Introduction

• Dynamic program analysis can be constructed from a set of primitives
  - Tracing
  - Checkpointing and replay
  - Dynamic slicing

• Applications
  - Dynamic information flow tracking
  - Abnormal behavior detection
  - Malware analysis

• Resulting in:
  - Relatively lower learning curve.
  - Precision.
  - Applicability.
  - Scalability.

Program Tracing

• Tracing is a process that faithfully records detailed information of program execution (lossless).
  - Control flow tracing
    • the sequence of executed statements.
  - Dependence tracing
    • the sequence of exercised dependences.
  - Value tracing
    • the sequence of values that are produced by each instruction.
  - Memory access tracing
    • the sequence of memory references during an execution

• The most basic primitive.
Why Tracing

- Malware analysis
- Abnormal behavior detection
- Forensic analysis

Outline

- What is tracing
- Why tracing
- How to trace
- Reducing trace size

Tracing by Printf

Max = 0;
for (p = head; p; p = p->next)
{
    printf("In loop\n");
    if (p->value > max)
    {
        printf("True branch\n");
        max = p->value;
    }
}

The Minimum Set of Places to Instrument

if (...)      if (...)
  S1               S1
else           else
  S2       if (...)else
  S3           S3
if (...)     S4       else
else
  S5

Tracing by Source Level Instrumentation

- Read a source file and parse it into abstract syntax trees (ASTs).
- Annotate the parse trees with instrumentation.
- Translate the annotated trees to a new source file.
- Compile the new source.
- Execute the program and a trace produced.

An Example

Source:
for i := 1 to 10 do
  a[i] := b[i] * 5;
end

AST:
An Example

Source:
```c
for i = 1 to 10 do
    if i = b[i] * 5;
    and
    printf("in loop\n")
```

AST:
```
  1
 /\  
 /  
```

Limitations of Source Level Instrumentation

- Hard to handle libraries.
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- Hard to handle multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
- Requires source code
  - Worms and viruses are rarely provided with source code.

Tracing by Binary Instrumentation

- What is binary instrumentation
  - Given a binary executable, parses it into intermediate representation. More advanced representations such as control flow graphs may also be generated.
  - Tracing instrumentation is added to the intermediate representation.
  - A lightweight compiler compiles the instrumented representation into a new executable.
- Features
  - No source code requirement
  - Easily handle libraries.

What is intermediate representation?

- An IR is designed to be conducive for further processing, such as optimization & translation.
- A "good" IR must be
  - accurate – capable of representing the source code without loss of information, and
  - independent of any particular source or target language.
- An IR may take one of several forms: an in-memory data structure, or a special tuple- or stack-based code readable by the program.

Static vs. Dynamic Instrumentation

- Static: takes an executable and generate an instrumented executable that can be executed with many different inputs
- Dynamic: given the original binary and an input, starts executing the binary with the input, during execution, an instrumented binary is generated on the fly; essentially the instrumented binary is executed.
Dynamic Binary Instrumentation - Valgrind

- Developed by Julian Seward at Cambridge University.
  - Google-O’Reilly Open Source Award for "Best Toolmaker" 2006
  - A merit (bronze) Open Source Award 2004
- Open source
  - works on x86, AMD64
- Easy to execute, e.g.:
  - valgrind --tool=memcheck ls
- It becomes very popular
  - One of the two most popular dynamic instrumentation tools
  - Very good usability, extendibility, robust
  - Mozilla, MIT, Berkeley-security, and many other places
- Overhead is the problem
  - ~5-10X slowdown without any instrumentation
- Reading assignment
  - Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation (PLDI07)

Valgrind Infrastructure

Input

Binary Code

VALGRIND CORE

BB Decoder

BB Compiler

Trampoline

Tool 1

Tool 2

......

