Lecture 10. Dynamic Analysis

Wei Le
Thank Xiangyu Zhang, Michael Ernst, Tom Ball
for Some of the Slides

Iowa State University

2014.11
Outline

▶ What is dynamic analysis?
▶ Instrumentation
▶ Analysis
  ▶ Representing dynamic information
  ▶ Dynamic slicing
▶ Applications: specification generation and Diakon
What is dynamic analysis

Learn from executions:

▶ Run programs
  ▶ where are the test inputs?
  ▶ what are the coverage?
  ▶ what parts of the code should be run?
  ▶ how to create runtime environment?

▶ Collect data
  ▶ online or offline (log analysis)?
  ▶ what is the overhead (memory and performance)
  ▶ how to capture and replay the data: compress, statistical approach

▶ Analyze data
  ▶ dynamic slicing
  ▶ model-based
What is the meaning of *Run*?

- Abstract interpretation and static analyses run a program over an abstract domain:
  \[ \text{OUT} = F(\text{IN}, s) \]

- Dynamic analysis: abstraction used in parallel with, not in place of, concrete values:
  \[ \text{OUT} = F(\text{IN}, s_i, v) \]
How dynamic analysis is used?

- Profiling-guided code optimization: which paths are frequently executed?
- Performance analysis
- Mining program invariants, protocols
- Fault localization, debugging (replay concurrency bugs)
- Bug finding
- ...
Running the programs

- The test suite determines the expense (in time and space)
- The test suite determines the accuracy (what executions are never seen)
- Coverage or frequency: statements, branches, paths, procedure calls, types, method dispatch
- Values computed: parameters, array indices
- Run time, memory usage
Instrumentation

A technique that inserts extra code into a program to collect runtime information.

```c
Max = 0;

for (p = head; p; p = p->next)
{
    if (p->value > max)
    {
        max = p->value;
    }
}
```
Instrumentation

A technique that inserts extra code into a program to collect runtime information.

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

A technique that inserts extra code into a program to collect runtime information.

```c
Max = 0;
for (p = head; p; p = p->next)
{
    count[0]++;
    if (p->value > max)
    {
        count[1]++;
        max = p->value;
    }
}
```
Source and Binary Instrumentation

- Source instrumentation: instrument source programs
- Binary instrumentation: instrument executables directly:
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes
Binary Instrumentation Tool Is Dominant

- Libraries are a big pain for source code level instrumentation e.g., proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries).
- Easily handle multi-lingual programs: source code level instrumentation is heavily language dependent.
- More complicated semantics: Turning off compiler optimizations can maintain a almost perfect mapping from instructions to source code lines.
- Worms and viruses are rarely provided with source code
Static Instrumentation

- Perform the instrumentation before the code is run
  - New binary = original binary + instrumentation
  - Raise binary to IR, transform IR, transfer back to binary

- All libraries are usually statically linked
  - The size of the binary is big

- Program representations are usually built from the binary
  - CFG
  - Call graph
  - PDG is hard to build from binary
    - Points-to analysis on binary is almost impossible
    - Simple DFA is possible
    - CDG is almost precise (PDG=CDG+DDG)
Dynamic Instrumentation

- A trampoline is required.
- Does not require recompiling or relinking
  - Saves time: compile and link times are significant in real systems.
  - Can instrument without linking (relinking is not always possible).
- Dynamically turn on/off, change instrumentation
  - From \( t_1 - t_2 \), I want to execute \( F' \), \( t_3 - t_4 \), I want \( F'' \)
    - Can be done by invalidating the mapping in the dispatcher.
- Can instrument running programs (such as Web or database servers)
  - Production systems.
- Can instrument self-mutating code.
  - Obfuscation can be easily get around.
Instrumentation Tools

- Source instrumentation tools: AIMS, Paradyn, Pablo ... [1]
- Binary instrumentation tools:
  - Static (insert in the code and then run): ATOM, EEL, Etch, Morph
  - Dynamic (run while insert the instrumentation): Dyninst, Vulcan, DTrace, Valgrind, Strata, DynamoRIO, PIN
Valgrind

See Valgrind slides
See PIN slides
Instrumentation: a Summary

- Source-level:
  - Pattern-matching over parse tree or AST and rewriting
  - Full access to source information
  - A* [Ladd, Ramming], Astlog [Crew],

- Binary-level:
  - ATOM [Srivastava], EEL [Larus], Diablo, Bluto
  - Analyze programs from multiple languages
  - Typically done at some IR level

