Web Application Slicing in presence of Dynamic Code Generation

PAOLO TONELLA tonella@itc.it
FILIPPO RICCA ricca@itc.it
ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, Trento, Italy

Abstract. The computation of program slices on Web applications may be useful during debugging, when the amount of code to be inspected can be reduced, and during understanding, since the search for a given functionality can be better focused. The system dependence graph is an appropriate data structure for slice computation, in that it explicitly represents all dependences that have to be taken into account in slice determination.

Construction of the system dependence graph for Web applications is complicated by the presence of dynamically generated code. In fact, a Web application builds the HTML code to be transmitted to the browser at run time. Knowledge of such code is essential for slicing. In this paper an algorithm for the static approximation of the dynamically generated HTML code is proposed. The concatenations of constant strings and variables are propagated according to special purpose flow equations, allowing the estimation of the generated code and the refinement of the system dependence graph.

Keywords: Program Slicing, Web Applications, System Dependence Graph.

1. Introduction

Program slicing is a static analysis technique for the extraction of the program statements relevant to a specified computation. Program slicing was introduced for the first time by Weiser (Weiser, 1984) and has now many applications such as debugging, code understanding, program testing, reverse engineering, software maintenance, reuse, software safety and metrics (Binkley and Gallagher, 1996; Cimitile et al., 1995; Gallagher and Lyle, 1991; Gallagher, 1996; Gupta et al., 1996; Harman and Danicic, 1995; Kamkar et al., 1993; Tip, 1995; Weiser, 1982). Two excellent surveys on program slicing are (Binkley and Gallagher, 1996; Tip, 1995).

A Web application consists of a set of static HTML pages displayed to the user and (possibly) of server side programs (e.g., written in PHP), which perform some computation, finally resulting in the production of dynamic pages transmitted to the browser for display. Moreover, HTML pages may embed client procedures which are either executed during page loading or in response to graphical interface events.

Web application slicing (Ricca and Tonella, 2001) is conducted on an extension of the program representation used with traditional software, the System Dependence Graph (SDG) (Horwitz et al., 1990). Its basic elements are similar to those introduced in (Liu et al., 2001) for data flow testing, and are described extensively in (Ricca and Tonella, 2002), where Web specific dependences and nodes are defined. Call dependences are categorized either as returning or non-returning, and graphical interface events and event loop are explicitly represented in the SDG.

Web applications comprising dynamically generated pages are remarkably more difficult to analyze than static Web sites. The server script that is executed when a page is requested builds the HTML structure of the resulting page at run time. The tags inserted into the page can, in general, depend on the state of the program and on the input values. Moreover, even the names of the HTML variables in the generated page, the `FORM` actions and the hyperlinks can be constructed dynamically, and vary from execution to execution. If the SDG is constructed with no regard to the generated code, important nodes and edges are possibly missed. For example, when tags are produced dynamically, the related nodes are not in the SDG built for the server script. When variable names are generated dynamically, the data dependences associated to them are absent in the SDG. When `FORM` actions are inserted dynamically in a page, the related call dependences are not reported in the SDG.

The problem of statically constructing a precise SDG for a dynamic Web application is in general undecidable. This means that there exist Web applications for which it is not possible to build a precise SDG just by analyzing the source code, because the dynamically generated SDG nodes cannot be approximated statically. However, under reasonable assumptions it is possible to devise static analysis techniques that are able to precisely produce the SDG for a subset of all possible Web applications, including most of the real world ones. This is the main topic of this paper.

Another completely different approach to the problem of determining the dynamically constructed SDG portions relies on *dynamic slicing*. When a specific execution of a Web application is considered, with given input values, the problem of approximating the generated HTML code disappears, since this can be obtained by actually running the application. Thus, slices can be computed directly on the generated code, which is known. However, resulting slices hold only for the input values used in the execution. Some of the excluded portions of the Web application could be relevant to the selected computation for different inputs. This is why we focus on static slicing.

In this paper, a technique is proposed for the static construction of the SDG for Web applications in presence of dynamic code generation. The code produced at run time is "extruded" from the statements that generate it. However, code extrusion is not sufficient when the generated code is accumulated into a string variable before being printed out. A flow analysis algorithm, called *string-cat propagation*, is proposed to handle such cases. Combined with code extrusion, it allows treating several programming patterns commonly used with Web applications, such as string variables, where the HTML code is temporarily stored, and general purpose functions, that are parameterized at invocation in order to generate the desired HTML fragment.

Preliminary experimental results have been obtained by applying the string-cat propagation and code extrusion techniques to some existing Web applications. Then, Web slices have been computed. The ratio between the size of the slice and the size of the original Web application has been used as an indicator of the potential of Web slicing to restrict the search space during program understanding. Moreover, one example of Web slice has been studied in detail, by considering the sliced source code and the results produced by slice execution.

The remainder of this paper is organized as follows: Section 2 summarizes the main elements in the SDG of a Web application. The algorithms for code extrusion and string-cat propagation are presented in Section 3. Some examples of PHP programs are analyzed in Section 4. Conclusions are drawn in Section 5.

2. Web application slicing

According to its original definition (Weiser, 1984), a *program slice* is a reduced, executable program obtained from a given program by removing statements, so that it replicates part of the behavior of the initial program. The main requirements in this definition are that a slice be still a legal program and that its behavior be preserved with respect to a computation of interest.

Similar constraints are enforced in introducing the notion of slice for Web applications. Slicing a Web application results in a portion of the Web application which exhibits the same behavior as the initial Web application in terms of information of interest displayed to the user, programs generating it (in case it is dynamic) and interaction with the user (client side code).

Definition 1: *a* Web application slice *is obtained from a given set of Web pages and server programs by removing HTML and server/client*

code statements, so that part of the behavior of the initial Web application is replicated.

The criterion for slice computation is an information item of interest displayed in a given Web page, and the resulting slice reproduces the same information item, if the user performs the same navigation actions and provides the same input in the original and in the sliced Web application. Formally, a slicing criterion (n, x) consists of a statement n, displaying the information item of interest, and a variable x, used for the production of such an information item.

Several server side languages are available for the construction of a Web application (e.g., PHP, Java, Perl, VBscript, etc.). The same is true for the client side code (e.g., Java, Javascript, etc.). In this paper, we will consider PHP for the server and Javascript for the client. Such a choice is by no means restrictive, since these languages offer all the basic features available in the others.

2.1. Overview

```
SLICE(n: SDGNode): Set<SDGNode>
1   S ← {n'| a data dep. holds between n' and n }
2   while S increases
3       if a control/data/transitive dep. holds between n' and n ∈ S
4           S ← S ∪ {n'}
5       endif
6       if a parameter-in dep. holds between n' and n ∈ S,
            and the associated call dep. starts at n''
7           S ← S ∪ {n', n''}
8       endif
9   endwhile
10  while S increases
11      if a control/data/parameter-out/transitive dep. holds between n' and n ∈ S
12          S ← S ∪ {n'}
13      endif
14  endwhile
15  return S
```

Figure 1. Pseudocode of a slicing algorithm based on the computation of the backward reachable nodes in the SDG.

Program slices can be computed by incrementally adding consecutive sets of transitively relevant statements. Statements that directly affect the computation at another statement are connected to the latter by a dependence relation, which can be explicitly represented in a graph called System Dependence Graph (SDG) (Horwitz et al., 1990). Different kinds of dependences are distinguished in the SDG (control, data, call, parameter-in, parameter-out, etc.) and the problem of static slicing can be restated in terms of a two-step reachability problem in the SDG (Horwitz et al., 1990). Figure 1 gives the pseudocode of such an algorithm. The slicing criterion (n, x) is simplified into n, under the

(non restrictive) assumption that every SDG node uses at most one variable. The two steps are respectively focused on all dependences, including calling functions, but excluding called functions (step 1, lines 2-9), and on all dependences, including called functions, but excluding calling functions (step 2, lines 10-14). This allows the algorithm to be calling context sensitive (see (Horwitz et al., 1990)), but requires the availability of transitive dependences. The exact meaning of the involved dependences (except for the transitive dependences) will become clear in the next sections, where they are presented in more detail. The interested reader can refer to (Horwitz et al., 1990) for more information about the transitive dependences and the rationale behind the two step computation of the algorithm. A one step algorithm, computing the transitive closure over all dependences, is applicable if all invocations are no-return invocations. This is quite typical for Web applications, where the control is transferred from page to page and does not get back to the previous page. However, usage of server side languages which support the full spectrum of function calls (e.g., PHP) might make the two steps necessary.

The program dependences that are represented in the SDG can in turn be computed by means of proper flow analysis algorithms that work on the Control Flow Graph (CFG). A Web application consists of a set of server side programs, generating HTML pages dynamically, and of a set of fixed HTML and client side code. The CFGs are built for all procedures in the server/client side programs. Then, flow analysis is conducted on such CFGs to determine data dependences, control dependences, etc., and finally the SDG is built and used for slicing. The flow analysis algorithms proposed in Section 3 to handle dynamic HTML code generation work on the CFG, similarly to the other flow analyses. Their output is used to construct an accurate SDG.

2.2. Nesting, data and call dependences

Control dependences for traditional programs connect the predicate tested at a conditional or loop statement to the instructions the execution of which directly depends on the truth value of the predicate. This continues to hold for the server/client side code, but Web applications are characterized by an additional kind of dependence, the nesting dependence, which holds between an HTML tag and all directly enclosed statements.

