Application-Only Call Graph Construction

Karim Ali and Ondřej Lhoták

David R. Cheriton School of Computer Science, University of Waterloo

Abstract. Since call graphs are an essential starting point for all interprocedural analyses, many tools and frameworks have been developed to generate the call graph of a given program. The majority of these tools focus on generating the call graph of the whole program (i.e., both the application and the libraries that the application depends on). A popular compromise to the excessive cost of building a call graph for the whole program is to ignore all the effects of the library code and any calls the library makes back into the application. This results in potential unsoundness in the generated call graph and therefore in any analysis that uses it. In this paper, we present CGC, a tool that generates a sound call graph for the application part of a program without analyzing the code of the library.

1 Introduction

A call graph is a necessary prerequisite for most interprocedural analyses used in compilers, verification tools, and program understanding tools [19]. However, constructing a sound, precise call graph for even a small object-oriented program is difficult and expensive. For example, constructing the call graph of a Java "Hello, World!" program using SPARK [20] can take up to 30 seconds, and produces a call graph with 5,313 reachable methods and more than 23,000 edges. The key reason is dynamic dispatch: the target of a call depends on the runtime type of the receiver of the call. Because the receiver could have been created anywhere in the program, a sound algorithm must either analyze the whole program [1,6,17,21,34], or make very conservative assumptions about the receiver type (e.g., Class Hierarchy Analysis [9]). Additionally, due to the large sizes of common libraries, whole-program analysis of even trivial programs is expensive [7,27,28]. Practical programs generally have many library dependencies, and in many cases, the whole program may not even be available for static analysis. Our aim is to construct sound and precise call graphs for the application part of a program without analyzing the libraries that it depends on.[1]

Construction of partial call graphs is an often-requested feature in static analysis frameworks for Java. On the mailing list of the SOOT framework [34], which analyzes the whole program to construct a call graph, dozens of users have requested partial call graph construction [4]. One popular approach for generating

[1] In the rest of this paper, we will use the singular "library" to mean all of the libraries that a program depends on.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 688–712, 2012.

partial call graphs, used for example in the WALA framework [17], is to define an analysis scope of the classes to be analyzed. The analysis scope then represents the application part of the program. The effects of code outside this scope (i.e., the library) are ignored. As a consequence, the generated call graph lacks edges representing call-backs from the library to the application. Methods that should be reachable due to those call-back edges are ignored as well. Since this approach ignores the effects of the library code, any store or load operation in the library that involves an application object is ignored. Therefore, the points-to sets of the application objects will be incomplete, potentially causing even more call graph edges to be missing.

In contrast, we aim to produce a partial call graph that soundly overapproximates the set of targets of every call site in the analysis scope, and the set of reachable methods in the analysis scope. Our call graph uses a single summary node to represent all methods in the library. However, the analysis should be accurate for the application code. The goal of our work is to make less conservative assumptions about the library code, which is not analyzed, while still generating a precise and sound call graph for the application.

The essential observation behind our approach is that the division between an application and its library is not arbitrary. If the analysis scope could be any set of classes, then the call graph would necessarily be very imprecise. In particular, a sound analysis would have to assume that the unanalyzed code could call any non-private method and modify any non-private field in the analysis scope.[2]

A realistic yet very useful assumption is that the code of the library has been compiled without access to the code of the application. We refer to this as the *separate compilation assumption*. From this, we can deduce more specific restrictions on how the library can interact with the application, which we will explain in detail in Section 3. In particular, the library cannot call a method, access a field, or instantiate a class of the application if the library author does not know the name of the method, field, or class. It is theoretically possible to discover this information using reflection, and some special-purpose "libraries" such as JUnit [18] actually do so. We assume that such reflective poking into the internals of an application is rare in most general libraries.

In this paper, we evaluate the hypothesis that this assumption of separate compilation is sufficient to construct precise call graphs. We provide a prototype implementation for call graph construction, CGC, that uses a pointer analysis based on the separate compilation assumption. We evaluate soundness by comparing against the dynamic call graphs observed at run time by *J [11]. Since a dynamic call graph does not represent all possible paths of a program, it does not make sense to use it to evaluate the precision of a static call graph. Therefore, we evaluate precision by comparing against call graphs constructed by whole-program analysis (using both the SPARK [20] and DOOP [6] call graph construction systems). We also compare the performance of our prototype

[2] Some field modifications could theoretically be ruled out if an escape analysis determined that some objects are not reachable through the heap from any objects available to the unanalyzed code.

partial call graph construction system with whole-program call graph construction. However, our prototype implementation is optimized for adaptability and for producing call graphs comparable to those of other frameworks in terms of soundness and precision and not specifically for performance. Given the positive research results from our prototype, an obvious next implementation step would be to optimize and embed the analysis within popular analysis frameworks such as Spark, Doop, and Wala.

In summary, this paper makes the following contributions:

- It identifies the *separate compilation assumption* as key to partial call graph construction, and specifies the assumptions about the effects of library code that can be derived from it.
- It presents our prototype implementation of a partial call graph construction system, Cgc.
- It empirically shows that the separate compilation assumption is sufficient for constructing precise and sound application-only call graphs.

2 Background

2.1 Call Graph Construction

The targets of method calls in object-oriented languages are determined through dynamic dispatch. Therefore, a precise call graph construction technique requires a combination of two inter-related analyses: it must determine the targets of calls, and also determine the run-time types of referenced objects (i.e., points-to analysis). In Figure 1, both analyses are divided further and their dependencies are made more explicit. Determining the targets of calls is divided into two relations: *reachable methods* and *call edges*. On the other hand, the points-to analysis is defined by two other relations: *points-to sets* and *points-to constraints*.

The main goal of a call graph construction algorithm is to derive the call edges relation. A call edge connects a call site, which is an instruction in some

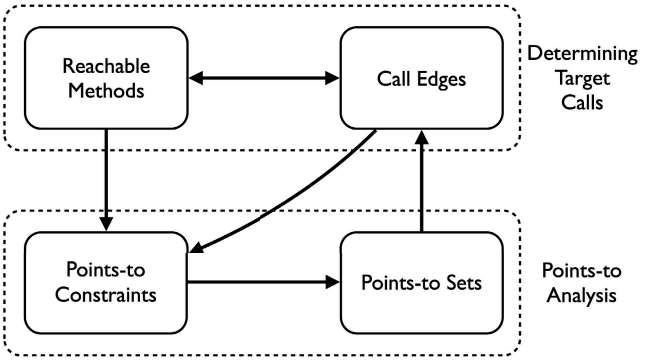

Fig. 1. Inter-dependent relations that make up a call graph construction algorithm

method, to a method that may be invoked from that call site. Figure 1 shows that the call edge relation depends on two relations: reachable methods and points-to sets. First, we are only interested in call sites that may actually be executed and are not dead code. A precise call graph construction algorithm therefore keeps track of the set of methods that are transitively reachable from the entry points of the program, e.g., its `main()` method. Second, the target of a given call depends on the run-time type of the receiver of the call. A precise call graph construction algorithm therefore computes the points-to sets abstracting the objects that each variable could point to. There are two common methods of abstraction to represent objects: either by their allocation site (from which their run-time type can be deduced) or by their run-time type. Thus, the points-to set of a variable at a call site indicates the run-time types of the receiver of that call.

