CIS 700 (Fall 2017) Project Milestone 2

Robert Zajac (rzajac), Graham Mosley (gmosley)

1. What is your project trying to accomplish?

No change from milestone 1.

2. What have you done so far in the project?

During the last milestone, we spent most of our time learning about the available static analysis tools as well as gathering sample programs to use in our evaluations. After further review of the programs we collected, we decided to focus on the benchmarks created by the Toyota InfoTechnology Center USA. These benchmarks contain multiple tests for various types of program analysis. The benchmarks also have “correct” versions of the tests in order to test false positives.

We looked through through the source code code of many static analysis tools like flawfinder, cppcheck and infer in order to get a better understanding of how they work. Flawfinder is quite simplistic and consists of a single python file that reads in a c source file and looks for common vulnerabilities using regex. Cppcheck is more complicated. It’s written in C++ and uses a combination of xml rule definitions and and regex parsing in order to find bugs. Infer is the more complicated tool and uses more powerful techniques including it’s own internal representation and abstract interpretation for analysis.

We ran our selected static analysis tools on the benchmarks and collected the results. Unfortunately, the output formats the tools create are vastly different from one another. Additionally, each tool is run in a different way. For example, cppcheck and flawfinder just scans the source files, but Facebook Infer hooks into the Makefile during compilation. It would nice if we could automate the process of running multiple different static analysis tools on the same code samples as well as converting the output to a standard format.

Our progress is in a Git repository (https://github.com/gmosley/cis700sa-project).

3. What challenges have you encountered?

We have run into some challenges getting these tools to run on general programs. We've found that the tools each require a different configuration and build in order to work properly. In particular, some tools demand a certain structuring of the source directory, while other tools require some instrumentation of the compiler. In addition, most of these tools can be run in several different modes/configurations, each of which may find different classes of bugs. It's not immediately clear which mode we should use for each tool to find the kinds of bugs that we're interested in. Furthermore, each tool has its own output format that we have to parse, since we don't have tooling to do this automatically. In order to make our study consistent we have to reconcile these different output formats and compile results into a consistent format that will let us present and analyze results more clearly, and this has proven to be challenging.

In an attempt to get around these challenges we have tried to develop some scripts/tooling to automate running the tools and analyzing their outputs. We are trying to do this in an attempt to make our study easily reproducible and extensible. This has proven to be challenging because of the issues described previously - namely, the variability in configuration and output format of tools.

We have also been attempting to look at the source code of these tools to figure out what features of them are most important for finding security bugs, as described in our previous milestone submission. In particular we have looked at the source code of Flawfinder, CPPCheck, and Facebook Infer. We are running into trouble because the codebases for these tools are quite unfamiliar to us and generally they are quite large and obfuscated. Our goal is to distinguish the key features of these tools, but we are finding that to do that we need to understand their whole codebase first, which is taking considerable time and effort. One feature

that we have managed to identify is that CPPCheck and Flawfinder seem to be regex-based whereas Infer seems to be building an IR of code and analyzing it. We think this will incur some additional challenges because we will have to get up to speed on these two different strategies of static analysis and try to understand how they can detect security bugs.

4. What do you plan to do by the next milestone?

Since the next milestone is the final submission, we plan to finish the project. In particular this will involve:

- Developing a final strategy for running these tools, potentially supported by scripts and other systems to automatically instrument and run the tools.
- Developing a common output format to unify the output formats of each of the different tools.
- Determining a subset of the source code of these tools that we can read in-depth and understand. This subset should be sufficient for explaining some differences between the classes of bugs that the tools find so that we can report it in our submission.
- Writing a final report detailing: the tools we used, the programs we analyzed, the steps we took to conduct the analysis, and our results (i.e. how good are these tools at finding security bugs, as well as what are the differences between these tools and why).

5. Do you have any additional comments?

We haven't heard any feedback on our proposal or milestone 1 submission, so we're still not sure if we're on the right track. We hope that we are!