Call Graph Construction for Java Libraries

Applied Static Analysis 2016

Michael Reif

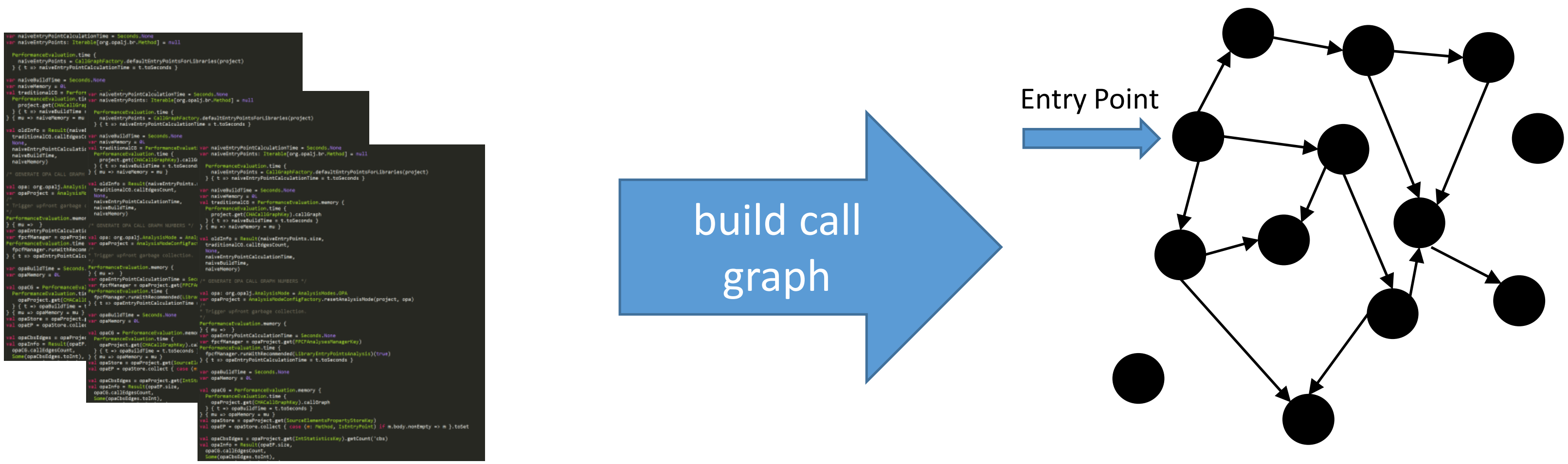
Dr. Michael Eichberg, Ben Hermann, Johannes Lerch,