- Dynamic instrumentation (run-time): Valgrind, PIN

- Java instrumentation tools:
  - ASM (http://asm.objectweb.org/), Small and fast, Supports Java 1.5
  - BCEL (http://jakarta.apache.org/bcel/), Works well, No longer under active development
  - SERP (http://serp.sourceforge.net/), Not designed for speed, Somewhat memory intensive
  - Soot (http://www.sable.mcgill.ca/soot/), Supports source, Designed for optimizations
Design Your Instrumentation

- How much to generate?
  - Everything
  - Just the necessary facts
  - Less than necessary

- On-line vs. off-line analysis
  - What/When to instrument?
  - Source code, IR, assembly, machine code
  - Preprocessor, compile-time, link-time, executable, run-time
Control Flow Trace

N=2:

1_1: sum=0
   i=1

3_1: while ( i<N ) do

4_1: i=i+1
   sum=sum+i

3_2: while ( i<N ) do

4_2: i=i+1
   sum=sum+i

3_3: while ( i<N ) do

6_1: print ( sum )

A More Compact CFT: < T, T, F >
Dynamic Dependence Graph

```
1:  z=0
2:  a=0
3:  b=2
4:  p=&b
5:  for i = 1 to N do
6:      if ( i %2 == 0) then
7:          p=&a
8:      endif
9:  endfor
10:  a=a+1
11:  z=2*(*p)
12:  print(z)
```

Input: N=2

```
11:  z=0
12:  a=0
13:  b=2
14:  p=&b
15:  for i=1 to N do
16:      if (i%2==0) then
17:          a=a+1
18:      endif
19:      z=2^{(*p)}
20:  endif
21:  print(z)
```

One use has only one definition at runtime; One statement instance control depends on only one predicate instance.
Input: N=2
1  $1_1$: $z=0$
2  $2_1$: $a=0$
3  $3_1$: $b=2$
4  $4_1$: $p=&b$
5  $5_1$: for $i = 1$ to N do
6  $6_1$: if $(i \% 2 == 0)$ then
7  $8_1$: $a=a+1$
8  $9_1$: $z=2^{(*)p}$
9  $5_2$: for $i = 1$ to N do
10 $6_2$: if $(i \% 2 == 0)$ then
11 $7_1$: $p=&a$
12 $8_2$: $a=a+1$
13 $9_2$: $z=2^{(*)p}$
14 $10_1$: print($z$)
Dynamic Slicing

See Xiangyu’s slides
Specification Generation-Diakon

- Daikon infers invariants from a program trace
- Looks for invariants between each combination of variables
- Polynomial in the number of variables
- Easy to implement offline, first pass finds equal variables
- More complex to implement incrementally.
Dynamic invariant detection [Ernst 99]

- Machine learning over values the program computes
  - Generate-and-test strategy
  - Statistical tests to combat overfitting
- Daikon implementation is publicly available:
  http://pag.csail.mit.edu/daikon/
- Many applications:
  - verifying safety properties [Vaziri 1998, Nimmer 2002]
  - automating theorem-proving [Ne Win 2003]
  - identifying refactoring opportunities [Kataoka 2001]
  - predicate abstraction [Dodoo 2002]
  - generating test cases [Xie 2003, Gupta 2003, Pacheco 2005]
  - selecting and prioritizing test cases [Harder 2003]
  - explaining test failures [Groce 2003]
  - predicting incompatibilities in component upgrades [McCamant 2003]
  - error isolation [Xie 2002, Liblit 2003]
  - choosing modalities [Lin 2004]
public class StackAr {
    Object[] theArray;
    int topOfStack;
    ...
}

Object properties

theArray != null
theArray.getClass() == java.lang.Object[].class
topOfStack >= -1
topOfStack <= theArray.length - 1
theArray[0..topOfStack] elements != null
theArray[topOfStack+1..] elements == null

Pre-conditions for the StackAr constructor

capacity >= 0
Post-conditions for the StackAr constructor

orig(capacity) == theArray.length
theArray[] elements == null
topOfStack == -1
theArray[0..topOfStack] == []

Post-conditions for the isFull method

theArray == orig(theArray)
theArray[] == orig(theArray[])
topOfStack == orig(topOfStack)
(return == false) <=> (topOfStack < theArray.length - 1)
(return == true)  <=> (topOfStack == theArray.length - 1)
Data Analysis Techniques

Data mining (mining database) fields, related to information retrieval, text mining

- Associate rules mining
- Sequential patterns mining
- Beyond:
  Structure patterns (tree, graphs) mining

- Definition: what is it?
- Which algorithm? there exist off-the-shelf packages
Associate Rules Mining

- Initially used for Market Basket Analysis to find how items purchased by customers are related: Bread → Milk [sup = 5%, conf = 100%]
- Applications: Object, properties
  - patient / symptoms
  - movies / ratings
  - web pages / keywords
  - basket / products
  - what are examples in computing?
The Model: Data

- \( I = \{i_1, i_2, \ldots, i_m\} \): a set of items.
- Transaction \( t \):
  - \( t \) a set of items, and \( t \subseteq I \).
- Transaction Database \( T \): a set of transactions
  \( T = \{t_1, t_2, \ldots, t_n\} \).
The Model: Rules

- A transaction \( t \) contains \( X \), a set of items (itemset) in \( I \), if \( X \subseteq t \).
- An association rule is an implication of the form:
  \[
  X \rightarrow Y, \text{ where } X, Y \subseteq I, \text{ and } X \cap Y = \emptyset
  \]
- An itemset is a set of items.
  - E.g., \( X = \{ \text{milk, bread, cereal} \} \) is an itemset.
- A \( k \)-itemset is an itemset with \( k \) items.
  - E.g., \( \{ \text{milk, bread, cereal} \} \) is a 3-itemset
Algorithms and Extensions

- Apriori
- Eclat
- FP-growth

Relevant concepts:
- Frequent itemset mining
- Maximal itemset mining [Bayardo, 1998]
- Closed itemset mining [Pasquier, 1999]
Patterns in Sequences

- Substrings
- Regular expressions
- Partial orders
- Structured patterns, directed acyclic graphs (mining for frequent subgraphs)
- Episodes
- Constraint-based sequential pattern mining (e.g., the item satisfies certain values)
Sequential Pattern Mining [2]

Sequential pattern mining: given a set of sequences, find the complete set of frequent subsequences

A sequence:

\[
\langle \text{(ef)} \langle \text{ab} \rangle \text{(df)} \text{c b} \rangle
\]

A sequence database

<table>
<thead>
<tr>
<th>SID</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>\langle a\langle ab \rangle(c)\langle ac \rangle(d)\langle cf \rangle \rangle</td>
</tr>
<tr>
<td>20</td>
<td>\langle (ad)c\langle bc \rangle\langle ae \rangle \rangle</td>
</tr>
<tr>
<td>30</td>
<td>\langle (ef)\langle ab \rangle\langle df \rangle\langle cb \rangle \rangle</td>
</tr>
<tr>
<td>40</td>
<td>\langle eg\langle af \rangle\langle cbc \rangle \rangle</td>
</tr>
</tbody>
</table>

An element may contain a set of items. Items within an element are unordered and we list them alphabetically.

\langle a\langle bc \rangle\langle dc \rangle \rangle is a subsequence of \langle a\langle ab \rangle\langle ac \rangle\langle d \rangle\langle cf \rangle \rangle

Given support threshold min_sup = 2, \langle ab \rangle c is a sequential pattern
Sequential Mining Algorithms

- Concept introduction and an initial Apriori-like algorithm
  - Agrawal & Srikant. Mining sequential patterns, ICDE’95
- Apriori-based method: **GSP** (Generalized Sequential Patterns: Srikant & Agrawal @ EDBT’96)
- Pattern-growth methods: FreeSpan & **PrefixSpan** (Han et al.@KDD’00; Pei, et al.@ICDE’01)
- Vertical format-based mining: **SPADE** (Zaki@Machine Learning’00)
- Constraint-based sequential pattern mining (SPIRIT: Garofalakis, Rastogi, Shim@VLDB’99; Pei, Han, Wang @ CIKM’02)
- Mining closed sequential patterns: **CloSpan** (Yan, Han & Afshar @SDM’03)
Software

C++ Implementations of Apriori, Eclat, FP-growth and several other algorithms are available on http://www.adrem.ua.ac.be/goethals/software/ and on http://fimi.cs.helsinki.fi/
Designing Your Dynamic Analysis

Dynamic analysis is, at its heart, an experimental effort:

▶ Have insight
▶ Build tool
▶ Evaluate efficiency and effectiveness
▶ Rethink