ORACLE®

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.

Scaling Points-to to large Java Libraries

Challenges and Solutions

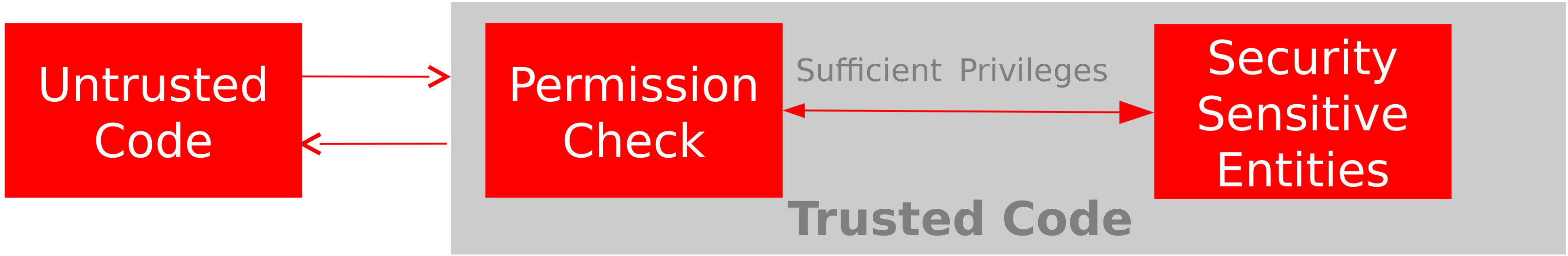
Raghavendra Kagalavadi
Postdoctoral Researcher,

Joint work with Paddy Krishnan, Bernhard Scholz, Yi Lu, Behnaz Hassanshahi

Oracle Labs, Brisbane
November, 2015

Goal: Java Security Analysis

- Automated security analysis for Java JDK™
  - Java Secure Coding Guidelines
- Find security bugs at development time before they are exploited



How to statically analyze?

- Points-to analysis fundamental to analyzing Java code

#	OpenJDK7-b147	Jython
Variables	1.5M	275K
Invocations	629K	121K
Object creation sites	193K	48K
Methods	171K	28K
Classes	17K	3558