Karim Ali Ph.D., Sebastian Proksch

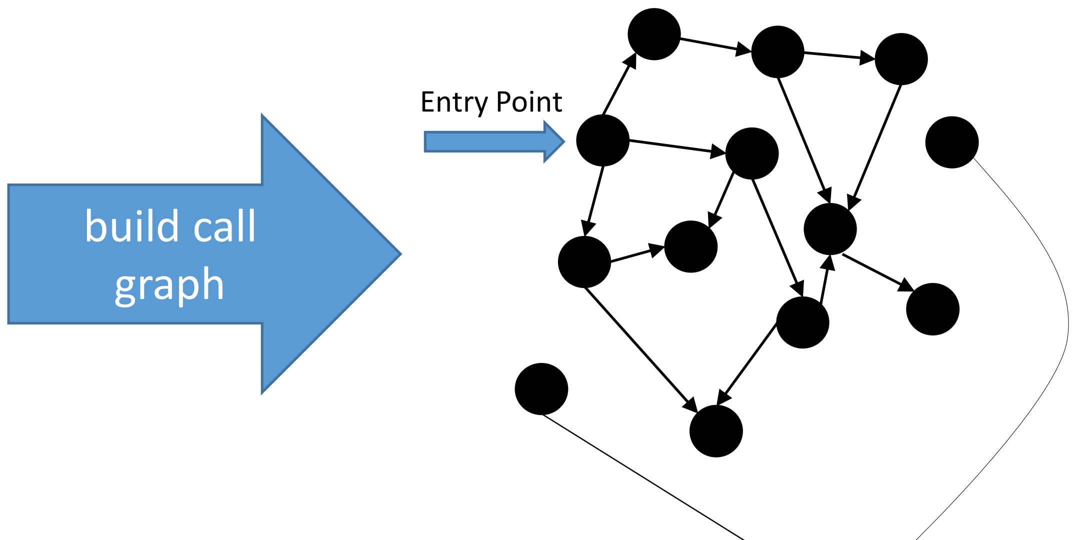
TECHNISCHE UNIVERSITÄT DARMSTADT

SOFTWARE TECHNOLOGY GROUP

An Application of Call Graphs is Dead Method Detection



An Application of Call Graphs is Dead Method Detection



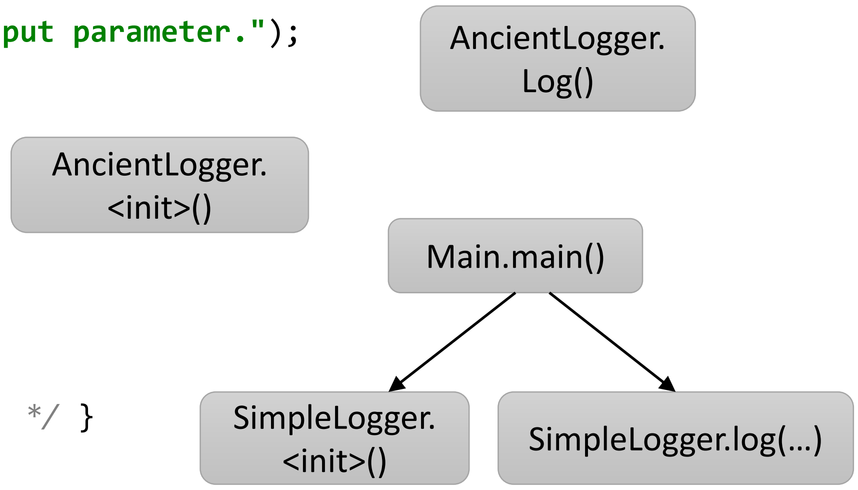
Dead Methods are all non-entry point methods that are not called by another method (excluding self-recursive calls).

Dead Code Detection in Applications is straight forward

```java
public class Main {
    public static void main(String[] args) {
        new SimpleLogger().
                      log("arguments", args.length + " input parameter.");
        // ...
    }
}

public interface Logger {
    void log(String category, String message);
}
public class SimpleLogger
implements Logger{
 public void log(String category, String message){ /* ... */ }
}

public class AncientLogger{
 public void log(String category, String message){ /*... */ }
}
```

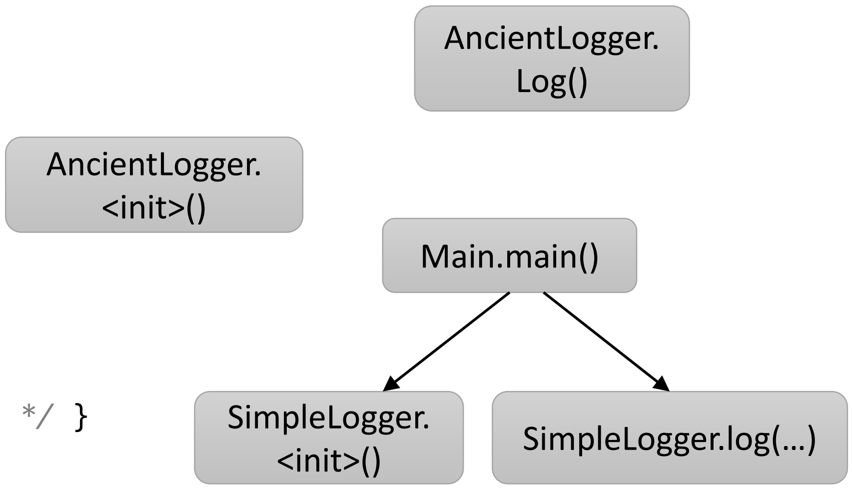


Dead Code Detection in Applications is straight forward

```
public class Main {
 public static void main(String[] args) {
   new SimpleLogger().
   log("arguments", args.length + " input parameter.");
        // ...
    }
}

public interface Logger {
    void log(String category, String message);
}
public class SimpleLogger
implements Logger{
 public void log(String category, String message){ /* ... */ }
}

public class AncientLogger{
 public void log(String category, String message){ /*... */ }
}
```



What to do when analyzing libraries?

All non-privates methods are assumed to be called by a client!

```
public class Main {
    public static void main(String[] args) {
        new SimpleLogger().
                  log("arguments", args.length + " input parameter.");
        // ...
    }
}

public interface Logger {
    void log(String category, String message);
}
public class SimpleLogger
         implements Logger {
 public void log(String category, String message){ /* ... */ }
}

public class AncientLogger {
 public void log(String category, String message){ /*... */ }
}
```

When Code is intended to be library code, there is no single entry point!

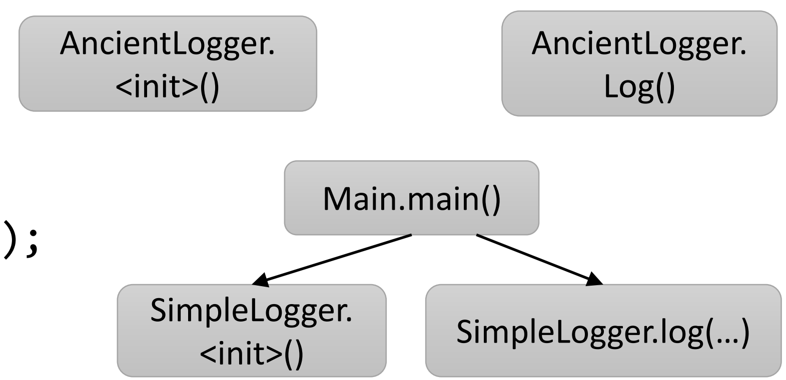
On the Challenges when analyzing Libraries

```java
public class Main {
 public static void main(String[] args) {
   new SimpleLogger().
     log("arguments", args.length + " input parameter.");
        // ...
 }
}

public interface Logger {
 void log(String category, String message);
}
public class SimpleLogger
         implements Logger {
 public void log(String category, String message){ /* ... */ }
}

public class AncientLogger {
 public void log(String category, String message){ /*... */ }
}
```

They call no methods, but are entry points!



