Applied Static Analysis

Inter-procedural Analysis - Basic Call Graph Algorithms

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

> If you find any issues, please directly report them: GitHub

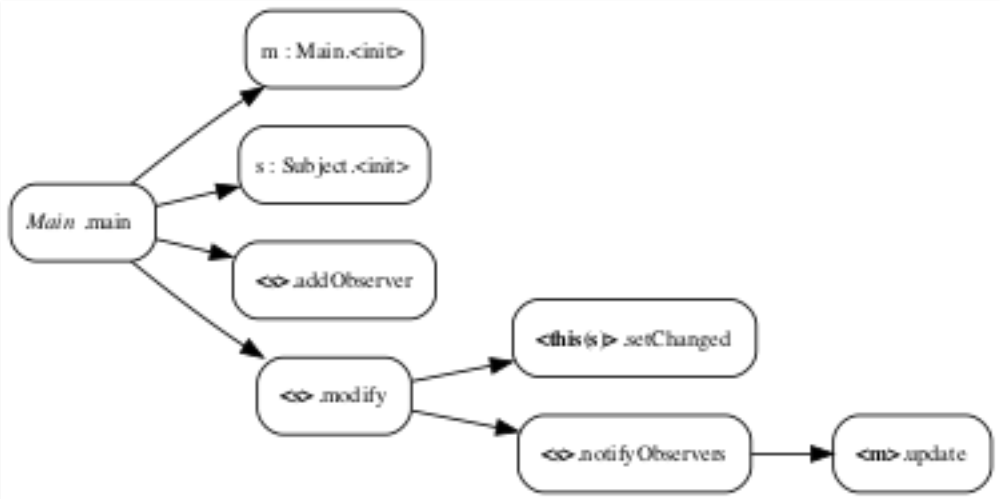
A Call Graph

- Core data structure to conduct an inter-procedural analysis.
- A call graph is a static abstraction of all method calls that a program may execute at runtime.
  - (call sites in) *methods* are nodes
  - *calls* are edges

Example - Monomorphic Calls

In the following call graph, all calls can be resolved to a single call target:

```
public class Main implements Observer {
   public static void main(String[] args) {
      Main m = new Main();
      Subject s = new Subject();
      s.addObserver(m);
      s.modify();
   }
   public void update(Observable o, Object arg) {
      System.out.println(o+" notified me!");
   }
   static class Subject extends Observable  {
      public void modify() {
          setChanged();
          notifyObservers();
}  }  }
```

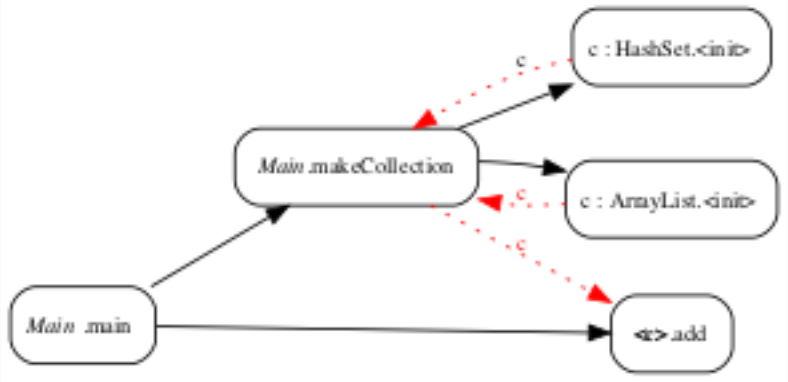


Example - Polymorphic Calls

In the following call graph, the call target is unknown. The target of the `add` call could either be a `HashSet` or an `ArrayList` object.

