Intra-procedural Static Taint Analysis of JavaScript Programs

Program Analysis, University of Stuttgart
Winter 2020-21
Project mentors:
Aryaz Eghbali (aryaz.eghbali@iste.uni-stuttgart.de),
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, which we have assigned to three milestones. At each milestone, every student has a progress meeting with her/his mentor, where the student should briefly present what she/he has done, and where the mentor is answering questions and will give feedback.

3.1. Milestone 1 (Dec 14, 2020)

During the project meeting on this day, the course participants are expected to have at least done the following:

- Create your own **private** repository at GitHub and copy the contents of the following repository into it: `https://github.com/michaelpradel/static-taint-analysis-project`. 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: `https://github.com/google/closure-compiler#building-it-yourself`
- Download the latex template for the project report.

3.2. Milestone 2 (Jan 11, 2021)

During the project meeting on this day, in addition to the tasks mentioned in milestone 1, the course participants are expected to have at least done the following:

- The taint analysis should be added as a compiler pass. Read how to add a compiler pass to closure-compiler here: `https://github.com/google/closure-compiler/wiki/Writing-Compiler-Pass`.
- Go through the JavaDoc comments present at `src/com/google/javascript/jscomp/DataFlowAnalysis.java`.
- Write your analysis as a subclass of `DataFlowAnalysis`. For inspirations refer to `src/com/google/-javascript/jscomp/MustBeReachingVariableDef.java`.
- Given a valid JavaScript file, the final output of the analysis should be a JSON file. Your analysis should be able to handle at least *test_1* present in the *benchmark* folder of the repository, i.e., it should output *one_out.json*. Each entry of the output JSON files 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 *one_out.json*.

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

  Explanation: The number after `@` denotes the line number of the definitions, i.e., `getInformation@1` means the function definition starts from line 1. 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.
- Finish writing the introduction section of the final report.

3.3. Milestone 3 (Jan 25, 2021)

For this meeting, in addition to the tasks in the previous milestones, the course participants are expected to do the following:

- Extend the analysis to handle test cases that contain loops and multiple functions. The analysis should correctly handle at least eight of the given test cases.
- Finish writing the approach section of the final report. Add examples and figures that explain the approach.

In addition to the given test cases, we strongly recommend that each student creates more test cases on her/his own. To grade the project, we will use additional test cases not shared in the repository.

4. Deliverables and Grading

The final submission of the project should be a GitHub repository containing the implementation and must be submitted via Ilias by February 12, 2021 (end of day, Stuttgart time). In addition to the implementation, the repository should 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 that explains 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, figures, etc.

Each person must present the project on the week of February 15-19, 2021, in a short talk, followed by a question and answer session.

Grading will be based on the following criteria:

Criterion	Contribution
Progress meetings and final presentation (progress made, clarity, illustration, quality of answers)	25%
Implementation (completeness, documentation)	25%
Results (discussion and interpretation, soundness, reproducibility)	25%
Report (clarity, illustration)	25%