On the Challenges when analyzing Libraries

```
public class Main {
    public static void main(String[] args) {
        Logger logger = …
        logger.log("arguments", args.length + " input parameter.");
        // ...
    }
}

public interface Logger {
    void log(String category, String message);
}
public class SimpleLogger
        implements Logger{
 public void log(String category, String message){ /* ... */ }
}

public class AncientLogger{
 public void log(String category, String message){ /*... */ }
}
```

Libraries are intended to be extended! Applications can introduce new subtypes!

Call-By-Signature (CBS) Resolution

When a method is invoked all methods that share the same method signature are determined as call targets. The method signature includes the method's name, return type and parameter types.

```
public static void main(String[] args) {
   Class<?> cls = args.getClass();
   cls.getName();
}


public class Person{
 String name;
 public String getName(){ /* ... */ }
}
```

Interface Invocations have to be resolved by method signature

Library

```
public interface Logger {
    void log(String category, String message);
}
public class SimpleLogger
       implements Logger{
 public void log(String category, String message){/* ... */}
}

public class AncientLogger{
 public void log(String category, String message){ /*... */ }

/* Somewhere in the library */
Logger l = …; l.log(…);
```

Interface invocations require **call-by-signature** resolution!
Hence, *log* should also point to *AncientLogger.log(…)*.

Application

```
public class MyLogger
       extends AncientLogger implements Logger {
    /* log method is inherited */ }
}
```

But why do we have to use only on interfaces invocations?

Library

```
public interface Logger {
 default void log(String category, String message) {/*...*/}
}

public abstract class SimpleLogger {
 abstract public void log(String category, String message);
}

/* Somewhere in the library */
Logger l = …; l.log(…);
```

That doesn’t work! Abstract methods have to implemented by the subclass!

Application

```
public class MyLogger
       extends SimpleLogger implements Logger {

    /* log
}
```

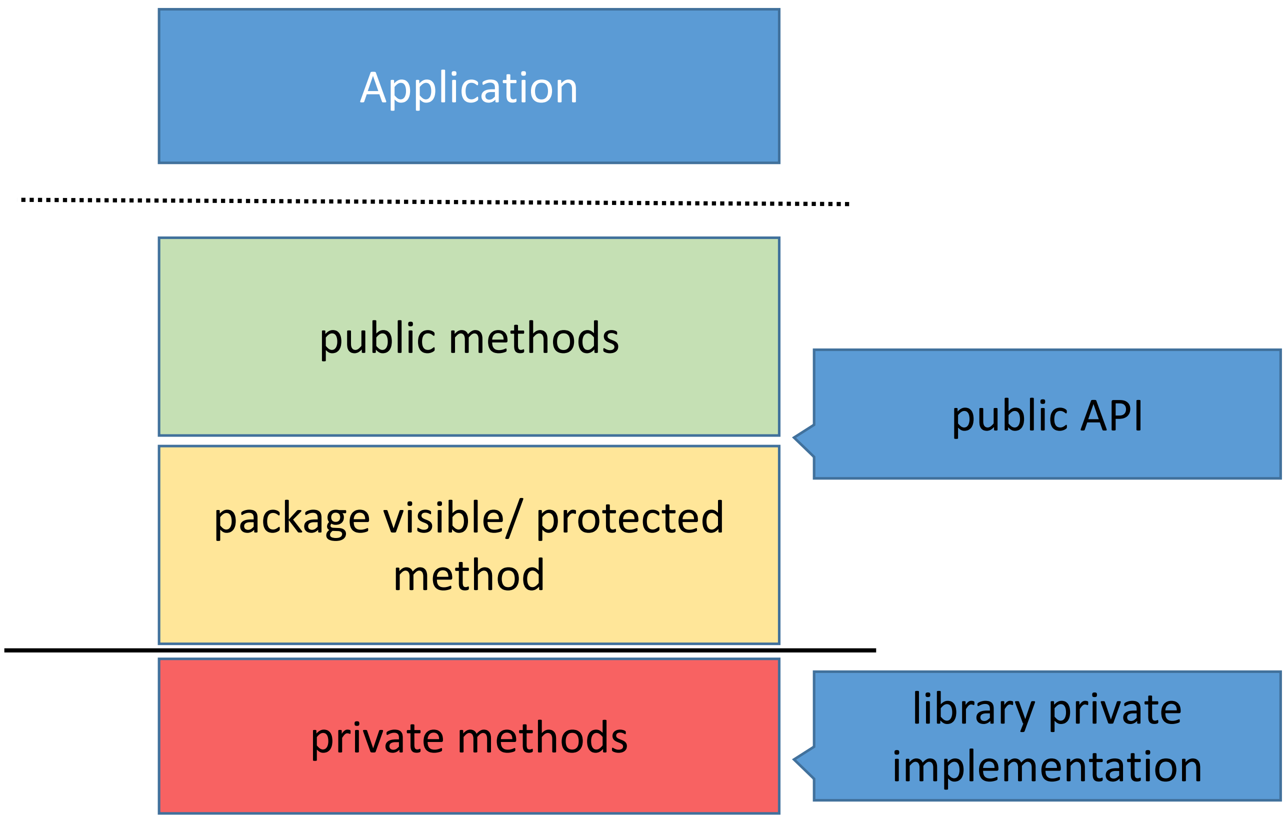
Other virtual calls are resolved soundly by each call graph algorithm!