Definition 2: *a* nesting dependence *holds between two HTML statements if the latter is nested inside the former.*

Data dependences for traditional programs hold when a statement defines the value of a variable, which is used at another statement after being propagated to it along a *definition clear path*: a path containing no redefinition of the given variable. In addition to such a situation, which can still be found in server/client code, data flows may occur in a Web application from server/client side statements to the HTML code. On the other side, a variable definition at an HTML statement can reach a server/client instruction only through a procedure invocation, as a submission parameter, or through a DOM (Document Object Model) element access.

Definition 3: *a* data dependence *holds between two statements if the former defines the value of a variable which is used by the latter, and a definition clear path exists between the two.*

When a server side program is invoked from an HTML statement (e.g., via form submission or `HREF`), control of execution is transferred to the server and never returns back to the Web page issuing the call. In other words, a server program invoked from an HTML statement cannot produce any effect on the variables of the calling page, since the invocation is a no-return invocation. A consequence is that the *calling context problem*, i.e., the problem of keeping the data flows associated to the different call sites separated, is absent, since call sites are not affected by the invocation.

Calls issued within server (or client) code, such as from a PHP procedure to another PHP procedure (or from a Javascript function to another Javascript function), similarly to traditional software, are subject to the calling context problem (see previous section for an algorithm handling such cases).

Mouse or keyboard events occurring during or after page loading can trigger the execution of a client side procedure, can produce the loading of another page (hyperlinks), or the submission of a form to a server side program. The graphical user interface of the browser handles such events by means of a so called *event loop*, which dispatches the events as soon as they occur. The event loop is activated at the beginning of page loading.

Invocations within the event loop can be issued before page loading is complete, if the user clicks on some interface widget while page loading is in progress. This means that it is not possible to assume that all HTML and client statements, interpreted during page loading, have been encountered when a callback is activated. The consequence in the construction of the SDG is that data dependences on callback parameters need be computed in a non-killing (flow insensitive) way. That is, when a statement redefines the value of a variable, it is not

possible to consider the previous definitions as invalid for the callback parameters, since execution could be interrupted by the user before the invalidating statement is loaded, and the actual parameter value is the previous one. The visible result in the SDG consists of some extra data dependence edges, which end at parameter-in nodes, and are due to the non-killing nature of the related statements.

Definition 4: (1) *a* call dependence *holds between each statement of type* call *and the server/client program or procedure invoked. (2) A* parameter-in *dependence holds between any actual parameter of a call and the respective formal parameter of the invoked program or procedure. (3) A* parameter-out *dependence holds between any value returned from an invocation and the related variable in the call site.*

Output parameters are necessary only when the invocation returns to the call site, where the effect of the call has to be propagated. On the contrary, for no-return invocations (such as those associated with `FORM` submissions and hyperlinks) only input parameters (and parameter-in dependences) have to be created.

aa.php:

```
1  <?
2  z = "aa";
3  ?>
4  <FORM method="POST" action="bb.php">
5      <INPUT TYPE="Submit"><BR>
6      <INPUT TYPE="Text" NAME="x" VALUE="<?print $z?>"><BR>
7      <INPUT TYPE="Text" NAME="y" VALUE="<?print $z?>"><BR>
8  </FORM>
```

bb.php:

```
9  <?
10 $xx = $x;
11 $yy = $y;
12 ?>
```

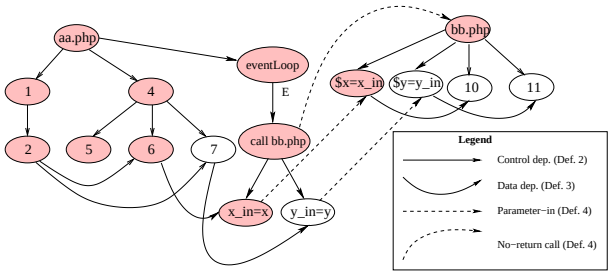


Figure 2. Fragment of PHP/HTML code and associated control, nesting and data dependences. Call/parameter-in dependences are depicted with dashed lines. Grayed nodes/statements belong to the slice at 10 on `$x`.

Nesting/control dependences (Def. 2) for the HTML/PHP code in Figure 2 are indicated by solid, straight lines. For example, nesting of the three `INPUT` tags at lines 5, 6, 7 inside the `FORM` at line 4 is

represented by the three edges from node 4 to nodes 5, 6, 7. Data dependences (Def. 3) are indicated by solid, curved lines. The value of `$z` defined at statement 2 is used at statements 6 and 7, and no intermediate redefinition occurs, so that two data dependences connect node 2 with 6 and 7. The input gathered at statements 6 and 7 determines the values of the parameters `$x` and `$y`, passed to the script `bb.php`, which is the action of the `FORM` at statement 4. In fact, such a `FORM` installs `bb.php` as the response to a click on the `Submit` button. Consequently, the event loop of this page contains only one call, triggered by the click event (indicated generically with `E`). Since this is a no-return invocation, it is indicated with a curved dashed line. Straight lines are used for normal procedure invocations. Two input variables are submitted by the form and accessed by the server program invoked: `x` and `y`. They are associated with two parameters (`x_in` and `y_in`), and with two parameter-in dependences (Def. 4), depicted with straight dashed lines in the figure.

A slice computed at node 10 on variable `$x` results in the nodes/statements with gray background in Figure 2. The "Submit" button (node 5) is required by the form at node 4.

2.3. The problem of dynamic code generation

Figure 3 shows an example where the HTML code of the dynamic page produced by the execution of the PHP code has not a fixed structure. There are some fixed parts, the form opened at line 1 and closed at line 8, and the submit button created at line 7, but the input text fields of the form are created dynamically at lines 2-6, within a loop iterating a number of times, `$N+1`, which is also computed at run time (not shown). The names of the variables associated with these inputs are constructed dynamically from the loop index (`x0, ..., xN`). Their initial value is equal to the loop index (`0, ..., N`). Consequently, the number of parameters passed to the server program `bb.php` is variable, function of `N`. On the server side, the program `bb.php` constructs dynamically the names of the parameters from `$N`, assigning them to the array `$z` (line 15). Then, parameter values are retrieved (line 16) and accumulated in variable `$sum` (line 17).

Construction of the SDG for this code fragment is remarkably difficult, due to the dynamic generation of a part of the code itself, namely the input elements of an HTML form and the parameters of a server program. Figure 3 shows what an "ideal" code analyzer is expected to generate, that is, a variable number of parameters (ellipsis) with

```
aa.php:
1   <FORM method="POST" action="bb.php">
    ...
2   <? for ($i=0; $i<=$N; $i++) {
3       $y[$i]="x" . $i;
4       $v[$i]=$i; ?>
5       <INPUT TYPE="Text" NAME="<? print $y[$i] ?>" VALUE="<? print $v[$i] ?>"> <BR>
6   <? } ?>
7   <INPUT TYPE="Submit"><BR>
8   </FORM>
    ...

bb.php:
12  <?
13  $sum = 0;
14  for ($i=0; $i<=$N; $i++) {
15      $z[$i]="x" . $i;
16      $v[$i]=$HTTP_POST_VARS[$z[$i]];
17      $sum=$sum+$v[$i];
18  } ?>
```

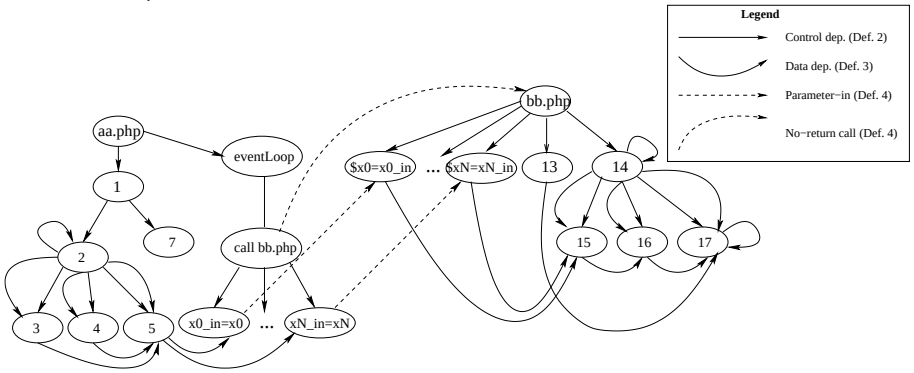


Figure 3. Generation of HTML code from PHP.

an associated set of data and parameter-in dependences. In its most general form, the problem is that the code under analysis generates the code actually interpreted by the browser. Such a generation process includes constant strings (such as `" VALUE="`), concatenated with strings computed dynamically. Consequently, the code generation instructions (lines 1, 5, 7, 8 of Figure 3) build the strings that are the real object of the analysis. For a static analysis of the source, the variable parts of such strings have to be reduced to constants, whenever possible. The algorithms that can be used for this purpose are the object of the next section. A possible alternative to handle dynamic code generation is resorting to *dynamic* slicing. The related algorithm has been described in (Ricca and Tonella, 2002). The main disadvantage of this second solution is that the result of dynamic slicing holds only for the selected input values (and for the related execution). On the contrary, static slicing provides a result which is valid for all input values and for all executions. This justifies the investigation of more complex algorithms, able to handle dynamic code generation statically.

3. Dynamic code generation

```
aa.php:
1   <HTML>
2   <FORM method ="POST" action ="universal.php">
3       <TEXTAREA name ="x" rows="40"cols="80"> </TEXTAREA>
4       <INPUT type ="submit">
5   </FORM>
6   </HTML>

universal.php:
7   <?
8       print $x;
9   ?>
```

Figure 4. Universal HTML code generator: any HTML page can be generated by this PHP script.

The problem of statically determining the HTML code generated dynamically by a Web application is in general undecidable. Consequently, it is in general impossible to build an accurate SDG for a Web application that generates some HTML code at run time. Figure 4 shows the worst possible case. The PHP script `universal.php` can produce any HTML page in output. In fact, this script just prints the value of its input variable `$x`, and the resulting string is sent to the browser. Since `$x` holds the text provided in input inside a text-area (line 3), any HTML code can be contained in it, provided that the user inserts it into the text-area.

3.1. Code generation patterns

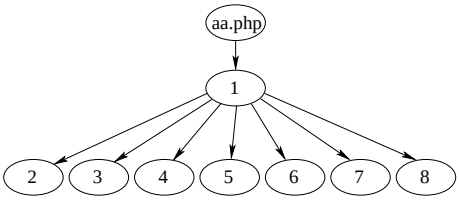
Since it is not possible to determine the HTML code generated by a PHP script in the general case (see Figure 4), the typical patterns of code generation are considered and a technique to handle them is presented. They have been determined by manually inspecting a set of Web applications sampled from those available on the Internet.

3.1.1. *Code extrusion*

Figure 5 (top) shows a PHP script that generates the output HTML code by means of `print` statements (`echo` is another possibility), where the printed strings are either constant strings (lines 2, 3, 6, 7, 8) or the concatenation of constant strings and variable values (lines 4, 5). The SDG associated to this code fragment contains a flat sequence of nodes, corresponding to the `print` statements. The control and data dependences of the generated HTML code are missing, and the invocation of `bb.php` inside the `FORM` at line 3 is also absent. This results in an inaccurate SDG (see Figure 5, top), which may produce wrong slices.

aa.php:

```
1   <?
2       print "<HTML>\n";
3       print "<FORM method = \"POST\" action = \"bb.php\">\n";
4       print "      <INPUT name = \"x\" value = \"" . $x . "\">\n";
5       print "      <INPUT name = \"y\" value = \"" . $y . "\">\n";
6       print "      <INPUT type = \"submit\">\n";
7       print "</FORM>\n";
8       print "</HTML>\n";
9   ?>
```



aa.php: generated code

```
10  <HTML>
11  <FORM method = "POST" action = "bb.php">
12      <INPUT name = "x" value = "<? print $x ?>">
13      <INPUT name = "y" value = "<? print $y ?>">
14      <INPUT type = "submit">
15  </FORM>
16  </HTML>
```

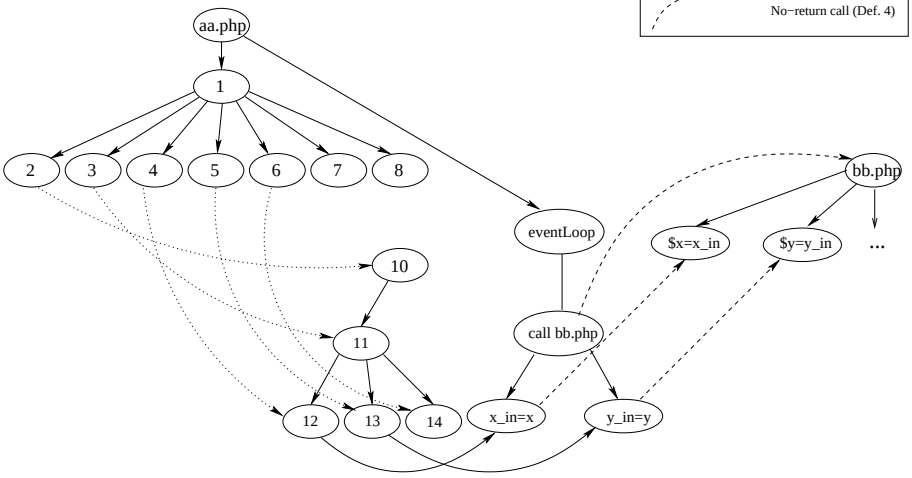


Figure 5. Code generation in the case of constant strings (possibly concatenated with variables).

The accurate SDG of this program can be obtained if a preliminary code transformation is applied to it, which we name *HTML code extrusion*. The *HTML code extrusion* consists of the replacement of each `print` or `echo` statement in the PHP code with the unquoted version of the printed strings, in case of constant strings. If variables are also

concatenated, the concatenation is replaced by a `print` within PHP delimiters. Unquoting the constant strings requires some minor lexical adjustments (such as mapping \" into ").

Figure 5 (bottom) shows the result of code extrusion. The related SDG now contains a *call* node linked to `bb.php`, as well as nodes for the actual parameters and edges for the data dependences terminating at the call parameters. The nesting dependences of the generated HTML code are also present.

Since slicing is conducted on the original program, when code extrusion is applied, the mapping between original and extruded statements has to be kept. To this aim, an additional dependence is introduced in the SDG, the *build dependence.*

Definition 5: *a* build dependence *holds between a server program statement and an HTML statement if the former generates the latter.*

During slice computation, build dependences are traversed backward similarly to the other dependences. In this way, the statements in the original program responsible for generating the HTML statements included in the slice will be also part of the slice.

With reference to the example in Figure 5, the following build dependences are represented in the SDG: $(2, 10)$, $(3, 11)$, $(4, 12)$, $(5, 13)$, $(6, 14)$, associated with the `print` statements in Figure 5 that produce the nodes that are the targets of the dependences. Generation of the close tags is considered implicitly, since these are not represented in the SDG. The related statements are included in a slice if the corresponding open tags are also in the slice.

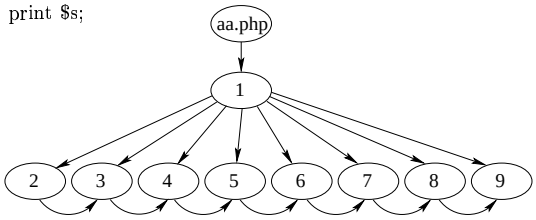
3.1.2. *String-cat propagation*

Figure 6 (top) contains an alternative implementation of the program in Figure 5, on which the HTML code extrusion is completely ineffective. In fact, the output of code extrusion is identical to the original program, and no simplification can take place. Still, the related SDG is inaccurate. Data dependences associated to the variable `$s` are properly represented, but HTML control dependences, as well as call dependences are missing.

In order to handle this variant, which is often used in practice, it is possible to resort to an extension of the copy-constant propagation algorithm (Aho et al., 1985), known from the field of flow analysis. We name such an analysis *string-cat propagation*, and its detailed description is delayed until the next subsection. The output of this analysis pairs program variables with string-concatenations (*string-cat*) that are possibly assigned to them. In the case of the code in Figure 6, the string-cat propagation algorithm produces the following pair at the node 9,

aa.php:

```
1   <?
2       $s = "<HTML>\n";
3       $s .= "<FORM method = \"POST\" action = \"bb.php\">\n";
4       $s .= "     <INPUT name = \"x\" value = \"" . $x . "\">\n";
5       $s .= "     <INPUT name = \"y\" value = \"" . $y . "\">\n";
6       $s .= "     <INPUT type = \"submit\">\n";
7       $s .= "</FORM>\n";
8       $s .= "</HTML>\n";
9       print $s;
10  ?>
```



aa.php: generated code

```
11  <HTML>
12  <FORM method = "POST" action = "bb.php">
13      <INPUT name = "x" value = "<? print $x ?>">
14      <INPUT name = "y" value = "<? print $y ?>">
15      <INPUT type = "submit">
16  </FORM>
17  </HTML>
```

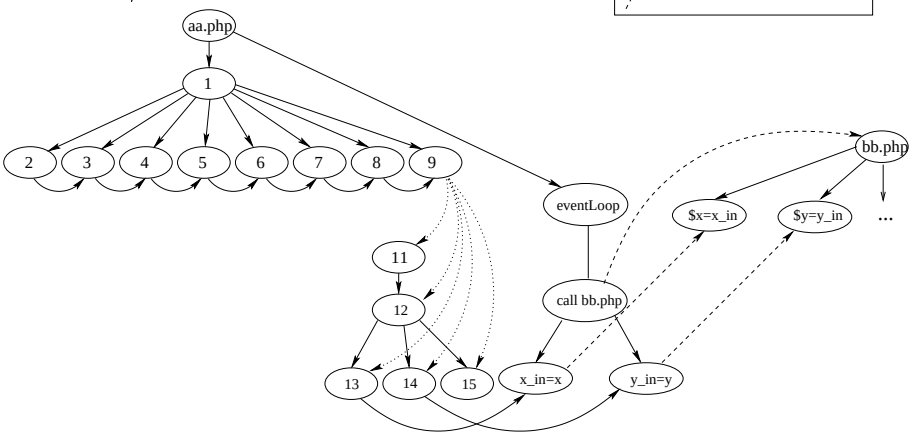


Figure 6. Code generation with string accumulation in a variable. Statements of the kind `$s .= "aa"` *are equivalent to* `$s = $s . "aa"`.

where `$s` is printed:

```
($s, "<HTML>\n<FORM method = \"POST\" action = \"aa.php\">\n
   <INPUT name = \"x\" value = \"" . $x . "\">\n
   <INPUT name = \"y\" value = \"" . $y . "\">\n
   <INPUT type = \"submit\">\n </FORM>\n</HTML>\n").
```

By replacing `$s` with the paired *string-cat* inside the `print` statement at line 9, a PHP statement is obtained on which the HTML code extrusion can be successfully applied. The resulting program is provided in Figure 6 (bottom), along with the associated SDG.

