# Piacente Cristian 866020

## Assignment A4 – Data flow analysis

### Software Quality, Academic Year 2023-2024, University of Milan – Bicocca

With reference to the sample program, consider the problem of identifying which definitions of variable sanitized can be used at the return statement at the end of the program.

| | | | | | init | | PASS 1 | | PASS 2 | | PASS 3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | STATEMENT | PREVIOUS STATEMENT IDs | GEN | KILL | REACH | REACH-OUT | REACH | REACH-OUT | REACH | REACH-OUT | REACH | REACH-OUT |
| A | String sanitizeUserInput(String userInput, List<Checker> checkers) throws UncheckedInputException { | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ |
| B | String sanitized = userInput; | A | sanitized_B | ∅ | ∅ | ∅ | ∅ | sanitized_B | ∅ | sanitized_B | ∅ | sanitized_B |
| C | int warnings = 0; | B | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B |
| D | if (!checkers.isEmpty()) { | C | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B |
| E | boolean done = false; | D | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B |
| F | for (int i = 0; | E | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B |
| G | i < checkers.size() && !done; | F, O | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| H | Checker c = checkers.get(i); | G | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| I | if (c.isActive()) { | H | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| J | sanitized = c.apply(userInput); | I | sanitized_J | sanitized_B, sanitized_J | ∅ | ∅ | sanitized_B | sanitized_J | sanitized_B, sanitized_J | sanitized_J | sanitized_B, sanitized_J | sanitized_J |
| K | done = true; | J | ∅ | ∅ | ∅ | ∅ | sanitized_J | sanitized_J | sanitized_J | sanitized_J | sanitized_J | sanitized_J |
| L | } else if (c.riskDetected(userInput)) { | I | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| M | LOGGER.log("Possible issues for: " + userInput); | L | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| N | ++warnings; } | M | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| O | ++i) } | K, L, N | ∅ | ∅ | ∅ | ∅ | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| P | } else { throw new UncheckedInputException(); } | D | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B | sanitized_B |
| Q | if (warnings > 0) { | G | ∅ | ∅ | ∅ | ∅ | sanitized_B | sanitized_B | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J | sanitized_B, sanitized_J |
| R | sanitized = this.defaultChecker.apply(userInput); } | Q | sanitized_R | sanitized_B, sanitized_J | ∅ | ∅ | sanitized_B | sanitized_R | sanitized_B, sanitized_J | sanitized_R | sanitized_B, sanitized_J | sanitized_R |
| S | return sanitized; } | Q, R | ∅ | ∅ | ∅ | ∅ | sanitized_B, sanitized_R | sanitized_B, sanitized_R | sanitized_B, sanitized_J, sanitized_R | sanitized_B, sanitized_J, sanitized_R | sanitized_B, sanitized_J, sanitized_R | sanitized_B, sanitized_J, sanitized_R |

In detail (consider only variable sanitize):

- Identify an appropriate type of data flow analysis, indicating its characteristics (forward/backward, all/any path)
  - **Reaching definitions**, since we need to consider the problem of identifying which definitions can reach the return statement, w.r.t every possible program path. It is a **forward**, **any-path** analysis.
- Define kill and gen operations
  - **Kill**: the variable sanitized is redefined, killing any previous definitions.
  - **Gen**: a value is assigned to the variable sanitized, generating a new definition.
- Identify the kill and gen sets for the code
  - See the table above.
- Report the results of the analysis for each statement in the code
  - See the table above.

# Piacente Cristian 866020

## Assignment A4 – Data flow analysis

### Software Quality, Academic Year 2023-2024, University of Milan – Bicocca

Furthermore, refer to your results to answer the following questions:

- Do your results indicate the danger that the intial value of variable sanitized (assigned at the very beginning of the program) can be returned without modifications in between?
  - **Yes**: the initial definition of sanitized (sanitized_B) belongs to the reaching definitions of the statement S.
    This means there is the danger that the initial value of sanitized could be returned without modifications in between.
- If yes, can you indicate all definition-clear program paths that correspond to such dangerous behavior, by considering only the program paths that iterate at most once through the loop in the program?
  - Considering only the program paths that iterate at most once through the loop, the following paths could lead to returning the initial value of sanitized:
    - A-B-C-D-E-F-G-Q-S
    - A-B-C-D-E-F-G-H-I-L-O-G-Q-S
    - A-B-C-D-E-F-G-H-I-L-M-N-O-G-Q-S

    These program paths were found taking into account that we must skip the statements J and R, since they would kill any previous definitions of sanitized, and also we must avoid to execute the statement P, because it would lead to throwing an exception.
- For each program path that you indicated at the above answer (if any) say also if it is a false alarm or a true alarm, and why.
  - Let's analyze each program path from the previous answer, to find out if it's a feasible path or not.
    - A-B-C-D-E-F-G-Q-S: we have 0 iterations of the loop.
      This means the condition **i < checkers.size() && !done is evaluated to false from the beginning**: since the variables done and i are initially defined as false and 0, we have that **checkers.size() must be 0**. It is a **contradiction**: after executing the statement D we make sure the list checkers is not empty, so when reaching the statement G we will always have a list with a size greater than 0. This program path is **infeasible**, so it's a **false alarm**.
    - A-B-C-D-E-F-G-H-I-L-O-G-Q-S: we have 1 iteration of the loop, in particular the list must contain exactly **1 checker** which is **not active** and which **does not detect a risk** on the user input. After the loop, to avoid executing statement R, the variable **warnings must be less or equal to 0**. The program path is **feasible**: it is possible to have such a checker in the list, and in this case the variable warnings keeps the initial value of 0. We have a **true alarm** here.
    - A-B-C-D-E-F-G-H-I-L-M-N-O-G-Q-S: we have 1 iteration of the loop, in particular **1 checker that is not active** and that **detects a risk** on the user input. After the iteration, the variable **warnings must be less or equal to 0**. This is a **contradiction**: since a risk is detected, the variable warnings is incremented at the statement N and so after the loop it has the value of 1, that is greater than 0 and the statement R would be executed, killing the initial definition of the variable sanitized. This program path is **infeasible**, so it's a **false alarm.**
  - In conclusion, from our analysis we can say that the initial definition of sanitized gets returned iff all the following conditions are true:
    - the parameter checkers is a list which contains **at least 1 checker** (to skip the exception at the statement P that gets thrown in case of an empty list)
    - **all checkers** in the list **are not active** (to skip the statement J)
    - **every checker does not detect a risk** on the user input (to skip the statement N, which results in skipping the statement R).