Lab 1: Intra-Procedural Analysis

In this lab, you will implement two intra-procedural analyses using the SootUp static analysis framework `https://github.com/soot-oss/SootUp`.

Introduction to SootUp

An intra-procedural analysis operates within a single method without considering the impact of other methods being called inside that method. When running an analysis with the SootUp framework, SootUp translates the analyzed Java code into Jimple intermediate representation (IR). It then extracts a Control-Flow Graph (CFG) from the IR of each method and runs the intra-procedural analysis on the individual CFGs.

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

- `Body`: represents a method body.
- `Stmt`: represents a code fragment (statement/instruction).
- `JavaSootClass`: represents a class.
- `JavaClassType`: represents an identifier, i.e., it contains the fully qualified classname for a JavaSootClass – used for loading and referencing JavaSootClasses.
- `JavaSootMethod`: represents a method.
- `MethodSignature`: represents an identifier for a JavaSootMethod – used for loading and referencing.
- `JInvokeStmt`: represents a statements that contains a AbstractInvokeExpr.
- `Subclasses of AbstractInvokeExpr`: represent a special/interface/virtual/static invoke expression.

You can find more SootUp information online `https://soot-oss.github.io/SootUp/`, especially for Jimple IR `https://soot-oss.github.io/SootUp/v1.1.2/jimple/` and in the javadoc `https://soot-oss.github.io/SootUp/apidocs/`.

General Instructions

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

- Make sure that you have Java 8, 9 or higher 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-SootUp` 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. Do not modify them.
- Target classes in the packages `target.exercise*.` of the folder `src/test/java`. Those contain bad code on which the test cases are run. Do not modify them.
- 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`.

Exercise 1

In this exercise, your goal is to develop an analysis that identifies misuses of cryptographic APIs in Java programs. Specifically, you need to focus on the incorrect use of the `javax.crypto.Cipher` class. This class is commonly used for **AES** encryption, but it requires more than just specifying the encryption algorithm for secure configuration. A common mistake developers make is to provide incomplete specifications. For instance, many use just the string **"AES"** instead of the full and secure **"AES/GCM/PKCS5Padding"** when calling `Cipher.getInstance()`. This leads to vulnerabilities in the code.

Your analysis should be able to precisely detect such misuses. Implement your solution in the file `MisuseAnalysis.java`. This implementation should be able to identify the misuses in the provided example code `Misuse.java`. To validate your analysis, ensure that the JUnit tests `testMisuse()` and `testNoMisuse()` in `Exercise1Test.java` are successful. (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 on 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 that 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)

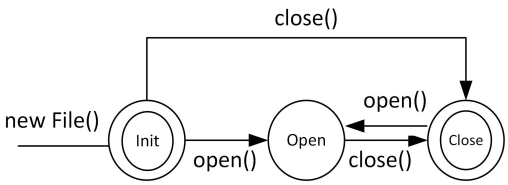

Figure 1: The Finite State Machine for File objects

```
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)