Applied Static Analysis

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

Simple Static Analyses

You can prototype the analyses using the console or develop small stand-alone analyses.

> If you use the console, don't use the methods which execute the analysis in parallel (e.g., `Project.parForeachMethodWithBody`). The console is broken when multiple threads are used and will run into a deadlock!)

If you want to develop it as a real application, you should use `MyOPALProject` as a template. That project is preconfigured to use the latest snapshot version of OPAL. You can clone the project using:
`git clone --depth 1 https://bitbucket.org/OPAL-Project/myopalproject Project`

An integrated JavaDoc of the latest snapshot version of OPAL that spans all subprojects can be found at: www.opal-project.de

For further details regarding the development of static analysis using OPAL see the OPAL tutorial.

Exercise: Security Checks in Private or Final Methods

Develop an analysis that finds violations of the following rule taken from The CERT Oracle Secure Coding Standard for Java:

> MET03-J: Methods that perform a security check must be declared private or final.

A method is considered to perform a security check if it calls one of the `check*` methods on an instance of the `SecurityManager` returned by `System.getSecurityManager()`.

> Exception: Classes that are declared final are exempt.

Non-compliant example:

```
public void processSensitiveFile(){
    String f = "FileName";
    try {
        SecurityManager sm = System.getSecurityManager();
        if(sm != null) {
            sm.checkRead(f);
        }
        // process file
    } catch (SecurityException se) {
        // handle exception
    }
}
```

Compliant example:

```
public final void processSensitiveFile(){
    String f = "FileName";
    try {
        SecurityManager sm = System.getSecurityManager();
```

```
            if(sm != null) {
                sm.checkRead(f);
            }
            // process file
        } catch (SecurityException se) {
            // handle exception
        }
    }
```

Tasks

1. Develop the analysis using the simplest possible approach. It is ok if it is subject to false positives, but it should not be subject to false negatives!
2. Test your analysis by running it against the entire JDK.
3. Ask yourself which classes, beyond those that are explicitly declared final, are also effectively final and should also be exempt.

Exercise: Ignored Return Value

Develop an analysis that finds violations of the following rule taken from The CERT Oracle Secure Coding Standard for Java:

> EXP00-J: Do not ignore values returned by methods.

Non-compliant example:

```
File f = new java.io.File("MyTempFile.txt");
f.delete(); // <= Return value ignored
```

Compliant example:

```
File f = new java.io.File("MyTempFile.txt");
if(!f.delete()) {System.out.println("File could not be deleted")};
```

Tasks

1. Develop the analysis by analyzing a method's bytecode. The approach should `only` handle the vast majority of standard cases; it must not handle every possible case.
2. Test your analysis using the class `IgnoredReturnValue`.
3. Test your analysis by running it against the entire JDK. What do you think about the result?

Exercise: Comprehending CFGs

The goal of this exercise is to get a better understanding of the shape of real-world CFGs.

Tasks

1. Develop an analysis to help you understand why a basic block ends? Does it end due to a potentially thrown exception, a return instruction, a control transfer instruction or because the next instruction is a regular instruction to which some branch instruction jumps to? If a basic block ends due to multiple reasons, count all of them.

2. Develop a small analysis which computes the following metrics for some set of methods (**always w.r.t. the number of instructions**; i.e., *not in terms of the length of the underlying bytecode array*):
   - The number of basic blocks in the code base
   - Most frequent basic-block length
   - Maximum length of a basic block
   - Average length of basic blocks in terms of the number of instructions
   - The average in-degree of basic blocks; i.e., how many predecessors – in average – a basic block has

3. Test your analysis using the `IrreducibleCF` class found in folder `2-Java-Bytecode` in the lecture's GitHub repository.
4. Run the analysis against the JDK.