CS5218: Assignment 3

Comparison of the CLANG and INFER Analyzers

Deadline: 23:59 Sunday 7th April 2024

In this assignment, you will compare two well-established Static Analysis tools **Clang** and **Infer** on a set of programs with different sizes. Clang and Infer static analysers use flow and partial path-sensitive analyses. The set of benchmark programs contain defect types related to dynamic memory allocation, error handling, multi-threading, etc.[1].

You are to organize yourselves into *three-person teams* (with an exception of one size 4 team). Please indicate the team structure for this project in the document: Google Spreadsheet by 23:59 Monday 25th March 2024.We need this information by the given deadline as we are going to share different sets of programs to each team individually.

1 Academic Programs

We consider two academic programs P1 and P2. The programs are written with a macro `TEAM_ID`, whose value is to be instantiated by each team by its identification number, so each team will consider a slightly different version. We wish to determine if the *error point* (indicated) is *reachable*. The expression Ψ`(x)` represents some predicate on the variable `x`. This is the *error condition*. An analyzer reports an *alarm* if it cannot determine if the error point is unreachable; otherwise, it reports *bug-free*.

P1

```
#define TEAM_ID ...  // (substitute your team ID here
#define N ...  // (N is statically fixed)
main() {
int x = TEAM_ID;
for (int i = 1; i <= N; i++) x += 1;
if (x >= 0 && error(x))
    x = 1 / 0; // error point
}
int error(int x) { Ψ(x) }
```

[1] Clang and Infer ppts are available on Canvas

P2

```
#define TEAM_ID ...   // (substitute your team ID here
#define N ...
int a[] = {1, 3, 5, 7, 9};
main() {
     int x = TEAM_ID, choice;
     for (int i = 0; i < N; i++) {
          scanf("%d", &choice);
          if (choice) x += a[i];
     }
     if (error(x)) x = 1 / 0; // error point
}
int error(int x) { Ψ(x) }
```

Program P1 has a straight line execution sequence. Thus whether an error is detected will depend on the analyzer to sufficient accurately track the value of `x` to the error point, and determine if `error(x)` is *definitely false* (and so no alarm is raised). Program P2, on the other hand, has many possible (in fact 2^N) execution paths can reach the error point. Thus to determine that there is no error, the analyzer needs to track up to 2^N different values of `x`.

Now consider the possible answers (*alarm or bug-free*) an analyzer can report, and what these answers can mean (*true or false*):

FP: *alarm* but in fact the error point is unreachable. This is a *false positive*. (This occurrence is frequent in practice and is the biggest challenge to static analysis for bug finding).

TP: *alarm* AND in fact the error point is reachable. This is a *true positive*. (This occurrence is probably due to "pure dumb luck"!)

FN: *bug-free* but in fact the error point is reachable. This is a *false negative*. (This occurrence is probably due to the [criminal?] decision to reduce the frequency of false positives by remain silent.)

TN: The analyzer reports *bug-free* AND in fact the error point is unreachable. This is a *true negative*. (Whether or not this is pure dumb luck is debatable, but this occurrence is the happiest outcome.)

The four outcomes above can be summarized as a "confusion matrix".

The following are example conditions for Ψ`(x)` and what the results are for the analyzer **Infer**.

- P1

`x < TEAM_ID - 99`	returns BUG-FREE; result is a *True Negative*
`x != TEAM_ID + N`	returns ALARM; *False Positive*
`x == TEAM_ID + N + 1`	returns ALARM; *False Positive*
`x == TEAM_ID + N`	returns ALARM; *True Positive*

- P2

`x >= TEAM_ID`	returns ALARM; result is a *True Positive*
`x == TEAM_ID + 23`	returns BUG-FREE; *True Negative*
`x == TEAM_ID + 22`	returns BUG-FREE; *False Negative*
`x <= TEAM_ID`	returns ALARM; *True Positive*

Task 1:

An error condition `e1(x)` is a predicate on a variable `x`, and it is more general than another error condition `e2(x)` if $e_2(x) \longrightarrow e_1(x)$. Your task is to revise the (eight) error conditions with new conditions so that your conditions (a) *return the same answers* (ie. same Infer outcome and same result), and (b) are *as general as possible*. As a trivial example, you could revise the "99" in the first error condition to "100".