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Collection c = makeCollection(args[0]);
      c.add(args[1]);
   }
   static Collection makeCollection(String s) {
      if(s.equals("list")) {
         return new ArrayList();
      } else {
         return new HashSet();
}  }  }
```



Call Graph Construction

> Compute a sound and precise approximation of all call edges for a given project.

I.e., the call graph should only contain call edges related to calls that may happen at runtime (precise) and it should contain all call edges that may happen at runtime (sound).

In practice, call graphs for real-world programming languages are often neither sound nor precise. And even the most precise ones are still rather imprecise. If soundness is required, e.g., to prove that some software satisfies some specific properties, then it is often required that only the soundly handled subset of a specific programming language such as Java, C# or C++ is used.

Basic idea:

- Given the set of (immediate) entry points,
- compute all methods that can transitively be reached from this set.

An entry point is each method that may be called with an unknown or only partially known context. In case of a simple command line application it is the `main` method. In case of an application with a graphical user interface the set of entry points (at least) also contains the event handlers.

Depending on the kind of code-base the set of entry points may be known immediately or may be discovered along the way.

Call Graph Construction Algorithms

- Basic Algorithms:
  - Call-by Signature
  - Class Hierarchy Analysis (CHA)
- Advanced Algorithms (require fixed point computations):
  - Runtime Type Analysis (RTA)
  - The family: (C|M|F|X)TA (C=Class,M=Method,F=Field,X~{Method And Field})
  - Variable Type Analysis (VTA)
  - ...
- Context-sensitive Algorithms:
  - k-CFA (CFA=Control Flow Analysis)
  - ...

Call Graph Algorithm: Call-by Signature

Basic definition:

> All methods which are entry points are reachable.
>
> For a call site `cs` in a reachable method M, assume a call edge to any method M' that has the same signature as `cs` describes. Add M' to the set of reachable methods.
> The signature is defined by the name of the method and the types of the parameters.

(In Java bytecode and frameworks which analyze Java bytecode, the signature also encompasses the return type.)

Call-by Signature - assessment

- Basic Properties:
  - Trivial to implement
  - Very fast
  - (In theory) Computes a sound over approximation
- Issues:
  - very imprecise

Call-by signature resolution is in some cases required to correctly approximate the call graphs of libraries. The latter is in particular required if you want to perform security-related analyses.

Call Graph Algorithm: Class Hierarchy Analysis (CHA)

Basic definition:

> All methods which are entry points are reachable.
>
> Given a reachable method M, for a polymorphic (virtual) call site `cs` on the declared type `T`, assume a call edge to any method M' with the signature of `cs` that belongs to a subclass of `T`.

The subclass relation is reflexive.

Several implementation-level decisions may affect the overall precision: e.g. an abstract class `SubT` which implements `m` and is a subtype of `T` may not be considered a call target if all concrete subtypes of `SubT` also implement `m` and the set of subtypes is also known and fixed.

CHA based call graphs are frequently used in Java because CHA is very fast to compute and trivially to implement.
In Java, the call edges related to `Object`'s `toString` method along with `Iterator`'s `next` and `hasNext` methods usually make up a very significant part of the call graph (~30% in case of the JDK). E.g., the `toString` method is overwritten more than 2323 times in the Open JDK 11.0.3 and there are more than 13130 call sites!

Class Hierarchy Analysis (CHA) - assessment

- Basic properties:
  - Simple to implement
  - Very fast
  - (In theory) computes a sound over approximation
- Issues:
  - More precise than *Call-by Signature*, but still rather imprecise.

When implementing CHA or even more precise algorithms, decisions have to be made regarding the handling of incomplete class hierarchies which are common place when analyzing real-world projects.

Rapid Type Analysis (RTA)

Basic definition (constraint based):

> Let R be the set of all methods and S be an approximation of the set of all classes instantiated by some program run.

- $staticType(r)$ denotes the static type of the expression r (the receiver of the call).
- $subTypes(t)$ denotes the set of all sub types of t including t.
- $methodResolution(C, m)$ denotes the (unique) method M' that is found when method resolution is performed given the declared class C and the method signature m.

(1) $entryPoints \subseteq R$

(2) For each method M, each virtual call site $r.m$ in M,
and each class $C \in subtypes(staticType(r))$
where $methodResolution(C, m) = M'$:
$(M \in R) \wedge (C \in S) \Rightarrow (M' \in R)$

(3) For each method M, and for each (successful) $newC()$ occurring in M:
$(M \in R) \Rightarrow (C \in S)$

The implementation requires a fixed point computation.

Rapid Type Analysis - example

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Collection c = makeCollection(args[0]);
      c.add("x");
      new LinkedList();
   }
   static Collection makeCollection(String s) {
      if(s.equals("list")) {
         return new ArrayList();
      } else {
         return new HashSet();
}  }  }
```

In the above case, we will have a call edge from `c.add` to `ArrayList.add`, `HashSet.add` and `LinkedList.add`, but not, e.g., `Vector.add`, because no instance of `Vector` is ever created.
In the first step, however, i.e., before we have analyzed `makeCollection`, no call edge will be added to `c.add`!

