# Differential Slicing using Trace Alignment Algorithm to Find Traces

**N. Surya[1] S. Mehala[2]**
[1,2]Department of Information Technology
[1,2]Panimalar Engineering College

*Abstract*— It is common for an analyst to identify or understand two runs of same program which produces differences in output. Same exists when two related programs are exhibited under two different environments, shows different behavior. The main difference exists depending on the input which is given to two similar programs producing different output state. This paper proposes trace alignment algorithm based on execution indexing that analysis the differences of such executions. There exists two traces as passing trace and failing trace. It finds out root cause of the failing execution.

***Key words:*** Execution Indexing, Trace Alignment Algorithm, Differential Slicing, Passing Trace, Failing Trace.

## I. INTRODUCTION

Always a security analyst has to understand why there exists differences when two similar programs are made to run with different input or made to run in different environment shows differences in their output.For example,we have two similar programs when made to run in same environment but with different input,we get two situations.One input trace makes program to run successfully,and other input trace makes program to crash.

Here same program shows differences while different input is given.We have to understand the crash and why one program input trigerred it and another input did not do it.And we have to identify the way for the bug which caused the crash can be avoided.

Malware often contains a hidden behavior which is activated when only a proper trigger is applied. So, the main crashing areas i.e., malware can be found out only by giving different inputs as triggering. In many malware programs, certain code paths implementing malicious behaviors will only be executed when certain trigger conditions are met. Those type of behaviors are called triggered based behavior.

If a piece of malware is made to execute under two different, environment A does not exhibit malicious behavior , and another trace of an execution of the same malware in environment B doesnot exhibit malicious behavior. However , knowing how to trigger the hidden behavior is not sufficient for identifying the bugs or malware.There are many differences between environments A and B. But we need to understand which subset of environment differences are truly relevant to trigger,and also fix the changes that the malware performs on the environment differences.

Here in this paper two similar programs are made to compare with code match,and the differences between them are stated. Here we find two cases , the execution trace which contain unexpected behavior i.e., crash and another with expected behavior.The execution difference is called as target difference.

To automate this analysis a method called differential slicing approach is used.Two different traces is occurs from which needed information is obtained as 1)parts of the program which are affected by the input and 2)the sequence of the events that led to the target difference.

Here two similar programs are compared to find aligned and disaligned regions.Then same program is made to run using differential slicing tool,and it fixes the exact place of error.

To handle the growing flood of malware, security vendors and analysts rely on tools that automatically identify and analyze malicious code. Current systems for automated malware analysis typically follow a dynamic approach, executing an unknown program in a controlled environment (sandbox) and recording its runtime behavior. Since dynamic analysis platforms directly run malicious code, they are resilient to popular malware defense techniques such as packing and code obfuscation. Unfortunately, in many cases, only a small subset of all possible malicious behaviors is observed within the short time frame that a malware sample is executed. To mitigate this issue, previous work introduced techniques such as multipath or forced execution to increase the coverage of dynamic malware analysis. Unfortunately, using these techniques is potentially expensive, as the number of paths that require analysis can grow exponentially.

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 allows us 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, whereby finding 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.

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

### *A. Problem Overview:*

We consider the following problem setting. We are given execution traces of two runs of the same program that contain some target execution difference to be analyzed. The two execution traces may be generated from two different program inputs or from the same program running in two different system environments[1].

For example, in crash analysis, a security analyst may have two execution traces obtained by running a program with two similar inputs where one input causes a crash and the other one does not. Here, the analyst's goal is first to understand the crash (informally, what caused it and how it came to happen), so that she can patch or exploit it.In a different application, a security analyst is given execution traces of a malware program running in two system environments, where the malware behaves differently in both environments, e.g., launches a denial of service attack in one environment but not in the other.

Here, the analyst has access to two environments that trigger the different behaviors, but still needs to understand which parts of the environment as well as which checks caused the different behavior, so that she can write a rule that bypasses the trigger.We can unify both cases by considering the system environment as a program input.

To differentiate between them we say the expected behavior trace as passing trace and the unexpected behavior(crash) as failing trace.The corresponding inputs or environment are called passing input and failing input.

### *B. Background – Execution Indexing:*

Execution Indexing captures the structure of the program at any given point in the execution, uniquely identifying the execution point, and uses that structure to establish a correspondence between execution points across multiple executions of the program [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. Note that a statement may be the immediate post dominator of multiple branches or call statements and can thus pop multiple entries from the stack. For example, are turn instruction is the immediate post-dominator of all the branches in the stack for the current function invocation. Xin et al. also propose optimizations to minimize the number of push and pop operations for cases that include, among others, avoiding instrumenting instructions with a single static control dependence and using counters for loops or repeated predicates[3].

Execution Indexing captures the structure of the execution starting at an execution point that is called an anchor point. To compare the structure of two executions, Execution Indexing requires as input a point in each execution considered semantically equivalent (i.e., already aligned). These can be automatically defined or provided by the analyst[1].

