CS510 Software Engineering

Dynamic Program Analysis

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Scott A. Carr
Slides inspired by Xiangyu Zhang

http://nebelwelt.net/teaching/15-CS510-SE

Spring 2015

Table of Contents

Overview

Dynamic program analysis tackles software dependability and productivity problems by inspecting *software execution*.

- A program execution captures runtime behavior of a program (think class and object).
- Dynamic analysis follows path through the program: each statement is executed $\{0, N\}$ times.
- The analysis is restricted to a single path.
- All variables are instantiated (solving the aliasing problem of static analysis).

Advantages

- Relatively low learning curve.
- Precision.
- Applicability.
- Scalability.

Disadvantages?

- Neither generalizable nor complete.
- Limited to available test-cases.
- Possible runtime constraints (Heisenbugs)

Table of Contents

Dynamic Program Analysis Primitives

- Tracing
- Profiling
- Checkpoint and replay
- Dynamic slicing
- Execution indexing
- Delta debugging

Applications

- Taint tracking
- Dynamic information flow tracking
- Automated debugging

Table of Contents

Tracing definition

Tracing

Tracing is a lossless process that faithfully records detailed information of a program's execution.

Tracing is a basic and simple primitive.

Types of Tracing

- Control-flow tracing (sequence of executed statements);
- Dependence tracing (sequence of exercised dependences);
- Value tracing (sequence of values produced by each instruction);
- Memory access tracing (sequence of memory accesses during execution).

Table of Contents

Use-cases for Tracing

- Debugging: time-travel to understand interactions;
- Code optimizations: hot program paths, data compression, value speculation, data locality for cache optimization;
- Security: malware analysis;
- Testing: code coverage.

Table of Contents

Tracing by printf

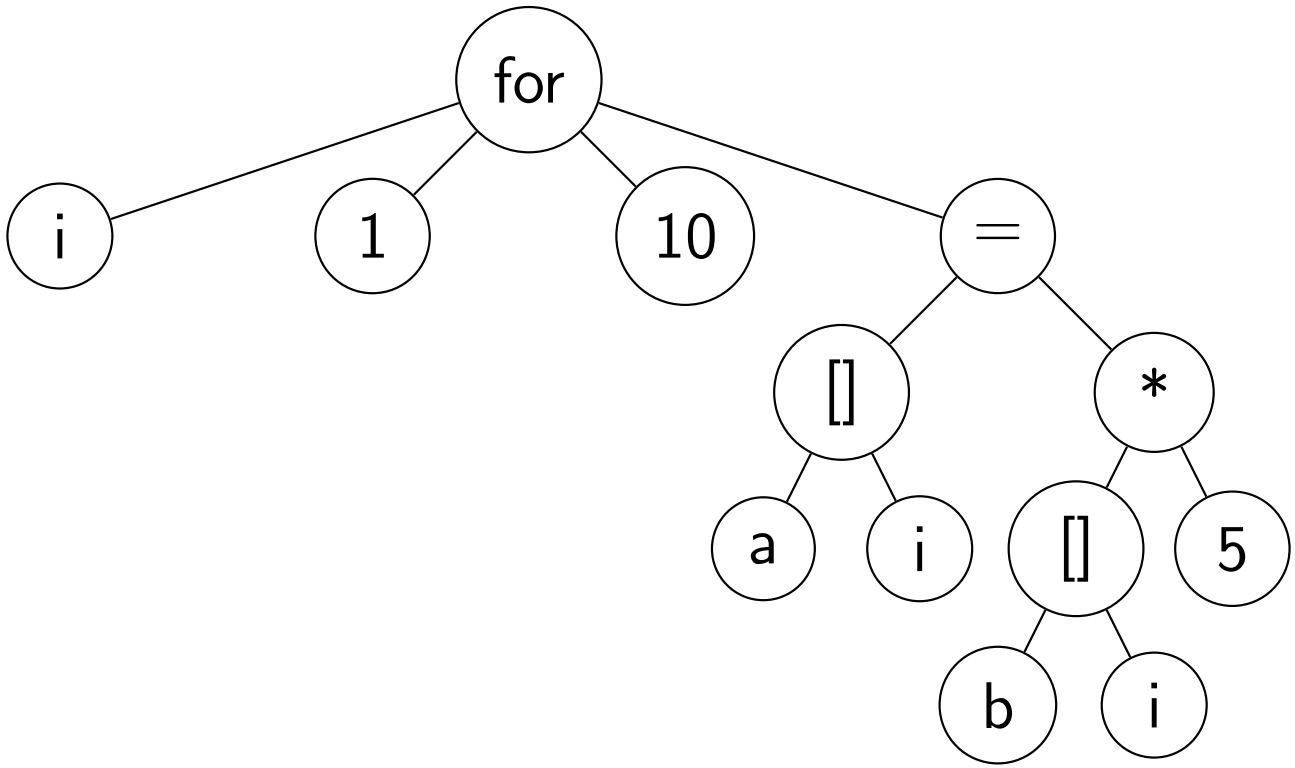
```
1 int max = 0;
2 for (p = head; p; p = p->next) {
3   printf("in loop\n");
4   if (p->value > max) {
5     printf("True branch\n");
6     max = p->value;
7   }
8 }
```