The Problem with Libraries is that they are not closed worlds

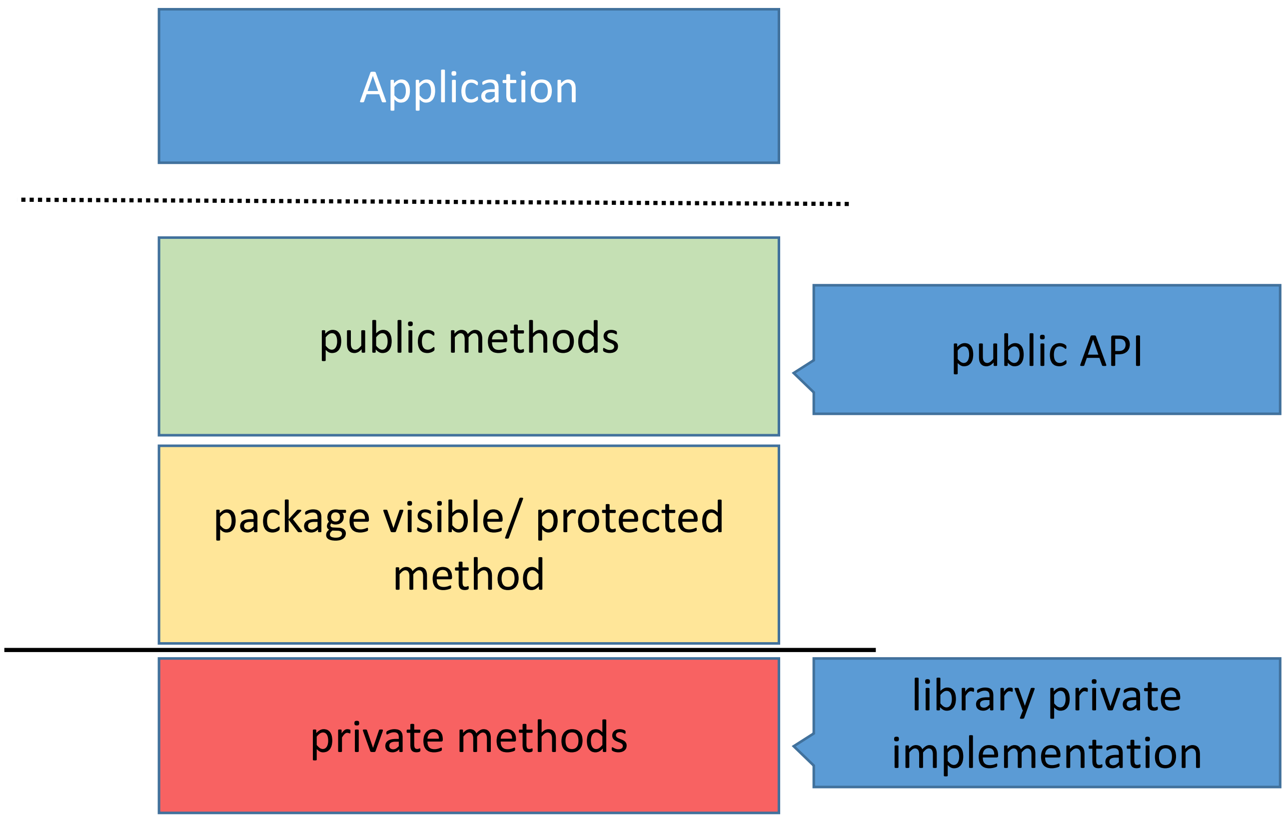
- Libraries are intended to be extended by future applications (inheritance)
  - **call-by-Signature** becomes necessary in library call graph construction

- The public interface of libraries is quite huge because every non-private method becomes an entry point

To start the call graph construction, we need the entry points into the library



Assumption: the application developer does not contribute to the used libraries



Contributing to a library

Library

```java
package de.tud.example

public class Example {
  public void getPublicInfo(){ /*... */ }

  void getInternalInfo(){ /*... */ }

  private void getSecretInfo(){ /*... */ }
}
```