2 Benchmarks

For this section of the assignment, you have to test ITC-Benchmarks (static analysis benchmarks from Toyota ITC). We will email different sets of 20 C programs to each team separately. This set contains C programs in two categories of a) *With defects* (Injected into programs), and b) *Without defect*s (The same errors have been resolved).

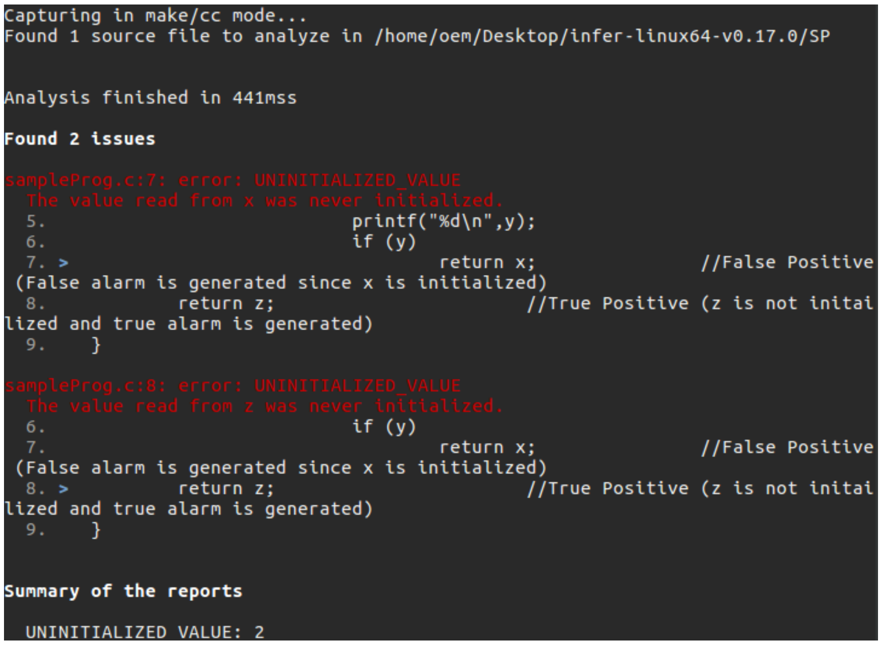
Example 1: For example, consider the program below.

```
1    int f(int y){
2        int x, z;
3        if (y)
4          x=1;      //True Negative (No error because x is defined)
5        printf("%d\n",y);
6        if (y)
7          return x;    //False Positive (False alarm is generated since x is
   initialized)
8    return z;     //True Positive (z is not initailized and true alarm is generated)
9  }
```

In this program **Example 1**, there is a bug only in one statement i.e., UNINITIALIZED_VALUE of z in statement-8. However, when we ran the program with Infer analyzer, it reported alarm for two statements. Please check Figure 1. between which, the first reported alarm for statement-10 is a FALSE POSITIVE because this path is unreachable and this bug will never get invoked. On the other hand, the second alarm for statement-12 is a true bug and correctly got identified by the analyzer as a TRUE POSITIVE. Statement-4 has no bug and Infer has not given alarm for it, so it is an example of TRUE NEGATIVE. The fourth case of FALSE NEGATIVE is difficult to show with a small example program. But you can refer to this article to get a some more understanding on FALSE NEGATIVE.