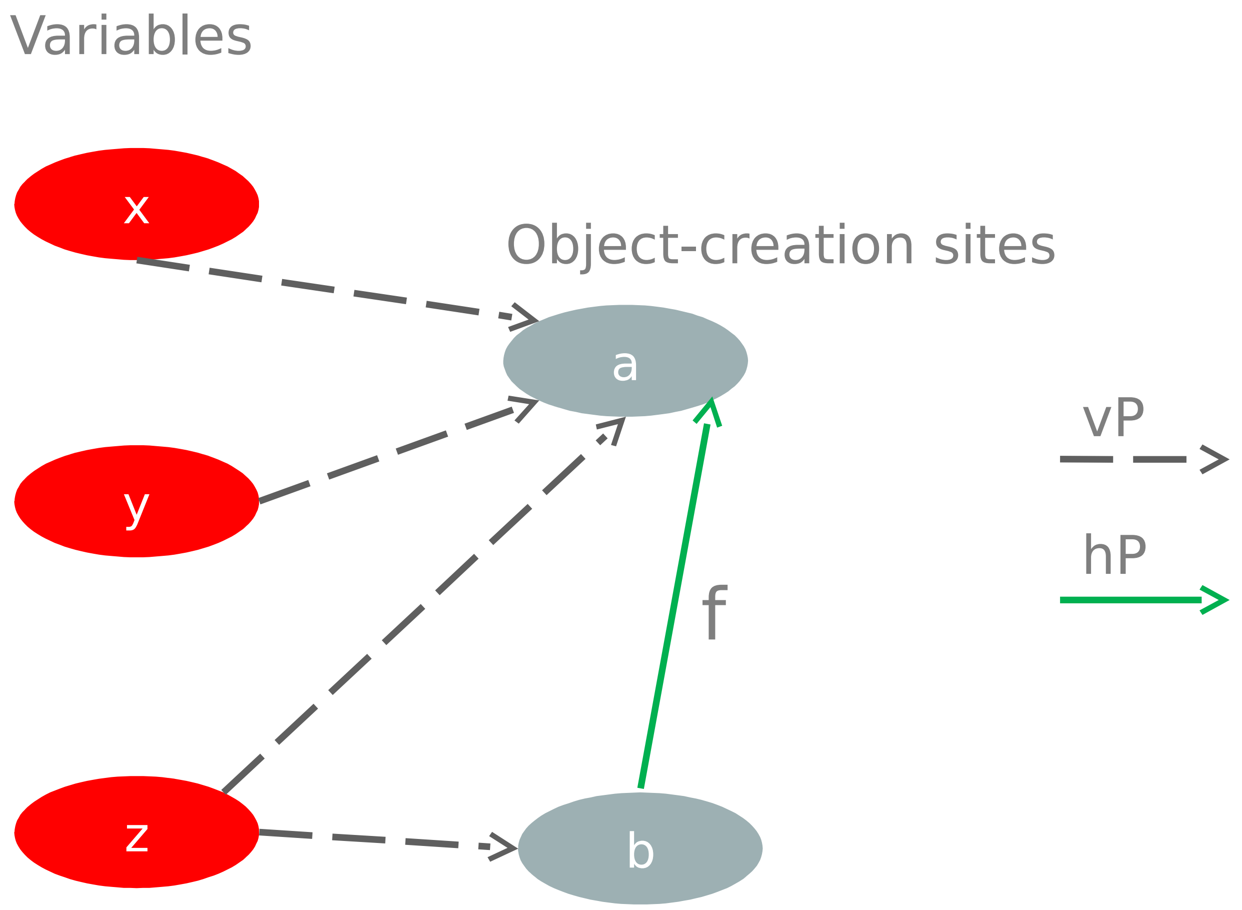
Scale
points-to analysis
to
large java library

Background

Context-insensitive,
flow-insensitive
Anderson's style points-to
for Java

Points-To Example

```
a:x=new Foo()
y=x;
if (cond) {
   z = y;
} else {
   b:z=new G();
   z.f = y;
}
```



