International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 5– No.7, May 2013 – www.ijais.org

Identifying the Behavioral Difference using Differential Slicing

N. Suguna
Computer Science and Engineering,
Annamalai University
Chidambaram, Tamilnadu, India,

RM.Chandrasekaran
Computer Science and Engineering,
Annamalai University
Chidambaram, Tamilnadu, India,

ABSTRACT
The programmer has to understand the behavior of two similar programs and then identify the execution difference which produces difference in output. When two similar programs are executed under two different environments which shows different behavior in output. The main difference exists in the program behavior is due to two different types of input. This paper proposes differential slicing based on trace alignment algorithm which produces the execution differences and generates a casual difference graph. We implement differential slicing for C# programs and identify the execution difference. The results shows that differential slicing identifies the input difference and casual difference graph reduces the amount of time for the programmers to understand the execution difference. Our experimental results show the proposed differential slicing performs better than existing approach.

Keywords
Casual Difference Graph (CDG), Program Dependence Graph (PDG)

1.Introduction:
Differential program Analysis is the process of finding two similar programs to determine the behavioral difference between them.Programs are frequently maintained by the programmers who are far removed from software development process. The purpose of modifications in the program development is not always clear. The actual effect on the programs behavior is the presence of a particular part of the program may be unknown,a particular line may be crucial or it may have no effect at all. The previous research shows that difference in behavior is caused by the presence or absence of particular element would help the maintainers in understanding the program. A set of automated techniques is needed to analyze the effect of modifications. We use differential slicing to focus on differences between two similar programs [1].

We proposed a differential slicing for C# programs. It identifies the aligned and disaligned region and generates a casual difference graph. We compare the differential slicing and existing approach and the results are tabulated. Our results also show that differential slicing identifies the input difference and CDG decreases the time and effort needed for an analyst to understand the observed difference.

The paper is organized as follows. Section 2 describes the existing differential slicing techniques. Section 3 explains about Dynamic slicing. Section 4 explains about differential slicing. Section 5 describes about slice alignment. Section 6 describes the performance result of differential slicing. Section 7 describes conclusion.

2. Related Work

Differential program analysis is the task of analyzing two related programs which identified the behavioral difference between them. Joel Winsted et al presents a technique to find an input for which the two programs will produce different outputs ,Thus illustrating the behavioral difference between the two programs [1]. A combination of static and dynamic techniques are used to find the differential inputs. Sumner and Zhang [4] finds the casual path of two executions by first patching the failing execution dynamically. In order to find this by modifying variables and predicates at runtime to produce a passing execution.

Zhang et al proposed a technique to find the execution omission errors[5].If a patch is found then both runs are aligned and similar variables are identified through value mutations. There are many existing techniques for finding trigger based behavior in malware. Temporal search identifies the time based behavior byperturbing thesystem time of a virtual machine and observing for different behaviors[6]. Similarly [7][8] uses dynamic analysis to explore multiple execution paths in order to find the hidden behavior in malware.

The most widely used debugging technique proposed by Weiser is program slicing [9] which produces a slice containing parts of a program that are relevant to find the value of a particular variable called slicing criterion. Korel and Laski presents a technique called dynamic slicing which works on single execution and outputs the executed statements relevant to slicing criterion. There are four forms of dynamic slicing based on the dependencies in the slice. Thin slicing [10] contains a subset of data dependencies, data slicing [11] consists of all data dependencies, full slicing [12] include data and control dependencies and relevant slicing [13], [14] includes data and control dependencies in addition to that predicates and chains of potential dependencies rooted at these predicates.

Delta debugging is a technique used to identifying the failure inducing inputs automatically [15]. Zeller et al proposed a failure analysis method that uses delta debugging to compare the status of a faulty and correct execution at the fault is found [15][16]. Xin et al [17] proposed a technique called execution indexing to find the correspondence between points across executions. Trace Alignment algorithm is based on execution

indexing. Liang et al [18] uses execution traces for fault localization.

In this paper we study the execution difference of two similar programs and generate a casual difference graph (CDG). First, We generate CDG for identifying casual difference for DotNet programs. Second we compare the dynamic slicing and the proposed differential slicing and the results are tabulated to analyze the performance.

