ISSN: 1832-5505

Applied GIS

Vol-8 Issue-01 Jan 2020

Slice Difference using Trace Alignment

N.Suresh[1],G.Manish[2],B.A.Sushil kumar[3]
Assistant Professor[#1],
CK College of Engineering & Technology Cuddalore

Abstract- It is not uncommon for an analyst to have to figure out why two separate executions of the same program provide different results. The same holds true when comparing the behavior of two similar programs in two distinct settings. It's the input that makes the difference between two otherwise identical programs with wildly different results. In order to analyze the differences between these executions, the authors of this study offer a technique for trace alignment that is based on execution indexing. There are two possible traces: a successful one and an unsuccessful one. The problem with the execution is tracked down and fixed.

Keywords- Indexing of Execution, Algorithm for Aligning Traces, Differential Slicing of Traces, Successful Traces, and Failed Traces.

I. INTRODUCTION

When two similar programs run with different input or operate in various environments, the security analyst must always figure out why there are discrepancies.Consider two identical programs, each of which produces two distinct outcomes when executed in the same environment but with different input.One input trace causes program to run successfully,and other input trace makes program to crash.

Here, the same software produces distinct results in response to varied input.We need to determine what caused the crash and why one set of inputs to the application triggered the problem while another set of inputs did not.We need to find the source of the crash and figure out how to prevent it.

Hidden behaviors in malware often become active when the right conditions have been met.So, only by using a variety of triggers can the primary crash zones, i.e., malware, be identified.Some dangerous activities in malware programs are only activated when predetermined circumstances are satisfied.Trigger-based behavior describes this sort of behavior.

If malicious code is designed to run in two distinct environments, it is said to be "environmentally Neither A nor B displays any signs of malevolent conduct, and C is not an executionof the same infection in environment B doesnot demonstrate harmful behavior. However, only knowing how to reproduce the problem or malicious code is insufficient.The two settings, A and B, are quite different from one another. In order to remedy the alterations that the malware makes to the environment differences, we need to first learn which subset of environment differences are really significant to trigger.

In this article, we'll look at two related programs, create a code-to-code comparison between them, and explain the key differences. Here we see two different kinds of behavior, one of which is unexpected (a crash in the execution trail) and the other of which is anticipated.Target difference refers to the variance in how an intended action is carried out.

Differential slicing methodology is utilized to automate this study.There are two types of traces that may be used to gather the necessary data: 1) the sections of the program that are changed by the input, and2)The timeline of what happened to cause the desired dissimilarity.Here, we compare two related systems that aim to locate aligned and disaligned areas.The identical program is then run using the differential slicing tool, which pinpoints and corrects the precise location of the issue.

Security companies and researchers depend on automated malware detection and analysis techniques to cope with the rising tide of harmful software. Most modern malware analysis solutions use a dynamic approach, running unknown code in a safe environment (sandbox) and analyzing its actions in real time. Dynamic analysis systems are resistant to commonly used malware security measures like packing and code obfuscation because they execute dangerous code directly. The problem is that the time it takes to run a malware sample is sometimes too short to see all conceivable dangerous behavior. Previous work has proposed solutions such as multipath or forced execution to help with this problem.

expand the coverage of dynamicmalware analysis. Unfortunately, the cost of using such methods may expand exponentially as the number of pathways necessitating examination does.

Our solution is based on the insight that we can leverage behavior observed while dynamically executing a specific malware sample to identify similar functionality in other programs. More precisely, when we observe malicious actions during dynamic analysis, we automatically extract and model the parts of the malware binary that are responsible for this behavior. We then leverage these models to check whether similar code is present in other samples. This allowsus to statically identify *dormant functionality*functionality that is not observed during dynamic analysis) in malicious programs.

This paper contains the following contributions:

1) An algorithm has been developed called Trace alignment algorithm based on *Execution Indexing* that aligns the execution traces for two runs of similar programs.It outputs the two regions that describes the similarities and differences between both executions.
2) This paper proposes a *differential slicing* technique through which the programs can be subjected to test,and find out whether it contains any bugs.
3) The byte number is identified using Execution Indexing to fix where the error or *bug* is found exactly.
4) A tool has been developed to compare and execute similar programs, wherebyfinding aligned regions and also exact statements where the error occurs.

