Intra-procedural Analysis

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

Introduction to Soot

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 Soot framework, Soot translates the analyzed Java code into Jimple intermediate representation (IR). It next 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 Soot constructs that you will need:

- `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 statement that 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 in 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 is in the file `session2-intraprocedural.zip`[1][2] and 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 `intraprocedural` 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 detects cryptographic-API misuses in

[1] https://owncloud.fraunhofer.de/index.php/s/ADrlUrAgXO3B2Pu
[2] Password: 3qTGm8S%m8

Java programs. The class `javax.crypto.Cipher` can be used to encrypt data with the encryption algorithm AES. However, in practice, the class needs configuring with more than an encryption algorithm. Developers often fail to provide a complete specification. For example, instead of passing **"AES/GCM/PKCS5Padding"** to the method `Cipher.getInstance()`, many just use the String **"AES"**, leading to a vulnerable configuration and, by extension, code. You should implement an analysis that detects this misuse exactly. 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.

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.

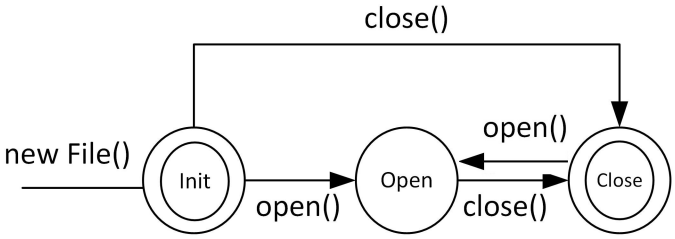

Figure 1: The Finite State Machine for File objects

```
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