Tracing by Source-Level Instrumentation

- Parse a source file into an AST.
- Annotate the AST with instrumentation.
- Translate the annotated trees into a new source file.
- Compile the new sources.
- Execute the program and produce a trace as side-effect.

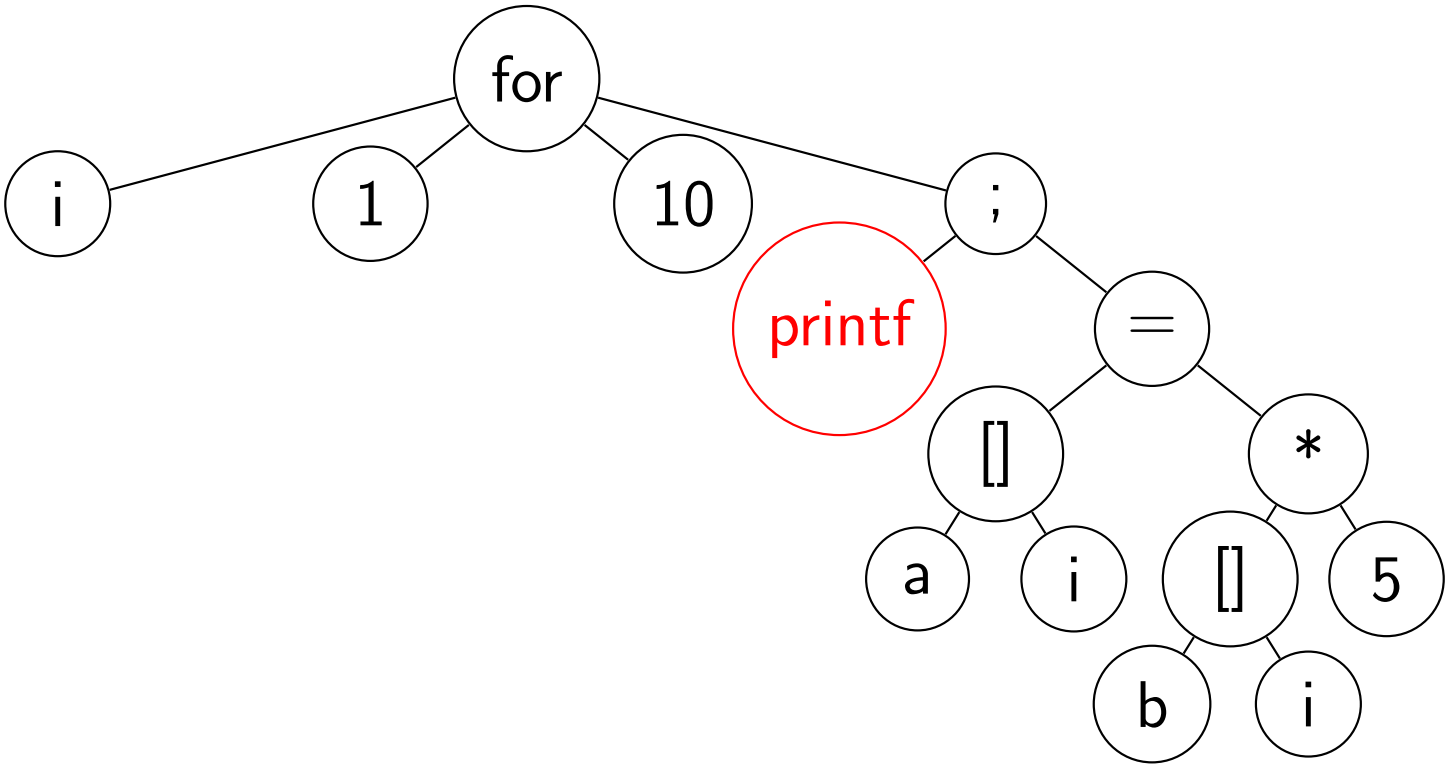
Source-Level Instrumentation Example

```
1 for (i = 1; i < 10; i++) {
2   a[i] = b[i] * 5;
3 }
```



Source-Level Instrumentation Example (2)

```
1 for (i = 1; i < 10; i++) {
2   printf("In loop\n");
3   a[i] = b[i] * 5;
4 }
```



Characteristics of Source-Level Instrumentation

- Detailed type and variable information available.
- Detailed control-flow structures available.
- No support for pre-compiled libraries or binaries.
- Limited support for multi-lingual programs.
- Requires full source-code.

