Program Analysis
Dynamic Analysis Frameworks

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2021/2022
Warm-up Quiz

```javascript
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```
Warm-up Quiz

```javascript
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

Result: a
Warm-up Quiz

```javascript
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

**Result:** `a`
Outline

1. Introduction
2. Special-Purpose Dynamic Analysis
3. General-Purpose Frameworks

Relevant papers:

- **Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript**, Sen et al., FSE 2013
Dynamic Analysis

- Execute an instrumented program to gather information that can be analyzed to learn about a property of interest

- **Precise**: All observed behavior actually happens

- **Incomplete**: Very difficult to cover all possible behaviors
Examples

- **Coverage**: Track which lines or branches get executed
- **Call graph**: Track which functions call which other functions
- **Slicing**: Track dependencies to produce a reduced program
- We’ll see more in upcoming lectures
Examples

- **Coverage**: Track which lines or branches get executed
- **Call graph**: Track which functions call which other functions
- **Slicing**: Track dependencies to produce a reduced program
- We’ll see more in upcoming lectures

Different goals, similar challenges: Use a common framework
Outline

1. Introduction
2. Special-Purpose Dynamic Analysis
3. General-Purpose Frameworks

Relevant papers:

- Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, Nethercote et al., PLDI 2007
- Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript, Sen et al., FSE 2013
Coverage Analysis

Goal: Track which **branches** are executed

```java
x = readInput();
if (x > 0) {
    y = 2;
    y = 3
    while (y > 0) {
        y = y - x;
    }
} else {
    y = 3
}
```
Instrumented Program

Add instrumentation code at **beginning** of each basic block

```c
x = readInput();
if (x > 0) {
    y = 2;
y = 3
    while (y > 0) {
        y = y - x;
    }
} else {
    y = 3
}
```

```c
cov = [false, false, false, false];
x = readInput();
if (x > 0) {
    cov[0] = true;
y = 2;
y = 3
    while (y > 0) {
        cov[3] = true;
y = y - x;
    }
    cov[2] = true;
} else {
    cov[1] = true;
y = 3
}
```
entry

\( x = \text{readInput}(1) \)

\( x > 0 \)

\( y = 2 \)

\( y = 3 \)

\( y > 0 \)

\( y = y - x \)

exit

Input 1: 5

Input 2: -5

3/4 branches covered

1/4 branches covered
Given the input 1, what’s the branch coverage?

```plaintext
cov = [false, false, false, false, false];
x = readInput();
if (x > 0) {
    cov[0] = true;
    y = 2;
y = 3
    while (y > 0) {
        cov[3] = true;
        y = y - x;
    }
cov[2] = true;
} else {
cov[1] = true;
y = 3
}
```
Quiz

 cov = [false, false, false, false];
 x = readInput();
 if (x > 0) {
   cov[0] = true;
   y = 2;
   y = 3
   while (y > 0) {
     cov[3] = true;
     y = y - x;
   }
   cov[2] = true;
 } else {
   cov[2] = true;
   } else {
   cov[1] = true;
   y = 3
} }

Given the input 1, what's the branch coverage?

Answer:
[true, false, true, true, true]
Call Graph Analysis

Goal: Track “calls” relationships between functions

```javascript
n = readInput();
function a() {
    b();
}
function b() {
    if (n == 5) {
        c();
    }
}
function c() {
    if (n == 5) {
        c();
n--;
    }
}
a();
```
Static overapproximation of call graph
Add instrumentation code at each call site

```
n = readInput();
function a() {
    b();
}
function b() {
    if (n == 5) {
        c();
    }
}
function c() {
    if (n == 5) {
        c();
        n--;
    }
}
a();
```

```
calls = new Set();
n = readInput();
function a() {
    calls.add("a->b");
    b();
}
function b() {
    if (n == 5) {
        calls.add("b->c");
        c();
    }
}
function c() {
    if (n == 5) {
        calls.add("c->c");
        c();
        n--;
    }
}
calls.add("main->a");
a();
```
Outline

1. Introduction
2. Special-Purpose Dynamic Analysis
3. General-Purpose Frameworks

Relevant papers:

- **Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript**, Sen et al., FSE 2013
Commonalities

Different dynamic analyses, but many commonalities

- Specific runtime events to track
- Analysis updates some state in response to events
Commonalities

Different dynamic analyses, but many commonalities

- Specific runtime events to track
- Analysis updates some state in response to events

Can avoid re-implementing everything from scratch for each new analysis?
Dynamic Analysis Frameworks

- Set of **kinds of runtime events**
- Analysis can **register** for specific events
- At runtime, instrumented program invokes event handlers

Program P $\xrightarrow{\text{instrument}}$ Program P$'$

Analysis A

$\downarrow$ invoke event handlers
## Typical Runtime Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operation</td>
<td>2+3</td>
</tr>
<tr>
<td>Boolean operation</td>
<td>a &gt; 0</td>
</tr>
<tr>
<td>Branch</td>
<td>if (c) ...</td>
</tr>
<tr>
<td>Function call</td>
<td>g()</td>
</tr>
<tr>
<td>Return from function call</td>
<td>x = g()</td>
</tr>
<tr>
<td>Write into variable or field</td>
<td>x.f = z</td>
</tr>
<tr>
<td>Read of variable or field</td>
<td>x.f = z</td>
</tr>
</tbody>
</table>

(and many others)
Example

Runtime events:
- Arithmetic operations
- Boolean operations
- Reads of variables
- Writes into variables
- Function calls

Input: -26
What sequence of events get triggered?