Rapid Type Analysis (RTA) - assessment

- Basic properties:
  - the implementation requires the maintenance of a global set and a fixed point computation
  - significantly more precise than CHA (many types which are not used in a program are soundly ignored)
- Issues:
  - more precise than CHA, but a parallelization of the implementation is non-trivial (no data-parallelism)

Separate sets for methods and fields (XTA)

Basic definition[1] (constraint based):

> Let S_M denote a distinct set for each method M and S_f denote a distinct set for each field f.

- $paramTypes(M)$ denotes the set of static types of the arguments of the method M.
- $subTypes(T) = \cup_{t \in T} subTypes(t)$.
- $returnType(M)$ denotes the declared return type of M.

(1) $entryPoints \subseteq R$

---

(2) For each method M, each virtual call site $r.m$ in M, and each class $C \in subtypes(staticType(r))$ where $staticLockup(C, m) = M'$:

$$(M \in R) \wedge (C \in S) \Rightarrow \begin{cases} M' \in R \wedge \\ subTypes(paramTypes(M')) \cap S_M \subseteq S_{M'} \wedge \\ subTypes(returnTypes(M')) \cap S_{M'} \subseteq S_M \wedge \\ C \in S_{M'} \end{cases}$$

(3) For each method M, and for each (successful) $newC()$ occurring in M:
$(M \in R) \Rightarrow (C \in S_M)$

(4) For each method M in which a read of a field f occurs:
$(M \in R) \Rightarrow (S_f \in S_M)$

(5) For each method M in which a write of a field f occurs:
$(M \in R) \Rightarrow (subTypes(staticType(f)) \cap S_M) \subseteq S_f$

XTA - example

```
class Main {
    static Collection c;
    static Object o;
    public static void main(String[] args) {
        c = initC1();
        set0(c);
        c = initC2(new String("x"));
        process();
    }
    static Collection initC1() {
        return new ArrayList<Object>(o);
    }
    static Collection initC2(Object o) {
        List<Object> l = new ArrayList<Object>(o);
        return l;
    }
    static String set0(Object o){
        Main.o = o;
        o.toString();
    }
    static void process() {
        List<Object> l = new LinkedList<Object>();
        System.out.println(l.size());
    }
}
```

The field `o` may store values of type `String` and `ArrayList`, but not of type `LinkedList`. Therefore the call `o.String` will not be a call to `<LinkedList>.toString`. (The analysis is not flow sensitive!)

XTA - assessment

- Basic Properties:
  - the implementation requires the maintenance of a multiple (many) sets and a fixed point computation
  - typically does not identify significantly more unreachable methods than RTA
  - reduces the number of call graph edges (on average) by 7%.
  - using XTA the number of virtual call sites that are classified as polymorphic is ~7%, which is only marginally better than RTA.
- Issues:
  - Runtime is on average ~5 times longer than RTA.

Soundiness

> Soundy is the new sound.[2]

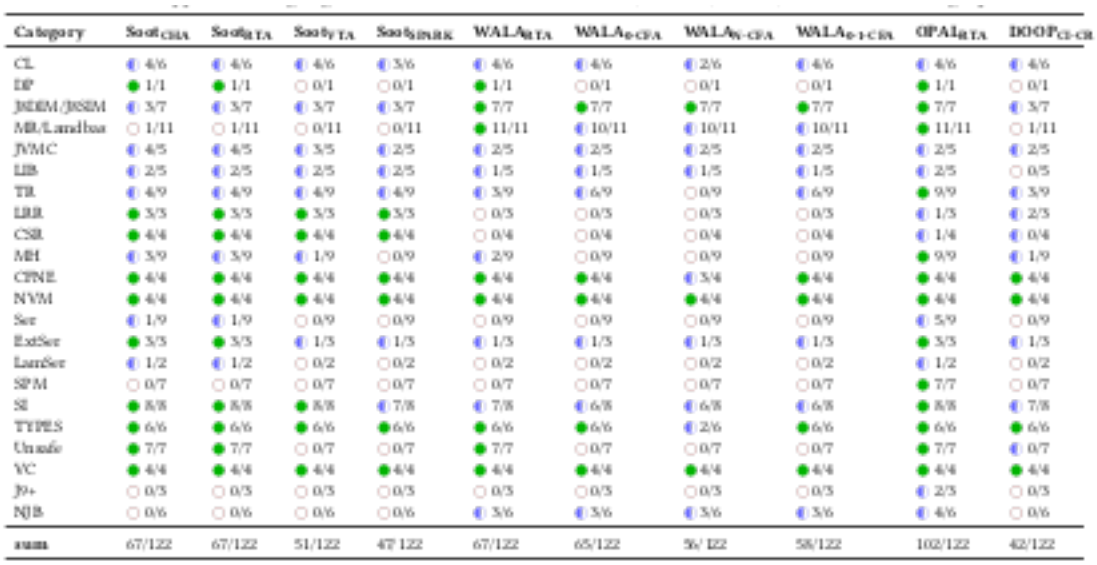
As discussed at the very beginning, most real-world static analyses are *just* soundy; i.e., they don't all features supported by the programming language/environment.