Data and call dependences are properly represented in the SDG, as well as the HTML nesting dependences. Since nodes 11, 12, 13, 14, 15 have been produced by expanding `$s` in the print statement (line 9 in Figure 6) and extruding the HTML code, inclusion of any of these nodes in a slice will imply the inclusion of node 9 of Figure 6. This is accounted for by the build dependences (Def. 5) that connect node 9 to nodes 11, ..., 15.

Figure 7 (top) shows another possible variant, which is also quite relevant, representing one of the typical programming patterns used in practice. Generation of the string to be printed is delegated to a separate function, `f`, invoked at the `print` statement (line 13).

This case can be still handled by means of string-cat propagation, once a parameter-out location is introduced for the value returned by the called function. We name such a location `f_out` for the function `f` invoked by the code in Figure 7. The return statement at line 10 results in a value of `f_out` equal to the final value of `$s` (after statement 9), which can be represented by the same *string-cat* already given for the example in Figure 6. Correspondingly, the SDG that can be constructed after HTML code extrusion is that in Figure 7 (bottom), inclusive of build dependences.

3.2. String-cat propagation algorithm

As described in the previous section, the code extrusion transformation can be successfully applied only after performing a preliminary computation of the string concatenations (*string-cat*) that are printed out. In this section, the details of the string-cat propagation algorithm are provided. The algorithm is an instantiation of the general flow analysis framework (Aho et al., 1985).

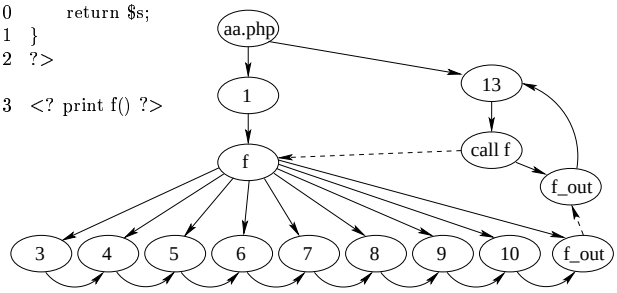
The purpose of string-cat propagation is associating the variables used in `print` statements to string concatenations. Function calls in `print` statements can be handled by introducing a fictitious location associated with the `return` statements in the called function.

We will first define the flow information that is propagated in the program's CFG, and then we will give the equations used for flow propagation. Finally, we will describe how alternatives and loops are handled.

aa.php:

```
1   <?
2   function f() {
3       $s = "<HTML>\n";
4       $s .= "<FORM method = \"POST\" action = \"bb.php\">\n";
5       $s .= "     <INPUT name = \"x\" value = \"" . $x . "\">\n";
6       $s .= "     <INPUT name = \"y\" value = \"" . $y . "\">\n";
7       $s .= "     <INPUT type = \"submit\">\n";
8       $s .= "</FORM>\n";
9       $s .= "</HTML>\n";
10      return $s;
11  }
12  ?>

13  <? print f() ?>
```



aa.php: generated code

```
14  <HTML>
15  <FORM method = "POST" action = "bb.php">
16      <INPUT name = "x" value = "<? print $x ?>">
17      <INPUT name = "y" value = "<? print $y ?>">
18      <INPUT type = "submit">
19  </FORM>
20  </HTML>
```

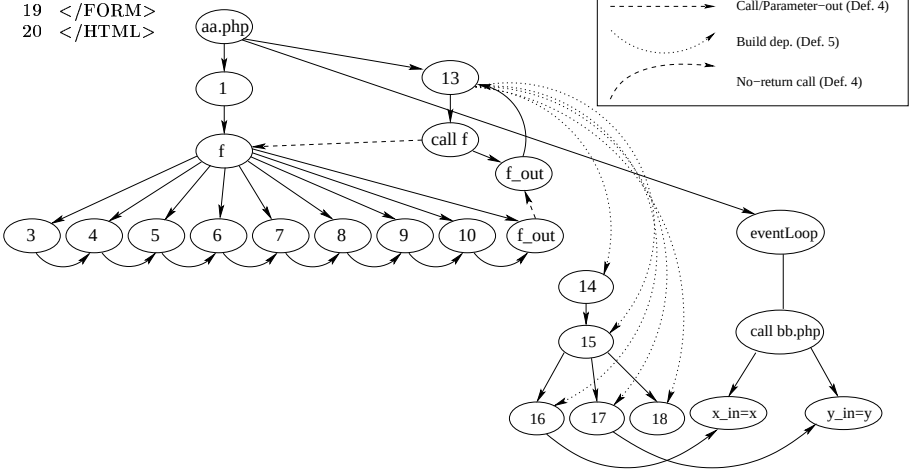


Figure 7. Code generation with function invocation.

3.2.1. *Flow information*

A string concatenation, *[stringcat]*, can be defined recursively as follows:

[stringcat] ::= *[stringlit]*
| *[stringexpr]*
| *[stringcat]* . *[stringcat]*
[stringexpr] ::= *[var]*
| *[funcall]*

A *[stringcat]* can be a constant string (*[stringlit]*) or an expression evaluating to a string (*[stringexpr]*) such as a variable holding a string, *[var]*, or the value returned by a function, *[funcall]*). Moreover, the concatenation of two *[stringcat]*s is still a *[stringcat]*.

The string-cat propagation algorithm aims at associating variables to string concatenations. Correspondingly, the flow information to be propagated in the program's CFG consists of pairs relating variables to the string-cat's they (possibly) hold:

$$FlowInfo = \{(v, s) | v \in StringVar, s : [stringcat]\}$$

where *StringVar* is the set of program variables of string type.

With reference to the example in Figure 6, Statement 2 generates the flow information `($s, "<HTML>\n")`, associating variable `$s` to a constant string. Statement 4 generates:
`($s, $s . " <INPUT name = \"x\" value = \"" . $x . "\">\n")`, associating variable `$s` to the concatenation of `$s`, a constant string, variable `$x`, and another constant string.

3.2.2. *Flow propagation in a sequential statement block*
Assignment statements generate and invalidate (kill) flow information, according to the following equations:

$$\begin{aligned} STMT[n] &: \texttt{v .= r;} \\ GEN[n] &= \{(v, s.r) | (v, s) \in IN[n]\} \\ KILL[n] &= \{(v, s) \in IN[n]\} \end{aligned}$$

If r is assigned to v by means of the `.=` assignment operator, the generated flow information is obtained by concatenating (the related "dot" operator is explained below) the incoming information, $s \in IN[n]$, with r. Any incoming information associated with v becomes invalid (*KILL[n]*). When the `=` assignment operator is used instead, equations are slightly modified as follows:

$$\begin{aligned} STMT[n] &: \texttt{v = r;} \\ GEN[n] &= \{(v, r)\} \\ KILL[n] &= \{(v, s) \in IN[n]\} \end{aligned}$$

Flow propagation can then be achieved according to the standard equations:

$$\begin{array}{lcl} IN[n] & = & \bigcup_{p \in pred[n]} OUT[p] \\ OUT[n] & = & (IN[n] \setminus KILL[n]) \bigcup GEN[n] \end{array}$$

Incoming information is the union of the flow information outgoing from the predecessors of n ($pred[n]$). Outgoing information is obtained by deleting the killed information from the incoming one, and adding the generated information. Flow propagation is repeatedly performed, until a fixpoint is reached.

When r is a call to function f, the assignment `v = f();` (`v .= f();`) is interpreted as `v = f_out;` (`v .= f_out;`), where the location `f_out` is associated to the value returned by `f`.

The . operator used in the equation above for $GEN[n]$ in case of `.=` assignment works as follows:

$$\begin{array}{llcll} (1) & s.r & = & \texttt{"aabb"} & \text{if } s = \texttt{"aa"},\ r = \texttt{"bb"} \\ (2) & s.r & = & \texttt{"aa" . \$x} & \text{if } s = \texttt{"aa"},\ r = \texttt{\$x},\ \nexists\ \texttt{(\$x, } t\texttt{)} \in IN[n] \\ (3) & s.r & = & \texttt{"aa"} \,.\, t & \text{if } s = \texttt{"aa"},\ r = \texttt{\$x},\ \forall\ \texttt{(\$x, } t\texttt{)} \in IN[n] \end{array}$$

Rule (3) is called the *copy-rule*, in that it allows replacing variables with a copy of the stored values. It is mutually exclusive with rule (2). It is possible to disable it (using rule (2) unconditionally), obtaining a less accurate result computed more efficiently.

Some trivial generalizations of rules (1), (2), (3) are required to make them applicable in cases where s and r have exchanged roles (s being not a string constant), and where s or r are concatenations themselves, instead of being a single constant string or variable.