3. Dynamic slicing

Program slicing is the task of finding all statements in a program that directly or indirectly influence the value of a variable occurrence. The set of statements that can affect the value of a variable at some point in a program is called a program backward slice. In several software engineering applications, such as program debugging and measuring program cohesion and parallelism, several slices are computed at different program points[2]. Program slicing can be static or dynamic. In the static program slicing it is required to find a program slice that involves all statements that may affect the value of a variable at a program point for any input set. In dynamic program slicing the slice is found with respect to a given input set. Many algorithms have been introduced to find static and dynamic slices. These algorithms compute the slices automatically by analyzing the program data flow and control flow.

Definition (Dynamic slicing criterion):

A dynamicslicing criterion is a tuple (T, xn) where T is a test case, xis a variable, and n is the index of the executed statement of interest.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Step2
{
class Program
{
static void Main(string[] args)
{
1:int  i = 0;
2:int prod=1, sum = 0,size;
 3:while (i<size) {
 4:sum = sum + a[i];
 5:i = i + 1; }
 6:prod = sum * (size + 1);
 7;if (size > 0) {
 8:all = sum + size;
 9:} else {
10:all = sum * size; }
}}}
```

Fig. 1. Sample dotnet program prod with an array a of size as input, and prod, all as outputs.

Algorithm Dynamic Slice:

Inputs: An execution trace $ET = \langle (1)1, \ldots, s(m)m \rangle$of a program Π and a test case T, and a slicing criterion $(T, xn), n \leq m$.

Outputs: A dynamic slice.

1) Compute the execution trace graph $ETG$for ET

using the definitions of $\rightarrow C$and $\rightarrow D$.

2) Mark the node (k)in$ETG$where $x \in ((k)k)$ and there is no $s(i)i$, $k<i \leq n, x \in ((i))$ in ETG.

3) Traverse the graph $ETG$from the marked node in the reverse direction of the arcs in ETGuntil no new nodes can be marked.

4) Let Sbe the set of all marked nodes.

5) Return the set $\{(i) | s(i)i \in S\}$ as the result.

Fig. 2.Algorithm dynamic slice.

For example in fig 1. shows the sample DotNetprogram for product of numbers. The dynamic slice for our example and the slicing criterion (({a = [1, 1]; size = 2}, {prod = 6, all =4}), $prod^8$) is the following program fragment:

1. i = 0;
2. sum = 0;
3. while (i<size) *{*
4. sum = sum + a[i];
5. i = i + 1; *}*
6. prod = sum * size;

3.1 Program Dependency Graph (PDG)

The program dependence graph (pdg) consists of nodes and direct edges. Each program's simple statement and control predicate is represented by a node. Simple statements include assignment, read, and write statements. Compound statements include conditional and loop statements and they are represented by more than one node. There are two types of edges in a pdg: data dependence edges and control dependence edges. A data dependence edge between two nodes implies that the computation performed at the node pointed by the edge directly depends on the value computed at the other node. This means that the pointed node has the definition of the variable used in the other node. A control dependence edge between two nodes implies that the result of the predicate expression at the node pointed by the edge decides whether to execute the othernode or not. We generate a program dependency graph for our sample program. Fig 3 shows the program dependenc graph.

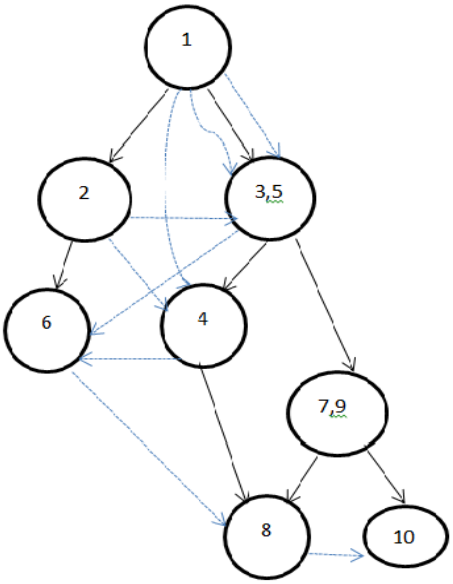

Fig.3 Program Dependency Graph

4: Proposed Approach