Sample programs has been taken in C# language and lines codes are written *.NET*platform.The tool is developed in such a way that *C# programs* are compared in *.NET*framework.And program are made to execute under different input and traces are found.

II. SYSTEM OVERVIEW

In this section we describe about the problem overview and a general overview of the approach.

A. *Problem Overview*

B. Here, we take into account the following scenario. Target execution differences between two iterations of the same program are provided as execution traces for analysis. Two separate programs' inputs or the same program executed in two different system contexts may both provide execution traces[1].

C. In crash analysis, for instance, a security analyst may collect two execution traces from the same program executed with different inputs, one of which produces a crash while the other does not. Here, the analyst's primary objective is to comprehend the crash (informally, what triggered it and how it occurred) in order to fix it or take advantage of it later.In another scenario, a security analyst is provided with execution traces of malware executing in two distinct system contexts, with the virus exhibiting varying degrees of behavior in each.

D. In this case, the analyst has access to two environments that trigger the divergent behaviors, but she still has to identify which aspects of the surroundings and which checks produced the divergent behavior so that she may build a rule that avoids the trigger.By using the system environment as an input to the program, we can bring together the two scenarios.

E. Traces of anticipated activity are referred to as "passing traces," whereas traces of unexpected behavior (crash) are referred to as "failing traces."The matching inputs or environment are termed passing input and failing input.

F. *Background – Execution Indexing*

To create a connection between execution points across numerous executions of the program, Execution Indexing records the structure of the program at a certain point in the execution, giving the execution point a unique identifier [2]. Xin et al. propose an online algorithm to compute the current execution index as the execution progresses, which uses an indexing stack, where an entry is pushed to the stack when a branch or method call is seen in the execution, and an entry is popped from the stack if the immediate post-dominator of the branch is executed or the method returns. It is important to keep in mind that a single statement may pop several items from the stack if it is the immediate post dominator of numerous branches or call statements. In the context of the present function invocation, for instance, the turn instruction is the dominator of all branches on the stack. Avoiding instrumenting instructions with a single static control dependent and employing counters for loops or repeated predicates are only two examples of the improvements Xin et al. suggest to reduce the amount of push and pop operations.

Execution Indexing captures the structure of the execution beginning at an execution point that is termed an anchor point. Execution Indexing is a tool for comparing the structure of several executions by taking as input a point in each execution that is regarded semantically identical (i.e., already aligned). It is up to the analyst or the system to define these[1].

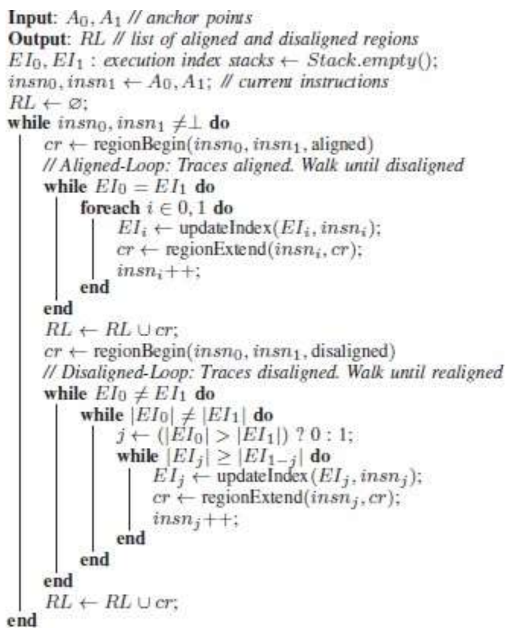
Trace Alignment Algorithm

The first step in our differential slicing approach is to align the failing and passing execution traces to identify similarities and