Task 2:

Your task is to analyze the provided set of programs; and report if the tools have found any errors; and then reason about the scalability of the tools. For each program, record the number of outcomes according



```
Capturing in make/cc mode...
Found 1 source file to analyze in /home/oem/Desktop/infer-linux64-v0.17.0/SP


Analysis finished in 441mss

Found 2 issues

sampleProg.c:7: error: UNINITIALIZED_VALUE
  The value read from x was never initialized.
  5.                              printf("%d\n",y);
  6.                              if (y)
  7. >                                    return x;               //False Positive
 (False alarm is generated since x is initialized)
  8.             return z;                        //True Positive (z is not initai
lized and true alarm is generated)
  9.     }

sampleProg.c:8: error: UNINITIALIZED_VALUE
  The value read from z was never initialized.
  6.                              if (y)
  7.                                     return x;               //False Positive
 (False alarm is generated since x is initialized)
  8. >           return z;                        //True Positive (z is not initai
lized and true alarm is generated)
  9.     }


Summary of the reports

  UNINITIALIZED_VALUE: 2
```

Figure 1: Terminal output of Infer for the sample C-Program

to the "confusion matrix" described above.

To better understand the two analysers performance on specific defect types, please group the given ITC programs based on these nine categories:
1. Static memory defects
2. Dynamic memory defects
3. Stack related defects
4. Numerical defects
5. Resource management defects
6. Pointer related defects
7. Concurrency defects
8. Inappropriate code
9. Misc defects

You can also follow the supplementary official specification to know more about the test bench and defect sub-types.

Task 3:

Finally, *for each tool*, you should generate *two* (small, academic) programs, one for which the tool is good, and another one for which the tool is bad. The aim here is to understand the pros and cons of the different tools in terms of a concrete example for 4-5 different types of bugs.

3 Suggested Command-line Arguments for Each Tool

Some useful command-line arguments for each of the tools are presented here.

3.1 Clang

Clang is free and open-source. Please download it from here: http://clang-analyzer.llvm.org/

Important options are as follows:

1. Run a program with default command options.

   Command Line: ∼/Desktop/$ clang –analyze dead_store.c[2]

For more detail please run *clang –help*.

3.2 Infer

Install Infer from here (https://github.com/facebook/infer/blob/main/INSTALL.md)

1. Run a program with default command options.

   Command Line: ∼/Desktop/Infer/infer-linux64-v0.17.0$ bin/infer run -o dead_store – clang -c dead_store.c

2. Run a program to find specific type of bug. For example, to run div_zero.c program to find the errors where the division operations denominator has value 0, in result the output will be undefined.

   Command Line: ∼/Desktop/Infer/infer-linux64-v0.17.0$ bin/infer run –enable-issue-type DIVIDE_BY_ZERO -o div_zero – clang -c div_zero.c

3. Run a program to find the mix errors, with convenient command options. For example, run mixtype.c program with command options where DEAD_STORE and UNINITIALIZED_VALUE types errors are disabled and DIVIDE_BY_ZERO type error is enabled to report.

   Command Line: ∼/Desktop/Infer/infer-linux64-v0.17.0$ bin/infer run –enable-issue-type DIVIDE_BY-_ZERO disable-issue-type DEAD_STORE disable-issue-type UNINITIALIZED_VALUE -o mixtype – clang -c mixtype.c

First try to run both tools (Clang and Infer) on their 'default' settings, i.e., **`clang --analyze`** and **`infer run`**. If this yields unsatisfactory results, then try to enable the non-default checkers[3][4] and re-run the test. It will generate better results (ie. detect more true positives).

[2] dead_store.c is a sample program

[3] https://releases.llvm.org/13.0.1/tools/clang/docs/analyzer/checkers.html

[4] https://fbinfer.com/docs/all-issue-types

4 Submission

Submit a report (up to 10 pages, in PDF) on your findings where you can emphasize your evaluation of the systems (`project.pdf`). It should describe your experiment setting, results and, importantly, *some discussion* on which strategy you found more useful. Include the statistics for the tested programs. This report is to be submitted by 23:59 Sunday 7th April 2024.

Make sure your submission contains information about your name, matric number, and email. Your zip files should have the format *Surname-Matric*-asg3.zip. Submit all files above to the appropriate Canvas Submission Link.