We take execution traces of two runs of the same program difference to be analyzed. The two execution traces are generated from two different program inputs or from the same program running in two different system environments.

To understand the target difference which consist of

1. Finding the input difference that caused the target difference.

2. Understanding the sequence of events that led from input difference to target difference

To find the casual difference graph using differential slicing. Fig 4 represents the architecture of our approach. It consists of three phases [3].

1.Preparation

2.Trace Alignment

3.Slice Alignment

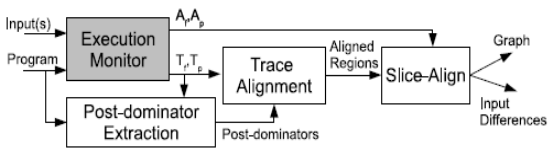

Fig. 4. Architecture of Differential Slicing

4.1. Preparation:

The preparation phase consists of two steps. In the first step the program is executed twice on the given inputs inside the execution monitor. The execution monitor tracks the executions of each program and produces execution traces T_f,T_p,A_f and A_p.T_f consists of all executed instructions and T_p is the contents of each instructions operands. A_f and A_p provides information about the heap allocation/deal location operands performed by the program during each run.The second step in post dominator extraction which takes input as program and execution traces and computes the control flow graph for each function and outputs the post dominator information.

4.2. TraceAlignmentAlgorithm:

The third step is trace alignment which identifies the execution differences that form the casual difference graph. The TraceAlignment algorithm is based on execution indexing[29]. It outputs the aligned and disaligned region in a single pass over the traces. Given size of m and n instruction the passing (*p*) and failing(*f*) traces respectively, a pass of statement from these traces(p_x,f_y).s.t that $x \in [1,n]$, $y \in [1,m]$ are aligned if they are correspond to each other. We say that a statement in disaligned in one trace if it has no corresponding statement in other trace are represented by a pair(p_x,$\perp$) or ($\perp$,f_y).Trace alignment marks aligned or disaligned for each statement. We group them into regions based alignment.

An aligned region is a maximal continual sequence of aligned statements:

$(p_x,f_y),(p_{x+1} , f_{y+1}) \ldots (p_{x+k} , f_{y+k})$ st$\forall_i \in [0,k]$ $p_{x+i} , f_{y+i} \neq \perp$

A disaligned region is a maximal continual sequence of disaligned statements.

$(p_x,\perp) \ldots (p_{x+k},\perp)$ or $(\perp,f_x) .. (\perp,f_{x+k})$

A disaligned region is present after the aligned region. The last statement in the aligned region is termed as divergence point, because it creates a disaligned by changing the control to different statements in both the traces. Fig.5 shows the example DotNet program for alignment. The Fig.7 shows that the two executions are aligned until #4 executes. Here statements #2-#4 in each trace form an aligned region. Branch statement #5 is divergence point.It checks the condition true in passing trace and false infailingrun by creating a disaligned region, because Stmt#5 executes in the passing run but not in failing run (an execution omission). The two executions are realigned at statement #6 and remain aligned until statement #7 produces crash in failing trace. Thus statement #6, #7 form another aligned region.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Step2
{
class Program
{
static void Main(string[] args)
{
long n,I,osum=0,esum=0;
Console.WriteLine("enter the limit:");
n = Convert.ToInt64(Console.ReadLine());
for(i=1;i<=n;i++)
{
if(i % 2 == 0)
esum = esum + i ;
else
osum = osum + i ;
Console.WriteLine(" The odd numbers sum is:"+osum);
Console.WriteLine(:The even numbers sum is :"+esum);
}
}
```