ISSN: 1832-5505

Vol-8 Issue-01 Jan 2020

differences between the executions. Our trace alignment algorithm builds on the previously proposed Execution Indexing technique [2], where an execution index uniquely identifies a point in an execution and can be used to establish correspondence across executions. Unlike previous work, we propose an efficient offline alignment trace algorithm that requires just a single pass over the traces and works directly on binaries without access to source code.Our trace alignment algorithm compares two execution traces representing different runs of the same program.



```
Input: $A_0, A_1$ // anchor points
Output: $RL$ // list of aligned and disaligned regions
$EI_0, EI_1$ : execution index stacks $\leftarrow Stack.empty()$;
$insn_0, insn_1 \leftarrow A_0, A_1$; // current instructions
$RL \leftarrow \varnothing$;
while $insn_0, insn_1 \neq \perp$ do
    $cr \leftarrow$ regionBegin($insn_0, insn_1$, aligned)
    // Aligned-Loop: Traces aligned. Walk until disaligned
    while $EI_0 = EI_1$ do
        foreach $i \in 0, 1$ do
            $EI_i \leftarrow$ updateIndex($EI_i, insn_i$);
            $cr \leftarrow$ regionExtend($insn_i, cr$);
            $insn_i$++;
        end
    end
    $RL \leftarrow RL \cup cr$;
    $cr \leftarrow$ regionBegin($insn_0, insn_1$, disaligned)
    // Disaligned-Loop: Traces disaligned. Walk until realigned
    while $EI_0 \neq EI_1$ do
        while $|EI_0| \neq |EI_1|$ do
            $j \leftarrow (|EI_0| > |EI_1|)$ ? 0 : 1;
            while $|EI_j| \geq |EI_{1-j}|$ do
                $EI_j \leftarrow$ updateIndex($EI_j, insn_j$);
                $cr \leftarrow$ regionExtend($insn_j, cr$);
                $insn_j$++;
            end
        end
    end
    $RL \leftarrow RL \cup cr$;
end
```

Fig.1 Trace Alignment Algorithm

Algorithm: In this research, we present an efficient implementation of trace alignment that uses a single run across both traces in parallel to calculate the execution index and the alignment. Figure 1 depicts our trace alignment method. After each trace, the Execution Indexing stack is refreshed via the update Index function. If the current instruction is a control-transfer instruction, it examines the current and next instructions to determine the right post-dominator and then pushes that instruction into the stack as the destination of the control flow transfer.

Since the current instruction is a post-dominator, it pops the dominant post off the top of the stack. Based on our findings, it is critical to provide reliable call stack tracking code [3] in order to deal with unstructured control flow (e.g.,setjmp/longjmp).The next steps of the method for trace alignment are as follows. The Aligned-Loop is used to first analyze both anchor points. By repeatedly iterating over both traces until a misaligned instruction is encountered, this loop produces the resulting aligned area. Both instructions are added to the current alignment region (cr) and the Execution Index is updated for each trace (updateIndex) while the Execution Index (EI) for the currentinstruction is the same in both traces (insn0, insn1).

Disaligned-Loop is entered when the current region is added to the output (RL), and a new disaligned region is produced (cr). The realignment point between the two traces is being sought for by this loop. Any new entries added to the stack after the moment of disalignment must be discarded before realignment can occur, hence the top entry (at the time of disalignment) must be popped before realignment can occur.Intuitively, this indicates that when the executions diverge, thefirst feasible location they may realign is at the post-dominatorof the divergence point. The Disaligned-Loop traverses both traces individuallyuntil the top item in the stack at the moment the disalignmentpoint was located has been deleted. A re-alignment of the traces has occurred if the stacks are now equal.

right after the dominant one. At this new alignedpoint, the current isalignment region terminates and Aligned-Loop resumes. If the call stack sizes are different, the bigger call stack will be explored until it is equal to or less than the smaller call stack. Once the two call stacks are the same size, the operation is complete. After that, we compare the current Execution Indexes. If they are not the same, the Disaligned-Loop will run again until the two stacks are the same size, at which point it will compare the Execution Indexes again.