Sources of unsoundness w.r.t. call graph construction (for Java programs) are:

- Reflection (often very simple reflection is handled, but advanced usages are not)
- Class loading
- On-the fly class generation
- System events (`Thread.start` → `Thread.run` , finalization of objects)
- Serialization
- Native methods
- New/advanced language constructs (e.g., `invokedynamic`)

A comprehensive discussion of sources of unsoundness and the effect of implementation decisions on call graphs can be found in [3].

Supported Language Features Across Different Frameworks:



Category	$Soot_{CHA}$	$Soot_{RTA}$	$Soot_{VTA}$	$Soot_{SPARK}$	$WALA_{RTA}$	$WALA_{0\text{-}CFA}$	$WALA_{N\text{-}CFA}$	$WALA_{0\text{-}1\text{-}CFA}$	$OPA1_{RTA}$	$DOOP_{CI\text{-}CR}$
CL	4/6	4/6	4/6	3/6	4/6	4/6	2/6	4/6	4/6	4/6
DP	1/1	1/1	0/1	0/1	1/1	0/1	0/1	0/1	1/1	0/1
JSEM/JSSM	3/7	3/7	3/7	3/7	7/7	7/7	7/7	7/7	7/7	3/7
MB/Landlua	1/11	1/11	0/11	0/11	11/11	10/11	10/11	10/11	11/11	1/11
JVMC	4/5	4/5	3/5	2/5	2/5	2/5	2/5	2/5	2/5	2/5
LB	2/5	2/5	2/5	2/5	1/5	1/5	1/5	1/5	2/5	0/5
TR	4/9	4/9	4/9	4/9	3/9	6/9	0/9	6/9	9/9	3/9
LBR	3/3	3/3	3/3	3/3	0/3	0/3	0/3	0/3	1/3	2/3
CSR	4/4	4/4	4/4	4/4	0/4	0/4	0/4	0/4	1/4	0/4
MH	3/9	3/9	1/9	0/9	2/9	0/9	0/9	0/9	9/9	1/9
CPNE	4/4	4/4	4/4	4/4	4/4	4/4	3/4	4/4	4/4	4/4
NVM	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
Ser	1/9	1/9	0/9	0/9	0/9	0/9	0/9	0/9	5/9	0/9
ExtSer	3/3	3/3	1/3	1/3	1/3	1/3	1/3	1/3	3/3	1/3
LamSer	1/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	0/2
SPM	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	7/7	0/7
SI	8/8	8/8	8/8	7/8	7/8	6/8	6/8	6/8	8/8	7/8
TYPES	6/6	6/6	6/6	6/6	6/6	6/6	2/6	6/6	6/6	6/6
Unsafe	7/7	7/7	0/7	0/7	7/7	0/7	0/7	0/7	7/7	0/7
YC	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
J9+	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	2/3	0/3
NJB	0/6	0/6	0/6	0/6	3/6	3/6	3/6	3/6	4/6	0/6
sum	67/122	67/122	51/122	47/122	67/122	65/122	56/122	58/122	102/122	42/122

Implementation Differences and their Effect

Often the same conceptual algorithms are implemented very differently across frameworks (e.g., WALA, OPAL, SOOT) and, therefore, the call graphs vary widely in the their precision.

Given the following example:

```
Collection c1 = new LinkedList();
Collection c2;
if(some_condition){
   c2 = new ArrayList();
} else {
   c2 = new Vector();
}
c2.add(null);  // CALL SITE
```

In the above example, the declared receiver type of `c2` is collection and, therefore, Wala will create an edge to all subtypes of `Collection` which define an `add` method. Soot instead computes the least upper type bound of `ArrayList` and `Vector` (which is `List`) and will only add an edge to `add` methods defined by subtypes of `List`. OPAL supports union (and intersection) types and is able to determine that the call will either go to `ArrayList.add` or `Vector.add`.

