A Seminar Report on Dynamic Program Slicing

Mitul Bipin, 7023803

01.09.2022

1 Introduction

Software applications have become a source of information in today's generation, and people store personally identifiable information and perform large cash transactions. Modern web and mobile applications are packed with advanced features that require complex algorithms and high computational power. With the increase in complexity of the application, the chances of witnessing bugs are higher. Therefore we need reliable methods to detect those bugs. Two well-known approaches to finding bugs - static and dynamic analysis are popular among developers and testers. This report will elaborate on the program slicing technique, one of the static analysis techniques. Program slicing is a technique for simplifying programs by focusing on a specific functionality[2]. It conveniently allows the developer or tester to evaluate the part of a code for a given variable. The program slicing consists of two methods - static and dynamic slicing. While static analysis involves finding a slice manually and also through automation, dynamic slicing detects a bug during run-time. Furthermore, dynamic slicing can also be performed offline by tracking the execution for a specific variable.

To further illustrate with an example, let us consider a program under test that has been encountered with an error. The error prints an incorrect value of the variable at a particular statement. Ideally, a tester requests for the test case under which the buggy functionality is tracked. This approach is much more efficient than knowing the value and statement of the incorrect variable. This approach suggests that - while debugging, we probably try to find the dynamic slice of the program in our minds [1].

The program slicing technique is widely used by developers, testers, and researchers to detect bugs in the code of an application. There exist several static analysis tools that assist in detecting bugs. In addition, security researchers use program slicing techniques to discover security vulnerabilities in the systems. Section 4 of the report will focus on vulnerability analysis with dynamic slicing.

The report's structure is as follows: Section 2 briefly highlights the program dependence graph and static program slicing. The section will also compare static and dynamic program slicing with an example. Section 3 will introduce one of the approaches of dynamic slicing included in paper [1]. Real-world usage of the slicing technique with regards to vulnerability detection is outlined in Section 4. Finally, the report is concluded in Section 5 with some benefits and drawbacks of program slicing.

2 Program Dependence Graph and Static Slicing

A program dependence graph is a tree-like representation of a program under test. It has one node for each simple (assign, read, write) and compound statement (if, if-else, while). Furthermore, the program dependence graph consists of two types of edges - the data dependence and the control dependence edges. With regards to the *vertex* v_i and *vertex* v_j, the data dependence edge implies that the computation performed at v_i depends on the value computed at v_j, whereas the control dependence edge indicates whether the node v_i is executed based on the Boolean outcome of node v_j.

```
        Begin
S1:           read(X);
S2:           if(X<0)
              Then
S3:             Y:= f(X);
              Else
S4:             If(X=0)
                Then
S5:               Y:=g(X);
                end_if;
              end_if;
S6:           write(Y);
        End
```

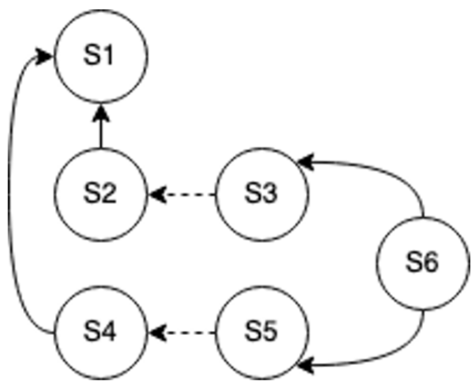

Figure 1: A code snippet with its corresponding program dependence graph.

Figure 1 shows the program dependence graph of a code snippet. The solid arrows in the graph denote a data dependence edge, and the dotted lines denote a control dependence edge for a set of statements. The goal is to achieve a *"slice"* of the code under test. In this context, the *"slice"* is a piece of code that is being tested for a particular variable. It may represent a functionality. While it is possible to obtain the *"slice"* directly from the code snippet, the program dependence graph makes it simpler. Furthermore, it is easy to distinguish between the data dependence and control dependence edges.

A static slice of a program with respect to a variable var at node n consists of all the nodes whose execution could possibly affect the value of a var at n[1]. Static slicing considers every possible program execution, making it longer to detect a bug. For example, to find the static slice of the code snippet in Figure 1 concerning variable Y at node 6, we first find all the possible nodes reaching Y and node 6. These nodes are node 3 and node 5. Furthermore, we find all the reachable nodes from the two nodes. Looking at the program dependence graph in Figure 2, we obtain the set of all nodes {S1,S2,S3,S4,S5} which is our static slice of the program.

The obtained static slice of a program concerning variable Y will include all the reachable nodes regardless of their execution. We already know that for any value of X in Figure 1, only one statement (S3 or S5) will be executed. When we consider X as -1 to be executed at S1, the dynamic slice will only contain node 2. Therefore the obtained slice would be {S1,S2,S3,S6}, as opposed to the static slice obtained earlier that included an irrelevant statement (S4). In a much larger scenario where several statements are involved, the dynamic slice would help localize the bug more quickly than the static slice.

A dynamic slice of a variable var consists of all the statements affecting the value of var at any point for particular program execution. In the next section, we will look at various approaches to dynamic slicing. We shall use superscripts to distinguish between multiple occurrences of a node. For example, a set of dynamic slices with multiple occurrences will look like $\{1, 2, 3, 4, 1^1, 2^2\}$.

3 Dynamic Slicing Approach

This approach is the most simple way to obtain a dynamic slice of a program. Considering the example in Figure 1, we have two assignment statements - S3 and S5. However, only one of the statements is executed for a given test case. For the test case where X is -1, the execution history yielded is {1,2,3,6}. Initially, all the nodes are denoted by dotted shapes. As the statements are executed, the corresponding nodes are made solid. Once the "solid" nodes are obtained, the graph is traversed only through those nodes. Lastly, all the nodes traversed for the test case are highlighted in bold. This is obtained by traversing along the solid nodes till the last definition of Y in the execution history.

Figure 2 outlines all the executed nodes in bold, which denotes the dynamic slice.

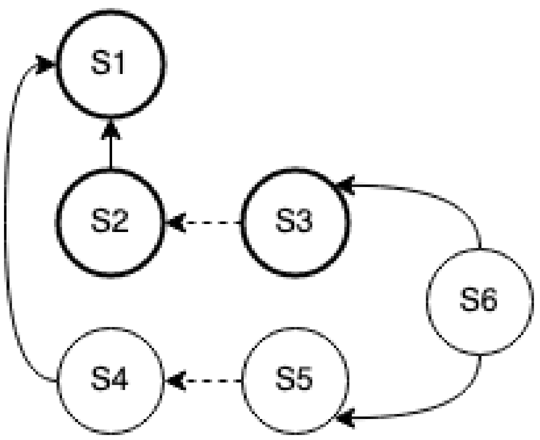

Figure 2: Bold Nodes Denote the Dynamic Slice

This is a naive approach as it includes additional statements that do not affect the execution of a particular dynamic slice. Such an issue is mainly observed in the case of conditional statements, as it will enforce all the statements to be executed within the loop. This approach can potentially include additional statements irrelevant to the test case, so it does not generate an accurate slice.

```
      Begin
S1:           read(N);
S2:           Z := 0;
S3:           Y := 0;
S4:           I := 1;
S5:           while(I<=N)
              Do
S6:              Z := f(Z,Y);
S7:              Y :=g(Y);
S8:              I := I + 1;
              End_while;
S9:           write(Z);
      End
```

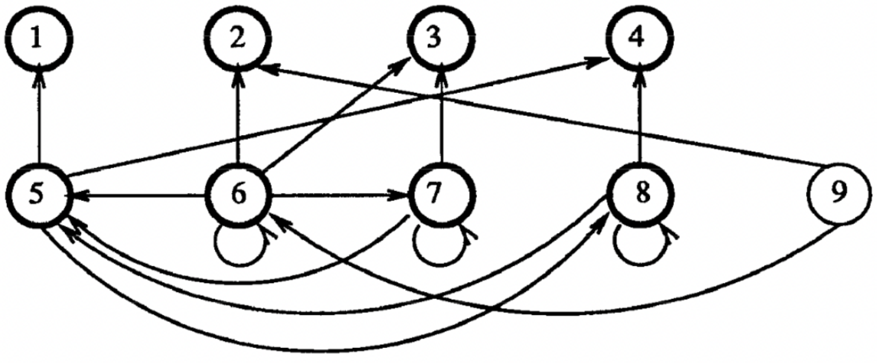

Figure 3: A code snippet with its corresponding program dependence graph.

To illustrate the flaw with an example, let us consider the code snippet and its corresponding program dependence graph in Figure 3 [1]. For the test-case where N = 1, we obtain the execution history {1,2,3,4,5^1,6,7,8,5^2,9}. Looking at the execution history, we see that statement 7, which assigns a value to a variable Y, does not appear in the further execution iterations. This statement should not be included in the dynamic slice as it increases the difficulty detecting a bug in the program.

Addressing the issue observed in this approach, we shall consider an alternate approach(es). The idea is to examine dependencies for only a single execution path for a given input, and the inclusion of a statement within the dynamic slice should lead to only those statements that affect the program's execution. Therefore, we shall follow the same method observed in Figure 2. First, all the edges are dotted. The edge is only made "solid" when the corresponding dependency is ever activated during execution. Lastly, the edges highlighted in bold are included in the dynamic slice. This is achieved by traversing through the "solid" edges obtained earlier. Figure 4 represents the solution in two parts. The graph on the left represents the initial state (unmarked edges) before execution. On the other hand, the graph on the hand is obtained after execution. Note that some of the edges and nodes are solid and bold respectively. The bold nodes denote the slice of a program. In comparison to the naive approach described above, its solution omits the statement S7, which does not affect the outcome of the program.

