SE576 Himanshu Gupta HW4 Reflection

Q1) We read a few papers on static analysis and related techniques (symbolic execution) that may have left you with a certain level of expectation for how "smart" a static analysis might be in general. How does your experience with Infer compare to your earlier expectations?

Ans) I have divided my experience into the pros and cons with Infer:

Pros:

1) I expect a good static analysis tool would be able to do some analysis on run-time situations and Infer does fulfill this requirement.
2) Even though, the installation process is hard for Windows/Linux users, the browser version of Infer (CodeBoad) seems to work well for small programs.
3) Although Infer is not very successful with data structures and heap memory, I can definitely see it be more useful in C where unlike Java, memory allocation is not automatic. I did test infer on a small C program where I have allocated space for my structure, but never freed it. Infer produced an error in this example.

Cons:

1) From the perspective of a Linux user, it does not work well and the whole installation process was full of traps making it nearly impossible to install it locally on Linux/Windows.
2) Most of the time, developers use System IO in the programs. In my experience, there are many situations where Infer produced False negatives and most of the cases are based on some mathematics calculations for example:

```
// if (Math.random(5) > 4) stream.close;
/* Infer produced no issues in the above example, although there is a very
high probability of producing a run-time error in this example. */
```

3) Infer is based upon data flow analysis which in practice makes it very brittle in many situations, for example, one made SQL table for storing int and somehow stored a null, followed by manipulation of int value using built-in functions. This would be passed by the Infer because value type is int and the built-in functions are eligible to take that value. However, in the execution, this example would fail.

Q2) Consider the approaches you took for your false positive and false negative (i.e., what approach did you take to "confusing" the analyzer?). How does this compare to the strengths and weaknesses of systems like KLEE and SAGE? Are they the same, different, or hard to tell?

Ans)

Sage produced good results for the run-time errors while Infer fails to understand them nicely.

For Producing False Positive,

I have used an approach of loops for confusing the program. Basically, initializing a variable value to be null, then change that variable value to something valid inside of a loop at a very specific value of the loop variable. Infer is based upon the data flow analysis which fails to recognize the change in the variable value. Both the KLEE and the Sage are based upon the

dynamic symbolic execution which dynamically generated tests for covering every possible path condition in the code. It is very likely because of the symbolic execution tree, both the other systems are not going to produce False-positive like Infer. Also, both KLEE and the Sage intent to find the tricky boundary condition code by eliminating the values given in code and generate inputs themselves for those conditions catching more bugs with more precision. On the other hand, Infer never intended to be absolute, it does work well with the other more sophisticated tools.

For Producing False Negative,

I have used an approach to complex data structures. As the Infer is based upon the Data flow analysis, it should not be able to read the values stored for the data types. To implement this approach in code, I have created a hashmap of <String, Integer> and put <" list", null> in my map. Infer analyses the map and marks it good based upon the value data type to be Integer, it does not go inside to check if it is null or not. This creates an error when I assign a primitive int with my value(null) stored in the map. Infer passed that without any trouble while the program fails to run. However, KLEE and Sage work with generating their own range of inputs for the path formed in the execution tree using dynamic symbolic execution. This ensures that KLEE or Sage would test our code with the value "null", it is going to fail on their execution. This is how they going to get rid of false negatives.

Q3) How does Infer's intended usage compare to what you understand of Coverity? Do some of Coverity's lessons seem like they would apply to Infer in the long run? Has Infer seemingly already adopted some of those lessons? Do some of the lessons seem like they may not be relevant to Infer?

Ans)

The first main difference is the purpose of both Infer and Coverity are different. Infer is an open-source free for everyone static analysis tool, while Coverity is commercially made tool for the case to case implementation based on the environment requirements. With the purpose so different, there might be some areas of overlap, but in general, Coverity is a more sophisticated tool made for handling big chunks of code, run them for indefinite cycles, producing reports which are more descriptive than those produced by Infer. On the other hand, Infer tends to not absolute tool, it is complementary to other tools, maybe using Infer before running big programs on Coverity like tools would give one an idea about the code quality in a very quick time.

Some laws of the Coverity overlaps with that of Infer. The law, "You can't check code you don't see", is very adapted in the Infer with the Infer to pick the code during the build process itself. The law, "You can't check code you can't parse" of the Coverity bug finding seems to flow the same challenge into Infer or any static analysis tool. With the companies having legacy systems and using old technologies with newer specifications, there would be code that can't be parsed and tested using tools. Infer not only face this problem for older systems, but also for newer languages complex data structures and heap flows. Overall, I would also like to add the myth mentioned in the Coverity paper that More analysis is always good. Automated error detections should be easy to diagnose, and leave the hard part manually.