### *C. Trace Alignment Algorithm:*

The first step in our differential slicing approach is to align the failing and passing execution traces to identify similarities and 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 ← Stack.empty();
insn_0, insn_1 ← A_0, A_1; // current instructions
RL ← ∅;
while insn_0, insn_1 ≠ ⊥ do
    cr ← regionBegin(insn_0, insn_1, aligned)
    // Aligned-Loop: Traces aligned. Walk until disaligned
    while EI_0 = EI_1 do
        foreach i ∈ 0, 1 do
            EI_i ← updateIndex(EI_i, insn_i);
            cr ← regionExtend(insn_i, cr);
            insn_i++;
        end
    end
    RL ← RL ∪ cr;
    cr ← regionBegin(insn_0, insn_1, disaligned)
    // Disaligned-Loop: Traces disaligned. Walk until realigned
    while EI_0 ≠ EI_1 do
        while |EI_0| ≠ |EI_1| do
            j ← (|EI_0| > |EI_1|) ? 0 : 1;
            while |EI_j| ≥ |EI_1-j| do
                EI_j ← updateIndex(EI_j, insn_j);
                cr ← regionExtend(insn_j, cr);
                insn_j++;
            end
        end
    end
    RL ← RL ∪ cr;
end
```

Fig. 1: Trace Alignment Algorithm

#### *1) Algorithm:*

In this paper we propose an efficient trace alignment implementation that performs a single pass over both traces in parallel, computing the execution index and the alignment along the way. Our trace alignment algorithm is shown in Fig.1. The function update Index updates the Execution Indexing stack for each trace. If the current instruction is a control-transfer instruction, it selects the correct post-dominator by looking at the current and next instruction (i.e., the target of the control flow transfer)and pushes the post-dominator into the stack.

While the current instruction corresponds to the post-dominator at the top of the stack, it pops it. Our experience shows that it is important to handle unstructured control flow (e.g.,setjmp/longjmp), which requires building robust call stack tracking code [3].The trace alignment algorithm proceeds as follows. It starts with both anchor points being processed in the Aligned-Loop. This loop creates an aligned region by stepping through both traces

 

until a disaligned instruction is found. While the Execution Index (EI) for the current instruction in each trace (insn0,insn1) is the same,both instructions are added to the current alignment region (cr) and the Execution Index is updated for each trace (updateIndex).

At a divergence point, the current region is added to the output (RL), a new disaligned region is created (cr) and Disaligned-Loop is entered. This loop searches for the realignment point in the two traces. Realignment can only happen after the top entry (at the time of disalignment) on the stack has been popped, because in order for the Execution Indexes to match, any additional entries added to the stack after this point will first need to be dropped. Intuitively, this means that when the executions diverge, the first possible place they can realign is at the post-dominator of the divergence point. The Disaligned-Loop walks both traces individually until the top entry in the stack at the time the disalignment point was found has been removed. If the stacks are equal at this point, it means that the traces have realigned at the immediate post-dominator. The current is alignment region ends and Aligned-Loop continues at this new aligned point. If the call stacks are unequal in size, the trace with the larger call stack is traversed until its call stack matches or falls below the size of the other trace's call stack. This process is repeated until the two call stacks are equal in size. Then, the current Execution Indexes are compared. If not equal, the Disaligned-Loop repeats, popping the current top entry until the two stacks are equal in size, then recomparing the Execution Indexes.

*2) Anchor Point Selection:*

To use Execution Indexing for alignment, we need an anchor point: two instructions (one in each trace) that are considered aligned. For example, if we always start tracing a program at the first instruction for the created process, then we can select the first instruction in both traces as anchor points, as they are guaranteed to be the same program point. Sometimes, starting execution traces from process creation may produce execution traces that are too large. In those cases, we can start the traces when the program reads its first input byte, so the first instruction in each trace is an anchor point.

## II. DIFFERENTIAL SLICING

Differential Program Analysis is the task of analyzing two related programs to determine the behavioral difference between them. One goal is to find an input for which the two programs will produce different outputs, thus illustrating the behavioral difference between the two programs. Because the general problem is undecidable, an unsound or incomplete analysis is necessary[4].

When making a change to a program, either to correct a known error or to add a new feature, the consequences of the change are not always fully understood. The change may have unintended side effects that were not anticipated by the programmer, or may fail to accomplish the intended goal. The change may even have no effect at all. In order to prevent unintended side effects and verify that changes have the intended effect, it would be helpful to have an automated analysis showing the actual effect of the modification on the program's behavior.

When two similar programs are compared to match code, trace alignment algorithm produces two traces as, passing trace and failing trace, which we call as aligned region and dis-aligned regions.

Then using same algorithm we find the binary difference value i.e., byte position where the exact bug has occurred. In simple way to say as where the statements are missing. Then using differential slicing tool we run the C# programs (sample programs). This tool makes the program to execute and produce when there is no error in it. But if any error is found in the code, while giving input the tool exactly identifies the error in the program. The main advantage of this tool is that any C# programs can made to run using this tool without using the C# environment.It identifies the error and notifies it.

The lines in aligned and disaligned region depends upon the lines of code in exact program.All the consecutive regions are placed in aligned region.All crashing statements are placed in disaligned region.

## III. 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.

## IV. CONCLUSION

In this paper we have proposed a differential slicing tool for security of programs. A trace alignment algorithm has been developed to find the aligned and disaligned regions. This algorithm is developed using the previous approach of Execution Indexing which is discussed earlier. This tool also identifies and shows the exact place, line number,byte number of the error prone ares.And finally tool executes error free programs and notifies the error when occurred.

## V. ACKNOWLEDGEMENT

We would like to thank Mrs.N.Suguna for her valuable feedback on our paper. Through her feedback we were able to improvise our paper from 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†,Pongsin Poosankam§†, 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 program execution indexing. In PLDI, Tucson, AZ, June 2008.

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

[4] H. Cleve and A. Zeller. Locating causes of program failures.In ICSE, Saint Louis, MO, May 2005.

 

[5] J. Winstead and D. Evans. Towards differential program analysis. In Workshop on Dynamic Analysis, Portland, OR,May 2003.
[6] G. Liang, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In CC, Vienna, Austria, March 2006.
[7] 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.In Operating Systems Review, pages 25–36. ACM Press,2006.
[8] W. N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In FASE, York, United Kingdom, March 2009.
[9] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In PLDI, San Diego, CA,June 2007.



 