Application

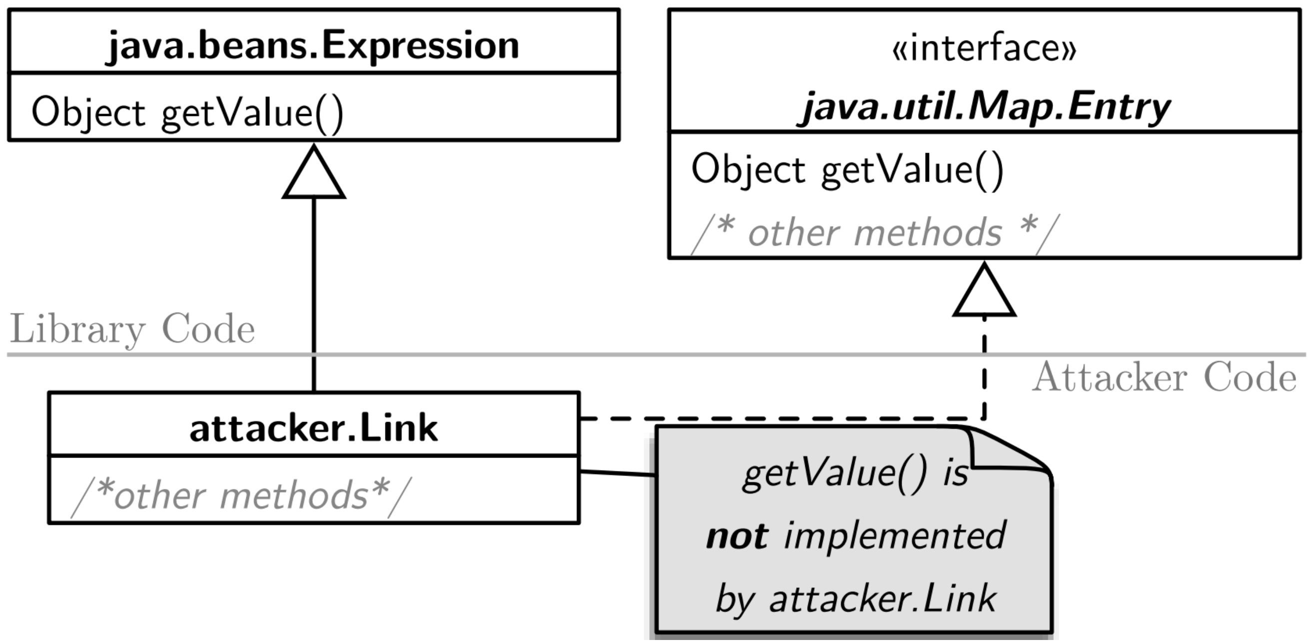
```java
package de.tud.example

public class Demo{
  Example ex = new Example();
  public void printInfo(){
     System.out.println(
       ex.getInternalInfo());
  }
}
```

Since both classes are defined in the same package. Protected and package visible members can be accessed!

But attackers want to create dependencies within the libraries if it enables exploitation

CVE-2010-0840



Some theory: Java has a stack-based security model

SecurityManager.checkPermission
SecurityManager.checkWrite
FileOutputStream.<init>
Attacker.doEvil
MyApplet.init

$\cap = \emptyset$

Code that leads to the exploit

CVE-2010-0840

```
HashSet<Map.Entry<Object, Object>> set = new HashSet<>();
set.add(new Link(System.class, "setSecuritymanager", null));
JList list = new JList(new Object[]{
 new HashMap<Object, Object>(){
   public Set<Map.Entry<Object, Object>> entrySet(){
          return set;
 }}
});
JFrame frame = new Jframe():
frame.getContentPane().add(li
frame.setSize(50, 50);
frame.setVisible(true);
```

```
java.lang.SecurityManager.checkPermission(RuntimeP...)
java.lang.System.setSecurityManager(SecurityManager)
java.beans.Expression.invoke()
java.beans.Expression.getValue(Object)
java.util.AbstractMap.toString()
...
javax.swing.JList.paint(Graphics)
...
java.awt.EventDispatchThread.run()
```

Only trusted caller on the stack!

Developers really want to write code that is not accessed by an application…

```java
package java.awt.datatransfer;

/**
 * A Multipurpose Internet Mail Extension (MIME) type, as defined
 * in RFC 2045 and 2046.
 *
 * THIS IS *NOT* - REPEAT *NOT* - A PUBLIC CLASS! DataFlavor IS
 * THE PUBLIC INTERFACE, AND THIS IS PROVIDED AS A ***PRIVATE***
 * (THAT IS AS IN *NOT* PUBLIC) HELPER CLASS!
 */
class MimeType implements Externalizable, Cloneable {

/* A lot of Code! */

}
```

Oracle JDK 8u77

Library developers write intentionally library private code!

Library private implementation in two different scenarios

- open-package assumption (OPA)
  - all private methods and fields
- closed-package assumption (CPA)
  - All classes, methods and fields that have at most package visibility
  - All public and protected methods or fields that are in a package visible class, unless:
    - The class inherits from a public class or interface which defines the respective method
    - The class has a public subclass which inherits the respective method

- java.* packages are always closed

Hardcoded in the JVM

Which methods are visible w.r.t. the given Assumption?

```
class SimpleLogger {
  public void log(){/*...*/}
  public void error(){/*...*/}
  void internal() {/*...*/}
}
public class ComplexLogger
    extends SimpleLogger(){
  public void log(){/*...*/}
  private void init() {/*...*/}
}
```

method	OPA	CPA
ComplexLogger.log()	✓	✓
ComplexLogger.init()	✗	✗
SimpleLogger.log()	✓	✗
SimpleLogger.error()	✓	✓
SimpleLogger.internal()	✓	✗

Which methods are visible w.r.t. the given Assumption?

```
public interface Logger { public void log(); }

class SimpleLogger {
  public void log(){/*...*/}
}
class ComplexLogger
    extends SimpleLogger() implements Logger {
  public void internal(){/*...*/}
}
public class Factory{
  public Logger createLogger(){ return new ComplexLogger(); }
}
```

method	OPA	CPA
SimpleLogger.log()	✓	✓
ComplexLogger.internal()	✓	✗

Design Space for Library Call Graphs

	Analysis Context		Closed-Package Assumption	Call-By-Signature Resolution
Library	Security Issues	in *our* library	✗ Someone will try to break it.	✓
		in 3rd party libraries	✗ Other libraries may try to break it.	✓
	Software Quality	in our library	✓ If someone behaves badly, we don‘t care.	✓
		in 3rd party libraries	✓ We use it as intented!	✓
Applicaton	Both security and general issues		(implicitly)	(Not relevant). ✗