Challenges and solutions

1 Library analysis

- Type information based analysis

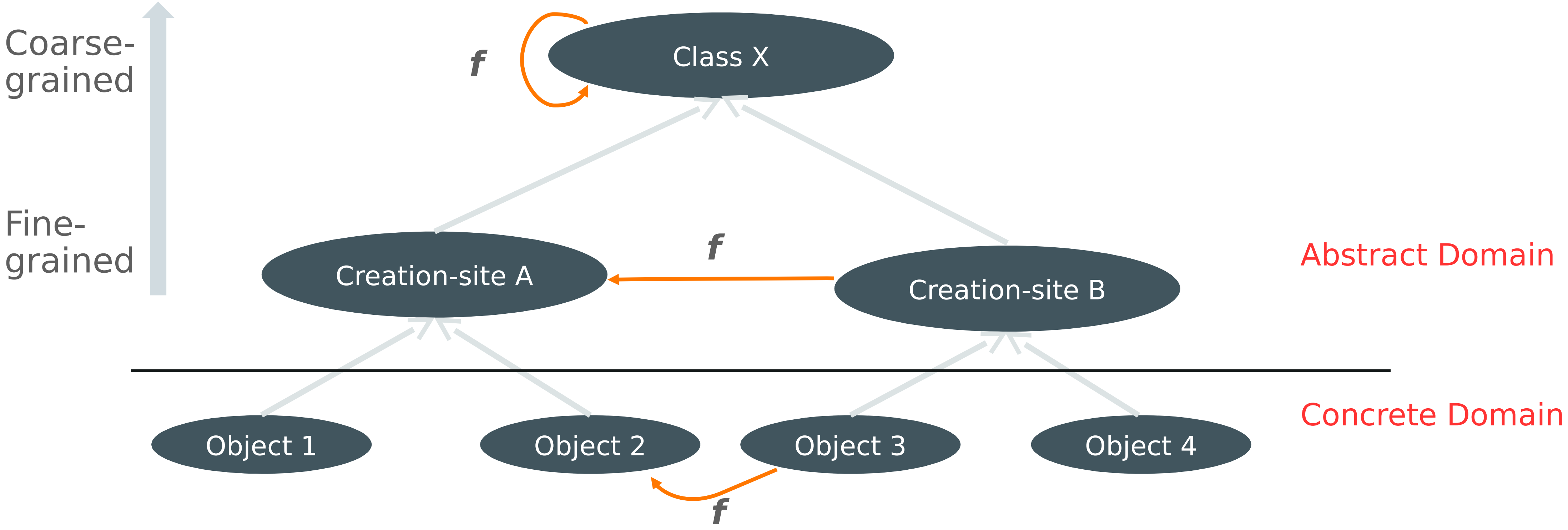
2 Scaling context sensitive points-to

- Demand-driven slicing

3 Implementation optimizations

Amalgamate Points-To with Type Abstraction

Assume creation-site A and B create instances of class X



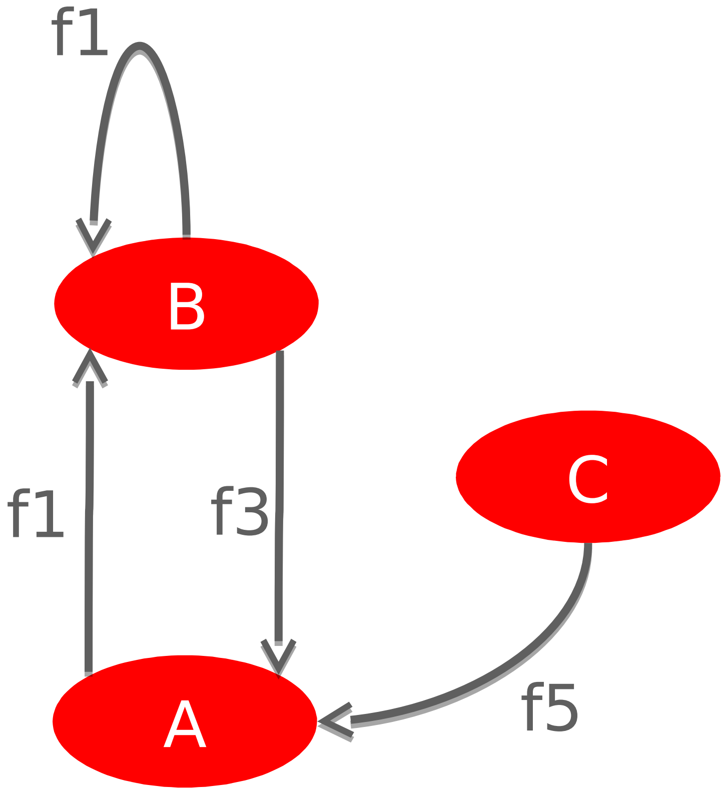
Heap Abstraction for Most General Application (MGA)

Example

```
class A {
 public B f1;
 private C f2; }

class B extends A {
 public A f3;
 private A f4 ; }

class C {
 public A f5; }
```



