Attack Diagnosis on Binary Executables Using Dynamic Program Slicing

Shan Huang, Yudi Zheng, Ruoyu Zhang
School of Software, Shanghai Jiao Tong University
{babalaka, monkey_yugi, holmeszry}@sjtu.edu.cn

ABSTRACT

Nowadays, the level of the practically used programs is often complex and of such a large scale so that it is not as easy to analyze and debug them as one might expect. And it is quite difficult to diagnose attacks and find vulnerabilities in such large-scale programs. Thus, dynamic program slicing becomes a popular and effective method for program comprehension and debugging since it can reduce the analysis scope greatly and drop useless data that do not influence the final result. Besides, most of existing dynamic slicing tools perform dynamic slicing in the source code level, but the source code is not easy to obtain in practice. We believe that we do need some kinds of systems to help the users understand binary programs. In this paper, we present an approach of diagnosing attacks using dynamic backward program slicing based on binary executables, and provide a dynamic binary slicing tool named *DBS* to analyze binary executables precisely and efficiently. It computes the set of instructions that may have affected or been affected by slicing criterion set in certain location of the binary execution stream. This tool also can organize the slicing results by function call graphs and control flow graphs clearly and hierarchically.

Keywords: Dynamic program slicing, binary analysis, attack diagnosis.

1. INTRODUCTION

1.1 Dynamic Program Slicing

Dynamic program slicing [1] is a technique that can do the program slicing in the actual process of running the program. After making the certain register or the memory address as the slicing criterion, the user can extract the instructions and basic blocks related to the slicing criterion during the execution of the target program by using dynamic program slicing. And these extracted instructions and basic blocks will constitute a subset of the total code, which is real executable. It can reduce the difficulty and improve the efficiency by analyzing this subset stead of the whole program because slicing removes large amount of parts unrelated to the slicing criterion.

After the dynamic slicing was proposed, there have been many researches in this field. In order to improve the precision of program slicing, Mock [2] proposes an approach to refine the program slicing with dynamic points-to data. And since efficiency is one of the major concerns in dynamic program slicing, much work has been done to improve the analysis efficiency. Rountev and Chandra [3] propose the mechanism of variable substitution by replacing a set of program variables which are guaranteed to have the same points-to sets with a single variable. By this mean, the size of the problem is greatly reduced and the efficiency has been improved.

Comparing with the static method, the dynamic way has much more advantages. First, the result of dynamic slicing is more precise and effective. Second, the dynamic slicing is more convenient than the static way, and it reduces the code size greatly. *PSE* [9] is a static analysis technique for diagnosing program failures. It can be viewed as a program slicing technique, however, it is more precise because of its consideration of error conditions. A motivating example is dereferencing a NULL value. It is similar to Das's earlier work, *ESP* [10], a symbolic dataflow analysis engine. The recent researches on dynamic program slicing are mostly based on source code, such as some existing dynamic slicing tools [4][5][6][7], but the source code is difficult to get in practice, which makes the slicing on source code level be not practical. The dynamic program slicing based on binary executables, which is proposed and implemented in this paper, has made up this deficiency. By using the binary analysis platform *DynamoRIO* [8], we can insert the analyzing code into the binary stream of target program, track the program running states and analyze them.

Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies, edited by Zhu Zeng, Yuting Li, Proc. of SPIE Vol. 8350, 83500Z · © 2012 SPIE · CCC code: 0277-786X/12/$18 · doi: 10.1117/12.923765

In this paper, we present an approach of diagnosing program attacks using dynamic program slicing based on binary executables, and provide a dynamic binary slicing tool named *DBS* to analyze binary executables precisely and efficiently. The tool also can organize the slicing result by function call graph and *CFG* (Control Flow Graph) to make it clear and hierarchical, which makes the debugging and vulnerability locating work much easier for programmers.

1.2 Main Contributions

- *Propose a method of diagnosing attacks by using dynamic backward binary slicing*

In this paper, a method of attack diagnosis on binary executables using dynamic backward program slicing is described. We set the attack point as the slicing criterion and execute the dynamic backward program slicing in order to locate the vulnerabilities, which make the attack happen, from the slicing results. During the target program execution, we construct the function call graph of it to implement the interprocedural slicing technique, and construct CFGs to organize the basic blocks in procedures.