Yes, there is really a need to analyze software libraries

Library Call Graph Algorithms

Standard Call Graph Algorithms (CHA, RTA, VTA…) do not work as they are

There is **no all-in-one** call graph algorithm

Unnecessarily huge

Too many entry points

Unsound

No call-by-signature

Not fit the use case

both

Steps to extend the CHA-based call graph to be sound for libraries

1. Compute the entry point set w.r.t. to the applied assumption (OPA/CPA)
2. Start to build the call graph starting from each entry point
   - Resolve each call sites by the class hierarchy
   - If the receiver type an interface, resolve it additionally with a constrained call-by-signature resolution

Entry Point Computation

Compute the entry point set under the open-package assumption

The following algorithm determines whether a given concrete method is an entry point.

```
def isEntryPoint(declType, method): Boolean =
      maybeCalledByTheJVM(method) ||
      Method.isStaticInitalizer ||
      (!method.isPrivate &&
             (method.isStatic || declType.isInstantiable))
```

Denotes the library private implementation

The JVM calls some methods implicitly during execution

```
maybeCalledByTheJVM(method)
```

- *finalize()* is called during Garbage Collection
- *readObject(), writeReplace(), readResolve(), writeObject()* etc. are called during (de-)serialization of *Serializable* or *Externalizable* classes

These methods are often private, hence, would not be considered as entry points!

Non-static methods can only be called if the declaring class is instantiable

```
declType.isInstantiable
```

- Irrelevant for static methods
- the class either has:
  - A non-private constructor
  - Or a (static!) factory method that returns instances of the class

A static method that calls the private constructor and returns a supertype (reflexive) of the class type

Examples for instantiablity in OPA

instantiable

```java
class  Foo{
 Foo(){/*…*/}
}
```

```java
class  Foo{
 private Foo(){/*…*/}

 public static Foo newInst(){
  return new Foo();
 }
}
```

A factory method!

not instantiable

```java
class  Foo{
 private Foo(){/*…*/}

 public static Foo newInst(){
  new Foo();
  return null;
 }
}
```

The object will be created but not returned!

Compute the entry point set under the closed-package assumption

The following algorithm determines whether a given concrete method is an entry point.

```
def isEntryPoint(declType, method): Boolean =
  maybeCalledByTheJVM(method) ||
  (Method.isStaticInitalizer && declType.isAccessible) ||
  (method.isClientCallable &&
       (method.isStatic || declType.isInstantiable))
```

Denotes the library private implementation

We can be more restrictive under CPA!

Static initializers are executed when the class or a subclass is accessed

```
declType.isAccessible
```

- access takes place when the name of the class or a subclass can appear in the application code
- accessible classes are either:
  - Public
  - Package private and have a public subclass (transitive)

Accessibility and the execution of a static initializer in case of OPA

Library

```
class Foo  { }
public class Bar extends Foo {
 public static final int num = 12;
 public static String s1 = “const”;
 public static final String s2= “realConst”;
 public static final Object obj = new Object();
}
```

Application

```
/* somewhere in the application */}

new Bar()      ✓
Bar.num;       ✗
Bar.s1         ✓
Bar.s2         ✗
Bar.obj        ✓
```

Access of primitive constants does trigger the static initializer

All method that can be called by a future application

`method.isClientCallable`

```
def isClientCallable(declType, method): Boolean =
  (method.isPublic || method.isProtected) &&
  (declType.isPublic ||
       declType.subclasses.exists { subC =>
         subC.isPublic && subC.inherits(m)})
```

If the method is not defined in a public class, it must be inherited by a public subclass, hence, the method is not overridden on the path from the declaring class to the public subclass!

Using visibility and inheritance concepts to determine when a method is client callable

Library

```
class Foo  {
 protected void protBar(){/* ... */}
 public void pubBar(){/* ... */}
}
public class Bar extends Foo {
 public void pubBar(){/* ... */}
 void defaultVisBar() {/* ... */}
 private void priv() {/* ... */}
}
```

Application

```
/* somewhere in the application */}

Bar.pubBar();            ✓
Bar.defaultVisBar();     ✗
Bar.priv();              ✗
Foo.protBar();           ✓
Foo.pubBar();            ✗
```

Overridden by **Bar.pubBar**!

instance methods can only be called if the declaring class is instantiable

```
declType.isInstantiable
```

- Same as in case of OPA
- The class has to be accessible as discussed earlier

Call-By-Signature Computation

Necessary CBS Resolution differs from a pure call-by-signature call graph

- call-by-signature is only used on interface invocations
- call targets are disjunct from the call targets of a more advanced call graph algorithm

How to compute call-by-signature call targets in the case of OPA

The following algorithm determines the call-by-signature call targets of call sites with an interface receiver.

```
def cbsTargets(declIntf, mSig): Set[Method] =
  project.findConreteMethods(mSig).filter { m =>
    m.isPublic &&
    !m.definingClass.isEffectivelyFinal &&
    !(m.definingClass <: declIntf)
}
```

All interface methods are public

<: denotes the (reflexive) subtype relation

The class is either final or has only private constructor(s)

How to compute call-by-signature call targets in the case of CPA

```
def cbsTargets(declIntf, mSig): Set[Method] =
  project.findConreteMethods(mSig).filter { m =>
    m.isPublic &&
    !m.definingClass.isEffectivelyFinal &&
    !(m.definingClass <: declIntf)
    &&
    ( m.definingClass.isPublic ||
      m.definingClass.subclasses.exists { subC =>
        subC.isPublic &&
        !(subC <: declIntf) && subC.inherits(m)})
  }
```

The subclass does not implement the interface!