Implementation of Call Graph Constructions - current state

The following table, which lists the size of some call graphs computed for some well known programs using different static analysis frameworks, is taken from Judge[^Judge]:

Framework	jasml	javacc	jext	proguard	sablecc
$Wala^{RTA}$	75817	76643	79513	80240	77607
$SOOT^{RTA}$	12134	12986	34470	36256	14088
$OPAL^{RTA}$	3195	4222	15705	7771	4932

Basically, call graphs computed by different frameworks are incomparable and, therefore, also the results of all analyses build on top of them.

Call Graph Construction for Libraries (LibCG)

The following challenges apply to all open code bases where new/unknown classes may make use of existing classes.

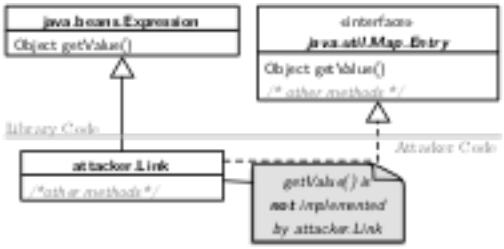
Challenges (when compared to building call graphs for applications):

- New (sub-)classes may introduce new caller/callee relations in the exiting library code between classes which are originally not in a sub/supertype relation.
- When we want to analyze the quality of a library, we typically want to treat the library's code base as **closed**.
- When we want to analyze the library w.r.t. security problems, we want to analyze it as **open** to get the complete threat surface.

Hence, the analysis techniques depend on the goal of the analysis

Most modern applications or frameworks (e.g., Eclipse, Netbeans, Glassfish, Tomcat, Spring,...) are explicitly extensible by so-called plugins. To interact with the core application/framework an API is provided which is – from the point-of-view of the plugin – a library.

LibCG - covering possible library extension



The given example is inspired by the `CVE-2010-0840`. This example exploits the fact that it is possible to create a class which *links* two classes which are not expected to be linked. For details see[4].

In this case, to build a sound call graph, we need call-by-signature resolution.

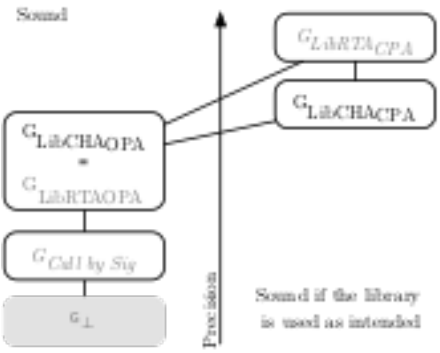
LibCG - the library private implementation

- A library's private implementation consists of all code that a library user *cannot* directly use. To determine this set two fundamentally different assumptions can be made:
  - open-package assumption (*OPA*)
  - closed-package assumption (*CPA*)

Under the open-package assumption (OPA), a library's private implementation consists of all methods and fields with private visibility. Under the closed-package assumption (CPA), the library's private implementation additionally includes (a) every code element (class, method or field) that has at most package visibility and (b) all protected and public fields- /methods of a package visible class, unless they are indirectly exposed to the library's user.

The distinction between the private and the public part becomes easier when Java 9+ modules or OSGi Bundles are used.

LibCG - algorithm comparison



In case of the open-package assumption, computing the call graph for a library using CHA effectively yields the same graph as using the RTA algorithm which is in case of applications with a single entry point generally more precise. The reason is, that under the open package assumption basically all classes can be instantiated by users of the library.

References

1. F. Tip and J. Palsberg; Scalable Propagation-Based Call Graph Construction Algorithms; OOPSLA 2000, ACM↩
2. Livshits, B., Sridharan, M., Smaragdakis, Y., Amaral, J. N., Møller, A., Lhoták, O., et al. (2015). In Defense of Soundiness: A Manifesto. Communications of the ACM, 58(2).↩
3. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs; Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, Mira Mezini, ISSTA 2019, ACM (to appear)↩
4. M. Reif, M. Eichberg, B. Hermann, J. Lerch and M. Mezini; Call Graph Construction for Java Libraries; FSE 2016; ACM↩