- *Implement the dynamic binary slicing tool, DBS*

We implement the dynamic binary slicing tool, *DBS*, and experimentally test its effect. After making the memory crash point as slicing source, *DBS* can slice out the basic blocks related to the slicing criterion and construct them into an executable subset. Besides, *DBS* can organize the slicing result by function call graphs and CFGs to make the result clear and hierarchical. Programmers can diagnose the attack and locate the software vulnerabilities easier based on the slicing result than the original total program.

2. SYSTEM DESIGN

2.1 Overview

The dynamic binary slicing tool that we implement in this paper, *DBS*, contains four major components: Function Analyzer, Slicing Engine, Call Graph Builder and CFG Builder (Basic Block Analyzer). We implement the tool with the help of *DynamoRIO*, which is a runtime code manipulation system that supports code transformations on any part of a program, while it executes. It makes our tool work on the binary executable stream directly and does not need any source code, which is more practical in real life. Dynamic analysis also enables *DBS* to eliminate false positives that is of importance because false positives are time-consuming to deal with.

As Fig.1 shows, *DBS* hooks and inserts analysis codes to the target program with the help of *DynamoRIO* at first. And then, *DBS* switches back to continue the execution of the target program with the analysis codes instrumented. With the analysis information from *DynamoRIO*, Function Recognizer and Function Analyzer deal with every function call during the target program execution. After that, the basic blocks of functions are sent to the Slicing Engine. In Slicing Engine, CFG Builder analyzes the basic blocks in every single procedure, organizes the basic block execution sequence and constructs the CFGs. Call Graph Builder can imitate the stack operations like Operation System to manage the function call relationship and use these basic block information to generate the call graph of the whole target program. Slicing Engine executes the dynamic program slicing algorithm on the basic block sets, slices out the ones related to the slicing criterion and generates hierarchical slicing result with the help of function call graph from the Call Graph Builder.

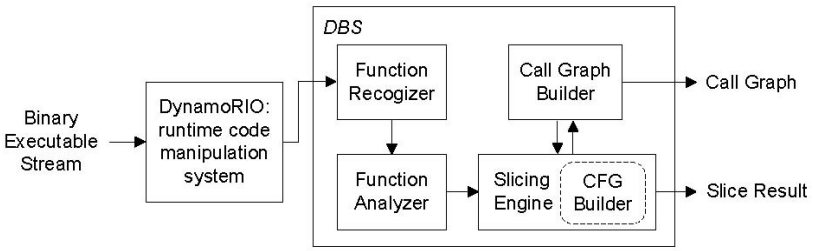

Figure 1. System Overview of DBS

During the execution, *DBS* gathers information such as operation codes and operands of the instructions in each basic block. If the target program reaches a fatal error during execution, the program terminates and an error is reported.

2.2 Dynamic Binary Program Slicing

We use *DynamoRIO* to help us implement the dynamic binary program slicing technique. We insert analyzing code before and after the next basic block dynamically by *DynamoRIO*, record the register and memory states during execution, thus, track and analyze the target program execution process. It is much more difficult to analyze binary programs than source code programs since they do not have advanced semantic information. However, *DynamoRIO* can accomplish the disassemble work so that the user could get assemble code instead of binary code to analyze and debug the target program. Of course, assemble code is not as intuitive to analyze as source code for programmer. Therefore, our next focus is on how to connect the assemble code in slicing result with the source code (if the user has it) of the target program.

We will explain our slicing algorithm in detail here. Program slicing can be divided into two categories: forward and backward program slicing. The primary function of our tool is locating the software vulnerability according to the attack occurred. We define a slicing criterion as <*n*, *V*>, where *n* is the memory point in which the attack happens, and *V* is the set of variables which the programmer would like to focus on. It means that we need make the attack point as the slicing source and slice out the basic blocks that will affect the slicing criterion. Thus, our tool use backward program slicing technique to find out the basic blocks related to slicing criterion during the execution. The set composed by these sliced basic blocks is a subset of original total program, and it is indeed executable. It is worth mentioning that our tool, *DBS*, is capable in forward program slicing as well, and we'll describe our approach in the case of the backward slicing in the rest part of the paper.

DBS uses the backward slicing algorithm called *Worklist Algorithm*, and it describes in Fig.2. After the backward Worklist algorithm finishes, set *S*, which includes the basic blocks sliced, is the result of this slicing algorithm.

