PHPSlicer: Slicing Dynamically Typed Programming Languages

Case Study on PHP Web Apps

Mohammed Sayagh, Bram Adams
Polytechnique Montreal, Canada
{mohammed.sayagh, bram.adams}@polymtl.ca

***Abstract*—Source code analysis and precisely the slicing technique is widely used for different source code analysis purposes. While there are many implementations for this technique proposed by the literature, they are used to analyze applications developed in C or Java. As the best of our knowledge, there are no existing solutions to slice dynamically-typed programming languages like PHP web applications. In this paper we propose an algorithm to dynamically slice PHP based web applications.**

I. Introduction

Slicing is a technique used to reduce the scope of source code to analyze, in order to better understand what impact a given source code line or variable. After the proposition of the slicing technique by Weiser [1], many research effort have been conducted to apply it in order to resolve different problems. The slicing technique is used for program comprehension, debugging, ...

The key point that gives slicing technique all that importance, is that it reduces the scope of source code to analyze, by finding the subset of the program source code lines that may have a relation with a given source code line, which we refer to by a slicing criterion. That subset of the whole source code lines represents the lines that are pertinents for the execution of a given slicing criterion.

Many tools implemented and evaluated this technique in different contexts. However, all of these presented tools focus only on programs that are developed in Java or C. As the best of our knowledge, the are no existing solution that considers dynamically-typed programming languages.

In this paper, we present a solution to slice a dynamically typed programming language, and precisely PHP-based web applications.

PHP-based web applications have many limits, which makes the application of static analysis on that kind of source code applications hard an inaccurate. The PHP code have many features that makes a static analysis hard, like the dynamic includes, which are include files statements, in which the included file could be not hard-coded, but it can be stored in a variable, it even could be a result of function call. Moreover, PHP has the feature of variable of another variable, which is the name of a variable that is stored in another one. To overcome these problems, we decided to use dynamic slicing instead of static slicing.

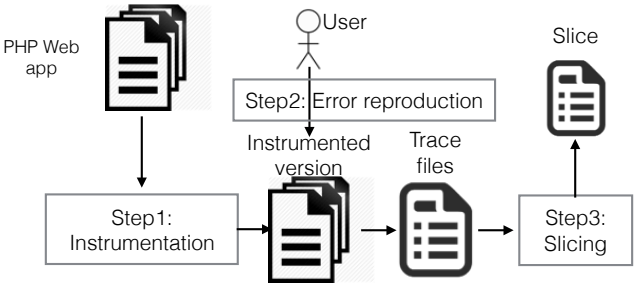

Figure 1: Overview of approach.

The paper is organized as follows: Section II presents the approach and our algorithm, which we implemented in the tool PHPSlicer, while we discuss the implementation details in Section III, before concluding in Section IV.

II. Approach

As shown in Figure 1, the first step of our approach consists of instrumenting the web application. The second step is manual, and involves the user reproducing a scenario, which generates a trace file. Such a file then is the input of the dynamic slicing step, whose goal is to recommend statements that are impacting the slicing criterion (which can be an error line or a debugging starting point).

The approach applies to other (dynamically typed) languages and systems as well. The rest of this section presents each of the three steps shown in Figure 1.

A. Step 1: Instrumentation

In order to generate execution trace files, we have to instrument the system once in the first step.

We transform the source code by adding trace statements that report for each source code line (1) the instrumented statement and (2) the names and values of variables and constants used in the statement. Figure 2 shows an example of instrumented statement (line 11), the statement in line 3 reports the instrumented line, whereas line 6 reports the variables and constants used in that instrumented line.

To avoid a blow-up in execution and analysis time due to excessive logging of information, we tried to reduce the trace

```
1  while nrIterations < thresHold && prevCond not null do
2      nrIterations++;
3      prevCond = predicateBefore(errorLine);
4      print(prevCond);
5      slicingAlgorithm (function, prevCond, null, request);
6  end
7  Function: slicingAlgorithm
   input  : function: Function, errorLine: string, specificVariables: string[], request:
            string
   output: recommended: global string[]
8  if specificVariables is null then
9      specificVariables := errorLine.getVariables();
10 end
11 recommended.add (errorLine.getOptions());
12 for var in specificVariables do
13     thereIsAssignmentForVar := false;
14     for line: errorLine back to firstLine (function) do
15         if line.assignsTo (var) then
16             thereIsAssignmentForVar := true;
17             print(line);
18             switch line.typeOfAssignment (var) do
19                 case 'assigned_function_return_value'
20                     called := f.getCalledFunction(line);
21                     parIndices := slicingAlgorithm (called,
                           called.getReturnLine(), null, request);
22                     pars := getPars (line,called,parIndices);
23                     slicingAlgorithm (f, line, pars, request);
24                 end
25                 case 'assigned_other_variables'
26                     slicingAlgorithm (f, line, null, request);
27                 end
28             endsw
29             break;
30         end
31     end
32     if thereIsAssignmentForVar == false then
33         switch typeOf(var) do
34             case 'global_variable'
35                 line := findWhereModifiedBefore (var, errorLine);
36                 slicingAlgorithm (line.getFunction(), line, var, request);
37             end
38             case 'function_parameter'
39                 listPars.add (f.getPars().indexOf (var)) ;
40             end
41             case 'attribute'
42                 line := findWhereModifiedBefore (var, errorLine,
                       f.getObjectId());
43                 slicingAlgorithm (line.getFunction(), line, var, request);
44             end
45         endsw
46     end
47 end
48 return listPars;
```

Algorithm 1: Our dynamic slicing algorithm.

```
1   <?php
2   // Print the exact statement
3   file_put_contents ("traceFile.txt", "Statement:
4               \$myVar = \$var + Constant; \n", FILE_APPEND);
5   // Print the variables' values
6   file_put_contents ("traceFile.txt", "Values:
7               \$myVar = " .$myVar. ",
8               \$var = " .$var. ",
9               Constant = ".Constant."\n", FILE_APPEND);
10  // Statement
11  $myVar = $var + Constant;
```

Figure 2: Example of instrumented PHP statement.

file size by relaxing the instrumentation requirements. First, we do not report all variable values, but only those of variables that are used as array indexes (in order to slice individual array elements instead of only the array as a whole). This is particularly important in the case multi-dimensional arrays.

```
1-  var1 = option1; var2 = option2; y = 19;
2-  if ( x == 10 && y == 20) {
3-  } else {
4-      if ( y > 15 ) {
5-          x = option3;
6-      }
7-  }
8-  if ( var2 == 20 ) {
9-      if ( var1 == 10 && y > 15 )  {
10-         print "error";
11-     }
12-}
```

Figure 3: A dynamic slicing example that finds the misconfigured option "option2".

Second, we replace long string literals in the trace statements with a short constant string. Finally, to know where an attribute of a given object is modified during the execution, we also report the object identifier in the trace statements.

B. Step 2: Error Reproduction

Having instrumented the web application, which is only necessary once a source code analysis is required, the user should try to reproduce his or her test scenario to generate execution traces. This generated execution trace file will be the inputs for the next step.

Note that instrumentation does not need to happen physically each time a dynamic slicing of the web application is required. Modern web and cloud systems use deployment techniques like canary or blue-green deployment that cleverly allow to redirect certain users to another instance of the system, just by modifying DNS entries [2]. As such, one just needs to instrument the system once and make this version run in parallel with the actual production system, ready for troubleshooting customers with problems.

C. Step 3: Dynamic Slicing on the Trace Files

1) The Slicing Approach: The main idea of our dynamic slicing approach is to (1) identify what conditions have been evaluated to true or false during the program's execution on its way to the slicing criterion (depending on the source code analysis purposes, it could be the code location where the system crashed or an error message was shown, or a debugging starting point); (2) verify what variables impacted these conditions; and (3) (recursively) identify the statement source code lines that impacted the value of these condition variables. In other words, we perform several dynamic slicing calculations, as outlined in Algorithm 1, and illustrated in Figure 3.

We first find the code line in the trace file, which is considered as a slicing criterion. We verify if it contains some variables to slice. We slice the variables of the closest, previously executed condition (line 3 of Algorithm 1), which could be in the current function's body or in the body of a function calling the current function. In the example of Figure 3, the closest condition to line 10 is line 9, in which we find (line 9 of Algorithm 1) the variables "var1" and "y". We find that they are modified respectively by "option1" (line **??** of Algorithm 1) and the hardcoded value "19". We then repeat

the analysis for the next closest condition, i.e., lines 8, 4 and 2, which yield "option2" (for "var2"). We repeat until we have checked a pre-defined number of conditions or arrive at the start of the execution.

If a seed instead is assigned the result of a function call (line 19 of Algorithm 1), we should not necessarily slice all parameters of the function call, since not all of them might impact the function's return value. Hence, we slice the called function from its return line (the one executed in the trace file), to obtain the list of function parameters that impact the return value.

If a variable is not modified within a function (line 32 of Algorithm 1), it can be either a global variable, a function parameter of its enclosing function, an attribute, or an array. We slice global variables by searching the closest statement that modified them before the seed line, then slice the found line (line 34 of Algorithm 1). We used the same principle to slice object attributes, except that we need to specify the identifier of the object whose attribute we are analyzing (line 41 of Algorithm 1). Finally, if the seed is a function parameter, we return it, in case "slicingAlgorithm()" has been called for a function call (line 38).

III. Implementation

This section presents notable details about how we implemented our approach.

To instrument a Web application, we used the TXL language [3] and a slightly modified PHP Grammar. TXL is a language designed to modify and transform source code, and has a wide variety of supported programming languages. To the best of our knowledge, there are no tools to slice PHP applications. Hence, we developed our own tool implementing Algorithm 1.

While the execution consists of three steps, i.e., instrumentation of the WP and plugin layers, reproducing the error and executing the slicing algorithm, the last step can be divided further into sub-steps, i.e., parsing the execution traces to a graph and serializing it, loading the serialized graph, and executing the slicing algorithm. All these steps are automatic, except the error reproduction (the second step).

IV. Conclusion

This paper presents an approach that implements the dynamic slicing approach on PHP-based web applications. We plan to evaluate our approach in future work for debugging objectives.

References

[1] M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, Ann Arbor, MI, USA, 1979, aAI8007856.

[2] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[3] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.