Software quality analyses improve in case of CPA over OPA

Oracle JDK 7u80

Algorithm	naïve/LibCHA$_{OPA}$	LibCHA$_{CPA}$
Reported Methods	218	2 119
Technical Artifacts	114	114
Swing PLAF related	4	1 325
Presumably Dead	100	680

Table 6: Number of dead methods found in the JDK

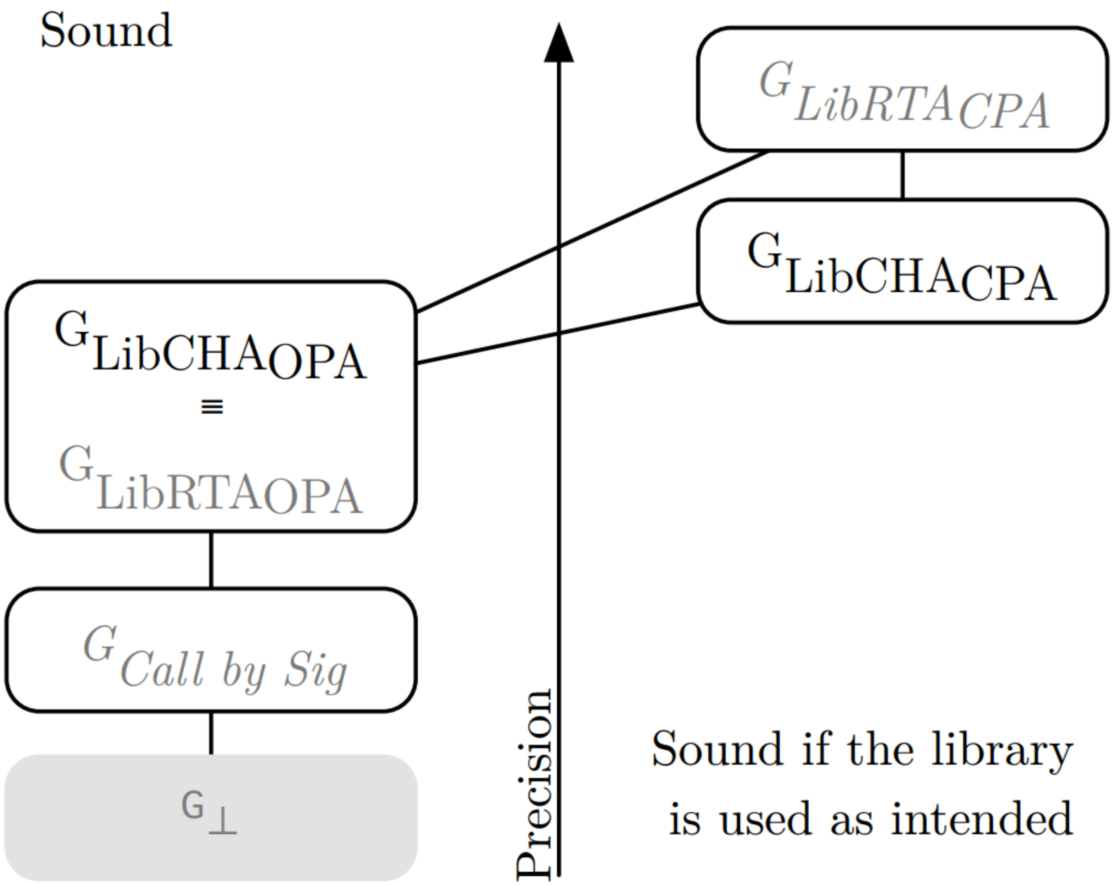
We don't resolve reflection...

6.8 times more dead methods found

In case of CPA it is also possible to report non-private methods as dead!

What is about more advanced call graph algorithms like RTA?

Why is CHA in case OPA equivalent to RTA?