Tool n

Instrumenter

Runtime

New pc

New BB

pc

BB

Trampoline

INPUT:

1: do {
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

OUTPUT:

1: print("1")
2:     i=i+1;
3:     s1;
4: } while (i<2)
5: s2;

OUTPUT:
Outline

- What is tracing
- Why tracing
- How to trace
- Reducing trace size

Fine-Grained Tracing is Expensive

```
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
6: endwhile
7: print(sum)
```

Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6
Space Complexity: 4 bytes * Execution length

Basic Block Level Tracing

```
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
6: endwhile
7: print(sum)
```

Trace(N=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6
BB Trace: 1 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 6
### More Ideas

- Would a function level tracing idea work?
  - A trace entry is a function call with its parameters.
- Predicate tracing

```
1: sum=0
2: i=1
3: while ( i<N) do
4:  i=i+1
5:  sum=sum+i
6:  print(sum)
```

* Lose random accessibility

### Program Slicing

**Instruction trace**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Predicate trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 6</td>
<td>F</td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td>T F</td>
</tr>
</tbody>
</table>

### Outline

- What is slicing
- Why slicing
- Static slicing
- Dynamic slicing
  - Data dependence detection
  - Control dependence detection
  - Slicing algorithms (forward vs. backward)
  - Chopping

### What is a slice?

- Analysis technique introduced by Mark Weiser in his PHD thesis (1979)
  - Idea derived when he was observing experienced programmers debugging a program
  - Result: Every experienced programmer uses slicing to debug a program
- Slicing reduces programs to statements relevant for partial computation
  - Irrelevant statements are deleted
- A slice $S$ includes all program statements affecting variables $V$ at position $n$ in Program $P$
  - E.g., the slicing criteria $S(a, 10)$ is a slice including all statements affecting the value of $a$ in line 10

### Why Slicing?

- Limit analysis scope: protocol reverse engineering
- Code Reuse: Extracting modules for reuse.
- Partial Execution replay: Replay only part of the execution that is relevant to a failure.
- Partial roll back: partially roll back a transaction.
- Information flow: prevent confidential information from being sent out to untrusted environment.
- Others.

```c
void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    printf("sum=%d\n", sum);
    printf("I=%d\n", I);
}
```
Static vs. Dynamic Slicing

**Static slicing**
- Slices derived from the source code for all possible input values
- No assumptions about input values.
- May lead to relatively big slices
- Contains all statements that may affect a variable for every possible execution

**Dynamic slicing**
- Uses information derived from a particular execution of a program
- Execution is monitored and slices are computed with respect to program history
- Relatively small slices
- Contains all statements that actually affect the value of a variable

How to Compute Slices?

**Dependence Graph**
- Data dep.
- Control dep.

X is data dependent on Y if (1) there is a variable v that is defined at Y and used at X and (2) there exists a path of nonzero length from Y to X along which v is not re-defined.

Y is control-dependent on X iff X directly determines whether Y executes
- X is not strictly post-dominated by Y
- There exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

Static Slices are Imprecise

- Don’t have dynamic control flow information
- Use of Pointers – static alias analysis is very imprecise
- Use of function pointers
Dynamic Slicing

- Korel and Laski, 1988
  - The set of executed statement instances that did contribute to the value of the criterion.
- Dynamic slicing makes use of all information about a particular execution of a program.
- Dynamic slices are often computed by constructing a dynamic program dependence graph (DPDG).
  - Each node is an executed statement (instruction).
  - An edge is present between two nodes if there exists a data/control dependence.
  - A dynamic slice criterion is a triple <\Var, \text{Execution Point}, Input>.
  - The set of statements (reachable in the DPDG from a criterion) constitute the slice.
- Dynamic slices are smaller, more precise, more helpful to the user.

An Example

1. \texttt{I=0}
2. \texttt{sum=0}
3. \texttt{I < N}
4. \texttt{sum=sum+I}
5. \texttt{I=I+1}
6. \texttt{print (sum)}
7. \texttt{print(I)}

Slice(I@7)={1,3,5,7}
DSlice(I@71,,N=0)={1,7}

Another Example

1. \texttt{I=0}
2. \texttt{sum=0}
3. \texttt{I < N}
4. \texttt{sum=sum+I}
5. \texttt{I=I+1}
6. \texttt{print (sum)}
7. \texttt{print(I)}

Slice(I@7)={1,3,5,7}
DSlice(I@71,,N=1)={1,3,5,7}

How to compute dynamic slices?