Points-to sets are computed by finding the least fixed-point solution of a system of subset constraints that model all possible assignments between variables in the program. Thus, an abstract object "flows" from its allocation site into the points-to sets of all variables to which it could be assigned. Eventually, the abstract object reaches all of the call sites at which its methods could be called. The dependency of the points-to set relation on the call edges relation is illustrated in Figure 1. The calculation of the points-to sets is subject to the points-to constraints. The points-to constraints model intra-procedural assignments between variables due to explicit instructions within methods. They also model inter-procedural assignments due to parameter passing and returns from methods. Since only intra-procedural assignments in reachable methods are considered, the set of points-to constraints depends on the set of reachable methods. The set of call edges is another dependency because it determines the inter-procedural assignments.

Finally, the set of reachable methods depends, of course, on the set of call edges. A method is reachable if any call edge leads to it. A precise call graph construction algorithm computes these four inter-dependent relations concurrently until it reaches a mutual least fixed point. This is often called *on-the-fly* call graph construction.

2.2 Partial Call Graph Construction

If a sound call graph is to be constructed without analyzing the whole program, conservative assumptions must be made for all four of the inter-dependent relations in Figure 1. A sound analysis must assume that any unanalyzed methods could do "anything": they could arbitrarily call other methods and assign arbitrary values to fields. Due to the dependencies between the four relations, imprecision in any one relation can quickly pollute the others.

Our call graph construction algorithm computes precise information for the application part of the call graph, but uses summary nodes for information about the library. It assumes that all library methods are reachable, and uses a single summary "method" to represent them. Calls from application methods to library methods and vice versa are represented as call edges to or from the library

```
public class Main {
   public static void main(String[] args) {
       MyHashMap<String,String> myHashMap = new MyHashMap<String,String>();
       System.out.println(myHashMap);
   }
}

public class MyHashMap<K,V> extends HashMap<K,V> {
   public void clear() { }
   public int size() { return 0; }
   public String toString() { return "MyHashMap"; }
}
```

Fig. 2. A sample Java program that will be used for demonstration

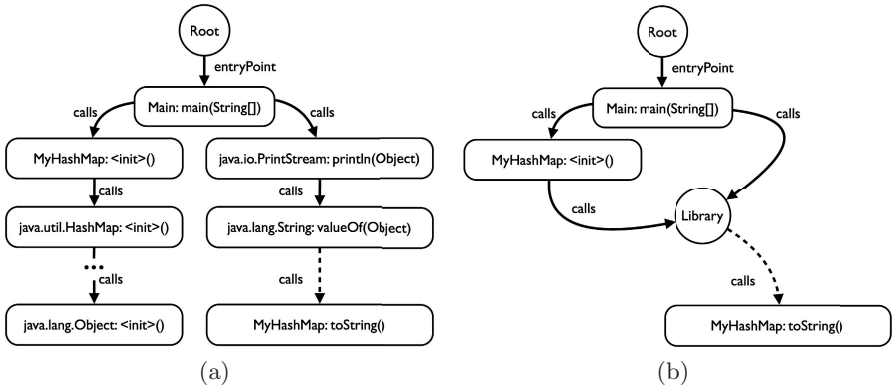

Fig. 3. Two branches from the call graph for the sample program in Figure 2 as generated by (a) SPARK and (b) CGC. The dashed line represents a call from a library method to an application method (i.e., a library call-back).

summary node. A call edge is created for each possible call between application methods, but no edges are created to represent calls within the library. It is implicitly assumed that any library method could call any other library method. Similarly, a single summary points-to set is used to represent the points-to sets of all variables within the library. Intra-library pointer flow, however, is not tracked precisely.

Figure 2 shows a sample Java program that we will use to demonstrate our analysis. Figure 3 compares two branches from the call graphs generated for this sample program by (a) SPARK and (b) CGC. We computed both branches by following the paths from the entry point method of the call graph, `Main.main()`, using the `CallGraphView` tool that comes with PROBE [19]. In Figure 3(a), we can see that the first branch shows all of the call edges from the method `MyHashMap.<init>()` all the way up to `java.lang.Object.<init>()`. Moreover,

the second branch in the call graph shows the call edges between library methods, e.g., the call edge from the method `java.io.PrintStream.println(Object)` to the method `java.lang.String.valueOf(Object)`. The target of that edge calls back to the application method `MyHashMap.toString()`.

On the other hand, Figure 3(b) shows how CGC represents the same branches. All of the edges beyond the predefined point of interest of the user (i.e., the application classes `MyHashMap` and `Main`) are considered as part of the library and are not explicitly represented in the graph. Therefore, all library methods are reduced to one library node that can have edges to and from application methods. The figure also shows that even for such a small sample program, the graph generated by CGC is easier to visualize and inspect. This will give the users a more focused view of the classes they are interested in, similar to what they would do during manual code inspection [15]. Having a more focused and precise view of the call graph should not ignore any of the potential call edges. Ignoring the library call-back edge in Figure 3(a), for example, will render the generated call graph unsound. Thus, it is crucial to precisely define, based on the separate compilation assumption, how the library summary node interacts with the application methods in the call graph.

3 The Separate Compilation Assumption

The input to our call graph construction algorithm is a set of Java classes designated as the *application classes*. The application classes may have dependencies on classes outside this set. We designate any class outside the set as a *library class*. We use the terms *application method* and *library method* to refer to the methods of application and library classes, respectively. The call graph construction algorithm analyzes the bytecode instructions of only the application classes. It does not analyze the instructions of any library class. However, the algorithm uses structural information (i.e., method signatures and field names) of each library class that is referenced in an application class, as well as its superclasses and superinterfaces. This is only a small subset of the full set of library classes. These referenced library classes are necessary to compile the application classes, and are readily available to the developer of the application.

The soundness of our approach depends on the **Separate Compilation Assumption**: all of the library classes are developed separately from the application classes. In particular, all of the library classes can be compiled in the absence of the application classes.

If a call graph construction algorithm does not analyze the whole program, it must make very conservative assumptions about the effects of the unanalyzed code. The separate compilation assumption makes these assumptions significantly less conservative. Without the separate compilation assumption, a sound algorithm would have to assume the following.

1. An unanalyzed class or interface may extend or implement any class or interface.

2. An unanalyzed method may instantiate an object of any type and call its constructor.
3. A local variable in an unanalyzed method may point to any object of any type consistent with its declared type.
4. A call site in an unanalyzed class may call any accessible method of any class.
5. An unanalyzed method may read or modify any accessible field of any object.
6. An unanalyzed method may read or modify any element of any array.
7. An unanalyzed method may cause the loading and static initialization (i.e., execution of the `<clinit>` method) of any class.
8. An unanalyzed method may throw any exception of any subtype of `java.lang.Throwable`.

The separate compilation assumption enables us to relieve these conservative assumptions in the following ways.

1. A library class cannot extend or implement an application class or interface. If it did, then the library class could not be compiled in the absence of the application classes.
2. An allocation site in a library method cannot instantiate an object whose run-time type is an application class. The run-time type of the object is specified in the allocation site, so compilation of the allocation site would require the presence of the application class.
   The only exception to this rule is reflective allocation sites in a library class (i.e., using `Class.forName()` and `Class.newInstance()`) that could possibly create an object of an application class. Since Java semantics do not prevent the library from doing this, our analysis should handle these reflective allocations without analyzing the library code. We assume that the library can reflectively instantiate objects of an application class if the library knows the name of this particular application class. In other words, if a string constant corresponding to the name of an application class flows to the library (possibly as an argument to a call to `Class.forName()` or `Class.newInstance()`), then the library can instantiate objects of that class.
3. Our algorithm computes a sound but non-trivial overapproximation of the abstract objects that local variables of library methods could point to. The library could create an object whose type is any library class. An object whose type is an application class can be instantiated only in an application method (except by reflection). In order for an object created in an application method to be pointed to by a local variable of a library method, an application class must pass the object to a library class in one of the following ways:
   (a) An application method may pass the object as an argument to a call of a library method. This also applies to the receiver, which is passed as the `this` parameter.
   (b) An application method called from a library method may return the object.