Anchor point selection. To use Execution Indexing foralignment, we need an anchor point: two instructions (onein each trace) that are considered aligned. For example, if we always start tracing a programat the first

instruction for the created process, then wecan select the first instruction in both traces as anchorpoints, as they are guaranteed to be the same program point. Sometimes, starting execution traces from process creationmay produce execution traces that are too large. In thosecases, we can start the traces when the program reads itsfirst input byte, so the first instruction in each trace is an
anchor point.

DIFFERENTIAL SLICING

Determining the behavioral differences between two similar programs is the goal of Differential Program Analysis. One objective is to discover an input for whichthe two programs will create distinct outputs, thereby illustratingthe behavioral difference between the two algorithms. Because the overall issue is undecidable, an unsound orincomplete analysis is necessary[4].

It's not always clear what will happen once a modification is made to a software, whether it's to fix a bug or introduce a new feature. The update might cause problems the developer hadn't foreseen or fall short of its target entirely. It's possible the adjustment won't do anything at all. It would be useful to have an automatic analysis revealing the real impact of the modification on the behavior of the program in order to avoid undesired side effects and verify that modifications have the intended effect.

When two programs are compared to find a matching line of code, the trace alignment technique creates what we term aligned area and dis-aligned region, respectively.The specific byte location where the error occurred is then located using the same procedure to get the binary difference value. What we're getting at here is where the statements are missing. The C# code (example code) is then run via a differential slicing tool.When there are no bugs in the code, this tool causes the program to run and provide the expected results.If a mistake is identified, however, the tool may pinpoint the precise location of the problem while the user is providing input. The primary benefit of this tool is that it can be used to make any C# application operate outside of the C# environment.The problem is detected, and a warning is sent.

The number of lines of code in the precise program determines the number of lines in the aligned and disaligned regions.The aligned region contains all the adjacent areas.All crashing statements are put in disaligned area.

IMPLEMENTATION

The trace alignment algorithm is designed and made to execute in .NET framework. The sample programs are written in C# language. The code for differential slicing is written in .NET language.

CONCLUSION

In this study, we present a differential slicing instrument for protecting computer programs. A trace alignment method has been designed to locate the aligned and disalignedregions. The Execution Indexing method, previously explained, was used in the development of this algorithm. The error prone areas are pinpointed in terms of location (coordinates), line number (LN), and byte number (BY).Finally, the tool ensures that applications run well and reports any problems that arise.

ACKNOWLEDGEMENT

We would like to thank Mrs.N.Sugunafor her valuable feedback on our paper. Through her feedback we were able to improvise our paperfrom the early existing models. And we would like to thank our panel members for their comments on this paper.

REFERENCES

[1] Differential Slicing: Identifying Causal Execution Differences forSecurity Applications Noah M. Johnson†, Juan Caballero‡, Kevin Zhijie Chen†, Stephen McCamant†,PongsinPoosankam§†, Daniel Reynaud†, and Dawn Song† †University of California, Berkeley ‡IMDEA Software Institute §Carnegie Mellon University.

[2] B. Xin, W. N. Sumner, and X. Zhang.Efficient programexecution indexing. In PLDI, Tucson, AZ, June 2008.

[3] J. Caballero. Grammar and Model Extraction for SecurityApplications using Dynamic Program Binary Analysis. PhDthesis, Department of Electrical and Computer Engineering,Carnegie Mellon University, Pittsburgh, PA, September 2010.

[6] H. Cleve and A. Zeller.Locating causes of program failures.InICSE, Saint Louis, MO, May 2005.

[4] J. Winstead and D. Evans.Towards differential programanalysis. In Workshop on Dynamic Analysis, Portland, OR,May 2003.

[5] G. Liang, A. Roychoudhury, and T. Wang. Accuratelychoosing execution runs for software fault localization. InCC, Vienna, Austria, March 2006.

[6] J. R. Crandall, G. Wassermann, D. A. S. Oliveira, Z. Su,S. Felix, W. Frederic, and T. Chong. Temporal search:Detecting hidden malware timebombs with virtual machines.InOperating Systems Review, pages 25–36. ACM Press,2006.

[7] W. N. Sumner and X. Zhang. Algorithms for automaticallycomputing the causal paths of failures. In FASE, York, UnitedKingdom, March 2009.