- Execution traces
  - control flow trace -- dynamic control dependences
  - memory reference trace -- dynamic data dependences
- Construct a dynamic dependence graph
- Traverse dynamic dependence graph to compute slices

Offline Algorithms – Data Dep

- Instrument the program to generate the control flow and memory access trace

  ```
  Void main ( ) {
    1 int I=0;
    2 int sum=0;
    3 while (I<N) {
      4 sum=\text{add}(sum,I);
      5 I=\text{add}(I,1);
      6 }
    7 printf ("sum=%d\n",sum);
    8 printf("I=%d\n",I);
  }
  ```

- Instrument the program to generate the control flow and memory access trace

  ```
  Void main ( ) {
    1 int I=0; trace("1 W  \&I");
    2 int sum=0; trace("2 W \&sum");
    3 while (trace("3 R \&I \&+\&N,I<N") {
      4 \text{sum=\text{add}(sum,I)};
      5 I=\text{add}(I,1);
      6 }
    7 printf ("sum=%d\n",sum);
    8 printf("I=%d\n",I);
  }
  ```

Trace (N=0)
1. \texttt{I=0}
2. \texttt{sum=0}
3. \texttt{I=N}
6. \texttt{print(sum)}
7. \texttt{print(I)}

DSlice(I@1,,N=0)={1,7}
Offline Algorithms – Data Dep

- Instrument the program to generate the control flow and memory access trace
  
  For a "R, addr", traverse backward to find the closest "W, addr", introduce a data dependency (DD) edge, traverse further to find the corresponding writes of the reads on the identified write.

  *"8, R &I" -> "5, W &I" -> "5, R &I" -> "1, W &I"

Offline Algorithms – Control Dep

- Assume there are no recursive functions and CD(i) is the set of static control dependence of i, traverse backward, find the closest x such that x is in CD(i), introduce a dynamic CD from i to x.

  Problematic in the presence of recursion.

Efficiently Computing Dynamic Dependences

- The previous mentioned graph construction algorithm implies offline traversals of long memory reference and control flow traces.

Efficient online algorithms
  - Online data dependence detection.
  - Online control dependence detection.

Efficient Data Dependence Detection

- Basic idea
  
  i: x=…   =>   hashmap[x]= i
  j: … =x…    =>    dependence detected j à hashmap[x], which is j à i

  Trace (N=0)
  1 W &I
  2 W &sum
  3 R &I &N
  4 R &I &sum W &sum
  5 R &I W &I
  6 R &I &N
  7 R &sum
  8 R &I

  HashMap
  I: 11
  I: 11   sum: 21
  31 à hashmap[I]=11
  I: 11   sum: 41
  41 à hashmap[sum]=21
  I: 51   sum: 41
  51 à hashmap[I]=11
  32 à hashmap[I]=51
  61 à hashmap[sum]=41
  71 à hashmap[I]=51

Efficient Dynamic Control Dependence (DCD) Detection

- Def: y DCD on x, iiff there exists a path from x to Exit that does not pass y, and no such paths for nodes in the executed path from x to y.
- Region: executed statements between a predicate instance and its immediate post-dominator form a region.

Postdominator

- X post-dominates Y if every possible program path from Y to End has to pass X.
  - Strict post-dominator, immediate post-dominance.
Region Examples

1. for(i=0; i<N, i++) {
2.     if(i%2 == 0)
3.         p = &a[i];
4.     foo(p);
5. }
6. a = a+1;

• A statement instance x, DCD on the predicate instance leading x,’s enclosing region.
• Regions are either nested or disjoint. Never overlap.

Efficient DCD Detection

• Observation: regions have the LIFO characteristic.
  — Otherwise, some regions must overlap.
• Implication: the sequence of nested active regions for the current execution point can be maintained by a stack, called control dependence stack (CDS).
  — A region is nested in the region right below it in the stack.
  — The enclosing region for the current execution point is always the top entry in the stack, therefore the execution point is control dependent on the predicate that leads the top region.
  — An entry is pushed onto CDS if a branching point (predicates, switch statements, etc.) executes.
  — The current entry is popped if the immediate post-dominator of the branching point executes, denoting the end of the current region.

Algorithm

Predicate (x_i)
{
    CDS.push(<x_i, IPD(x) >);
}

Merge (t_i)
{
    while (CDS.top().second==t) CDS.pop();

GetCurrentCD()
{
    return CDS.top().first;
}

An Example

1. if   ( p1  ||  p2 )  {
2.     s1;
3.     s2;
4. }
5. if  (p3)  {
6.     while   (p4)  {
7.         s3;
8.     }
9. } else {
10.     if   (p5) {
11.         return;
12.     }
13. }
14. s4;

Wrap Up

• We have introduced the concept of slicing and dynamic slicing
• Offline dynamic slicing algorithms based on backwards traversal over traces is not efficient
• Online algorithms that detect data and control dependences are discussed.

Forward Dynamic Slice Computation

• The approaches we have discussed so far are backwards.
  — Dependence graphs are traversed backwards from a slicing criterion.
  — The space complexity is O (execution length).
• Forward computation
  — A slice is represented as a set of statements that are involved in computing the value of the slicing criterion.
  — A slice is always maintained for a variable.
The Algorithm

- An assignment statement execution is formulated as
  \[ s_i: x = p_j \text{ op } (src_1, src_2, ...); \]
  - That is to say, the statement execution instance \( s_i \) is control dependent on \( p_j \) and operates on variables of \( src_1, src_2, \) etc.
- Upon the execution of \( s_i \), the slice of \( x \) is updated to
  \[ \text{Slice}(x) = \{s_i\} \cup \text{Slice}(src_1) \cup \text{Slice}(src_2) \cup ... \cup \text{Slice}(p_j) \]
  - The slice of variable \( x \) is the union of the current statement, the slices of all variables that are used and the slice of the predicate instance that \( s_i \) is control dependent on. Because they are all contributing to the value of \( x \).
- Such slices are equivalent to slices computed by backwards algorithms.
  - Proof is omitted.
- Slices are stored in a hashmap with variables being the keys.
- The computation of \( \text{Slice}(p_j) \) is in the next slide. Note that \( p_j \) is not a variable.

The Algorithm (continued)

- A predicate is formulated as
  \[ s_i: p_j \text{ op } (src_1, src_2, ...); \]
  - That is to say, the predicate itself is control dependent on another predicate instance \( p_j \) and the branch outcome is computed from variables of \( src_1, src_2, \) etc.
- Upon the execution of \( s_i \)
  - A tuple is pushed to CDS with the format of
    \[ <s_i, \text{IPD}(s_i), \text{Slice}(src_1) \cup \text{Slice}(src_2) \cup ... \cup \text{Slice}(p_j)> \]
  - The entry is popped at its immediate post dominator
  - \( \text{Slice}(p_j) \) can be retrieved from the top element of CDS.

Example

<table>
<thead>
<tr>
<th>Statements Executed</th>
<th>Dynamic Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( a=1 )</td>
<td>( \text{Slice}(a) = {1} )</td>
</tr>
<tr>
<td>2: ( b=2 )</td>
<td>( \text{Slice}(b) = {2} )</td>
</tr>
<tr>
<td>3: ( c=a+b )</td>
<td>( \text{Slice}(c) = {1,2,3} )</td>
</tr>
<tr>
<td>4: if ( a&lt;b ) then</td>
<td>( \text{push}(&lt;4,6, {1,2,4}) )</td>
</tr>
<tr>
<td>5: ( d=b*c )</td>
<td>( \text{Slice}(d) = {1,2,3,4,5} )</td>
</tr>
<tr>
<td>6: ( \ldots )</td>
<td># # #</td>
</tr>
</tbody>
</table>

Properties

- The slices are equivalent to those computed by backwards algorithms
  - The proof is omitted.
- The space complexity is bounded
  - \( O (\# \text{ of variables} + \text{MAX_CDS_DEPTH} \times \# \text{ of statements}) \)
- Efficiency relies on the hash map implementation and set operations.

Extending Slicing

- Essentially, slicing is an orthogonal approach to isolate part of a program (execution) giving certain criterion.
- Mutations of slicing
  - Event slicing – intrusion detection, execution fast forwarding, understanding network protocol, malware replayer.
  - Forward slicing.
  - Chopping.
  - Probabilistic slicing.

Limitations of Dynamic Slicing

- Execution omission
  \[ x=\text{input}(); \]
  \[ y=0; \]
  if \( x>10 \)
  \[ y=\text{y}+1; \]
  output (y);
- Horrible approximation of causality
  if \( x=10 \) \( y=1; \)
  if \( x=10 \) \( y=1; \)
  for \( i=0 \) to \( x\) \( \text{sum}++; i; \)
  if \( i>10 \) \( \text{sum}++; i; \)