Algorithm 1: Worklist Algorithm (Backward)

```
   Input: target program whole basic block set W,
          memory crash point address n
   Output: sliced basic block set S
 1 set V, S = ∅;
   // V saves the slicing criterions every step,
      and S saves the basic blocks selected during
      the slicing process
 2 i, j ∈ N, v_i ∈ V, w_j ∈ W;
 3 V ← n; // n is slicing start point
 4 while V ≠ ∅ do
 5     for v_i ∈ V do
 6         for j = i → 0 do
 7             if v_i is the destination operand of w_j then
 8                 V ← w_j's source operand;
 9                 S ← w_j;
10             end
11         end
12         delete v_i from V;
13     end
14 end
15 return S;
```

Figure 2. Worklist Backward Program Slicing Algorithm

2.3 Interprocedural Analysis

DBS can construct call graph and CFGs automatically during the dynamic slicing process, and imitate function call procedure, stack operations and basic block executing sequence using these two data structures. *DBS* collects function calling relationship and stores it in a tree where the callers are regarded as farther nodes and callees are as child nodes. With the call graph, users could know the function calling relationship of the target program execution hierarchically, and it is convenient for him to find the vulnerabilities and debug the program. A CFG describes the basic block execution sequence of a single function. Users can obtain the basic block dependency by the CFGs and ensure the proper execution of the slicing algorithm.

We use a bitmap to record the slicing states of all basic blocks to mark whether a certain basic block is belong to the slicing result set. This method is efficient and has low space cost. With the help of stack operation information from call graphs and CFGs, *DBS* can break through the function field limitation and implement interprocedural analysis. This expands the use scope of *DBS* greatly and enhances its practical applicability.

2.4 Pointers and Loops

Because *DBS* does the analysis work based on the basic block stream of actual running program process, it can get a lot of actual information, such as the actual value of pointers and so on. Thus, it is more precise and effective than a static method. *DBS* can deal with pointer variables since all the variables are assigned actual values according to the running program. A pointer variable is just a variable which stores the real instruction address value, and we can treat it as a common variable in dynamic binary analysis.

It is a big problem for static binary analysis that how to deal with loops since people cannot know how many times a loop executes actually, especially meeting endless loops. To address this problem, we use a solution that setting a limit value *N* for the allowable max number of a loop's execution times, and the loop will automatically terminate if the number is over *N*. Furthermore, for dynamic analysis, because the analysis objects are real executing basic blocks, the number of loop execution times depends on the specific condition of a certain execution. It is an actual value, and we do not need any special treatments. In summary, *DBS* can deal with pointers and loops perfectly.

3. EXPERIMENT

In this part, we choose several widely used applications to test *DBS* in order to validate its accuracy, efficiency and practicality.

We perform some experiments on both mature software and small applications. Since our tool has two releases, for Windows and Linux, we can do the experiments on the applications running on the two Operation Systems. *Ping* is a computer network administration utility used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer. *Netstat* (network statistics) is a command-line tool that displays network connections (both incoming and outgoing), routing tables, and a number of network interface statistics. It is used for finding problems in the network and to determine the amount of traffic on the network as a performance measurement. And the *tracert* command is used to visually see a network packet being sent and received and the amount of hops required for that packet to get to its destination. It is useful for troubleshooting large networks where several paths can be taken to arrive at the same point, or where many intermediate systems (routers or bridges) are involved. *Comp* is a simple command that compares two groups of files to find information that does not match. The *findstr* command is short for find string and is a command used in MS-DOS to locate files containing a specific string of plain text. These experiments treat designated function entry addresses as the slicing criterion, and slice out the instructions related to the input registers which we are interested in. From the results shown in Table 1, we can know that the slicing effect of *DBS* is pretty good. The numbers of sliced instructions in basic blocks have a great fall compared with the original ones in both forward slicing and backward slicing methods.

