Intra-procedural Static Taint Analysis of JavaScript Programs

Program Analysis, University of Stuttgart
Winter 2019-20
Project mentor: Jibesh Patra (jibesh.patra@iste.uni-stuttgart.de)

1. Introduction

Taint analysis is a general approach to check for integrity or confidentiality violations and is popular in the security community. For checking for integrity problems, taint analysis tracks the flow of information from an untrusted input called *source* to potentially sensitive locations called *sinks*. As an example, consider the following JavaScript program that takes a user input and runs a query on the input.

```
1  function getInformation(){
2    let input = getUserInput();
3    let q = 'SELECT * FROM ' + input;
4    let len = 20;
5    if (q.length > len){
6      let ans = query(q);
7    }
8  }
```

The source in this case is `input` and the sink is the `query` function. An adversary may leverage the potential flow of data from `input` to the `query` to run arbitrary queries. A proper way to defend against attacks would be to find program locations where user input influences sensitive sinks and take measures to *sanitize* the input.

2. Goal

The goal of this project is to implement a taint analysis approach that tracks whether a *source* value flows as an input to a *sink*. For the purpose of this project, we may assume that two special functions called `retSource()` and `sink(x)` are available. It may also be assumed that on each call, `retSource()` always returns a new *source* and does not accept any argument. In contrast, the `sink(x)` function accepts only one argument and does not return any value. The goal of this project is to implement a taint analysis that tracks whether a value returned by a `retSource()` call eventually flows as an input to the `sink(x)` function. For the example given above, the analogous functions to `retSource()` and `sink(x)` are `getUserInput()` and `query(q)` respectively.

A variable is said to be *tainted* if there exists a flow of data from a *source* to the variable. In the above example, variable `q` on line number 3 is said to be tainted. In contrast, the variable `len` on line number 4 is said to be *untainted* since there exists no flow of data from the source `input`. As a data-flow analysis problem, taint analysis can be formulated as a forward analysis that keeps a set of tuples containing a variable and a label denoting *tainted* or *untained* for each program point. The goal is to track if a variable with a tainted label flows into a sink.

3. Tasks and Implementation Details

More specifically, the project involves the following tasks:

- Create your own **private** repository at GitHub and copy the contents of the *static-taint-analysis-project* [1] repository into it. You must use a private repository, as the project is individual and your solution (or parts of it) must not be shared with other students.
- Build the *closure-compiler* provided in the repository on your local machine. Please use exactly the version of Closure provided in the repository. Build instructions can be found here [2].
- The taint analysis may be added as a compiler pass[3] to the *closure-compiler*. You may refer to the class `src/com/google/javascript/jscomp/MustBeReachingVariableDef.java` to get more ideas on writing the analysis. Specifically, you have to write your analysis as a subclass of the `src/com/google/javascript/jscomp/DataFlowAnalysis.java` class.

[1] https://github.com/michaelpradel/static-taint-analysis-project
[2] https://github.com/google/closure-compiler#building-it-yourself
[3] https://github.com/google/closure-compiler/wiki/Writing-Compiler-Pass

- Given a valid JavaScript file, the final output of the analysis should be a JSON file. Each entry of the JSON file should correspond to a function and contain two values, one corresponding to the sources that *must* reach a sink and the other corresponding the sources that *may* reach a sink. The following demonstrates the format of the JSON file for the above given example.

```
{
  "getInformation@1-8": {
    "sources_that_must_reach_sinks": [],
    "sources_that_may_reach_sinks": ["input@2"]
  }
}
```

  Explanation: The numbers after @ denote the line numbers of the definition i.e., *getInformation@1-8* means the function defined between line numbers 1 and 8. For the example given in Section 1, the flow to the sink is guarded by an *if* statement and this results in the *may* flow of the source `input` of line 2 to the sink.

4. Evaluation

The forked GitHub repository contains a folder called *benchmark* which currently contains only one test called *one.js*. You may start your evaluation using this single test case. More tests will be added to this folder during the semester. For your reference, this test case also contains a corresponding expected output JSON file called *one_ out.json*.

5. Deliverables and Grading

The final submission of the project should be a GitHub repository containing the implementation and must be sent to the project mentor via email by February 7, 2020 (end of day). In addition to the implementation, the repository should also contain the following:

- A closure compiler JAR file located in the root of the repository that accepts one JavaScript file as input and writes out a JSON file strictly matching the format explained in Section 3. The output JSON file must be written to the same folder as the input JavaScript file. For example, if the input JavaScript file path is `benchmark/test_1/one.js` then the output JSON file should be `benchmark/test_1/one_out.json`.
- A README file in the repository should explain how to execute the JAR.
- A project report as a PDF file. The report should be written like a (short) scientific paper and in English. The report should describe how the analysis addressed the taint analysis problem and explain details with figures, algorithms, etc. In addition, it should also discuss the results of the analysis, obtained on given the benchmarks, with tables, plots, etc. The strict page limit is four pages including references.

Each person must present the project on the week of February 10-14, 2020, in a short talk, followed by a question and answer session.

Grading will be based on the following criteria:

Criterion	Contribution
Report (structure, explanations, examples, writing)	25%
Results (discussion and interpretation, soundness, reproducibility)	25%
Implementation (completeness, documentation, extensibility)	25%
Presentation (clarity, illustration, quality of answers)	25%