```java
a = readInput();
b = a + 3;
if (b == -23) {
    foo();
} else {
    b = 5;
}
```
Runtime Events: Example

- call of `readInput`
- write -26 into a
- read of a (-26)
- arithmetic operation (-26 + 3 = -23)
- write of -23 into b
- read of b (-23)
- boolean operation (-23 == -23 -> true)
- call of `foo1()`
Extended Operational Semantics

- Tracking runtime events: Additional behavior performed during program execution
- Formally describe by extending the operational semantics
Extending Small-Step Operational Semantics

Events:  
- write to variable → "write 3 to x"
- branch → "true branch taken"

Extending configuration into:

\(<P, s, e>\) where \(P, s\) as before
\(e\) - sequence of events (represented as strings)
Replace all axioms & rules to use triple configuration, e.g.,

\[
\langle \! \! \mathrm{e}, s \rangle \rightarrow \langle n, s \rangle \quad \text{if } s(e) = n
\]

\(\text{(var)}\)

becomes

\[
\langle \! \! \mathrm{e}, s, e \rangle \rightarrow \langle n, s, e \rangle \quad \text{if } s(e) = n
\]

\(\text{(var)}\)

events remain the same
Revise some axioms & rules to create new events

1) \textit{writes to variables}:

\[
\langle \text{\texttt{\textbackslash l:=\texttt{n}}}, \text{s} \rangle \rightarrow \langle \text{\texttt{skip}}, \text{s}[\text{l\rightarrow \texttt{n}}] \rangle
\]

becomes

\[
\langle \text{\texttt{\textbackslash l:=\texttt{n}}}, \text{s}, \text{e} \rangle \rightarrow \langle \text{\texttt{skip}}, \text{s}[\text{l\rightarrow \texttt{n}}], \text{e}, \text{\texttt{"write \texttt{n} to \texttt{l}"}} \rangle
\]
Quit: Extend axioms & rules for tracking branches

\[
\text{\texttt{(if \ True \ then \ C_1 \ else \ C_2, s, e)}}
\]

\[
\rightarrow \text{\texttt{(C_1, s, e \cdot \text{"true branch taken"})}}
\]

(analogous for False)
Implementing Dynamic Analyses

How to implement a dynamic analysis framework in practice?
Option 1:
Source-to-Source Instrument

Source code

↓

compile

Option 2:
Bytecode/binary

Option 3:
Track events here
Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:

// Before: x=y; foo(); a=b;
// After: x=y; foo(); evt('call'); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt('call')")
Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example: Identify function calls

// Before: x=y; foo(); a=b;
// After: x=y; foo(); evt('call'); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt('call')")
Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:
Identify function calls

```
// Before: x=y; foo(); a=b;
// After: x=y; foo(); evt('call'); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt('call')")
```

Add call that logs the 'call' event
Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:

// Before: x=y; foo(); a=b;
// After: x=y; foo(); evt('call'); a=b;
regex = /; \(\w+\(\\));/g;
code.replaceAll(regex, "; $1; evt('call')")

Identify function calls

Add call that logs the 'call' event

Cumbersome and extremely brittle:
Don’t do this
AST-based Instrumentation

More reliable approach:

- **Parse** code into AST
- **Manipulate** AST, e.g., by adding subtrees
- **Pretty-print** AST into code again
if ($x == 0)$
{
  $y = 3$;
  $foo()$;
}

logCall("foo")
# Real-World Tools

<table>
<thead>
<tr>
<th>Name</th>
<th>Target language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>x86 binaries</td>
</tr>
<tr>
<td>Valgrind</td>
<td>x86 binaries</td>
</tr>
<tr>
<td>DiSL</td>
<td>Java</td>
</tr>
<tr>
<td>RoadRunner</td>
<td>Java</td>
</tr>
<tr>
<td>Jalangi</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Wasabi</td>
<td>WebAssembly</td>
</tr>
</tbody>
</table>
# Real-World Tools

<table>
<thead>
<tr>
<th>Name</th>
<th>Target language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>x86 binaries</td>
</tr>
<tr>
<td>Valgrind</td>
<td>x86 binaries</td>
</tr>
<tr>
<td>DiSL</td>
<td>Java</td>
</tr>
<tr>
<td>RoadRunner</td>
<td>Java</td>
</tr>
<tr>
<td>Jalangi</td>
<td>JavaScript</td>
</tr>
<tr>
<td>Wasabi</td>
<td>WebAssembly</td>
</tr>
</tbody>
</table>

To be used in course project

Developed by my group (main author: Daniel Lehmann)