Tracing by Binary Instrumentation

- Parse binary into intermediate representation, generate graph data structures like CFG.
- Instrument IR with tracing nodes.
- Compile/assemble back to an executable for static binary instrumentation or use a JIT to execute on-the-fly.

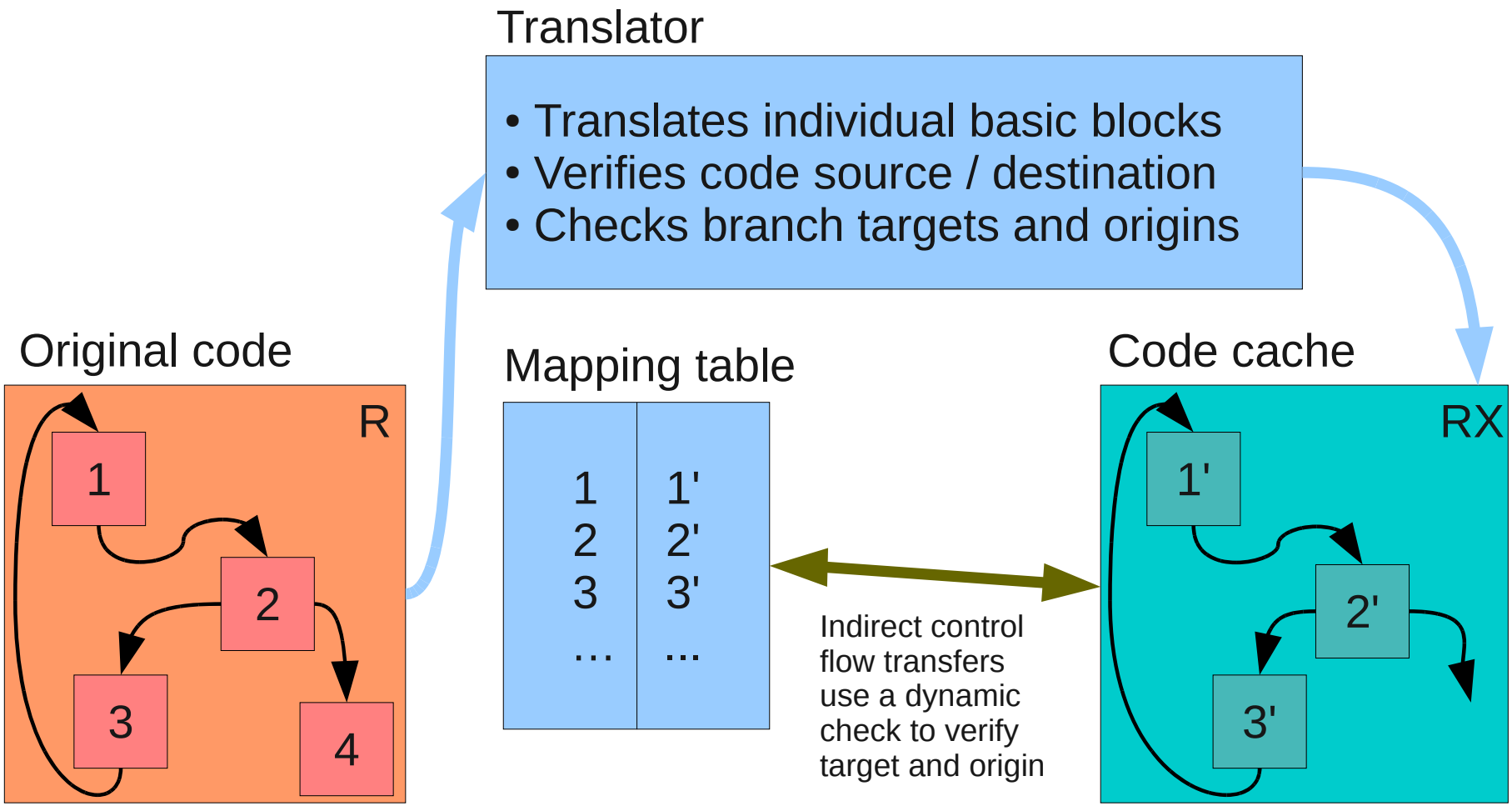
Characteristics of Binary-Level Instrumentation

- No source-code needed.
- Supports libraries and any executable.
- Possibly high overhead due to instrumentation and translation.
- Limited scope and high-level data structures available.