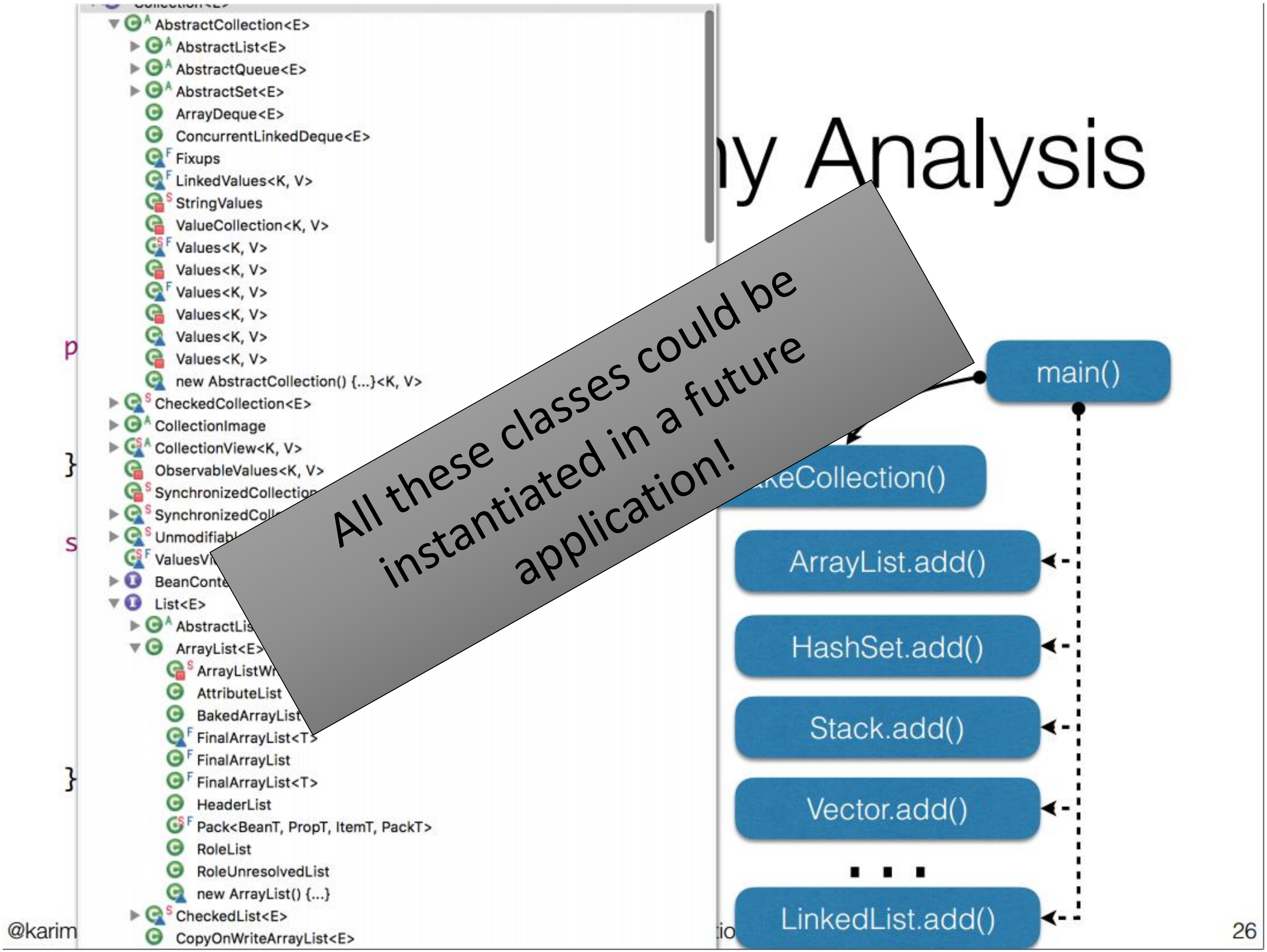


RTA is slightly better in case of CPA!

LibRTA degenerates to LibCHA in the case of OPA

```
public static void main(String[] args) {
   Collection c = makeCollection(args[0]);
   c.add(“elem”);
}


static Collection makeCollection(String s) {
  if(s.equals(”elem”)) {
   return new ArrayList();
  } else {
   return new HashSet();
  }
}
```



y Analysis
AbstractCollection<E>
AbstractList<E>
AbstractQueue<E>
AbstractSet<E>
ArrayDeque<E>
ConcurrentLinkedDeque<E>
Fixups
LinkedValues<K, V>
StringValues
ValueCollection<K, V>
Values<K, V>
Values<K, V>
Values<K, V>
Values<K, V>
Values<K, V>
Values<K, V>
new AbstractCollection() {...}<K, V>
CheckedCollection<E>
CollectionImage
CollectionView<K, V>
ObservableValues<K, V>
SynchronizedCollection
SynchronizedColl
Unmodifiabl
ValuesVi
BeanConte
List<E>
AbstractLis
ArrayList<E>
ArrayListWr
AttributeList
BakedArrayList
FinalArrayList<T>
FinalArrayList
FinalArrayList<T>
HeaderList
Pack<BeanT, PropT, ItemT, PackT>
RoleList
RoleUnresolvedList
new ArrayList() {...}
CheckedList<E>
CopyOnWriteArrayList<E>
All these classes could be instantiated in a future application!
main()
eCollection()
ArrayList.add()
HashSet.add()
Stack.add()
Vector.add()
. . .
LinkedList.add()
@karim
26

LibRTA can be more precise in some cases than LibCHA in the case of CPA

```
interface InternalCollection { public void add(String s); }

class List implements InternalCollection {
   public void add(String s){/*…*/}}
class Set implements InternalCollection {
   public void add(String s){/*…*/}}

public static void main(String[] args) {
   InternalCollection c = makeCollection(args[0]);
   c.add("elem");
}
static Internal makeCollection(String s) {
   return new List();
}
```

Never instantiated in this package.

All classes belong to the library private implementation!

Also VTA can be adapted to work with libraries

Recap: Main Idea is to propagate **types** from allocation sites to potential **variable references**

1. start with a pre-computed **library** call graph
   - The entry points changed accordingly to the assumption
2. entry point method **parameters** have to be resolved to all types in the type hierarchy that can be instantiated by the client!

Also VTA can be adapted to work with libraries

3. Build type propagation graph
4. Collapse strongly-connected components
5. Propagate types along the final Directed Acyclic Graph

Use **call-by-signature** on interface invocations when resolving call sites!