Lab 1: Intra-procedural Analysis

In this lab, you will implement two intra-procedural analyses using the Soot static analysis framework `https://github.com/Sable/soot`. To help with visualizing your analyses, you can optionally use the VisuFlow Eclipse plugin provided as part of this exercise.

Introduction to Soot

An intra-procedural analysis operates within a single method, without considering the connections between callers and callees. When running an analysis with Soot, the Soot frameworks translates the analyzed Java code into the Jimple intermediate representation. It then extracts a Control-Flow Graph from the Jimple code of each method and runs the intra-procedural analysis on the individual CFGs.

Here are some Soot constructs that you will need in this lab:

- `SootClass`: represents a class.
- `SootMethod`: represents a method.
- `Body`: represents a method body.
- `Unit`: represents a code fragment (statement/instruction).
- `Stmt`: represents a statement (Unit and Stmt are similar and can be considered the same).
- `InvokeStmt`: represents a statements conatins a special/interface/virtual/static invoke expression.
- `InvokeExpr`: represents a special/interface/virtual/static invoke expression.

You can find the documentation of those classes into the `soot` and `soot.jimple` packages of Soot `https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/`.

General Instructions

Set up: The code in folder `DECALab1` contains a maven project readily set up. To set up your coding environment:

- Make sure that you have Java 8 or 9 running on your machine.
- Import the maven project into your favorite development environment.
- To run the test cases, run the project as a Maven build, with the goals set to "clean test".
- **Important: Make sure that your test cases run via the command line.** To do so, run `mvn clean test` in the `DECALab1` directory.

Project: The project contains:

- four `JUnit` test classes in the package `exercises` of the folder `src/test/java`. Your goal is to make all of those test cases pass.
- target classes in the packages `target.exercise*.` of the folder `src/test/java`. Those contain bad code on which the test cases are run.
- two classes containing the flow functions of the two analyses you will develop in this lab. They are located in the packages `analysis.exercise*.` of the folder `src/main/java`.

Analysis framework: In this project, we use the Soot analysis engine to write two types of static analyses: a detection of cryptographic API misuses and a simple typestate analysis. Do do so, fill in the `flowFunction` methods.

Exercise 1

In this exercise, you should implement an analysis that detect cryptographic API-misuses in java

programs. The class `javax.crypto.Cipher` can be used to encrypt data with the AES algorithm. However in practice, it is often misused by developers. For example, instead of passing **"AES/GCM/PKCS5Padding"** to the method `Cipher.getInstance()`, many just use the String **"AES"**, which is a vulnerable configuration. You should implement an analysis which detects exactly this misuse. Implement your analysis in the file `MisuseAnalysis.java` so that the misuses in the example code `Misuse.java` are detected. The JUnit tests `testMisuse()` and `testNoMisuse()` in `Exercise1Test.java` must pass.
(2 points)

Exercise 2

A typestate analysis is used to detect invalid sequences of operations that are performed upon an instance of a given type. In the package `target.exercise2` under directory `src/test/java` you will find the class `File.java` which implements the operations that can be performed to a File object. A File can be initialized, opened and closed, therefore, the states of a File object are **Init**, **Open** and **Close**. The valid sequences of operations for File objects are presented in the finite state machine in Figure 1. The valid final states of a file object are **Init** and **Close**. Figure 2 shows an example which contains an invalid sequence of file operations, since the file object initialized at line 1 is not closed. In the file `TypeStateAnalysis.java`, implement a typestate analysis for the sequence of operations shown in Figure 1. Your analysis should detect **unclosed** file objects. The JUnit tests `testFileClosed()`, `testFileClosedAliasing()`, `testFileNotClosed()` and `testFileNotClosedAliasing()` in `Exercise2Test.java` must pass.
(8 points)

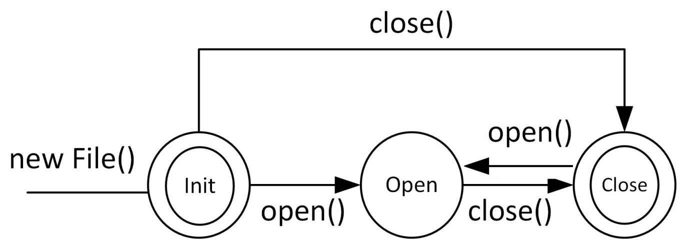

Figure 1: The Finite State Machine for File objects

```
1:  File a = new File();
2: a.open();
3: a = new File();
4: a.close();
```

Figure 2: An Example of Invalid Sequences of File Operations

Exercise 3

Static analysis is limited to what the analysis writer expects to find in the code. In practice, unexpected code constructs can make the analysis unsound (i.e. they do not report errors that are in the code). For example, malicious developers sometimes obfuscate their code to fool static analysis checkers. If you want to take it further, craft code examples that contain errors that your analyses cannot detect. This exercise will not be evaluated.
(0 points)