MP 1

Introduction

The purpose of this assignment is to get you acquainted with static analysis. In particular, it will help you get familiar with Soot, a popular framework for analyzing Java programs (including Android applications).

From a high level view, Soot transforms Java programs into an intermediate representation (IR), which can then be analyzed much easier than directly analyzing Java bytecode or the parse tree. The primary IR in Soot is called Jimple, which is a typed three-address code (but very easy to read). Soot provides many out-of-the-box analyses on Jimple.

Part 0: Getting Started

In this section, we'll get started by downloading Soot and setting up your development environment. We will use Eclipse in this tutorial. However, you are free to use any other IDEs such as Intellij IDE or VS Code for the development. If you are not familiar with Eclipse, read this tutorial.

Notice: be sure that the java version you install is no later than Java8, and for the eclipse you can use the latest version.

Installing Eclipse and Soot

In your Eclipse workspace, create a new Java project "`a1`". Download soot-3.3.0.jar into the directory "`a1/lib`". It is an all-in-one file that contains all the required libraries from Soot nightly build. Alternatively, you can run the following script to download soot:

```
$ mkdir lib
$ cd lib
$ sudo curl
https://soot-build.cs.uni-paderborn.de/public/origin/master/soot/soot-master/3.3.0/build/sootclasses-t
runk-jar-with-dependencies.jar -o soot-3.3.0.jar
```

In Eclipse, add `soot-3.3.0.jar` to the build path of project "`a1`".

Environment Setting

Add soot to the class path

Choose and right click your java project folder, select "Build Path" and then "Configure Build Path", in the appeared properties window, choose "Classpath" and click the "Add External JARs..." in the right-hand side, then browse your local folder to locate where your "soot-3.3.0.jar" is. Finally, click "Apply and Close" and the soot project will be added in your class path.

Add Java to the JRE library

The same as before, but this time choose "Modulepath - JRE System library" rather than "Classpath", then choose "Workspace default JRE (Java SE 8)" and click "Finish".

Create package and add files

Right click the "src" folder under your project, and choose "New - Package" to create a package and put all the java files in the following tasks into this package.

Note: If the "src" already has a default package, just put the java files in that package.

Quick Start with Soot

Download TestSoot.java and HelloThread.java, and add them to the project source folder in Eclipse.

The code in `TestSoot.java` uses Soot to analyze `HelloThread.java` and print the Jimple statements in each method of the class `HelloThread`.

The main method is shown below:

```
public class TestSoot extends BodyTransformer {
	public static void main(String[] args)  {
	    String mainclass = "HelloThread";

	    //set classpath
	    String bootpath = System.getProperty("sun.boot.class.path");
	    String javapath = System.getProperty("java.class.path");
	    String path = bootpath + File.pathSeparator +javapath;
	    Scene.v().setSootClassPath(path);

	      //add an intra-procedural analysis phase to Soot
	    TestSoot analysis = new TestSoot();
	    PackManager.v().getPack("jtp").add(new Transform("jtp.TestSoot", analysis));

	      //load and set main class
	    Options.v().set_app(true);
	    SootClass appclass = Scene.v().loadClassAndSupport(mainclass);
```

```
        Scene.v().setMainClass(appclass);
        Scene.v().loadNecessaryClasses();

        //start working
        PackManager.v().runPacks();
    }
```

`TestSoot` extends `BodyTransformer` and overrides the method`internalTransform`, which will be called when the Jimple body of each analyzed method is traversed by Soot.

```
@Override
    protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
        Iterator<Unit> it = b.getUnits().snapshotIterator();
      while(it.hasNext()){
          Stmt stmt = (Stmt)it.next();
          System.out.println(stmt);
      }
```

Practice with Soot Tutorial

Browse Instrumenting Android Apps with Soot if you would like to analyze Android apps.

If you want to learn more, work through this excellent tutorial: A Survivor's Guide to Java Program Analysis with Soot.

Part 1: Control Flow Analysis - Finding Dominators

In this section, we'll implement a classical control flow analysis: finding dominators of a statement. When you're done with this part, you should have a basic understanding of control flow analysis.

By definition, a node `d` dominates a node `n` if every path from the entry node to `n` must go through `d`, and every node dominates itself.

Your task: given a method `m` and a statement `s`, to find all the statements in `m` that dominate `s`.

