Scalable Context-Sensitive Points-To Analysis

Nicholas Allen
Bernhard Scholz
Paddy Krishnan

Oracle Labs
Brisbane, Australia

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Disclaimer

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Motivation

- Principal aim: Security analysis of JDK
- Security analysis: Flow of objects through the program
  - Objects created by untrusted code may not flow to a security sensitive operation
  - Sensitive objects may not escape to untrusted code

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Solution

- Value Flow Analysis
- Points-to analysis: Objects a variable may reference
  - Result of value flow analysis
- Points-to analysis: Security analysis
  - Taint: Variable in trusted code points to object created by untrusted code
  - Escape: Object created by trusted code pointed to by variable in untrusted code

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Points-To: Choices

- Context-insensitive
  - Not sufficiently precise: Numerous false positives
- Context-sensitive
  - Numerous choices: callsite, receiver/allocator object
  - 2-Object+1-Heap: *Does not scale for JDK*

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Problem Size: Open JDK7-b147

- 1.3 Million variables
- 200,000 methods
- 600,000 potential invocations
- 400,000 object creation sites

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Points-To: Open JDK7-b147

Intel Xeon E5-2660 (2.2GHz) 256GB RAM, Using DOOP and the LogicBlox Engine

Analysis	Time	Size of Result/Outcome
Context-Insensitive	20 minutes	≈ 1 Gigatuples
1-Callsite-Sensitive	20 hours	Does not terminate
1-Object-Sensitive	20 hours	Does not terminate
2-Heap+1-Object	20 hours	Runs out of memory

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Constraining Problem Space

Motivation: Security

- Computing points-to information **not** the ultimate goal
  - Client analysis needs points-to information to answer a query
  - Potential queries: Call-graph, Escape, Null, Taint
- Solution: Demand-driven analysis
  - Only compute information required to answer the query

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

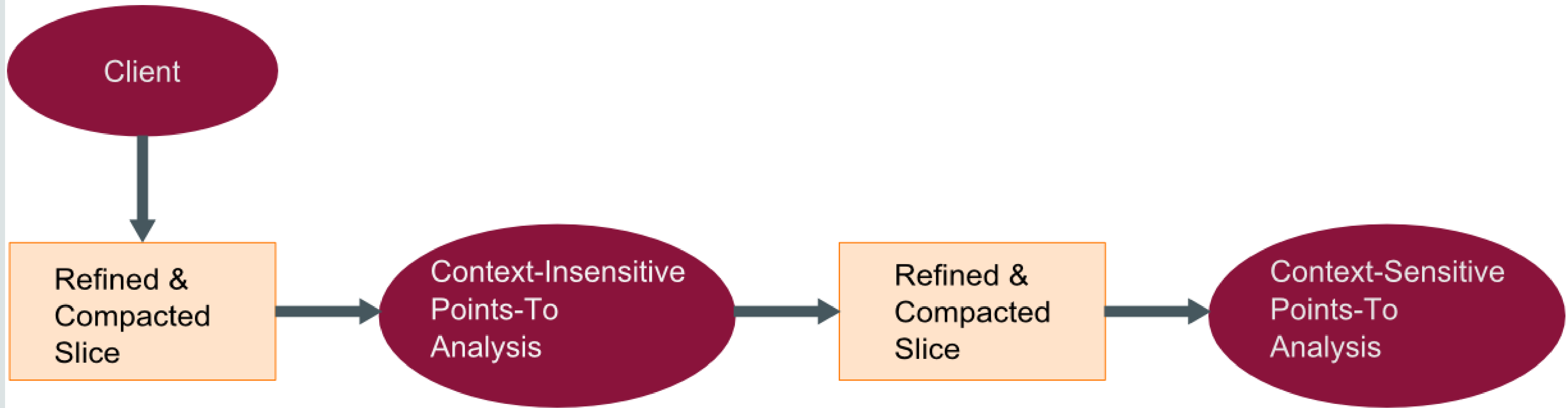
Approach

- Static program slicing and compaction
  - Client's queries as starting point
- Compute points-to in stages
  - Refinement approach
- Reduce program to semantically equivalent for points-to queries