With reference to the example in Figure 6, the incoming and outgoing flow information at the statements 2, 3, 4 is:

IN[2] = $\emptyset$
OUT[2] =
```
{($s, "<HTML>\n")}
```
IN[3] = *OUT[2]*
OUT[3] =
```
{($s, "<HTML>\n
   <FORM method = \"POST\" action = \"bb.php\">\n")}
```
IN[4] = *OUT[3]*
OUT[4] =
```
{($s, "<HTML>\n
   <FORM method = \"POST\" action = \"bb.php\">\n
   <INPUT name = \"x\" value = \"" . $x . "\">\n)}
```

The *string-cat* approximation of the value printed at statement 9 is thus:

IN[9] = `{($s, "<HTML>\n`
`<FORM method = \"POST\" action = \"bb.php\">\n`
`<INPUT name = \"x\" value = \"" . $x . "\">\n`
`<INPUT name = \"y\" value = \"" . $y . "\">\n`
`<INPUT type = \"submit\">\n </FORM>\n</HTML>\n")}.

3.2.3. *Flow propagation in the presence of control alternatives*

aa.php:

```
1   <?
2       $s = "<INPUT  ";
3       if ($c) {
4           $s .= "name = \"x\">\n";
5       } else {
6           $s .= "name = \"y\">\n";
7       }
8   ?>

9   <FORM method = "POST" action = "bb.php">
10      <? print $s; ?>
11      <INPUT type = "submit">
12  </FORM>
```

aa.php: generated code

```
13  <ALT>
14      <CASE alt = "1">
15          <INPUT name = "x">
16      </CASE>
17      <CASE alt = "2">
18          <INPUT name = "y">
19      </CASE>
20  </ALT>
```

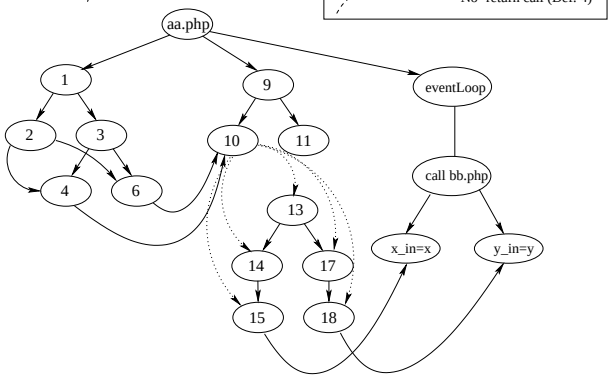


Figure 8. Alternative code generation.

It might happen that a given PHP variable is associated to more than one *string-cat* after flow propagation. This occurs, for example,

when alternatives are in the code. In the example in Figure 8 (top), two alternative strings are concatenated with `$s` at the statements 4 and 6. Correspondingly, the *string-cat*'s computed for `$s` at the `print` statement (line 10) are:

```
IN[10] = {($s, "<INPUT name = \"x\">\n"),
          ($s, "<INPUT name = \"y\">\n")}
```

In order to represent all possible alternatives in the generated code, two new pseudo-HTML tags are introduced: `ALT` and `CASE`. They indicate that the enclosed tags are alternatives and their actual presence in the generated code depends on some (unspecified) run-time condition. `ALT` and `CASE` tags implement the conditional statements of traditional programming languages in HTML.

Figure 8 (bottom) shows the extruded HTML code. The `ALT` tag (node 13) is generated, due to the presence of more than one pair in the set *IN[10]*. It signals the presence of some alternatives, and nodes 14 and 17 represent them.

3.2.4. *Flow propagation in the presence of loops*

If the *string-cat* propagation algorithm is applied to the code in Figure 9 (top), it never terminates. In fact, the flow information computed at the statement 4 does not ever reach a stable state, since it is increased during every new flow propagation. This is the initial sequence of *IN* and *OUT* sets computed at node 4 by the algorithm:

Iteration 1:

```
IN[4] = {($s, "")}
OUT[4] = {($s, "<INPUT name = \"x" . $i . "\">\n")}
```

Iteration 2:

```
IN[4] = {($s, ""), ($s, "<INPUT name = \"x" . $i . "\">\n")}
OUT[4] = {($s, "<INPUT name = \"x" . $i . "\">\n"),
          ($s, "<INPUT name = \"x" . $i . "\">\n
                <INPUT name = \"x" . $i . "\">\n")}
```

Iteration 3:

```
IN[4] = {($s, ""), ($s, "<INPUT name = \"x" . $i . "\">\n"),
          ($s, "<INPUT name = \"x" . $i . "\">\n
                <INPUT name = \"x" . $i . "\">\n")}
OUT[4] = {($s, "<INPUT name = \"x" . $i . "\">\n"),
          ($s, "<INPUT name = \"x" . $i . "\">\n
```

aa.php:

```
1   <?
2       $s = "";
3       for ($i = 0 ; $i < $n ; $i++) {
4           $s .= "<INPUT name = \"x" . $i . "\">\n";
5       }
6   ?>

7   <FORM method = "POST" action = "bb.php">
8       <? print $s; ?>
9       <INPUT type = "submit">
10  </FORM>
```

aa.php: generated code

```
11  <ALT>
12      <CASE alt = "1">
13      </CASE>
14      <CASE alt = "2">
15          <INPUT name = "x"<? print $i; ?>>
16      </CASE>
17  </ALT>
```

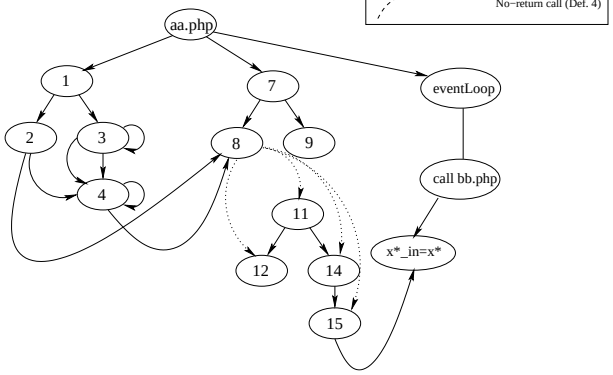


Figure 9. Code generation inside a loop.

```
        <INPUT name = \"x" . $i . "\">\n")}
($s, "<INPUT name = \"x" . $i . "\">\n
        <INPUT name = \"x" . $i . "\">\n
        <INPUT name = \"x" . $i . "\">\n")}
```

However, from the point of view of the SDG construction, the execution of the first iteration alone would be sufficient. The flow information generated successively is just a repetition of what is already in *OUT[4]*, and would result in a replication in the SDG of nodes already built.

It is thus possible to force the termination of the flow propagation algorithm without any information loss. A possible termination criterion is the k-times traversal of each loop, where k is properly set. In the example in Figure 9 a value of $k = 1$ suffices, but in more complex cases,

aa.php:

```
1   <?
2       $tab = array (
3           "aa" => $a
4           "bb" => $b
5           "cc" => $c);
6       ...
7       for (reset($tab ; $k = key($tab) ; next($tab))
8           echo "<INPUT name=\"" . $k . "\" value=\"" . $tab[$k] . "\">";
9   ? >
```

Figure 10. Code generation with data retrieved from a hash table.

with loop-carried dependences, $k = 2$ is a better choice. An alternative (complementary) termination criterion is the following:

> *When the* string-cat *added to a flow information is a substring of the previous* string-cat*, no new flow information is generated.*

By applying this rule to the example in Figure 9, termination is achieved at the second iteration. In fact, the *string-cat* added during the second iteration is identical to the *string-cat* already paired with `$s`. The algorithm can thus stop with the results of iteration 1. Correspondingly, the extruded code will include an alternative, since at node 8 two *string-cat*'s are associated to `$s`:

IN[8] = `{($s, ""), ($s, "<INPUT name = \"x" . $i . "\">\n")}`

The related SDG is depicted in Figure 9 (bottom). It can be noted that the name of the parameter passed to `bb.php` is only partially known. It is indicated as `x*`, since its prefix is `x`, but the remaining part is unknown from the *string-cat* propagation algorithm.

3.2.5. *Hash tables*

PHP provides native support to hash tables, a basic data structure to associate keys to values. Figure 10, lines 2-4, shows the creation of a hash table `$tab` with 3 entries. After creation, the hash table can be indexed by a key to retrieve the stored value. For example, `$tab[$x]`, with `$x` equal to `"aa"`, gives the value hold by `$a` at creation time. Hash tables are widely employed in PHP programming, thanks to their easiness of use.

Hash table keys and values may be accessed when HTML code is dynamically generated. For example, names and values of an `INPUT` field can be obtained as the keys and values in a hash table, as shown in lines 7-8 of Figure 10. The string-cat propagation algorithm described in the previous sections would give an empty string-cat associated to

`$k`, thus missing important information about `FORM` parameters. It is possible to extend the algorithm in order to model keys and values of hash tables in terms of string-cats. Specifically, each hash table (such as `$tab`) is associated with a set of pairs, coupling keys and values, in turn represented as string-cats. With reference to the example in Figure 10, the following pairs would be associated with the hash table `$tab`:

```
("aa", $a)
("bb", $b)
("cc", $c)
```

Then, each reference to the set of keys is replaced by the set of all possible string-cats in first position in the pairs. This means that statement 7 is associated with the following flow information:

GEN[7] = `{($k, "aa"), ($k, "bb"), ($k, "cc")}`

Given a key, represented as a string-cat, the related hash table values are all those paired with the key, if this is a constant string, while they are the set of all possible string-cats in second position, if the key is not a constant. For example, when the flow information `($k, "bb")` is processed at line 8, the string-cat associated to `$tab[$k]` is `$b`.

4. Preliminary results

We have considered 11 Web applications of small to medium size (108 PHP Lines Of Code, LOC, to 73180 PHP LOC), which have been obtained from the Web as open source software (see Table I). Examples of their application domains are the management of discussion groups (wmforum), chats (phpchat), emails (squirrelmail) and project teams (tutos), the publication of picture galleries (gallery), photographs (pw) and page banners (adsnew), the scheduling of appointments (ltw_calendar), the creation of user accounts (account), and the administration of data bases (myadmin). They have been randomly selected among those distributed by open source portals, such as `gotocode.com`, `sourceforge.net` and `freshmeat.net`. All of them exploit dynamic HTML generation to insert attribute values or texts computed at run time inside fixed HTML tags (column 3, *values*, of Table I). In other cases, HTML tags are determined dynamically, being inserted into a variable of type string which is successively printed. This is accounted for in column 4 (*tags*) of Table I. The case of a function returning a string which embeds some HTML tags is also considered in column

4. When the names of HTML variables or called functions/scripts are generated dynamically (e.g., `<INPUT name=<?print $x?>>`, `<FORM action=<?print $a?>>`), being inserted into a PHP variable or returned by a function, a tick is put in column 5 (*names*) of Table I. No case of "extreme" code generation (such as the one shown in Figure 4) has been detected in the 11 applications under analysis, and all generation patterns actually used fall within the categories examined in the previous section.

Table I. *Features of the HTML code generated by the Web applications considered.*

Web application	LOC	values	tags	names
account	108	√		
wmforum	535	√		
pw	702	√	√	
ltw_calendar	911	√		
phpchat	2475	√	√	√
phorum	11489	√	√	√
gallery	14431	√	√	√
adsnew	34094	√	√	√
squirrelmail	48315	√	√	√
tutos	69698	√	√	√
myadmin	73180	√	√	√

A prototype tool for Web application slicing has been developed. Its input is a PHP program where string-cat propagation and code extrusion have already been executed. This is in turn obtained as a transformation of the original PHP code, which is performed by a TXL (Cordy et al., 2002) program.

The current version of the string-cat propagation procedure has some limitations. The current implementation cannot handle string manipulations, other than the concatenation, such as formatted string production, substring selection, search and replacement of regular expressions. Moreover, it cannot handle the insertion and extraction of strings from hash tables. The Web slicing tool has been successfully applied to the first 5 applications in Table I: those which do not use programming features currently unsupported by our tool.

4.1. Size of slices

A preliminary step in program slicing consists of deciding the slicing criteria. The typical scenario in which Web slicing is used involves a question, such as: *Where does this piece of information, produced by the Web application, come from?* The question can be triggered by the occurrence of a fault, or by the need to understand a portion of the Web application, for example because it has to be evolved. Consequently, it seems quite natural to define a set of slicing criteria in the following way: every time a Web application communicates some piece of information to the external world, for example filling-in a Web page, a slice is computed on all variables that are used to produce the externalized data. The main mechanisms a Web application typically exploits to communicate information externally are the generation of HTML code, to be displayed by a Web browser, and the access to a database, where records are inserted, updated or deleted. Correspondingly, the slicing points can be defined as all the statements producing HTML code that is transferred to the browser (e.g., `print` and `echo` statements), as well as all the statements where an SQL query is executed to access a database. Slicing criteria are then completed by pairing these slicing points with each variable used in the related PHP statements.

Table II. *Number of slicing criteria and number of criteria involving aggregate variables that have been split into sub-criteria.*

Web application	criteria	aggregates	sub-criteria
account	4	1	4
wmforum	41	0	0
pw	37	14	62
ltw_calendar	80	0	0
phpchat	86	1	18

Table II, column 2 (*criteria*), shows the number of slicing criteria that have been generated in this way. Statements such as `print`, `echo`, `mysql_db_query`, etc., have been automatically located, together with all variables used by them. Some of the resulting slicing criteria involve variables of type string that *aggregate* the values of several variables (column 3, *aggregates*, of Table II gives their number). An example, taken from *pw*, is the following:

```
1:  $html = $html.'<select name="per_id">
           <option selected value="">Persons...</option>';
```

```
2:  $result = mysql_query("SELECT {$P}persons.per_id,
                per_name,count(pho_id) AS cnt FROM
                {$P}persons LEFT JOIN {$P}photos_persons
                USING (per_id) GROUP BY per_name");
3:  while (list($per_id,$per_name,$cnt) =
                  mysql_fetch_array($result)) {
4:    $html = $html."<option value=\"$per_id\">
              $per_name ($cnt)</option>\n";
    }
5:  $html = $html."</select>\n";
6:  print $html;
```

The automatically generated slicing criterion is: (6, `$html`). However, the string (`$html`) is a concatenation that involves three variables: `$per_id, $per_name` and `$cnt`. A programmer that uses slicing to understand just a portion of the output generated at statement 6 might be interested in a sub-criterion, where the value stored in the variable `$html` is disaggregated into one of the variables used to build it, i.e., `$per_id, $per_name` or `$cnt`. Correspondingly, three new slicing criteria are generated: (4, `$per_id`), (4, `$per_name`), and (4, `$cnt`). A similar case of aggregate criterion that can be split into sub-criteria occurs when a string variable is used to make an SQL query, and its content is built incrementally, out of several other string variables. Sub-criteria have been generated also in this situation. The number of sub-criteria produced for each Web application under analysis is given in the last column (*sub-criteria*) of Table II.

Figure 11 shows the histograms of the size of the slices computed for the analyzed Web applications, obtained using the criteria counted in column 2 of Table II (aggregates are not split into sub-criteria). The horizontal axis gives the size of the slices as a percentage of the total number of statements in each Web application. It is divided into intervals of length 5%. The vertical axis gives the percentage of slices that fall within each 5% size interval. For example, the Web application *account* has 25% of its slices with a size between 0 and 5% of the total number of statements; 25% of the slices have a size between 5 and 10% of the statements, 25% have a size between 15 and 20%, and finally another 25% is between 25 and 30%.

As apparent from Figure 11, the computed slices are typically small. Many of them have a size which is below 10% of the whole size of the application, and no slice have a size above 30%, except for *ltw_calendar*, which has a large proportion of slices (81%) with a size slightly below 70%. These large slices are due to accesses to object fields that are possibly defined at several different statements. Usage of an object-

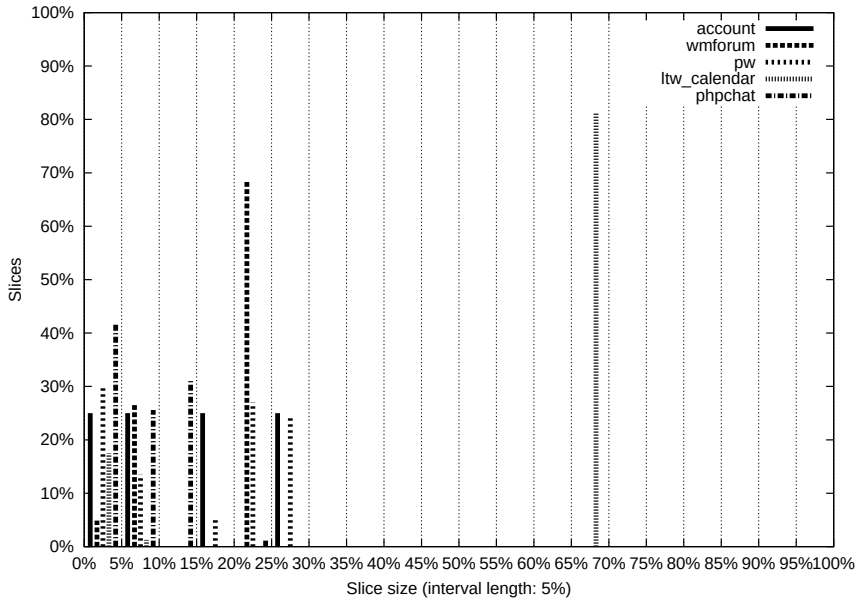


Figure 11. Histograms with the percentage of slices with given size vs. slice size, given as a percentage of the number of statements.

oriented programming style in this application gives raise to a high number of data dependences on class fields.

Figure 12 shows the histograms of slices obtained using only the sub-criteria (counted in the last column of Table II). As one would expect, they are even smaller than those obtained with aggregate variable values. Actually, most of them are below 10% of the application size, and only for *pw* some of the slices computed for the sub-criteria are larger, falling between 15 and 20% of the program size. Slice sizes are given as absolute values in Table III, for the identified criteria and sub-criteria. The average number of statements in the computed slices is reported in the table, as well as the related percentage.