We will use the following algorithm where `D(n)` denotes a set of dominators of a node `n`:

```
D(n0) := { n0 }
 for n in N - { n0 } do D(n) := N;
 while changes to any D(n) occur do
    for n in N - {n0} do
        D(n) := {n} U (intersect of D(p) over all predecessors p of n)
```

You are provided with sample code TestDominatorFinder.java, DominatorFinder.java and a test program GCD.java that computes the greatest common divisor of two integers. `TestDominatorFinder` first uses Soot to build a control flow graph for each method, and constructs a `DominatorFinder` object for each control flow graph. For each statement, it then computes its dominators via the `DominatorFinder` object and prints them out. The relevant code is shown below:

```
@Override
protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
    UnitGraph g = new ExceptionalUnitGraph(b);
    DominatorFinder analysis = new DominatorFinder(g);
    Iterator it = b.getUnits().iterator();
    while (it.hasNext()){
        Stmt s = (Stmt)it.next();
        List dominators = analysis.getDominators(s);
        Iterator dIt = dominators.iterator();
        while (dIt.hasNext()){
            Stmt ds = (Stmt)dIt.next();
            String info = s+" is dominated by "+ds;
            System.out.println(info);
        }
    }
}
```

The class `DominatorFinder` has been partially implemented. You will need to complete the method `doAnalysis`. You may also add new fields and methods.

```
public class DominatorFinder<N> {
    DirectedGraph<N> graph;
    protected Map<N,BitSet> nodeToFlowSet;
    Map<Integer,N> indexToNode;

    public DominatorFinder(DirectedGraph<N> graph){
        this.graph = graph;
        doAnalysis();
    }

    protected void doAnalysis(){
        //your code starts here
    }

    public List<N> getDominators(Object node){
        //reconstruct list of dominators from bitset
        List<N> result = new ArrayList<N>();
        BitSet bitSet = nodeToFlowSet.get(node);
        for(int i=0;i<bitSet.length();i++) {
            if(bitSet.get(i)) {
                result.add(indexToNode.get(i));
            }
        }
        return result;
    }
}
```

Hints

- Since dominator analysis is fundamental, it may have already been implemented in Soot. Try to search in Soot source code to find existing code that you can learn from.

Part 2: Data Flow Analysis - Call Graph Construction

A call graph is a pre-requisite for almost every interprocedural analysis for object-oriented programs such as Java. In this section, we'll learn how to use Soot to construct call graphs. After this section, you'll have a basic sense of inter-procedural data flow analysis.

A Simple Example

Consider Example.java, the code shown below. **Question**: which method does `animal.saySomething()` in the `main` method call?

```
public class Example {
    static Animal neverCalled() {
        return new Fish();
    }
    static Animal selectAnimal() {
        return new Cat();
    }
    public static void main(String[] args) {
        Animal animal = selectAnimal();
        animal.saySomething();
    }
}
abstract class Animal {
    public abstract void saySomething();
}
class Cat extends Animal {
    public void saySomething() {
        System.out.println("purr");
    }
}
class Dog extends Animal {
    public void saySomething() {
        System.out.println("woof");
    }
}
class Fish extends Animal {
    public void saySomething() {
        System.out.println("...");
    }
}
class Car {  // not an Animal
    public void saySomething() {
        System.out.println("honk!");
    }
}
```

Answer: `animal.saySomething()` calls `Cat.saySomething()`. But, how to write an analysis to find that?

A simple way is to do a class hierarchy analysis (**CHA**), that identifies, at a call `o.f()`, the type of `o` and all its subtypes, and returns all the methods `f()` defined in these types. However, this is not very precise. For example, it will also report `animal.saySomething()` calls `Fish.saySomething()`, which is not true.

Another way is to find out which object (created in the program) that the reference `o` can refer to and return method `f()` of that object. For example, we can find that `animal` can only refer to the `Cat` object created in method `selectAnimal()`, and hence we can determine that `animal.saySomething()` only calls `Cat.saySomething()`. The analysis of finding which object a variable may reference is called points-to analysis (**PTA**). The result obtained by PTA is often more precise than CHA. However, PTA is not as simple as CHA. It needs to analyze the data flow in the whole program, and often takes much more time and memory than CHA.

Besides CHA and PTA, there are a few other classical call graph construction algorithms, such as Reachabilty Analysis (**RA**), Rapid Type Analysis (**RTA**), Class Type Analysis (**CTA**), Separate Type Analysis (**XTA**), Variable Type Analysis (**VTA**), k-order Control Flow Analysis (**k-CFA**), etc. Please read this page for a good summary of them.