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Architecture



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Slicing/Refinement

Three Steps

- Slicing using variables of interest
  - Very cheap and imprecise (no points-to), removes variables
  - Context-insensitive points-to computed on this slice
- Slicing using context-insensitive points-to information
  - More expensive but also more effective
  - Removes variables and object creation sites
- Context-sensitive points-to analysis
  - Computes final result

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Step 1

- Client identifies required variables
- Over-approximated call graph based on class hierarchy analysis (CHA)
- Sound/Imprecise backwards value-flow trace from the required variables
- Variables not involved in the value-flow trace removed

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Step 2

- Context-insensitive points-to is computed
- Backwards trace from required points-to locations
  - Similar to step 1
  - Points-to provides more precise call graph and value-flow information
  - Variables and heap objects not involved in the trace removed

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Example

```
class SecurityApplication {

    private static void doSecurity(SecurityObject secObj1,
                                   SecurityObject secObj2) {

      SecurityAction action1 = new SecurityAction();
      SecurityAction action2 = new SecurityAction();
      action1.object = secObj1;
      action2.object = secObj2;

      Object res1 = action1.invoke();
      Object res2 = action2.invoke();

      doOtherThings(res1, res2);
    }
```

Any call to `invoke` with an untrusted `object` field?

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Example: Backward Slice

```
class SecurityApplication {

    private static void doSecurity(SecurityObject secObj1,
                                   SecurityObject secObj2) {

      SecurityAction action1 = new SecurityAction();
      SecurityAction action2 = new SecurityAction();
      action1.object = secObj1;
      action2.object = secObj2;

      Object res1 = action1.invoke();
      Object res2 = action2.invoke();

      doOtherThings(res1, res2);
    }
```

Any call to `invoke` with an untrusted field?

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Example: Caller of Relevant Methods

```
class SecurityApplication {

    public static void main(String[] args) {
      String result = setup(args);
      System.out.println(result);

      SecurityFactory uFactory = new UntrustedSecurityFactory();
      SecurityFactory tFactory = new TrustedSecurityFactory();

      SecurityObject uObject = uFactory.getSecurityObject();
      SecurityObject tObject = tFactory.getSecurityObject();

      doSecurity(uObject, tObject);
    }
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Example: Slicing Callers

```
class SecurityApplication {

    public static void main(String[] args) {
      String result = setup(args);
      System.out.println(result);

      SecurityFactory uFactory = new UntrustedSecurityFactory();
      SecurityFactory tFactory = new TrustedSecurityFactory();

      SecurityObject uObject = uFactory.getSecurityObject();
      SecurityObject tObject = tFactory.getSecurityObject();

      doSecurity(uObject, tObject);
    }
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Example: Propagate Slicing Information

```
class SecurityApplication {

    private static void doSecurity(SecurityObject secObj1,
                                   SecurityObject secObj2) {

      SecurityAction action1 = new SecurityAction();
      SecurityAction action2 = new SecurityAction();
      action1.object = secObj1;
      action2.object = secObj2;

      Object res1 = action1.invoke();
      Object res2 = action2.invoke();

      doOtherThings(res1, res2);
    }
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Final Slice

```
class SecurityApplication {
    public static void main() {
      SecurityFactory uFactory = new UntrustedSecurityFactory();
      SecurityObject uObject = uFactory.getSecurityObject();
      doSecurity(uObject);
    }

    private static void doSecurity(SecurityObject secObj1) {
      SecurityAction action1 = new SecurityAction();
      action1.object = secObj1;
      Object res1 = action1.invoke();
   }
}
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Experimental Results

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Experiment: OpenJDK 7-b147

Using the DOOP Framework and the LogicBlox Engine

- Clients derived from Java Secure Coding Guidelines
  - Caller Sensitive Methods (e.g., `Class.forName()`)
  - `AccessController.doPrivileged()`
  - Identify locations of interest
  - Combine with Escape and Taint analysis
- Aim: To compute context-sensitive points-to for these clients
  - Security analysis beyond the scope of this work

