the time spent to analyze each program. Experiments were done on a Linux PC with an AMD Athlon FX-53 processor (2.4 GHz) and 4 GByte memory.

[3]Currently, our analyzer fails if a pattern string argument cannot be approximated with a finite set of strings. It will be possible to adopt a rough approximation for that case.

Program	# lines	# nonterminals	# productions	Time (sec)
webchess	2224	300	450	0.36
schoolmate	8085	7985	9505	39.92
faqforge	843	180	443	0.16
phpwims	726	82	226	0.13
timeclock	462	656	1233	0.15
tagit	890	858365	6961180	4933.17

Table 2: Measurements: approximating outputs with a context-free grammar

In the programs, we found only one case where string operations occurred in a cycle of productions. The following is the simplified version of code found in the program tagit.

```
$a["abc"] = "ABC";
$a["xyz"] = "XYZ";
foreach($a as $key=>$value)
  $x = eregi_replace($key, $value, $x);
```

The variable `$x` appears on both left and right sides of the assignment with `eregi_replace`. The string operation `eregi_replace` thus appears in a cycle of productions extracted from the program. Hence, we can obtain only a very rough approximation for tagit with this code. The result in the table was obtained by commenting out this code.

The grammars obtained for the programs except tagit are not large and the analysis does not take much time. On the other hand, the analysis of tagit generates a huge context-free grammar. This comes from the following code:

```
$post = str_replace('[b]', '<b>', $post);
$post = str_replace('[/b]', '</b>', $post);
$post = str_replace('[i]', '<i>', $post);
$post = str_replace('[/i]', '</i>', $post);
...
```

where string replacement functions are repeatedly applied to the same variable 17 times. Each replacement grows a grammar by some factor and thus the factor by which the grammar is enlarged in total is exponential in the number of replacement functions applied. This exponential behavior appears in this program.

We also checked cross-site scripting vulnerabilities of the programs by the method described in Section 3. The programs other than tagit do not sanitize input at all and we could not find interesting bugs causing cross-site scripting vulnerabilities in those programs. In tagit, however, we found the following sanitization code.

```
$tagitname = strip_tags($tagitname);
$tagitname = stripslashes($tagitname);
```

The content of the variable `$tagitname` is sanitized. However, this variable is not used in the other files at all and instead the variable `$tagitnames` is used there. This causes a cross-site scripting vulnerability.

We have also checked the validity of the pages generated by the programs and found that all of the programs may generate invalid HTML pages.

- For most of the bugs, the required end tag matching a start tag is omitted or the nesting of tags is invalid. Although most of them may easily be found with a standard validation tool, some of them may be overlooked.

  For example, we found the following bug in webchess. If the `echo` function is executed, the code generates an invalid page. The code should generate matching `<p>` and `</p>`.

  ```
  if ($errMsg != "")
    echo("<p><h2><font color='red'>".
         $errMsg."</font></h2><p>\n");
  ```

  This `echo` function is executed only when some error occurs and thus the bug might be overlooked with tests by a standard validation tool.

- We also found some false positive alarms in HTML validation. The following is a simplified example of code that causes a false alarm.

  ```
  if ($x != 0) echo '<b> nonzero';
  if ($x == 0) echo '<b> zero';
  echo '</b>';
  ```

  Our analyzer currently ignores the conditions in `if`-statements. It is therefore impossible to find that exactly one of the bodies of the `if`-statements is executed and thus it generates a string with the matching `b` tag. This kind of false positive alarm was found only in the program schoolmate.

Finally, we applied the matching validation described in Section 4.2 to the programs. The results are shown in the following table. The programs were revised so that start and end tags always matched: the column "Bugs" in the table shows the number of bugs found in this revision and the column "Opt tags" shows the number of optional tags inserted so that start and end tags match. The table also shows that the maximum depth of pages the programs may generate and the time spent on checking.

Program	Depth	Bugs	Opt tags	Time (sec)
webchess	9	1	6	123.33
faqforge	10	30	0	45.64
phpwims	9	7	0	63.93
timeclock	14	11	0	145.61
schoolmate	17	14	3	7580.69

Table 3: Matching validation

Some PHP programs are not designed to generate a valid HTML page when the program stops its execution with the explicit exit commands: `exit` or `die`. Our analyzer has an

option not to analyze output if a program stops its execution with an explicit exit command. The programs timeclock and schoolmate were checked in this mode.

The time spent on checking is large compared to the time spent obtaining a grammar approximating the output of a program. This is because a large automaton is constructed for each depth and the computationally expensive algorithm of checking disjointness between a regular language and a context-free language is applied. The complexity of the algorithm is $O(m^3n^3)$, where m is the number of terminals and nonterminals in the grammar, and n is the number of states of the automaton [15]. The program tagit was excluded from this experiment because it is a bulletin board program where a user can write a text with some tags on the board, and then a page that it generates may have an arbitrary depth and be an invalid HTML text.

7. RELATED WORK

7.1 String Analysis

Christensen, Møller and Schwartzbach developed a string analyzer for Java, which approximates the value of a string expression with a regular language [7]. The approximation can be checked against the specification of expected string values embedded in the Java program. The key phase in their analysis is a transformation developed by Mohri and Nederhof [16] approximating a context-free language with a regular language. Our string analyzer differs from theirs in the following ways. Firstly, our analysis is based on context-free languages instead of regular languages and thus our approximations can be more precise. The analysis is also simplified by removing the transformation of Mohri and Nederhof. Secondly, our analyzer approximates the whole output of a program representing a Web page and thus can be used to check properties of a dynamically generated Web page as a whole.

Syntax checking based on string analysis was extended to type checking by Gould et al. in the context of SQL query strings [10]. They developed a system that guarantees that SQL query strings generated by a Java program at runtime are well typed. Their system is based on the Java string analyzer of Christensen, Møller and Schwartzbach. The context-free reachability algorithm is used for type checking. Their method can be incorporated into our analyzer.

It is possible to formulate string analysis as a type system. A type system based on regular expressions was studied by Tabuchi et al. for a minimal functional language with string concatenation and pattern matching over strings [23]. By adopting regular expressions as types, they could include rich operations over types in their type structure, and that made it possible to capture precisely the behavior of pattern matching over strings in their type system. However, a full type inference algorithm was not given in the paper. We have not considered pattern matching in our analyzer since PHP does not have it. It will be interesting to extend our analysis for pattern matching. The extension is not straightforward because our analysis is based on context-free grammars.

Recently, Thiemann presented a type system for string analysis based on context-free grammars and presented a type inference algorithm based on Earley's parsing algorithm [24]. The type inference is incomplete as our matching validation. In his type system, a program that may generate a document with an arbitrary depth can be type-checked against a context-free grammar. However, the grammar must be carefully crafted and a program must be written so that it is checked against it. The type system was designed for an applied lambda calculus with string concatenation, and it was not discussed how to deal with string operations other than concatenation.

7.2 Domain Specific Languages for HTML and XML

Extensive studies have been done in the area of designing domain specific languages for HTML and XML [13, 5, 6, 11, 3]. One of the key issues there is to guarantee validity of dynamically generated documents.

Ladd and Ramming proposed the MAWL language [13] where an HTML document is constructed from templates. A template is an HTML document with named gaps where a string is inserted. This approach was extended to higher-order templates and adopted in the `<bigwig>` and JWIG projects [5, 6]. To guarantee the validity of the operation filling a gap with a string, JWIG applies a simple form of string analysis.

Hosoya and Pierce designed a statically typed XML processing language called XDuce based on the theory of finite tree automata [11]. Finite tree automata have good mathematical properties including that it is decidable whether the language accepted by one tree automata is included in that accepted by another. The type system of XDuce has similarities with string analysis in the sense that it is based on the theory of formal languages. However, XDuce does not directly work on strings, but on trees representing XML documents.

7.3 Trust Checking

Vulnerabilities in a server-side program such as cross-site scripting and SQL injection are caused by applying critical operations to untrusted (or tainted) data submitted by a user. It is possible to avoid such potential unsafe operations by classifying data into trusted and untrusted data. Perl's taint mode keeps track of this classification at runtime and ensures that no untrusted data is used in potentially unsafe operations [26]. Program analysis to statically classify data was presented by Ørbæk [18] and Ørbæk and Palsberg [19]. This analysis was extended and adopted by the static analysis for PHP by Huang et al. [12]. These analyses are closely related to information flow analysis [8, 9] and can be considered as its duals.

A key concept in trust checking is conversion from untrusted to trusted data. Intuitively, sanitization functions correspond to this conversion in server-side programming. However, sanitization is done in a great variety of ways in practice and it is difficult to determine what are sanitization beforehand. On the other hand, security checks based on string analysis are more declarative: the key concept in this case is what are safe (or unsafe) strings.

8. CONCLUSION AND FUTURE WORK

We have designed and implemented a program analyzer that approximates the output of a program with a context-free grammar. The analyzer was successfully applied to publicly available PHP programs to detect cross-site scripting vulnerabilities and to validate pages dynamically generated

by them. We are planning to extend our analyzer to support the full features of PHP and to evaluate the analyzer further for a wider range of PHP programs.

The theory of context-free languages and transducers has a significant role in our analysis. The grammar obtained by the analysis is checked against a specification by checking the disjointness of regular and context-free languages. Transducers are used to model and approximate string operations in PHP, and to transform the grammars obtained by the analysis for applications.

The precision of the approximation will be improved with techniques available in compilers and other program analysis. We are planning to combine symbolic evaluation and more precise alias analysis to improve approximations.

9. ACKNOWLEDGMENTS

We would like to thank Nobuo Otoi for developing the formal language library used in our analyzer. This work has been supported by CREST of JST and the Kayamori Foundation of Informational Science Advancement.

10. REFERENCES

[1] M. Achour, F. Betz, et al. *PHP Manual*, 2005. http://www.php.net/docs.php.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63, 2003.

[4] J. Berstel. *Transductions and Context-Free Languages*. Teubner Studienbucher, 1979.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

[7] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18, 2003.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of ACM*, 19(5):236–243, 1976.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of ACM*, 20(7):504–513, 1977.

[10] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, 2004.

[11] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[12] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, pages 40–52, 2004.

[13] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th International World Wide Web Conference*, 1995.

[14] X. Leroy. *The Objective Caml system release 3.08: Documentation and user's manual*, 2004. http://caml.inria.fr/index-eng.html.

[15] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoretical Computer Science*, 248(1–2):29–98, 2000.

[16] M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, pages 153–163, 2001.

[17] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *34th Meeting of the Association for Computational Linguistics (ACL '96), Proceedings of the Conference*, 1996.

[18] P. Ørbæk. Can you trust your data? In *Proceedings of the 6th International Conference on Theory and Practice of Software Development: TAPSOFT*, volume 915 of *LNCS*, pages 575–590, 1995.

[19] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[20] L. Quinn. WDG HTML validator. http://www.htmlhelp.com/tools/validator/.

[21] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 2000.

[22] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[23] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In *Proceedings of International Workshop on Types in Programming*, ENTCS 75, 2002.

[24] P. Thiemann. Grammar-based analysis of string expressions. In *Proceedings of the ACM SIGPLAN-SIGACT Workshop on Types in Language Design and Implementation*, pages 59–70, 2004.

[25] The W3C markup validation service. http://validator.w3.org/.

[26] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl (3rd Edition)*. O'Reilly, 2000.