Comparing Different Call Graph Construction Algorithms

Soot has implemented both CHA and PTA, in addition to a few others. CHA is implemented in class `CHATransformer`, and PTA is based on the Spark pointer analysis toolkit. Read the spark paper for more details. You are provided with sample code TestSootCallGraph.java, which can test both CHA and PTA. In `TestSootCallGraph`, you are provided with the following code, which prints out every call edge and the total number of call edges in class `Example`.

```
int numOfEdges =0;
CallGraph callGraph = Scene.v().getCallGraph();
for(SootClass sc : Scene.v().getApplicationClasses()){
    for(SootMethod m : sc.getMethods()){
      Iterator<MethodOrMethodContext> targets = new Targets(callGraph.edgesOutOf(m));
           while (targets.hasNext()) {
            numOfEdges++;
            SootMethod tgt = (SootMethod) targets.next();
            System.out.println(m + " may call " + tgt);
      }
    }
}
System.err.println("Total Edges:" + numOfEdges);
```

Your task: run `TestSootCallGraph` on `Example` (add both TestSootCallGraph.java and Example.java to your project source folder) to compare the precision and speed between CHA and PTA. In addition, you need to read the Soot code of CHA and PTA to understand how they work.

Part 3: Program Instrumentation with Soot

Another powerful usage of Soot is program transformation: taking an input program (Java source code or bytecode), it supports adding/removing code at the Jimple level, and outputs a new program. One typical task we can do with this feature is to add instrumentation code into the program, e.g., to profile the program execution. In this section, you will learn how (easy it is) to do that with Soot.

Logging Method Calls

Suppose we want to log all the method calls in the main method of Example.java we used in Part 2. We need to add two print statements in the main method at lines 2 and 4 below:

```
1. public static void main(String[] args) {
2.    System.out.println("calling selectAnimal");
3.    Animal animal = selectAnimal();
4.    System.out.println("calling saySomething");
5.    animal.saySomething();
6. }
```

Since many Soot analyses operate at the Jimple level, we'll add the two print statements at Jimple. The Jimple code of the original main method is shown below:

```
public static void main(java.lang.String[]){
    java.lang.String[] r0;
    Animal r1;
    r0 := @parameter0: java.lang.String[];
    r1 = staticinvoke <Example: Animal selectAnimal()>();
    virtualinvoke r1.<Animal: void saySomething()>();
    return;
}
```

The Jimple code above is actually very intuitive and easy to understand. Similary, the Jimple code of the instrumented main method is shown below:

```
public static void main(java.lang.String[]){
    java.lang.String[] r0;
    Animal r1;
    r0 := @parameter0: java.lang.String[];
    $r2 = <java.lang.System: java.io.PrintStream out>;
    virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>("calling selectAnimal");
    r1 = staticinvoke <Example: Animal selectAnimal()>();
    $r3 = <java.lang.System: java.io.PrintStream out>;
    virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String)>("calling saySomething");
    virtualinvoke r1.<Animal: void saySomething()>();
    return;
}
```

As you can see, a print statement `System.out.println("calling selectAnimal")` corresponds to two Jimple statements: `$r2 = <java.lang.System: java.io.PrintStream out>` and `virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>("calling selectAnimal")`. What we need is to create these two Jimple statements and insert them before the method invoke statement `r1 = staticinvoke <Example: Animal selectAnimal()>()`.

The sample code below does these tasks for you. **Please read carefully and understand every statement**. You don't need to copy this code. We provide the full sample code in TestSootLogging.java for you to play with later.

```
1. SootClass systemClass = Scene.v().loadClassAndSupport("java.lang.System");
2. SootClass printClass = Scene.v().loadClassAndSupport("java.io.PrintStream");
3. SootMethodRef printMethod = printClass.getMethod("void println(java.lang.String)").makeRef();
4. SootFieldRef fr = Scene.v().makeFieldRef(systemClass,"out",RefType.v("java.io.PrintStream"),true);
5. StaticFieldRef sfr = Jimple.v().newStaticFieldRef(fr);
6. Local r = Jimple.v().newLocal("$r2",RefType.v("java.io.PrintStream"));
7. AssignStmt newAssignStmt = Jimple.v().newAssignStmt(r, sfr);

8. @Override
9. protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
10.    if(b.getMethod().getName().equals("main")){
11.             b.getLocals().add(r);
12.             Unit toInsert = b.getUnits().getFirst();
13.             while(toInsert instanceof IdentityStmt)
14.                toInsert = b.getUnits().getSuccOf(toInsert);
15.             b.getUnits().insertBefore(newAssignStmt,toInsert);
16.        Iterator<Unit> it = b.getUnits().snapshotIterator();
17.          while(it.hasNext()){
18.            Stmt stmt = (Stmt)it.next();
19.            if(stmt.containsInvokeExpr()){
20.               String name = stmt.getInvokeExpr().getMethod().getName();
21.               InvokeExpr printExpr = Jimple.v().newVirtualInvokeExpr(r, printMethod,
StringConstant.v("calling "+name));
22.               InvokeStmt invokeStmt = Jimple.v().newInvokeStmt(printExpr);
23.               b.getUnits().insertBefore(invokeStmt,stmt);
24.            }
25.          }
26.     }
27. }
```