Fig. 5. Example program to find sum of odd and even numbers

Input: A_o,A_1 //anchor points
Output: *RL* //list of regions
EI_0,EI_1 : execution index stacks←Stack empty();
ins_0,ins_1←A_0,A_1; //current instruction
RL← Ø;
while ins_0,$ins_1 \neq \perp$ **do**
cr←regionBegin(ins_0,ins_1,aligned)
// Aligned-Loop: Traces aligned. Walk until disaligned
while $EI_0 = EI_1$ do
foreach $i \in 0,1$ **do**
EI_iupdateIndex(EI_i , ins_i);
cr ← regionExtend(ins_i,cr);
ins_i++;
end
end
RL *RL U cr*;
crregionBegin(ins_0,ins_1,disaligned);
//Disaligned-Loop: Traces daligned, Walk until realigned
while $EI_0 \neq EI_1$do
while $|EI_0| \neq |EI_1|$ **do**
j ($|EI_0| > |EI_1|$) ? 0:1;
while $|EI_j| \geq |EI_{\cdot j}|$ **do**
EI_i←updateIndex(EI_i, ins_i);
cr←regionExtend(ins_i,cr);
ins_j++;
end
end
end
RL *RL*←*U cr*;
end

Fig. 6 Trace Alignment Algorithm

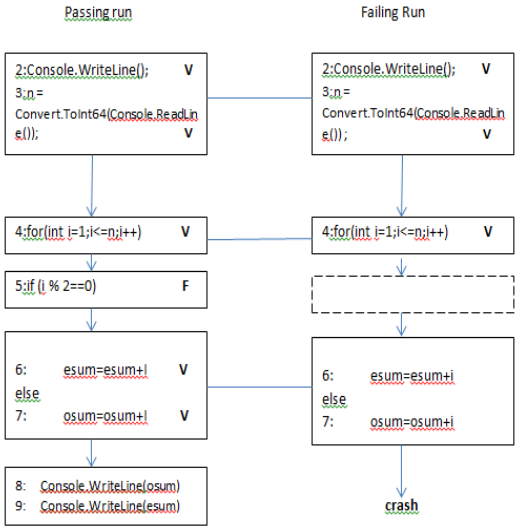

Fig.7. Traces and alignment example

A. Execution Differences:

Execution differences are values that differ across runs or statements executed in only one run. However for finding that a value differs or such a statement appears only in one trace needs to establish a correspondence between statements in both traces. It is quite difficult because the same statement may appear multiple times occur in loops, recursive functions, or calling of same function in different contexts. The process of identifying such correspondence is called trace alignment and for finding the execution differences. Given two aligned executions there are two types of execution differences: flow differences and value differences. A flow difference is simply a disaligned statement value difference is used in aligned statement that has a different value in both executions. For example statement #5 in Fig.7 is a flow difference n variable in statement #3 has value 2 in passing run and has value Ram in failing run. We say that statement has differences in value when it uses one or more variables that are value differences.

4.3. Slice Alignment:

The last step is slice Alignment which is used to create the casual difference graph dynamically as the execution traces are scanned backwards in lockstep, starting from the target difference.

B. Casual difference graph

The Casual difference graph consists of the sequence of execution differences to the target differences. The CDG stats from the input difference because those are root cause of all execution differences. It identifies the sequence of events that cause the target difference and the casual path which contains flow differences and value differences. The CDG is more precisely than the full list of execution differences between both runs, since not all execution differences may be relevant to the target difference. Fig.9. Shows the CDG for sum of odd and even numbers. As it processes from start to end with the target difference .statement #5 crashes in the failing run.

Input: *TD /* target difference */, RL /* alignment results */*

Output: *N, E // nodes and edges in causal difference graph*

Worklist: stack of instruction-pairs ← (TD_p, TD_f);

Processed: Boolean lookup table ←∅;

while *! worklist.is Empty()* **do**

($insn_p$, $insn_f$) ←worklist.pop();

$N_{p,f}$←$N_{p.f}$∪$insn_{p,f}$;

Processed ($insn_p$, $insn_f$) ← true;

if *isAligned ($insn_p$, $insn_f$, RL)* **then**

slice_operands←valdifferences ($insn_p$, $insn_f$);

forall *operand ∈slice_operands***do**

dep←immDataDeps (operand);

$E_{p,f}$←$E_{p,f}$∪ new Edge ($insn_{p,f}$←$dep_{p,f}$);

if*! Processed (dep_p, dep_f)* **then**

end *Worklist.push (dep_p, dep_f);*

end

else

dtype←divRegion Type ($insn_p$, $insn_f$, RL);

switch*dtype***do**

// See Table I for explanation for divergence types

end

case*ExtraExec or ExecOmission or ExecDiff*

div ←domDivPt (dtype, $insn_p$, $insn_f$,RL);

$E_{p,f}$←$E_{p,f}$∪ new Edge ($insn_{p,f}$→$div_{p,f}$);

if! Processed (div_p, div_f) then

Worklist.push (div_p, div_f);

Case*invalidPointer*

if*wildwrite ($insn_p$, $insn_f$)***then**

$aligned_p$←alignedInsn ($insn_f$, RL)

if*! Processed ($aligned_p$, $aligned_f$)* **then**

Worklist.push ($aligned_p$, $insn_f$);

end

end

end

end

end

Fig. 8 Algorithm for casual difference graph.