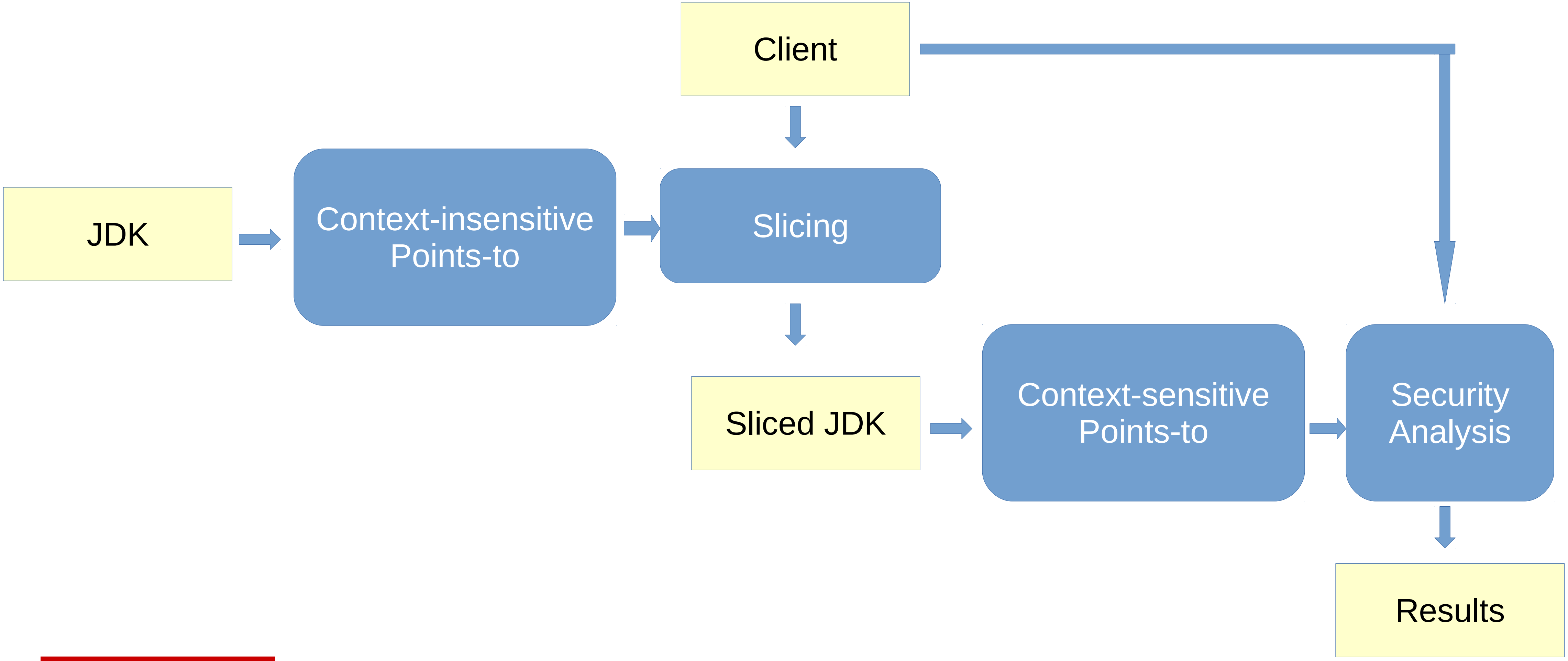
Context Sensitive Points-To for JDK

Even on OpenJDK7-b147 without Swing does not Scale. Times out (>1day)!

- Soufflé on Intel Xeon E5-2660 (2.2GHz) 256GB

Demand driven Analysis

Demand driven analysis



Experiment on OpenJDK7-b147 without Swing

- A Client derived from Java Secure Coding Guidelines
  - Identify program points of interest

	Before Slicing	After Slicing
Variables	1.3M	233K
Object creation sites	182K	35K

Using Soufflé
On Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz 32GB
20m 26G 8 cores

Implementation Optimizations

Optimizations for Soufflé

- Oracle Labs Datalog Engine
  - Efficient indexing
- Leveraging indexing in Soufflé
  - Reordering atoms
  - Manual Query planning

Reordering atoms

```
PotentialCallToExternalOverridableMethod(heaptype, callsite) :-
  VarPointsTo(heap, base),
  ExternalHeapAllocation(heap),
  OptVirtualMethodInvocationBase(callsite, base),
  HeapAllocationType(heap, heaptype).
```

Results of Implementation Optimizations

Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30 GHz, 396G using 8 cores

	Before	After
OpenJDK7-b147 without Swing	20m 26G	12m 17G
Full OpenJDK7-b147	Timeout	78m 255G

Conclusion

- Library analysis
  - Type information based analysis
- Scaling Context Sensitive Points-to
  - Demand-driven for Client
- Implementation Optimizations

Scaling Points-to to JDK is possible

Integrated Cloud

Applications & Platform Services

ORACLE®