FastBT

- Enable fast, efficient instrumentation at low overhead.
- Instead of converting machine code to an IR, translate using pre-generated tables.
- Define a set of translation actions that add instrumentation when dispatched.
- Use a code-cache to lower overhead.
- Challenge: define translation actions for instructions that change control-flow.

FastBT Overview



Reading material: Generating low-overhead dynamic binary translators, Mathias Payer and Thomas R. Gross, SySTOR'10 (see course homepage).

Table of Contents

Fine-grained Tracing is Expensive!

```
1 int sum = 0;
2 int i = 1;
3 while (i < N) {
4   i++;
5   sum = sum + i;
6 }
7 printf("Sum: %d\n", sum);
```

Trace ($N = 6$): 1, 2, 3, 4, 5, 3, 4, 5, 6, 3, 4, 5, 6, 3, 4, 5, 6, 3, 4, 5, 6, 3, 7.

Space complexity: $exec\_length * sizeof(void*)$

Basic block-level Tracing

```
1 int sum = 0;
2 int i = 1;
3 while (i < N) {
4   i++;
5   sum = sum + i;
6 }
7 printf("Sum: %d\n", sum);
```

BB Trace: 1-2, 3, 4-5, 3, 4-5, 3, 4-5, 3, 4-5, 3, 4-5, 3, 7
In this example only 13/19 storage needed.
Drawback: seeking inside basic block is more complicated.

Other options to reduce trace size?

- Function-level tracing (i.e., recording functions and their parameters)(What about side-effects?)
- Predicate tracing (i.e., record all branch predicates from beginning of execution (needs only one bit per branch)(Seeking is hard)
- Path-based tracing (record path through CFG)(Needs heavy-weight data structures)
- Compression using, e.g., deflate(Relies on decompression, no seeking)

Last n Values Predictor: Compression

- Buffer stores the last n unique encountered values.
- If the next value is one of the n values then the index into the buffer is emitted (prefixed with symbol 0).
- Otherwise (mis-prediction) store the encountered value to the encoded trace (prefixed with symbol m), update the buffer with a least used strategy.

Example:
123 456 456 456 456 123 123 789 456
Use last-2 predictor:
m 123 m 456 00 00 00 01 01 m 789 m 456

Last n Values Predictor: Decompression

- Take one bit from encoded trace.
- If m symbol then read next value and update buffer.
- If 0 symbol read index and print value from table.

n-Value Predictors are related to Run-Length Encoding (RLE).

Finite Context Method (FCM)

- Construct a lookup-table that predicts a value based on the last n values (2-FCM, 3-FCM).
- If the next value is correctly predicted using the left context, a 0-bit is emitted to the encoded trace.
- Otherwise (mis-prediction), an m-symbol and the original value are emitted to the trace. The lookup-table is updated accordingly.

Example (3-FCM):
1 2 3 4 5 3 4 5 ... 3 4 5 6
m 1 m 2 m 3 m 4 m 5 m 3 m 4 m 5 0 . . . 0 0 0 m 6

FCM Characteristics

- Length (compressed): $n/sizeof(void*) + n * (1 - predict\_rate)$.
- Predictors are better than deflate due to repetitive loop patterns.
- Drawback: trace is only forward traversable.

Bidirectional Compression

- Use a small sliding window of clear text on the compressed string (just like with FCM)[1]
- Keep both left-context and right-context lookup table (instead of just left-context lookup table).
- Moving forward: decompress next value using left-context lookup table (sliding window is now n+1), compress the first value using the right-context lookup table (sliding window is now n again).
- Moving backward: decompress using right-context, compress using left-context.

[1] The left and right side of the window stay compressed

Bidirectional Compression: Example

... A X Y 0 ...
Left-context: A X Y: Z
Compress right-context: X Y Z: A[2]

[2] If correct prediction, emit 0 to right-context stream, otherwise update table and emit m symbol

Bidirectional Predictor Characteristics

- Almost same compression rate as unidirectional predictors.
- (Possibly slightly worse due to different prediction rate for forward/backward).
- Fast compression/decompression (two times slower than unidirectional predictors).

Questions?

?