Code explanation:

- Lines 1-7 create the Jimple statement `$r2 = <java.lang.System: java.io.PrintStream out>`.
- Line 10 checks if the method is "`main`".
- Line 11 adds the created local variable "`$r2`" to the method.
- Lines 12-14 finds the beginning instruction (non-identity statement) in the method.
- Line 15 inserts our created Jimple statement before the beginning instruction.
- **Note**: *we insert before the begining instruction so that "`$r2`" can be reused when there are multiple invoke statements in the method.*Lines 16-19 traverse every statement in the method and check if it contains method invocation.
- Line 20 gets the name of the invoked method.
- Lines 21-22 creates the Jimple statement `virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>("calling selectAnimal")`, and Line 23 inserts it before the method invoke statement.

Your task:

- Understand the code in TestSootLogging.java and run it. It will generate a file `Example.jimple` in a folder "`sootOutput`" under your project directory. Read and understand the Jimple code in `Example.jimple`.
- In the `main` method of `TestSootLogging`, comment the statement `Options.v().set_output_format(1);` and run again. It will generate a Java class file `Example.class` in "`sootOutput`". Run `java Example` to see the output.

```
$ cd sootOutput
$ java Example
calling selectAnimal
calling saySomething
purr
```

Tracing Heap Accesses

In this section, we are interested in logging every read and write statement that accesses data on the heap. In particular, we are interested in adding instrumentation to log thread reads and writes to field variables. In Java, there are two types of field variables: static field and instance field. For example, in HelloThread.java shown below, `out` is static field of class `java.lang.System`, `x` is static field of class `HelloThread` and `y` is instance field of class `TestThread`.

```
public class HelloThread {
       static int x=1;
       public static void main(String[] args) {
           TestThread t = new TestThread();
           t.start();
           int z = t.y+1/x; // race here, may throw divide by zero exception
           System.out.println(z);
       }
       static class TestThread extends Thread{
           int y;
           public void run(){
              x=0;
              y++;
           }
       }
}
```

Your task: write code to instrument every field access in `HelloThread` to print out the access information. For example, a typical execution of the instrumented `HelloThread` will print:

```
Thread main wrote static field <HelloThread: int x>
Thread main read instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main]
Thread main read static field <HelloThread: int x>
Thread Thread-0 wrote static field <HelloThread: int x>
Thread main read static field <java.lang.System: java.io.PrintStream out>
Thread Thread-0 read instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main]
Thread Thread-0 wrote instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main]
```

Hints

- You are provided with a logging class Log.java that has already implemented the print statements. See the method `logFieldAcc` below. What you only need is to invoke it with proper arguments.

```
public static void logFieldAcc(final Object o, String name, final boolean isStatic, final boolean
isWrite){
    if(isStatic)
        System.out.println("Thread "+Thread.currentThread().getName()+(isWrite?" wrote":" read")+ "
static field "+name);
    else
       System.out.println("Thread "+Thread.currentThread().getName()+(isWrite?" wrote":" read")+ "
instance field "+name+" of object "+o);
}
```

- You are also provided with a sample code TestSootLoggingHeap.java that has implemented all the other functions excepts the instrumentation. Ideally you only need to write less than ten lines of code in the method `internalTransform` below:

```
@Override
protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
    //we don't instrument the Log class
    if(!b.getMethod().getDeclaringClass().getName().equals("Log")){
       Iterator<Unit> it = b.getUnits().snapshotIterator();
       while(it.hasNext()){
           Stmt stmt = (Stmt)it.next();
             if (stmt.containsFieldRef()) {
           //your code starts here
                    ...
             }
        }
    }
}
```

Bonus: can you also get the value of each read and write access? That is, print the log below:

```
Thread main wrote static field <HelloThread: int x> with value 1
Thread main read instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main] with
value 0
Thread main read static field <HelloThread: int x> with value 1
Thread Thread-0 wrote static field <HelloThread: int x> with value 0
Thread main read static field <java.lang.System: java.io.PrintStream out> with value
java.io.PrintStream@135fbaa4
Thread Thread-0 read instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main]
with value 0
Thread Thread-0 wrote instance field <HelloThread$TestThread: int y> of object Thread[Thread-0,5,main]
with value 1
```

This part is a bit challenging. You don't need to finish it in this assignment. However, you are encouraged to have a try if you are strongly motivated. The sample code will be released later.