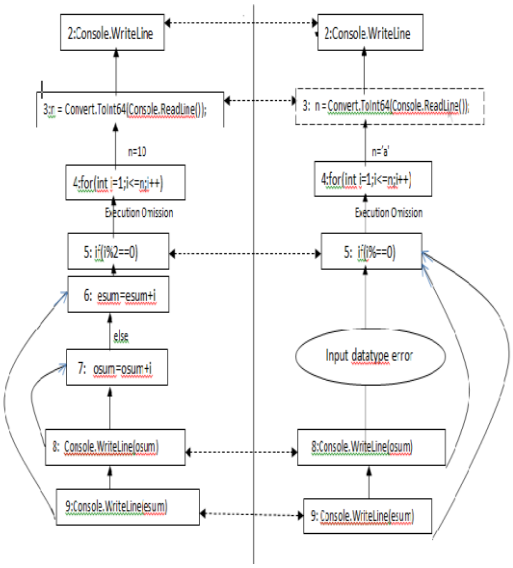

Fig. 9 Casual Difference Graph

Table1.Traces and time to generate slice align graph.

S.No.	LOC	Pass (KB)	Fail (KB)	Tracing		Trace Align (Sec)	Slice Align (Sec)
				(µs)	(µs)		
1.	15	0.7	0.4	10	8	7	6.1
2.	70	3	2.1	30	27	30	10.4
3.	40	2	1.6	22	20	26	15
4.	50	2	1.6	24	21	22	94
5.	72	3	1.9	30	27	31	20
6.	50	2	1.6	24	21	22	94
7.	25	1	0.6	18	19	17	14
8.	30	1.3	0.9	20	21	19	17
9.	25	1	0.6	18	19	17	14
10.	32	1.2	0.9	20	21	19.8	15
11.	28	1.1	0.7	18.7	19.3	17.5	14.2
12.	20	1	0.6	15	13.9	16.5	15.9
13.	15	0.7	0.4	10	8	10.1	12
14.	25	1	0.6	18	19	17	14
15.	20	1	0.6	15	13.9	16.5	15.9
16.	35	1.3	0.9	22.7	21.3	20	72
17.	27	1	0.6	18.7	19.3	16.9	13.1
18.	100	4	3.1	48.1	47.4	50	94
19.	32	1.2	0.9	20	21	19.8	12.2
20.	150	5	3.9	60.3	59.8	58	109

Performance:

Table 1 show the performance evaluation of trace alignment and slice alignment. It includes the size of passing and failing traces the time needed to align the traces of C# programs. It also includes the time taken to generate the slice align graph for example the time taken to trace for interface program is 26 (sec) but for slice align never takes 15 (sec). So we conclude that it saves the time of the security analyst when compared to analyst manual work.

Table 2: Performance evaluation

Name of Program	Number of lines in the program	Average slicing time (microseconds) Dynamic slicing	Average slicing time (microseconds) Differential slicing
1. Sorting.cs	15	475.2	250.6
2. Inheritance	15	475.2	247.3
3. Interface	20	477.6	227.3
4. String operations	25	490.5	231.3
5. Thread	25	483.3	203.2
6. Binarysearch	25	490.5	246.9
7. Operator overloading	27	480.1	223.1
8. Dining philosopher	28	481.3	225.3
9. Linked list	30	485.1	228.7
10. Animation	32	487.2	248.9
11.Radiobuttonlist	32	486.7	219.7
12. Tower of hanoi	35	489.5	228.7
13. Database	35	489.5	224.2
14. Matrix operations	40	550.6	225.2
15. Queue	50	563.4	210.7
16. Fileoperations	50	560.2	229.3
17. Tic-Tac-Toe	70	590.4	231.3
18. Binary tree	72	590.3	208.2
19. Stack	120	1250.8	951.2
20. calculator	154	1280.3	960.4

