Homework 6: Interprocedural Analysis

17-355/17-665/17-819: Program Analysis
Jonathan Aldrich*
aldrich@cs.cmu.edu

Checkpoint Due: Thursday, February 28, 2019, 11:59 pm

Full Assignment Due: Tuesday, March 5, 2019, 11:59 pm

200 points total

Assignment Objectives:

- Implement a larger-scale, more realistic program analysis
- Solidify and demonstrate understanding of the course material on alias and/or interprocedural analysis.

Handin Instructions. Place all your homework files in your private GitHub repository in a folder called `hw6`. After the deadline, we will clone your repository and run your analysis tool (including tests). You should include a way to build and run tests for your analysis (e.g., using `ant` or a `Makefile`). Include a `README.md` file explaining how to run your analysis against the test case(s) you provide.

Context-Sensitive Interprocedural Analysis Implementation

In this assignment, you will implement a context-sensitive interprocedural analysis. There are two options:

- **Sign Analysis.** You may implement a context-sensitive, interprocedural version of Sign Analysis. The advantage of this approach is that you can build on your existing intraprocedural Sign Analysis implementation. Like the analysis you implemented before, your Sign Analysis will warn if an array dereference uses a definitely-negative or possibly-negative index. However, it will be more precise than your prior analysis because it will consider interprocedural data flows in a context-sensitive way.
- **Constant-derived string analysis.** You may implement a context-sensitive, interprocedural constant-derived string analysis. This option allows you to explore a new kind of analysis that is important to real analysis tools; it is essentially the inverse of Taint Analysis. The goal of this analysis is to determine which strings in the program are entirely derived from string constants. A string is constant-derived if it is a literal string constant, if it flows from a literal string constant (via assignments to variables and fields), or if it results from combining two

*This homework was developed together with Claire Le Goues

constant-derived strings (e.g., in Java s1 + s2, where s1 and s2 are constants or derived from constants themselves).[1]

An important application of this kind of analysis is to ensure that certain uses of strings are secure. For example, a format string passed as the first argument to `java.util.Formatter.format` should always be a compile-time constant, or derived from one; otherwise, your code is likely to have a format string vulnerability. Likewise, a query passed to `java.sql.Connection.prepareStatement` should be a compile-time constant, or derived from one; otherwise your code is likely to have a SQL injection vulnerability. For example, I believe Google runs a static analysis similar to this on every Java compilation to ensure the lack of SQL vulnerabilities. Your analysis should output a warning if either of these methods are called with a string that is not (or might not be) constant-derived according to your analysis.

You may, if you wish, implement a general taint analysis of a constant-derived string analysis. The difference is that taint analysis tracks which strings may (or must) derive from user input, while constant-derived analysis tracks which strings may (or must) derive from compile-time string constants.

Implement your analysis in Soot or some other program analysis infrastructure. Write appropriate test cases to ensure your analysis is working properly. One or more test cases should require context sensitivity—i.e., the test case would fail if your analysis was interprocedural but not context-sensitive.

Write a `README.md` that describes what you did: what analysis you implemented, in what infrastructure, and for what language. Explain what context-sensitivity strategy you used. Describe how to compile your analysis and how to run the tests, in enough detail that the instructor should be able to compile and test it without contacting you.

Checkpoint. Commit a version of your analysis to your GitHub repository by the checkpoint deadline listed above. You should have some interprocedural analysis working, but it is OK if there are still bugs or if context sensitivity is not yet working. Your `README.md` file should, as above, describe how to compile and run your analysis on some test case that illustrates something working interprocedurally. The checkpoint is worth 40 points, i.e., 20% of the total credit for the assignment.

[1] Note that in Java, s1 + s2 is implemented as `new StringBuilder().append(a).append(b).toString();`—this is the code that will Soot will see.