(c) The application code may store the object in a field that can be read by the library code.

(d) If the type of the object is a subtype of `java.lang.Throwable`, an application method may throw the object and a library method may catch it.

Thus, our algorithm computes a set, `LibraryPointsTo`, of the abstract objects allocated in the application that a local variable of a library method can point to. Implicitly, the library can point to objects whose type is any library class since these can be created in the library. Only the subset of application class objects that are passed into the library is included in `LibraryPointsTo`.

4. Two conditions are necessary in order for a call site in a library class to call a method m in an application class c.

   (a) The method m must be non-static and override a (possibly abstract) method of some library class. Each call site in Java bytecode specifies the class and method signature of the method to be called. Since the separate compilation assumption states that the library has no knowledge about the application, the specified class and method must be in the library, not the application. The Java resolution rules [22, Section 5.4.3] could change the specified class to one if its superclasses, but this must also be a library class due to the previous assumption. Therefore, the only way in which an application method could be invoked is if it is selected by dynamic dispatch. This requires the application method to be non-static and to override the method specified at the call site.

   (b) The receiver variable of the call site must point to an object of class c or a subclass of c such that calling m on that subtype resolves to the implementation in c. Therefore, an object of class c or of the appropriate subclass must be in the `LibraryPointsTo` set.

5. Similar conditions are necessary in order for a library method to read or modify a field f of an object o of class c created in the application code.

   (a) The field f must originally be declared in a library class (though c can be a subclass of that class, and could therefore be an application class). Each field access in Java bytecode specifies the class and the name of the field. This class must be a library class due to the separate compilation assumption. The Java resolution rules could change the specified class, but again only to one of its superclasses, which must also be a library class.

   (b) It must be possible for the local variable whose field is accessed to point to the object o. In other words, the `LibraryPointsTo` set must contain the abstract object representing o.

   (c) In the case of a field write, the object being stored into the field must also be pointed to by a local variable in the library. Therefore, its abstraction must be in the `LibraryPointsTo` set.

   The library can access any static field of a library class, and any field of an object that was instantiated in the library.

6. If the library has access to an array, it can access any element of it by its index. This is unlike an instance field, which is only accessible if its name is known to the library. However, the library is limited to accessing only the elements of arrays that it has a reference to (i.e., ones that are in the `LibraryPointsTo` set). In the case of a write, the object that is written into the array element must also be in the `LibraryPointsTo` set.
7. Due to the separate compilation assumption, the library does not contain any direct references to application classes. Thus the library cannot cause a static initializer of an application class to be executed except by using reflection. When determining which static initializers will execute, our algorithm includes those classes that are referenced from application methods reachable through the call graph, as well as classes that may be instantiated using reflection as discussed above in point 2.
8. The library can throw an exception either if it creates the exception object (in which case its type must be a library class) or if the exception object is created in an application class and passed into the library (in which case its abstraction appears in the `LibraryPointsTo` set). We conservatively assume that the library can catch any thrown exception. Consequently, we add the abstractions of all thrown exception objects to the `LibraryPointsTo` set.

In addition to these conditions, our algorithm strictly enforces the restrictions imposed by declared types.

- When the library calls an application method, the arguments passed in the call must be in the `LibraryPointsTo` set, and must also be compatible with the declared types of the corresponding parameters.
- When an application method calls a library method, the returned object must be in the `LibraryPointsTo` set and compatible with the declared return type of the library method.
- When the library modifies a field, the object it may write into the field must be compatible with the declared type of the field.
- When an application method catches an exception, only exceptions whose type is compatible with the declared type of the exception handler are propagated.
- When the `LibraryPointsTo` set is used to update the points-to set of an application local variable, only objects compatible with the declared type of the local variable are included.

4 CGC Overview

We have implemented a prototype of the application-only call graph construction approach that we call CGC, and have made it available at `http://plg.uwaterloo.ca/~karim/projects/cgc/`. For ease of modification and experimentation, CGC is implemented in Datalog.[3] CGC uses a pointer analysis that is

[3] Datalog is a logic-based language for (recursively) defining relations.

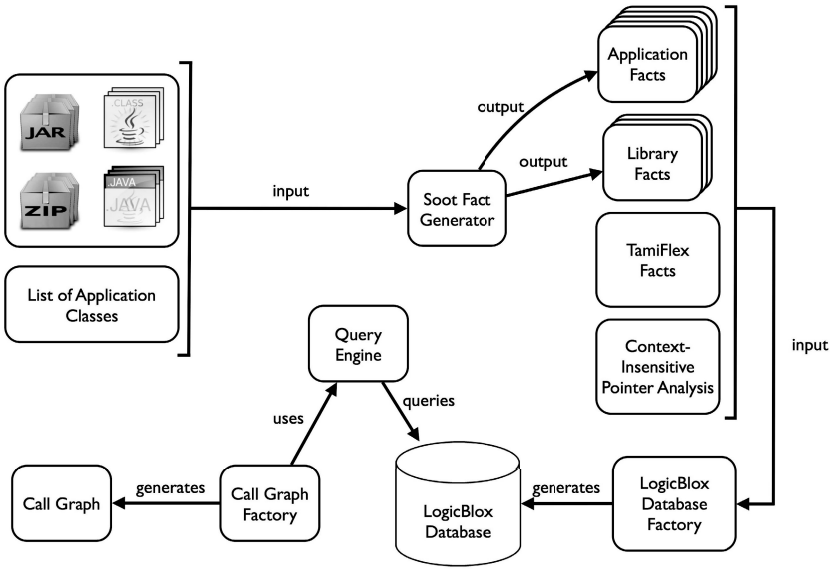

Fig. 4. An overview of the workflow of Cgc

based on the context-insensitive pointer analysis from the Doop framework [6]. However, the analysis is independent of Datalog, and could be transcribed into Java to be embedded into an analysis framework such as Soot or Wala. We have also implemented summary tools that summarize the call graphs generated by Spark and Doop. Each summary tool takes a list of application classes and the call graph as input. The output is a call graph with the library methods summarized into one node in the graph. Therefore, call graphs from Spark, Doop and Cgc can be compared. Additionally, Cgc can export the generated call graph as a GXL [16] document [4] or a directed DOT [31] graph file. The DOT graph can be visualized using Graphviz [13] or converted by Cgc to a PNG or a PS file that can be visualized using any document previewer.