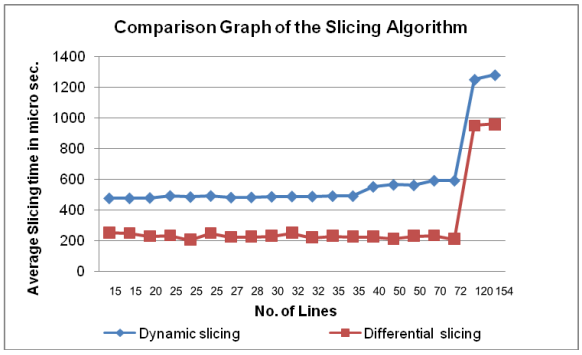

Fig. 10 Comparison Graph of the Slicing Algorithm

Table 2 and 3 shows the average slicing time for dynamic and differential slicing. For example database program the average dynamic slicing time is 489.5 (micro secs) but for differential slicing same program it tooks 224 (micro seconds). So we conclude that differential slicing takes less amount of time for slicing when compared to dynamic slicing. From the graph we can conclude that the size of the program increases, the average slicing time also increases. There may be sudden increase in the average slicing time as in our case, because of complexity of the program.

CONCLUSION

In our proposed work we indentify the behavioral difference between the two similar programs. We used trace alignment algorithm for producing the aligned and disaligned region and also to identify the flow difference & value difference of a variable. We proposed a slice alignment algorithm for c# programs to generate a casual difference graph. We compare dynamic slicing & differential slicing and the time taken to slice the c# programs. We conclude that slicing time for differential slicing is less when compared to dynamic slicing for DotNet programs.

REFERENCES

[1] J. Winstead and D. Evans. Towards differential program analysis. In *Workshop on Dynamic Analysis*, Portland, OR, May 2003.

[2] Franz Wotawa, "On the use of constraints in dynamic slicing for program debugging", Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011.

[3] Noah M.Johnson and Juan caballero et al, "Differential slicing: Identifying casual executions differences for security applications", IEEE symposium on security and privacy, 2011.

[4] W. N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FASE*, York, United Kingdom, March 2009.

[5] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI*, San Diego, CA, June 2007.

[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. In *Operating Systems Review*, pages 25–36. ACM Press, 2006.

[7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense"*, 2007.

[8] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M. Weiser. Program slicing. In *ICSE*, San Diego, CA, March 1981.H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6), June 1990.

[10] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, San Diego, CA, June 2007.

[11] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, Portland, OR, May 2003.

[12] B. Korel and J. Laski. Dynamic program slicing. *Info. Proc. Letters*, 29(3), October 1988.

[13] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London Incremental regression testing. In *ICSM*, Montreal, Canada, September 1993.

[14] T. Gyim´othy, A. Besz´edes, and I. Forgacs. An efficient relevant slicing method for debugging. In *ESEC*, Toulouse, France, September 1999.

[15] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28, February 2002.

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

[17] W. N. Sumner and X. Zhang. Memory indexing: Canonical zing addresses across executions. In *FSE*, Santa Fe, NM, November 2010.

[18] G. Liang, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, Vienna, Austria, March 2006.

[19] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, Washington, D.C., August 2003.

[20] 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.

[21] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 61–76, Washington, DC, USA, 2010. IEEE Computer Society.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987.

[23] M. J. Harrold, Y. G. Rothermel, Z. K. Sayre, Z. R. Wu, and L. Y. Z. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10:2000, 2000.

[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12:26–60, January 1990.

[25] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *VMSec*, Chicago, IL, November 2009.

[26] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[27] P. Porras, H. Saidi, and V. Yegneswaran. A foray into Conficker's logic and rendezvous points. In *LEET*, Boston, MA, April 2009.

[28] Symantec Corporation. W32.Netsky.C. http://www.symantec.com/security J. Winstead and D. Evans. Towards differential program analysis. In *Workshop on Dynamic Analysis*, Portland, OR, May 2003.

[29] response/writeup.jsp? docid=2004-022417-4628-99.

[30] TEMU: The Bit Blaze dynamic analysis component. http:// bitblaze.cs.berkeley.edu/temu.html.

[31] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test case generation: The optimizationbased approach. In *In International Workshop on Dependable Computing and Its Applications*, pages 169–180, 1998.

[32] D. Weeratunge, X. Zhang, and W. N. S. S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *ISSTA*, Trento, Italy, July 2010.

[33] J. Winstead and D. Evans. Towards differential program analysis. In *Workshop on Dynamic Analysis*, Portland, OR, May 2003.

[34] T. Xie and D. Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. Technical report, FSE Poster Session, 2002.

[35] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, Tucson, AZ, June 2008.