4.1.1. *Discussion*

According to the results of this preliminary study, Web slices seem to offer a remarkable help in restricting the focus of a search, in that a Web slice is substantially smaller than the whole application. On average, the slices computed for the analyzed applications contain 21.52% (18.62% if criteria are split into sub-criteria) of the original PHP/HTML statements (comments and blank lines excluded). A Web developer who aims at determining the origin of a given piece of information can thus limit the inspection of the source code to a small fraction of the entire program.

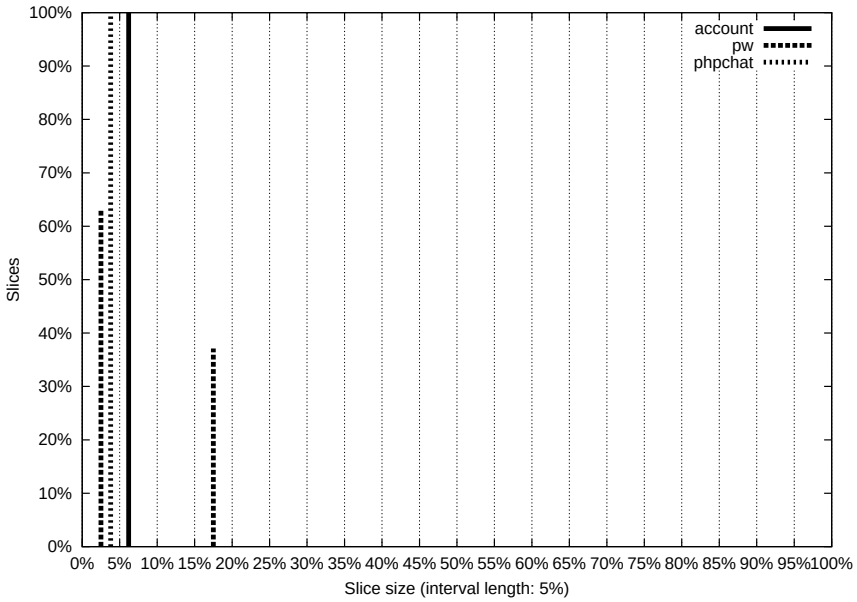


Figure 12. Slices vs. slice size (percentages) obtained after splitting aggregate criteria into sub-criteria.

Table III. *Average size of the slices, measured in number of statements, for the identified criteria and sub-criteria (percentages are given in brackets).*

Web application	criteria	sub-criteria
account	7.75 (15.19%)	4.00 (7.84%)
wmforum	57.90 (15.99%)	-
pw	46.05 (15.87%)	19.38 (6.68%)
ltw_calendar	198.72 (54.89%)	-
phpchat	39.59 (5.68%)	22.00 (3.15%)

Slices of traditional software systems are known to have a distribution with two peaks (see, for example, the report of the working sessions of SCAM 2002 (Cordy, J. and A. De Lucia, editors, 2002)), one positioned at small size values and the other one at high values. The empirical study described in (Binkley and Harman, 2003) reports an average slice size equal to 28.1% of the original LOC (with structure fields expanded), obtained on a code base consisting of 43 C programs, for a total of around 1MLOC. Such a datum is not directly comparable with ours, since we exclude non executable lines of code from the count, while in (Binkley and Harman, 2003) comments and blank lines are

included in the size measure. However, the authors of the study indicate that the reduction computed as a percentage of the SDG nodes is 4.7% less. The corresponding measure is still not directly comparable, though more similar, to ours. Assuming that the correction factor from LOC to executable statements may account for just a few percentage points, our (provisional) conclusion is that slices of Web applications are quite smaller (less than 2/3) than slices of C programs.

In order to investigate the reasons for the small slice size, some of the Web slices have been inspected manually, and the following conclusions have been drawn:

- Fixed HTML code is a substantial portion of any Web application.
- Data dependences tend to remain local to one or a few pages.

There are only a few cases where the HTML statements of a Web application are part of a slice. This happens, for example, for the HTML code of a `FORM` enclosing an `INPUT` field that is part of the slice and for any enclosing HTML statement. However, all HTML statements that are used to format a fixed content included in a page will never occur in any slice.

The limited capabilities of Web applications to track values from a request to the successive one, and the possibility for a user to interrupt the interaction at any time, induce the development of local navigation patterns. The data necessary to complete an interaction are possibly requested at an input page and used at the next one, without any further processing. When this is not possible, and more than one submission steps are necessary, their number tend to remain as small as possible. This is the second main reason why the Web slices we have computed are quite small.

4.1.2. *Limitations of current slicing tool*

Six of the Web applications in Table I exploit programming features of the PHP language that are currently not handled by our slicing tool. These are the main kinds of currently unsupported features:

- A template file is used, where fixed HTML tags are specified together with variable template tags. Variable template tags are replaced by dynamically available strings, when the template is transformed into an output HTML page by the PHP script invoked. Template tags are used for variable names as well.
- In some Web applications, the names of some HTML variables and the respective values are invocation parameters. When a script is

invoked, such names and values can be retrieved from `$HTTP_POST_VARS`. Then, they are possibly inserted into dynamically generated `FORM`s as hidden variables for further propagation.

- HTML variable names are extracted from hash tables in many of the six Web applications not analyzed.
- Advanced string manipulations, provided by PHP library functions, are sometimes used by the considered Web applications. Examples are the usage of the *sprintf* function or of regular expressions.

Static resolution of the cases where templates are used would require the ability to analyze the template format in addition to HTML/PHP. In presence of parameter access via `$HTTP_POST_VARS`, it can be conservatively assumed that each call node in the SDG be augmented with all the parameters possibly recorded in `$HTTP_POST_VARS`. Hash tables would require an extension of the string-cat propagation algorithm, so as to statically approximate the pairs *(key, value)* for each hash table in the program, as described at the end of Section 3.

Handling the cases described above is just a matter of technological improvement of the current version of our tool. Although such an improvement is far from immediate, requiring substantial implementative work, it introduces no conceptual problem beyond those discussed in Section 3.

4.2. Example of slice

Let us consider the Web application *wmforum*, the organization of which is shown in Figure 13. *Wmforum* supports the management of discussion forums. It allows storing user messages and replies into server side text files (no usage of any database). In this Web application, users can insert messages in a newsgroup, see the messages that have been posted previously, and can reply to them.

Wmforum is composed of three PHP programs: `index.php`, `add.php` and `small.php`. In the model shown in Figure 13, the dynamic page `add.php` is split into four different nodes, one for each different behavior of the script `add.php`. Actually, the model has been obtained by running our reverse engineering tool **ReWeb**, described elsewhere (Tonella and Ricca, 2002). **ReWeb** unrolls dynamic pages which exhibit different behaviors in different states into distinct nodes of the model. In the *wmforum* example, the values of the variables `top` and `go` decide on the behavior of `add.php`: when users insert a new message from the page `index.php`, both `top` and `go` are equal to zero. A page is dynamically

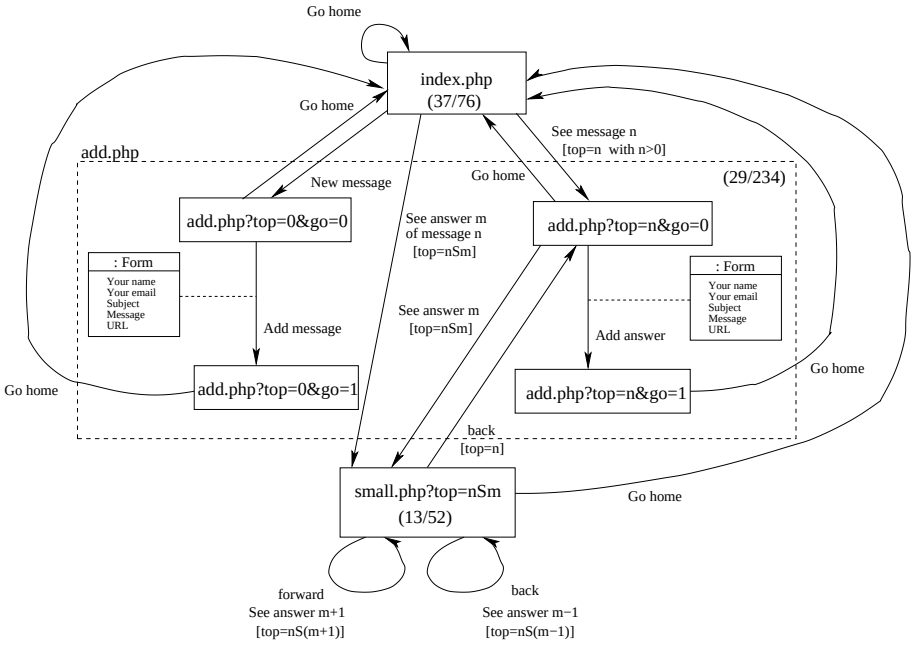


Figure 13. Model of the Web application wmforum. *The ratio between number of nodes in the slice and in the whole application is given in brackets.*

created by `add.php` for the insertion of the new message. The model element representing such a page is labeled `add.php?top=0&go=0` in Figure 13. In this page, users are requested to insert name, email, subject, text of the message and URL by means of a `FORM`, indicated on the outgoing, downward edge. After clicking the submit button, the message is added to the forum and a page is dynamically created by `add.php` to inform the user that the operation was completed successfully. The variable values `top=0, go=1` are interpreted by `add.php` as a request to build such a confirmation page, indicated in the model as a node labeled `add.php?top=0&go=1`.