4.1 Workflow

Figure 4 shows an overview of the work flow of Cgc. Like Doop, Cgc uses a fact generator based on Soot to preprocess the input code and generate the input facts to the Datalog program. The fact generator receives a collection of input files and a specification of the set of application classes. The rest of the classes are considered library classes. The fact generator then generates two sets of facts. The first set is for the application classes and contains all details about those classes: signatures for classes, methods, fields as well as facts about

[4] The DTD schema can be found at `http://plg.uwaterloo.ca/~karim/projects/cgc/schemas/callgraph.xml`

method bodies. The second set is dedicated to the library and contains the following: signatures for classes, methods, and fields in the library classes that are referenced in the application and their (transitive) superclasses and superinterfaces. We generate a third set of facts which holds information about reflection code in the application. This set is generated using TamiFlex [5], a tool suite that records actual uses of reflection during a run of a program, and summarizes them in a format suitable as input to a static analysis. CGC uses the output of TamiFlex to model calls to `java.lang.reflect.Method.invoke()`, and application class name string constants to model reflective class loading. We plan to add more support for other reflective application code in the future.

The three sets of facts along with the Datalog rules that define our pointer analysis are then used to initialize a LogicBlox [24] database. Once the database is created and the analysis completes, CGC queries it for information about the call graph entry points and the various types of call graph edges: application-to-application edges, application-to-library edges, and library-to-application call back edges. Finally, CGC uses those derived facts to generate the call graph for the given input program files and to save it as a GXL document.

4.2 Implementation

We will now outline the main parts of the CGC analysis implementation, listing the most important relations that are generated.

Object Abstraction. For precision, CGC uses a separate abstract object for each allocation site in the application classes. The abstract object represents all objects allocated at the site; it has an associated run-time type.

CGC must use a coarser abstraction to represent objects allocated in the library. CGC first computes the set L of all library classes and interfaces referenced in the application and their transitive superclasses and superinterfaces. It then creates one abstract object for each class in L.

The meaning of an abstract object in L is subtle. The abstraction must represent all objects created in the library, of any type, but L is limited to those types referenced in the application. Therefore, each abstract object $c \in L$ represents all concrete objects created in the library such that if the actual run-time type of the object is c', then c is the closest supertype of c' that is in L. In other words, each concrete object is represented by its closest supertype that is referenced in the application. From the point of view of analyzing the application, an object of type c' created in the library is treated as if its type were c. If the application accesses a field of the object, it must be a field that was already declared in c or one of its superclasses. Accessing a field declared only in c' would require the application to reference class c'. The situation is different for resolving a call to a library method as the analysis just models all library methods as a single node. Therefore, in the case of a call site in the application, the analysis does not need to determine which library method of c' or all its superclasses will be invoked.

Abstract objects allocated in the application always have a concrete class as their run-time type. An important but subtle detail is that the analysis must include abstract objects even for library classes that are declared abstract. This is because the actual run-time type of the concrete object could be a subtype of the abstract class, and not referenced by the application.

Points-to Sets. CGC uses the relations `VarPointsTo` and `StaticFieldPointsTo` to model the points-to sets of local variables and static fields within the application code, respectively. Library code cannot directly read object references from these sets. The relations `InstanceFieldPointsTo` and `ArrayIndexPointsTo` model the points-to sets of the fields of each abstract object and, in the case of an array, its array elements. The analysis distinguishes individual fields, but does not distinguish different elements of the same array.

We define a new relation, `LibraryPointsTo`, which models the set of abstract objects that the library may reference. This set is initialized with all of the abstract objects created in the library. In addition, the analysis adds abstract objects that are passed into a library method, returned to a library method from an application method, or stored in a static field of a library method. Finally, the analysis adds abstract objects that the library may read out of instance fields according to the conditions described in Section 3.

The analysis also includes rules to update `InstanceFieldPointsTo` and `ArrayIndexPointsTo` in order to model the instance field and array element writes that may occur within the library.

In addition to objects, the library can also instantiate arrays. The analysis adds to `LibraryPointsTo` an abstract array of type `T[]` whenever the application calls a library method whose return type is `T[]`, or the library calls an application method that takes `T[]` as a parameter.

Points-to Subset Constraints. CGC defines a relation called `Assign` that represents subset constraints between the points-to sets of local variables in the application code. This relation models assignment statements in reachable methods, and parameter passing and return at each method call edge. CGC defines two additional relations, `AssignToLibrary` and `AssignLibraryTo`, for assignments crossing the boundary between the application and the library. This includes parameter passing and return, as well as reading from and writing to static library fields within the application code.

For precision, we have found that it is very important to enforce declared types at the boundary between the application and the library. Since the input to CGC is Java bytecode, which is typed, there are few assignments (both intra-procedural and inter-procedural) within the application where explicit type checks are necessary (except for explicit casts in the bytecode). However, because the `LibraryPointsTo` set represents all references within the library, it does not have a declared type. Thus, at every assignment out of the library into an application local variable, instance/static field, or array element, CGC checks that the abstract objects respect the declared type of the destination.

Call Graph Edges. CGC defines the `ApplicationCallGraphEdge` relation to model calls within application methods. Additionally, two special relations, `LibraryCallGraphEdge` and `LibraryCallBackEdge` are defined to represent calls into and back out of the library. For call sites in the application, the points-to set of the receiver variable is used to resolve the dynamic dispatch and determine which methods may be called. Constructing the `LibraryCallBackEdge` set is more interesting since CGC knows nothing about the call site within the library. Following the conditions defined in Section 3, CGC uses the `LibraryPointsTo` set as an overapproximation of the possible receiver objects. CGC considers the signatures of all application methods that override a library method as possible targets for a `LibraryCallBackEdge`.

CGC computes the `Reachable` set of all methods that are transitively reachable through the call edges. The set contains application methods only.

4.3 Special Handling of `java.lang.Object`

Every constructor calls the constructors of its transitive superclasses. Therefore, every object ever created flows to the constructor of `java.lang.Object` and would be accessible to the library. However, this particular constructor is empty; it cannot leak a reference to other library code. The analysis therefore makes a special exception so that objects passed to this constructor are not added to `LibraryPointsTo`. The same exception is made for all other methods of `java.lang.Object` except `toString()`. We have determined by manual inspection that these methods do not leak object references to the library. Since the `java.lang.Object.clone()` method returns a copy if its receiver, we model it as follows: at any call site of the form `a = b.clone()`, all references pointed to by `b` flow to `a`.

5 Experiments

We evaluate CGC by comparing its precision and performance to that of SPARK and DOOP on two benchmark suites. We analyzed both the DaCapo benchmark programs, v.2006-10-MR2 [3], and the SPEC JVM 98 benchmark programs [29] with JDK 1.4 (jre1.4.2.11) which is larger than JDK 1.3 used by Lhoták and Hendren [20] and similar to JDK 1.4 used by Bravenboer and Smaragdakis [6]. We also evaluate the soundness of the call graphs generated by CGC by comparing them to the dynamic call graphs recorded at run time using the *J tool [11]. We ran all of the experiments on a machine with four dual-core AMD Opteron 2.6 GHz CPUs (runnning in 64-bit mode) and 16 GB of RAM. We exclude the benchmarks fop and eclipse from our evaluation because they do not include all code that they reference, so we were unable to analyze them with SPARK and DOOP. We exclude the benchmark jython because it heavily uses sophisticated forms of reflection, making any static analysis impractical.

5.1 Preliminaries and Experimental Setup

Since we are comparing the generated graph from the three different tools, CGC, SPARK and DOOP, we have to run them with similar settings so that the generated graphs are comparable. Therefore, there is a common properties file that holds the values for the input files, list of application classes, the benchmark to run and the name of the main class to be used across the three tools. The main class is an application class whose `main()` method is considered the entry point of the call graph.