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Experiment: Clients

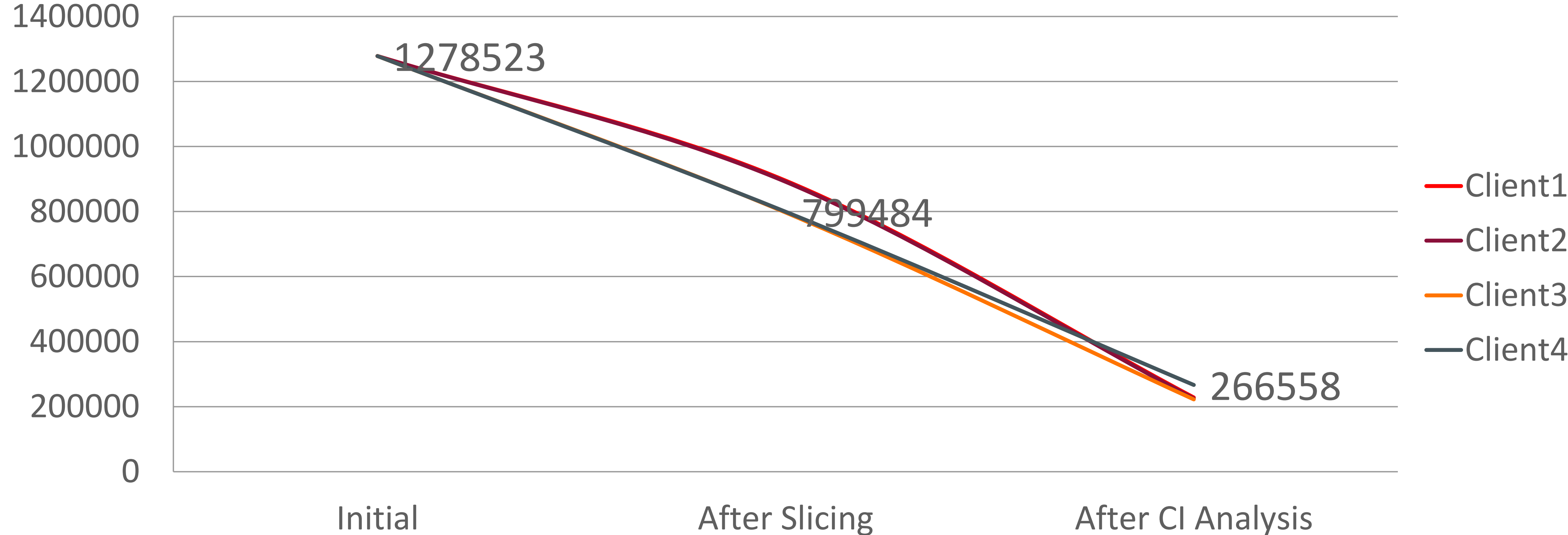
- Four clients chosen for experimentation
  1. Caller-sensitive-methods with tainted input, and escaping values
  2. Caller-sensitive-methods with only tainted input
  3. Caller-sensitive-methods with only escaping values
  4. `AccessController.doPrivileged()` with tainted inputs

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Results

Reduction in Number of Variables



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Results

Reduction in Number of Object Creation-Sites

	Initial	After CI Analysis
Client1		
Client2		
Client3		
Client4		

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Other Results

	Context-Insensitive	Context Sensitive	Context-Sensitive (Without Contexts)
Size of Points-To	120 Million	430 Million	2 Million
# Objects per Variable	140	-	8
# Call-graph Edges	330,000	80 Million	145,000

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Runtime

Intel Xeon E5-2660 (2.2GHz) 256GB RAM

Stage	Average Time
Variable Slicing	5 minutes
Context-Insensitive Analysis/Slicing	37 minutes
2-Heap+1-Object Analysis	3 hrs 53 minutes
Total	4 hrs 35 minutes

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Scalable Context-Sensitive Points-To Analysis

Demand-Driven, Slicing/Compaction Approach

Questions?

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |