Report for *HW3 CHA Call Graph*

Hongchen Cao
Shanghaitech University
Shanghai, China
caohch1@shanghaitech.edu.cn

***Abstract*—Report for *CS224 Homework3: CHA Call Graph*, including project structure, algorithm introduction, and Implementation description.**

I. Project Structure

Following is the directory tree of the source code of the project:

```
CHACallGraph
 └─src
 │  └─main
 │  │  └─java
 │  │     └─indi
 │  │        └─baseCG
 │  │        └─CHACallGraph
 │  │        └─Edge
 │  │        └─MethodHelper
 │  └─test
 └─out
 └─pox.xml
```

Here we give a brief description of each part of the project:

- **pom.xml:** Maven configuration of the project.
- **out:** Directory holds built artifacts(i.e., jar package).
- **src/main/java/indi:** Directory holds key codes for the project. Specifically, **CHACallGraph** and **baseCG** are the key files for the project
- **src/main/resources:** Directory holds resource files like MANIFEST.MF.
- **src/test:** Directory holds test files of the project.

For the two key java file (i.e., **CHACallGraph** and **baseCG**), we describe them in detail in Sec. III.

II. Algorithm

Listing 1. Algorithm for resolve method in CHA

```
T = ∅
m = method signature at $cs$
if $cs$ is a static call then
    T = {m}
if $cs$ is a special call then
    $c^m$ = class type of m
    T = {Dispatch($c^m$, m)}
if $cs$ is a virtual call then
    c = declared type of receiver variable at $cs$
    foreach $c$' that is a subclass of $c$ itself do
        add {Dispatch($c'$, m)} to T
return T
```

List. 1 shows the pseudocode of the key method *resolve()* in CHA.

III. Implementation

Next we introduce the two key classes in the project, and we describe the important member functions and attributes. For the other two classes, they are small and naive. Thus, we don't describe them in detail.

A. baseCG

This class contains three important functions.

• **baseCG().** Here we travel through all the classes in the input file. For each class, we travse its non-abstract methods. For each method, we store the map from its jimple IR uint to the method itself.

• **getCallSite(method).** For a given method, we travel through its units. If the unit contains a call to another method, we add the unit into call sites.

• **addEdge(unit, method, kind).** We first mark the method as reachable method. Then we create its corresponding edge which contains its kind, unit, and method. Then we add the edge into corresponding set through the map from callers to callees and calles to callers.

B. CHACallGraph

This class contains three important functions.

• **CHACallGraph(...).** We use worklist algorithm to create the call graph in this method. We iteratively get all the callees of each element(i.e., a caller method) in worklist by resolve() method and add edge into the cg.

• **dispatch(class, method).** We first try to find the method from all the methods in the given class by comparing the subsignatures. If we can't find, then we try to find it from the superclass of the given class.

• **resolve(unit).** This method totally follows the algorithm in List. 1. For a static call, we just return the set contains the invkoed method. For a special call, we return the set contains the disptach result. For a virtual call, it can be complex. If it is a interface, we first get all its implementers. If not, we first get its decalred type and then get all its direct and indirect subclasses. Then, for all the classes we get, we invoke dispatch() method and store them in a set.