Each experiment run is executed by a bash shell script that takes this properties file as an input. The script then runs the three tools consecutively and collects the results. For each benchmark program, the script records some statistics about the elapsed time for each tool to finish execution and the number of the various types of call graph edges generated. The script also reads in the dynamic call graph for the corresponding benchmark program to be used in evaluating the soundness of all three static analysis tools. The dynamic call graphs are generated using *J [11], a tool which attaches to the Java VM and records all method calls that occur in an actual run of the given benchmark program. The bash script also produces GXL files for the generated call graphs for CGC, SPARK, DOOP, and the dynamic call graphs. The script then computes and records the differences between them. This is done by generating four difference graphs: CGC-SPARK, SPARK-CGC, CGC-DOOP, and DOOP-CGC. A difference graph A-B is a graph that contains all of the edges that are in A and not in B.

The *J tool records a call from method a to method b if the method b ever executes on a thread during the execution of method a on the same thread. There are several situations in which the Java VM triggers such a method execution that are not due to method calls. We remove these edges from the *J call graph. First, we remove edges to the methods `java.lang.ClassLoader.loadClassInternal()` and `java.lang.ClassLoader.checkPackageAccess()`, which are called internally by the Java VM. We also remove edges to static initializers (`<clinit>`), because CGC treats static initializers as entry points, whereas *J treats them as methods that are called. Nevertheless, CGC considers static initializers as reachable methods and analyzes their effect, including any method calls that they make.

In addition, while converting SPARK's call graphs to CGC's format for comparison, we convert `NewInstanceEdges` to `LibraryCallBackEdges`. In SPARK, `NewInstanceEdges` represent implicit calls to constructors from the method `java.lang.Class.newInstance()`. In a SPARK call graph, the source of those edges is the calling site of the method `java.lang.Class.newInstance()`. On the other hand, those edges are `LibraryCallBackEdges` in CGC. Therefore, this conversion allows us to do a fair comparison between SPARK and CGC by resolving any inconsistencies in the way both model `NewInstanceEdges`.

In the following subsections, we evaluate and compare the soundness, precision, and size of the call graphs generated by the static tools, as well as the performance of the tools.

Table 1. Comparing the soundness of CGC, DOOP, and SPARK with respect to `ApplicationCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	3066	3733	482	1505	565	435	1894	2543	39	36	520	2384	5	317
DYNAMIC-CGC	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	0	0	0	325	244	193	153	250	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	59	0	155	0	59	0	0	0	0	0	0

Table 2. Comparing the soundness of CGC, DOOP, and SPARK with respect to `LibraryCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	372	475	168	119	148	99	157	325	4	17	76	148	5	13
DYNAMIC-CGC	0	0	0	0	0	1	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	0	0	0	5	53	45	43	42	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	1	0	27	0	9	0	0	0	0	0	0

Table 3. Comparing the soundness of CGC, DOOP, and SPARK with respect to `LibraryCallBackEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
DYNAMIC	11	49	7	3	13	5	36	85	0	1	0	6	3	0
DYNAMIC-CGC	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DYNAMIC-DOOP	3	0	0	1	6	3	29	78	0	0	0	0	0	0
DYNAMIC-SPARK	0	0	0	1	4	3	3	28	0	0	0	0	0	0

5.2 Call Graph Soundness

The separate compilation assumption makes it easier to reason soundly about library code than whole-program approaches. Whereas a whole-program analysis must soundly model all details of the library, CGC needs only to soundly handle its interface. We evaluate the soundness of CGC, DOOP, and SPARK by counting the call graph edges that are present in the dynamic call graph, but missing from the call graphs generated by each static tool. These counts are shown in the lines DYNAMIC-CGC, DYNAMIC-DOOP, and DYNAMIC-SPARK in Tables 1, 2 and 3.

This comparison shows that the call graphs generated by CGC soundly include all of the call edges that were dynamically observed at run time by *J, except for one single call edge from the application to the library in the lusearch benchmark (see Table 2). The dynamic call graph for lusearch has a `LibraryCallGraphEdge` from the method `org.apache.lucene.index.FieldInfos.fieldName()` to the constructor of `java.lang.NullPointerException`. There is no statement in this method that constructs this object, but the method tries to access the field of a `null` object. As a result, the Java VM creates a `java.lang.NullPointerException` and executes its constructor. The exception is caught elsewhere in the benchmark. This type of unsoundness can be resolved by improving our analysis to model the behavior of the Java VM by checking for when an application attempts to dereference `null`. This same unsoundness is also found in both DOOP and SPARK.

Table 4. Comparing the precision of CGC with respect to `ApplicationCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	6276	14430	1879	9015	923	1998	4594	11628	40	47	653	8194	6	400
DOOP	6293	12433	1811	6245	449	1507	3956	8911	40	47	646	8188	6	400
SPARK	6299	13419	6732	7792	1412	2724	6893	13020	40	47	646	8519	6	400
CGC-DOOP	13	1997	68	2445	230	298	485	2654	0	0	7	6	0	0
CGC-SPARK	2	1829	15	1394	0	250	179	1356	0	0	7	0	0	0

Table 5. Comparing the precision of CGC with respect to `LibraryCallGraphEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	649	874	571	982	243	345	431	1149	13	24	107	313	7	34
DOOP	649	841	529	787	152	251	348	818	13	24	98	313	6	34
SPARK	661	885	1060	869	336	392	797	1055	13	23	97	317	6	34
CGC-DOOP	0	33	42	190	38	50	40	289	0	0	9	0	1	0
CGC-SPARK	0	10	1	139	0	29	3	171	0	1	10	2	1	0

Table 6. Comparing the precision of CGC with respect to `LibraryCallBackEdges`

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	47	190	95	663	27	48	114	696	0	2	10	25	12	1
DOOP	39	84	55	24	15	26	37	61	0	2	0	23	8	0
SPARK	73	223	490	69	132	146	135	464	10	12	10	41	64	10
CGC-DOOP	5	106	40	638	6	19	48	557	0	0	10	2	4	1
CGC-SPARK	0	19	8	616	1	5	25	307	0	0	10	0	1	1

5.3 Call Graph Precision

In order for a call graph to be useful, it must also be precise in addition to being sound. We now compare the precision of call graphs generated by CGC to those generated by DOOP and SPARK. We would expect CGC to be at least as imprecise as DOOP and SPARK, since it makes conservative assumptions about the library code instead of precisely analyzing it. Since we found some dynamic call edges that were missing from the call graphs generated by both DOOP and SPARK (and one edge from CGC), we first correct this unsoundness by adding the missing dynamic call edges to the static call graphs. This enables us to compare the precision of the static call graphs by counting only spurious call edges, and to avoid confounding due to differences in soundness. In Tables 4, 5 and 6, the quantity CGC-DOOP represents the number of edges in the call graph generated by CGC that are missing in the call graph generated by DOOP, and are also not present in the dynamic call graph. The quantity CGC-SPARK is defined similarly. We say that CGC is *precise* when the call graph that it generates is identical to that generated through whole program analysis by DOOP or SPARK.

Application Call Graph Edges. Table 4 shows that CGC generates precise call graphs with respect to `ApplicationCallGraphEdges` when compared to both DOOP and SPARK for compress, db, jess, and raytrace. Additionally, CGC generates precise call graphs for luindex and javac when compared to SPARK. For all benchmark programs, CGC generates a median of 41 extra `ApplicationCallGraphEdges` (min: 0, max: 2656, median: 40.5) when compared to DOOP and a

median of 5 extra `ApplicationCallGraphEdges` (min: 0, max: 1829, median: 4.5) when compared to SPARK. Both medians are negligible as they represent 2.42% and 0.13% of the median number of `ApplicationCallGraphEdges` generated by DOOP and SPARK respectively.

Library Call Graph Edges. Table 5 shows that CGC generates precise call graphs with respect to `LibraryCallGraphEdges` when compared to DOOP for antlr, compress, db, javac, and raytrace. On the other hand, CGC generates precise call graphs when compared to SPARK for antlr, luindex, compress, and raytrace. Across all benchmark programs, CGC generates a median of 21 extra `LibraryCallGraphEdges` (min: 0, max: 288, median: 21) when compared to DOOP as opposed to a median of 2 extra `LibraryCallGraphEdges` (min: 0, max: 171, median: 1.5) when compared to SPARK.

Library Call Back Edges. Table 6 shows that CGC generates precise call graphs with respect to `LibraryCallBackEdges` when compared to DOOP and SPARK for compress and db. Additionally, CGC generates precise call graphs for antlr and javac when compared to SPARK. In general, CGC generates a median of 8 extra `LibraryCallBackEdges` (min: 0, max: 638, median: 8) when compared to DOOP and a median of 3 extra `LibraryCallBackEdges` (min: 0, max: 616, median: 7.5) when compared to SPARK. The former represents 72.9% as opposed to only 2.53% for the latter, of the median number of `LibraryCallBackEdges` generated by DOOP and SPARK respectively. In other words, the majority of `LibraryCallBackEdges` generated by CGC are spurious compared to DOOP. These extra edges are the root cause of the small amounts of imprecision that we observed in the `ApplicationCallGraphEdges` and `LibraryCallGraphEdges`.

We further investigate the specific causes of the extra `LibraryCallBackEdges` in the CGC call graph. In Tables 7 and 8, we categorize these edges by the name of the application method that is being called from the library. In particular, we are interested to know whether the library calls a wide variety of application methods, or whether the imprecision is limited to a small number of well-known methods, which could perhaps be handled more precisely on an individual basis.

Table 7 shows that the most frequent extra `LibraryCallBackEdges` in CGC when compared to DOOP target the commonly overridden methods (in descending order): `clone`, `<init>`, `toString`, `equals`, `remove`, and `hashCode`. The number of extra `LibraryCallBackEdges` generated by CGC compared to SPARK is smaller than that compared to DOOP. Table 8 shows that the most frequent extra `LibraryCallBackEdges` in CGC compared to SPARK target the methods: `<init>`, `finalize`, `run`, `close`, `write`, and `remove`.

The constructor `<init>` ranks highly in hsqldb, pmd, and xalan. This is because these benchmarks use class constants. CGC conservatively assumes that if a class constant is created (using `java.lang.Class.forName()`), an object of that type might also be instantiated and its constructor called.

The benchmark programs hsqldb and xalan have the highest frequency of imprecise `LibraryCallBackEdges`. In the case of hsqldb, most of those imprecise

Table 7. Frequencies of extra `LibraryCallBackEdges` in CGC when compared to DOOP (CGC-DOOP). *Other* methods include all methods that are encountered only in one benchmark program.

Method	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace	Total
`clone`	4	50	25		3	4		11				1			98
`<init>`				20			19	33							72
`toString`		2	2	7	2	2	2								17
`equals`	1		1	6		4	2	2							16
`remove`		12					2								14
`hashCode`			4	2		4	2	1							13
`write`				2				9							11
`run`				4		1		2						1	8
`close`				5				3							8
`printStackTrace`							5	2							7
`next`		4		1											5
`getAttributes`				1				3							4
`getType`				1				3							4
`read`				2								1			3
`clearParameters`				1				2							3
`accept`					1	1									2
`previous`		1		1											2
Other	0	37	8	585	0	3	16	486	0	0	10	0	4	0	1149
Total	5	106	40	638	6	19	48	557	0	0	10	2	4	1	1436

Table 8. Frequencies of extra `LibraryCallBackEdges` in CGC when compared to SPARK (CGC-SPARK). *Other* methods include all methods that are encountered only in one benchmark program.

Method	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace	Total
`<init>`				19			19	48							86
`finalize`				3	1	4		3							11
`run`				4		1		2						1	8
`close`				5				3							8
`write`				2				6							8
`remove`		5					2								7
`getType`				1				2							3
`clearParameters`				1				2							3
`previous`		1		1											2
Other	0	13	8	580	0	0	4	241	0	0	10	0	1	0	857
Total	0	19	8	616	1	5	25	307	0	0	10	0	1	1	993

`LibraryCallBackEdges` are to methods of classes in the package `org.hsqldb.jdbc` (DOOP = 560, SPARK = 555). In xalan, most of the imprecise `LibraryCallBackEdges` are to methods of classes in the packages `org.apache.xalan.*` (DOOP = 276, SPARK = 184) and `org.apache.xml.*` (DOOP = 263, SPARK = 112). Thus, in both of these benchmarks, the high imprecision is due to the fact that each benchmark contains its own implementation of a large subsystem (JDBC and XML) whose interface is defined in the library.

5.4 Call Graph Size

As we mentioned earlier, call graphs are a key prerequisite to all interprocedural analyses. Therefore, any change in the size of the call graph will affect the performance of the analyses that use it as input. Since CGC overapproximates the generated call graph, we evaluate the size of the generated call graph

Table 9. Comparing the size of the call graph generated by CGC, DOOP and SPARK for the same input program

	antlr	bloat	chart	hsqldb	luindex	lusearch	pmd	xalan	compress	db	jack	javac	jess	raytrace
CGC	6,972	15,494	2,545	10,660	1,193	2,391	5,139	13,473	53	73	770	8,532	25	435
DOOP	6,981	13,358	2,395	7,056	616	1,784	4,341	9,790	53	73	744	8,524	20	434
SPARK	7,033	14,527	8,282	8,730	1,880	3,262	7,825	14,539	63	82	753	8,877	76	444
CGC/DOOP	0.99	1.16	1.06	1.51	1.94	1.34	1.18	1.38	1	1	1.03	1	1.25	1
CGC/SPARK	0.99	1.07	0.31	1.22	0.63	0.73	0.66	0.93	0.84	0.89	1.02	0.96	0.33	0.98

(in terms of total number of edges) compared to those of DOOP and SPARK. Table 9 shows that CGC generates call graphs of equal or smaller size to DOOP and SPARK for the benchmark programs antlr, chart, compress, db, jack, javac, and raytrace. Additionaly, CGC generates call graphs of smaller size than SPARK for the benchmark programs bloat, luindex, lusearch, pmd, xalan, and jess.

It is counterintuitive that the call graphs generated by CGC are smaller than those generated by SPARK, which analyzes the whole program precisely. This result is primarily due to imprecisions in SPARK. To model objects created by the Java VM or by the Java standard library using reflection, SPARK uses special abstract objects whose type is not known (i.e., any subtype of `java.lang.Object`). SPARK does not filter these objects when enforcing declared types, so these objects pass freely through casts and pollute many points-to sets in the program. This affects the precision of the points-to sets of the method call receivers and leads to many imprecise call graph edges in SPARK. The extent of this imprecision spark was a surprise to the second author, who is also the author of SPARK. In response to this observation, we plan to improve the precision of SPARK by redesigning the mechanism that it uses to model these objects.

5.5 Analysis Performance

We evaluate the performance gain of not analyzing the method bodies of the library code. There are two major aspects that can measure the performance of an analysis: execution time, and the disk footprint of the analysis database. We define *execution time* to be the time taken by the analysis to finish computing the points-to sets and constructing the call graph for the input program. We measure this time by using the `time` Linux command [23] that measures the time taken by a given program command to finish execution. Based on the numbers shown in Figure 5, we can deduce that CGC is approximately 3.5x faster than DOOP (min: 0.86x, max: 18.42x, median: 3.45x), and almost 7x faster than SPARK (min:0.93x, max: 42.7x, median: 6.56x).

Figure 6 shows that CGC achieves this performance gain in execution time while using a database of facts approximately 7x smaller in size than DOOP (min: 2.19x, max: 31.75x, median: 6.83x). We measure the size of the database of facts used by CGC by calculating the disk footprint of the LogicBlox database files after the analysis completes. We calculate the size of the database of facts for DOOP similarly and compare it against its CGC counterpart. In all benchmarks, DOOP has a larger disk footprint as it analyzes the method bodies for all the

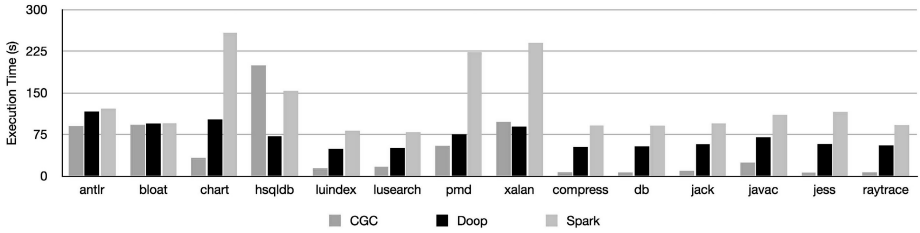

Fig. 5. Comparing the time taken by the analysis in each of Cgc, Doop and Spark to generate the call graph for each program from the DaCapo and SPEC JVM benchmarks. This only includes the time taken to compute the points-to sets as well to construct the call graph.

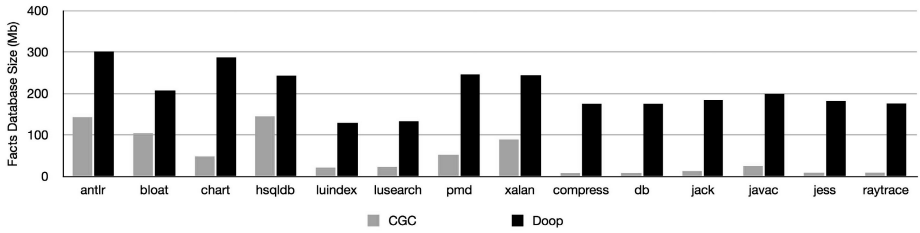

Fig. 6. The size of Cgc's facts database compared to Doop's

library classes while Cgc only analyzes the method signatures for the classes referenced in the application (and their transitive superclasses and superinterfaces). It is difficult to report similar statistics for Spark as it uses a different model to represent the facts its analysis uses to construct the call graph for an input program.

6 Related Work

Early work on call graph construction used only simple approximations of run-time types to model dynamic dispatch. Dean et al. [9] formulate *class hierarchy analysis* (CHA), which does not propagate object types. CHA uses only the subclass hierarchy to determine method targets. Bacon and Sweeney [2] define *rapid type analysis* (RTA), a refinement of CHA that considers as possible receivers only classes that are instantiated in the reachable part of the program. Sundaresan et al. [30] introduce an even more precise approach, *variable type analysis* (VTA). Like points-to analysis, it generates subset constraints and propagates points-to sets to approximate the run-time types of receivers.

Tip and Palsberg [32] provide a scalable propagation-based call graph construction algorithm. Separate object sets for methods and fields are used to approximate the run-time values of expressions. The algorithm is capable of

analyzing incomplete applications by associating a single set of objects, S_E, with the outside world (i.e., the library). The algorithm conservatively assumes that the library calls back any application method that overrides a library method. The set S_E is then used to determine the set of methods that the external code can invoke by the dynamic dispatch mechanism. A separate set of objects S_C is associated with an external (i.e., library) class if the objects passed to the methods in class C interact with other external classes in limited ways. An example of this case is the class `java.util.Vector`. This step requires the analysis of the external classes to model the separate propagation sets. In addition, this technique varies based on the library dependencies of the input program.

The previous algorithm was later used by Tip et al. [33] to implement Jax, an application extractor for Java. The external object set S_E inspired the `Library-PointsTo` relation in our algorithm. Expanding on the initial idea of analyzing incomplete applications, we formulated the separate compilation assumption, and worked out in detail the specific assumptions flowing from it. We also derived a set of constraints from those assumptions. In addition, we have empirically analyzed the precision and soundness of the partial call graphs generated compared to call graphs generated by analyzing the whole program.

Grothoff et al. [14] present Kacheck/J, a tool that is capable of identifying accidental leaks of sensitive objects. Kacheck/J achieves that by inferring the *confinement* property [35,36,38] for Java classes. A Java class is considered *confined* when objects of its type are encapsulated in its defining package. The analysis needs only to analyze the defining package of the given Java class to infer its confinement property. That is similar to the way CGC needs only to analyze the application classes to construct its call graph. Although the set of application classes can be thought of as one defining package, determining the confinement property for the classes in this package is not enough to construct the call graph. Constructing the call graph would still require the points-to set information. As part of the confinement analysis, Kacheck/J identifies *anonymous* methods which are guaranteed not to leak a reference to their receiver. A similar notion and analysis could be used in CGC to identify library methods that do not retain permanent references to their arguments, in order to improve the precision of the `LibraryPointsTo` set. In addition to the analysis that infers the confinement property, there is a large body of work on type systems that enforce encapsulation by restricting reference aliasing [8,10,12,25].

Our work is also related to the work of Zhang and Ryder [37]. They provide a fine-tuned data reachability algorithm to resolve library call-backs, $V^a - DataReach^{ft}$. Their algorithm also distinguishes library code from application code in the formulation of the constraints. The fundamental difference is that the purpose of their algorithm is to construct a more precise call graph than a whole-program analysis by analyzing the library more thoroughly. Each call into the library is treated as an isolated context. In contrast, our aim is not to analyze the library at all, and generate a possibly less precise but sound call graph. In CGC, we make up for not analyzing the library by enforcing the restrictions that follow from the separate compilation assumption.

Rountev et al. [26] present a general approach for adapting whole-program class analyses to operate on program fragments. Whereas our aim is to analyze the application without the library, they soundly analyze the library without the application. The authors create placeholders to serve as representatives for and simulate potential effects of unknown code. The fragment class analysis then adds the placeholders to the input classes and treats the result as a complete program which can be analyzed using whole-program class analyses. Although Rountev et al. proved that their fragment class analysis is correct, the precision of the fragment analysis is variable as it significantly depends on the underlying whole-program analysis. Therefore, a CHA-dependent fragment analysis is less precise than an RTA-dependent fragment analysis. In contrast, the precision of the call graph construction in CGC depends only on the separate compilation assumption and its consequences.

DOOP [6] implements various pointer analysis algorithms for Java programs, all defined declaratively in Datalog. DOOP constructs the call graph on-the-fly while computing the points-to sets. However, there is no way to exclude some classes (e.g., the library classes) from the analysis as DOOP analyzes all of the input program. The pointer analysis in CGC is an extended version of DOOP's context-insensitive pointer analysis. The major difference is the introduction of the library summary relation and the necessary associated Datalog rules.

Lhoták and Hendren introduced SPARK [20], a flexible framework for experimenting with points-to analyses for Java programs. SPARK provides a SOOT transformation that constructs the call graph of the input program on-the-fly while calculating the points-to sets. It is possible to setup up the SOOT classes so that SPARK ignores some of the input classes. However, this is usually achieved through setting the *allow_phantom_refs* option to *true* which means that the ignored class will be completely discarded. Therefore, crucial information about the signatures of the classes, methods, and fields is lost which would render the generated call graph unsound. Thus, SPARK does not support excluding some of the input classes (e.g., the library classes) from the process of constructing the call graph despite the demand in the SOOT community [4].

WALA [17] is a static analysis library from IBM Research designed to support various pointer analysis configurations. WALA is capable of building a call graph for a program by performing pointer analysis with on-the-fly call graph construction to resolve the targets of dynamic dispatch calls. WALA provides the option of excluding some classes or packages while constructing the call graph. In fact, WALA excludes all the user-interface related packages from the Java runtime library by default when constructing a call graph. When this option is set, WALA limits the scope of its pointer analysis to the set of included classes. This ignores any effects the excluded classes might have on the calculation of the points-to sets. Therefore, the generated call graphs may be unsound and/or imprecise. Moreover, it is impossible to exclude crucial classes (e.g., `java.lang.Object`) from the analysis as this will cause an exception to be thrown. We plan to empirically compare the precision, soundness, and speed of our call graph construction algorithm with WALA in future work.

7 Conclusions and Future Work

We have proposed CGC, a tool that generates an application-only call graph with the library code represented as one summary node. The main contributions are: (1) the separate compilation assumption that defines specific assumptions about the effects that the library code could have on the various application entities; and (2) empirically showing that the separate compilation assumption is sufficient for constructing sound (with respect to dynamic call graphs) and precise (with respect to call graphs generated by DOOP and SPARK) application-only call graphs. Experimental results show that not analyzing the library code does not affect the soundness of the resulting call graph. In fact, in many cases (antlr, hsqldb, luindex, lusearch, pmd, and xalan) CGC was found to be more sound than DOOP and SPARK. In many cases, the call graphs generated by CGC are almost as precise as call graphs generated by DOOP, and sometimes more precise than SPARK. However, when the application implements a large subsystem whose interface is defined in the library (e.g., JDBC in hsqldb and XML in xalan), CGC loses precision compared to a whole-program analysis.

Remaining imprecisions are mainly due to objects that are passed into the library, but the library does not retain permanent references to them. This could be remedied by identifying, either manually or with an analysis, specific Java standard library methods known not to retain permanent references. CGC would then not add objects passed into these methods to the `LibraryPointsTo` set. This approach was also suggested by Tip and Palsberg [32].

Further improvements could be achieved by defining multiple libraries that have dependencies between them. Each library will then have its own `LibraryPointsTo` set that can interact with other `LibraryPointsTo` sets or points-to sets of application entities. Although this might improve the remaining imprecision in CGC, it requires extensive analysis for the library code to define those multiple libraries. Moreover, it might not be practical to apply this technique for user libraries as they vary greatly from one application to another.

To make the analysis more useful to users, we plan to create an Eclipse plugin that wraps our analysis and the tools we created alongside. The plugin will provide users with a suitable user interface for presenting the analysis results as well as navigating the call graph. We are also planning to embed the analysis into some of the widely used analysis frameworks such as SOOT and WALA.

References

1. Agrawal, G., Li, J., Su, Q.: Evaluating a Demand Driven Technique for Call Graph Construction. In: CC 2002. LNCS, vol. 2304, pp. 29–45. Springer, Heidelberg (2002)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1996, pp. 324–341 (1996)

3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA 2006, pp. 169–190 (October 2006)
4. Bodden, E.: Soot-list: Stack overflow when generating call graph (May 2011), http://www.sable.mcgill.ca/pipermail/soot-list/2008-July/001831.html
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: 33rd International Conference on Software Engineering, ICSE 2011, pp. 241–250 (2011)
6. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, pp. 243–262 (2009)
7. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 133–146 (1999)
8. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: 13th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 48–64 (1998)
9. Dean, J., Grove, D., Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
10. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
11. Dufour, B., Hendren, L., Verbrugge, C.: *J: a tool for dynamic analysis of Java programs. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, pp. 306–307 (2003)
12. Genius, D., Trapp, M., Zimmermann, W.: An Approach to Improve Locality Using Sandwich Types. In: Leroy, X., Ohori, A. (eds.) TIC 1998. LNCS, vol. 1473, pp. 194–214. Springer, Heidelberg (1998)
13. Graphviz - Graph Visualization Software (November 2011), http://www.graphviz.org/
14. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 241–255 (2001)
15. Holmes, R., Notkin, D.: Identifying program, test, and environmental changes that affect behaviour. In: International Conference on Software Engineering, ICSE 2011, vol. 10 (2011)
16. Holt, R., Schürr, A., Sim, S.E., Winter, A.: Graph eXchange Language (November 2011), http://www.gupro.de/GXL/dtd/gxl-1.1.html
17. IBM: T.J. Watson Libraries for Analysis WALA (May 2011), http://wala.sourceforge.net/
18. JUnit Home Page (December 2011), http://junit.sourceforge.net
19. Lhoták, O.: Comparing call graphs. In: 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2007, pp. 37–42 (2007)

20. Lhoták, O., Hendren, L.: Scaling Java Points-to Analysis Using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
21. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18, 3:1–3:53 (2008)
22. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
23. Linux User's Manual: time(1) (October 2011), http://www.kernel.org/doc/man-pages/online/pages/man1/time.1.html
24. LogicBlox Home Page (November 2011), http://logicblox.com/
25. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
26. Rountev, A., Milanova, A., Ryder, B.G.: Fragment class analysis for testing of polymorphism in Java software. IEEE Trans. Softw. Eng. 30, 372–387 (2004)
27. Rountev, A., Ryder, B.G., Landi, W.: Data-flow analysis of program fragments. In: 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7, pp. 235–252 (1999)
28. Sreedhar, V.C., Burke, M., Choi, J.D.: A framework for interprocedural optimization in the presence of dynamic class loading. In: ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI 2000, pp. 196–207 (2000)
29. Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (May 2011), http://www.spec.org/jvm98/
30. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 264–280 (2000)
31. The DOT Language (November 2011), http://www.graphviz.org/content/dot-language
32. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2000, pp. 281–293 (2000)
33. Tip, F., Sweeney, P.F., Laffra, C., Eisma, A., Streeter, D.: Practical extraction techniques for Java. ACM Trans. Program. Lang. Syst. 24, 625–666 (2002)
34. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
35. Vitek, J., Bokowski, B.: Confined types. In: 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 82–96 (1999)
36. Vitek, J., Bokowski, B.: Confined types in Java. Softw., Pract. Exper. 31(6), 507–532 (2001)
37. Zhang, W., Ryder, B.G.: Automatic construction of accurate application call graph with library call abstraction for Java: Research Articles. J. Softw. Maint. Evol. 19, 231–252 (2007)
38. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. J. Funct. Program. 16(1), 83–128 (2006)