From page `index.php`, users can select one of the previously posted messages, displayed in the entry page (`index.php`). The result is a request sent to `add.php`, with variable `top` assigned a value $n > 0$ and `go=0`. The response to such a request is a dynamically constructed page, showing the newsgroup message number n, indicated in the model as `add.php?top=n&go=0`. At this point of the interaction, the user can reply to the message, by means of a `FORM`, or can select one of the available replies (if any) posted previously. In the first case, the reply is posted and a confirmation page is created (`add.php?top=n&go=1`). In the second case, if the user selects the reply number m to the message number n, variable `top` is assigned the value `nSm`, and the script `small.php` gen-

erates a dynamic page, indicated as `small.php?top=nSm`. In case there are other replies to the message number n, it is possible to navigate across them, using *forward* and *back* hyperlinks.

```
... here variable go is equal to 0
<?php
135 if ( $top > 0 ):
... print information of message number $top
190 :    if ($indx > 0 ):
191 :        for( $i=1; $i<=$indx; $i++) {
192 :            $indexfname=$patchs.$top."s".$i.".txt";
193 :            $fp=@fopen($indexfname,'r');
194 :            if ($fp <> 0) :
195:                 fclose($fp);
196 :                $content = file($indexfname);
197:                 $SenderData=$content[0];
198:                 $SenderNik=$content[1];
199:                 $SenderMail=$content[2];
200 :                $SenderSubj=$content[3];
201:                 $SenderUrl=$content[4];
202:                 $SenderText=$content[5];
203:                 $SenderEmozi=$content[6]; ?>
...
215 : <a href="<?php echo("small.php?top=".$top."s".$i); ?>"><?php echo($SenderSubj); ?></a>
<?php            endif;
             };
        endif;
228 :   $SenderSubj="Re:".$SenderSubj1;
    else:
230 :   $SenderSubj="";
    endif; ?>
236 <FORM action="<?php echo('add.php?top='.$top); ?>" method="post">
237 :   <TABLE border=0>
238 :       <TBODY>
258:            <INPUT value="<?php echo($SenderSubj); ?>" name=SenderSubj>
...
307 :           <INPUT type="Hidden" name="go" value="1">
308 :           <INPUT type="submit" value="Send" >
...
```

Figure 14. Portion of `add.php` *and slice with criterion: (258,* `$SenderSubj`*).*

The SDG of the *wmforum* application has been built, and, as an example, the slice at the output node 258 on variable `$SenderSubj` has been determined. The resulting PHP/HTML statements have gray background in Figure 14.

The `FORM` at line 236, included at the end of the dynamically generated page, allows posting a message and replying to a message already in the newsgroup. The default value of its input field `SenderSubj`, generated at the slicing point (line 258), is set to "Re:", concatenated with the subject of the current message (line 228), when variable `$top`, holding the number of the selected message is greater than zero (line 135). Otherwise (`$top` is equal to zero when a new message is created), the default value of `SenderSubj` is the empty string (line 230). In the former case, the original subject, stored into variable `$SenderSubj1`, is obtained at line 200 (part of the slice), after reading it from a text file (line 196). All statements in the original Web application involving other computations, such as variables `$SenderData`, `$SenderNik`,

`$SenderMail`, `$SenderUrl`, `$SenderText` and `$SenderEmozi` do not appear in the slice, being not relevant for the computation of the variable `$SenderSubj`. It should be noted that line 215 is included in the slice because it transmits the value of `$top` to `small.php`, which, in turn, might return it back to `add.php` as a URL parameter (see Figure 13).

The ratio between number of nodes in the slice and in the whole application is 79/362 (more precisely, `index.php`: 37/76, `add.php`: 59/234, `small.php`: 13/52), i.e. 78% of the nodes in the original Web application have been removed. Thus, this slice contributes to the histogram bar between 20 and 25% in Figure 11.

The original Web application and the sliced Web application have been navigated in parallel (some `echo` statements have been added to the latter, in order to visualize the selected information). Exactly the same behavior, restricted to the computation of variable `$SenderSubj`, was observed during the navigation.

5. Conclusions

Code extrusion and string-cat propagation are two fundamental algorithms for Web application slicing. All the analyzed applications build HTML code dynamically, by concatenating constant string fragments and values stored in variables. As a consequence, the SDG misses all nodes and edges associated to dynamically generated tags, if no static analysis is executed to determine them. Moreover, names of variables, `FORM` actions and hyperlinks are often generated by string concatenation as well.

The proposed code extrusion and string-cat propagation algorithms have been applied to 5 existing Web applications. Then, slices have been determined on the extruded HTML code. Interestingly, the size of the resulting slice is substantially lower than the original size of the application, and does not exceed 1/3 of the whole application (except for *ltw_calendar*). Thus, Web application slicing has the potential to offer a great help to Web developers, who inspect the source code to locate the statements responsible for a given piece of information. Manual inspection and execution of one of the slices computed for the Web application *wmforum* confirmed the potential to remarkably restrict the search space. The main reasons for the small size of Web slices are the presence of fixed HTML code, which is excluded from all slices, and the locality of the data flows.

Application of Web slicing to larger systems requires some major technological improvements of the currently available tools. Advanced

features of the PHP language for the manipulation of the strings have to be properly handled during string-cat propagation. Although the implementative work required by such extensions is non trivial, there is no conceptual gap to face. While specific features of Web applications, such as dynamic HTML code generation, make them more difficult to slice than traditional software, preliminary results indicate that the resulting size reduction is even higher.

Acknowledgements

The authors would like to thank Ira Baxter and Michael Mehlich for the interesting discussions about dynamic code generation, stimulated by the SCAM workshop in Montreal, October 2002. They would also like to thank the anonymous referees, who gave a substantial contribution to the improvement of the paper.

References

Aho, A. V., R. Sethi, and J. D. Ullman: 1985, *Compilers. Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley Publishing Company.

Binkley, D. and K. B. Gallagher: 1996, 'Program slicing'. *Advances in Computers* **43**, 1–50.

Binkley, D. and M. Harman: 2003, 'A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity'. In: *Proc. of the International Conference on Software Maintenance (ICSM).* Los Alamitos, CA, USA, pp. 44–53, IEEE Computer Society.

Cimitile, A., A. D. Lucia, and M. Munro: 1995, 'Identifying Reusable Functions Using Specification Driven Program Slicing: A Case Study'. In: *Proceedings of the International Conference on Software Maintenance.* Opio(Nice), pp. 124–133.

Cordy, J., T. Dean, A. Malton, and K. Schneider: 2002, 'Source Transformation in Software Engineering using the TXL Transformation System'. *Information and Software Technology* **44**(13), 827–837.

Cordy, J. and A. De Lucia, editors: 2002, *Proceedings of SCAM, 2nd International Workshop on Source Code Analysis and Manipulation.* Montreal: IEEE Computer Society.

Gallagher, K. and J. Lyle: 1991, 'Using Program Slicing in Software Maintenance'. *IEEE Transactions on Software Engineering* **17**(8), 751–761.

Gallagher, K. B.: 1996, 'Visual Impact Analysis'. In: *Proceedings of the International Conference on Software Maintenance.* Monterey, pp. 52–58.

Gupta, R., M. Harrold, and M. Soffa: 1996, 'Program Slicing-Based Regression Testing Techniques.'. *Journal of Software Testing, Verification, and Reliability* **6**(2), 83–112.

Harman, M. and S. Danicic: 1995, 'Using program slicing to simplify testing'. *Software Testing, Verification and Reliability* **5**, 143–162.

Horwitz, S., T. Reps, and D. Binkley: 1990, 'Interprocedural Slicing Using Dependence Graphs'. *ACM Transaction on Programming Languages and Systems* pp. 26–61.

Kamkar, M., P. Fritzson, and N. Shahmehri: 1993, 'Interprocedural dynamic slicing applied to interprocedural data flow testing'. In: *Proceedings of the International Conference on Software Maintenance*. Montreal, Quebec, Canada, pp. 386–395.

Liu, C.-H., D. C. Kung, P. Hsia, and C.-T. Hsu: 2001, 'An object-based data flow testing approach for web applications'. *International Journal of Software Engineering and Knowledge Engineering* **11**(2), 157–179.

Ricca, F. and P. Tonella: 2001, 'Web Application Slicing'. In: *Proceedings of the International Conference on Software Maintenance*. Firenze, Italy, pp. 148–157.

Ricca, F. and P. Tonella: 2002, 'Construction of the System Dependence Graph for Web Application Slicing'. In: *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. Montreal, Canada, pp. 123–132, IEEE Computer Society.

Tip, F.: 1995, 'A Survey of Program Slicing Techniques'. *Journal of Programming Languages* **3**(3), 121–189.

Tonella, P. and F. Ricca: 2002, 'Dynamic Model Extraction and Statistical Analysis of Web Applications'. In: *Proc. of the International Workshop on Web Site Evolution (WSE)*. Montreal, Canada, pp. 43–52, IEEE Computer Society.

Weiser, M.: 1982, 'Programmers use slices when debugging'. *Communications of the Association for Computing Machinery* **25**, 446–452.

Weiser, M.: 1984, 'Program slicing'. *IEEE Transactions on Software Engineering* **10**(4), 352–357.