Table 1. The Result of *DBS*

Application	Function Entry Address	Original Instructions	Time Cost (ms)	Forward Slice Criterion	Forward Sliced Instructions	Backward Slice Criterion	Backward Sliced Instructions
ping	0x1002b22	108	281	ebp	52	ebp	61
netstat	0x1004fe0	21	453	eax	12	eax	2
tracert	0x1001591	184	360	esp	84	esp	108
comp	0x1002ee7	52	250	esp	23	esp	21
findstr	0x01002ca7	51	271	esp	14	esp	18

Fig.3 presents a part of call graph from the *DBS* analysis result for the *Notepad* application in Windows XP. Since the result is too complex to show if we take all the function calls of *Notepad* into consideration, we just analyze the procedures which are invoked in the *EXE* module. In the call graph we have generated, every ellipse stands for a function entry address, and the whole graph shows the call relationships of these functions.

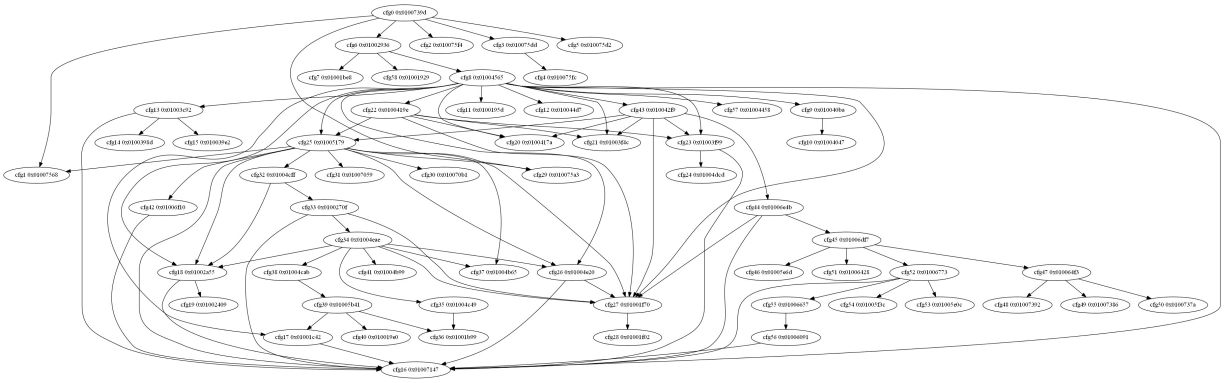

Figure 3. Notepad's Call Graph Generated by *DBS*

4. CONCLUSION

In this paper, we propose a method of attack diagnosis on binary executables using dynamic program slicing and develop the system of *DBS*. In this system, we construct the call graphs of the program to implement the interprocedural slicing technique, and construct CFGs to organize the basic blocks in procedures. The dynamic slicing is applied on the base of the call graph and CFGs. After making the attack point as slicing criterion, *DBS* successfully slices out the basic blocks related to the criterion and constructs them into an executable subset. Besides, *DBS* also organizes the slicing result to make it clear and hierarchical. Programmers could diagnose the attacks and locate the software vulnerabilities easily based on the slicing result.

5. ACKNOWLEDGMENTS

This work is supported by Key Lab of Information Network Security of Shanghai Jiao Tong University and Ministry of Public Security.

REFERENCES

[1] B. Korel and J. W. Laski, "Dynamic program slicing," IPL, 155–163 (1988).

[2] M. Mock, D. C. Atkinson, C. Chambers and S. J. Eggers, "Improving program slicing with dynamic points-to data," SIGSOFT FSE, 71–80 (2002).

[3] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," PLDI, 47–56 (2000).

[4] H. Agrawal, R. A. DeMillo and E. H. Spafford, "Debugging with dynamic slicing and backtracking," SPE, 23(6), 589–616 (1993).

[5] B. Korel and J. Rilling, "Application of dynamic slicing in program debugging," AADEBUG, 43–58 (1997).

[6] A. Nishimatsu, M. Jihira, S. Kusumoto and K. Inoue, "Call-mark slicing: An efficient and economical way of reducing slice," ICSE, 422–431 (1999).

[7] T. Wang and A. Roychoudhury, "Jslice: A java dynamic slicing tool," http://jslice.sourceforge.net/.

[8] Dynamorio: Dynamic instrumentation tool platform, http://dynamorio.org.

[9] R. Manevich, M. Sridharan, S. Adams, M. Das and Z. Yang, "Pse: explaining program failures via postmortem static analysis," SIGSOFT FSE, 63–72 (2004).

[10] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," PLDI, 57–68 (2002).