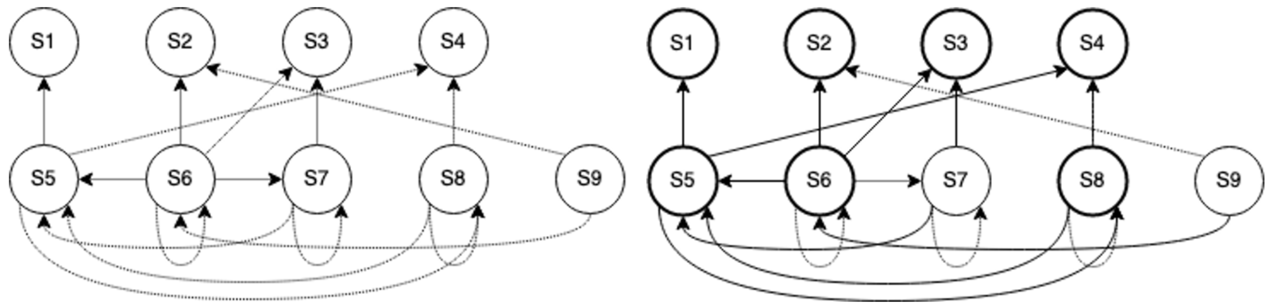

Figure 4: The graph on the left represents the unmarked edges and the graph on the right represents the slice marked in bold.

4 Vulnerability Detection with Slicing

One of the primary use cases of program slicing is in the field of vulnerability detection. A security analyst can detect significant vulnerabilities by obtaining the "slice" from a large code base. This approach will help reduce the efforts of iterating through the entire code base. Furthermore, the need to perform static analysis on a program under test is essential because a vulnerability scanner cannot detect bugs in a program's logic.

```
        Begin
S1:             N := function();
S2:             Y := "[";
S3:             I := 1;
S4:             while(I < 5)
                Do
S5:                 if(I < 4)
                    Then
S6:                     Y += f(I) + ","
S7:                     X = N;
                    Else
S8:                     Y += g(I) + "]"
S9:                     N = eval;
                    End_if;
                End_while;
S10:            write(${N});
        End
```

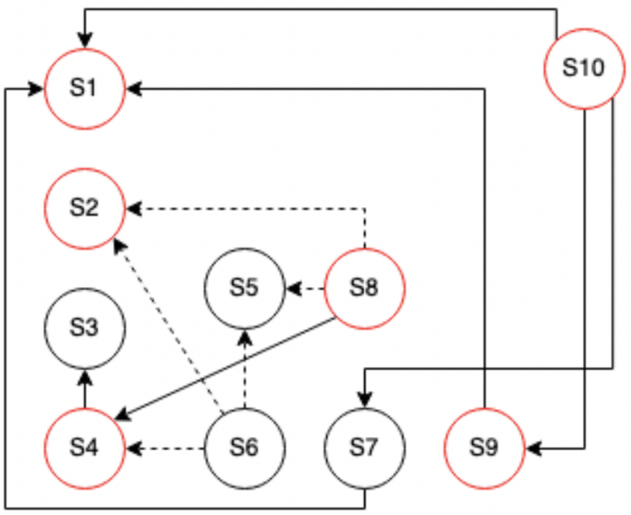

Figure 5: A vulnerable code snippet with its corresponding program dependence graph.

To illustrate with an example, Figure 5 consists of a vulnerable code snippet with its corresponding program dependence graph. The code is vulnerable to sensitive data exposure as *eval* is being misused. When the value of $I = 4$, the statements S8 and S9 are executed. An adversary, who learns about the vulnerable code, can perform arbitrary code executions through the *eval* function. With the approaches mentioned in 3, the dynamic slice {1,2,4,8,9,10} of the vulnerable code is obtained. We can also derive the slice by constructing a Control Flow Graph (CFG)[4], and detect vulnerabilities using forward slicing and backward slicing[5]. However, this report does not explain the concept of forward and backwards slicing.

5 Conclusion

The paper presented in this report successfully generates a program's static and dynamic slice under execution. In addition to that, the report also draws a comparison between static and dynamic analysis. The program dependence graph demonstrates the obtained ”slice” in a tree-like structure - making it easy to visualize the program. While the approach highlighted in this report does not deliver the most accurate slices, there exist improved approaches that can obtain ”slices” from complex programs. We then proposed a real-world usage of dynamic slicing as seen in Section 4 highlighting a vulnerability in a program that web vulnerability scanners cannot detect. Such types of vulnerabilities can also be detected by forward and backward slicing. Further research is required to understand the various approaches of dynamic slicing better to ensure the obtained slice is accurate, which enables the approach to derive dynamic slices out of complex programs.

References

[1] Hiralal Agarwal, Joseph R. Hogan: Dynamic Program Slicing

[2] Mark Harman, Robert M Hierons: An Overview of Program Slicing

[3] https://steemit.com/software/@cpuu/the-difference-between-soundness-and-completeness

[4] https://en.wikipedia.org/wiki/Controlflow_graph

[5] Rahaman, Sazzadur, et al: Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects.