Case Study: Static Security Analysis of the Android Goldfish Kernel

Tao Liu[1] and Ralf Huuck[2]

[1] University of New South Wales, Sydney, Australia
tao.liu4@unsw.edu.au
[2] NICTA* and Red Lizard Software, Sydney, Australia
ralf.huuck@nicta.com

Abstract. In this work we present an industry-driven case study of applying static program analysis to the Android kernel. In particular, we investigate the ability of open source tools as represented by *Cppcheck* and of commercial tools as represented by *Goanna* to detect security vulnerabilities. In our case study, we explore static security checking along the dimensions of setup effort, run time, quality of results and usability for large code bases. We present the results we obtained from analyzing the Android Goldfish kernel module of around 740 kLoC of C/C++ code. Moreover, we highlight some lessons learned that might serve as a guidance for future applications.

1 Introduction

The Android operating system as developed by Google has reached universal prominence as the most popular OS for mobile devices. More recently Android is advancing into adjacent domains including automotive, medical devices and home automation systems.

Given Android's ubiquitous presence the operating system's overall security is deemed to be paramount. However, while the design of Android has been relatively stable over time, firmware and drivers are constantly changing leading to different software versions on almost a monthly schedule. Given the size of the Android OS with millions of lines of code spread over $10,000$ files, managing the security implications manually is a arduous task. As such, many organizations either trust new updates and hope to detect any anomalies during integration or system testing, or they rely on complimentary automated tools such a *static program analysis* [1] to achieve a minimum level of assurance.

This work is a case study of using static program analysis tools for security checking of the Android kernel. It was developed in conjunction with the static analysis tools company Red Lizard Software and driven by their customers in the telecommunication and entertainment devices market. The goal of the case

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

study was to evaluate the overall lifecycle effort and benefits of using static analysis tools for security checking. This includes: setup and integration effort, run time and analysis bottlenecks, ease of evaluating the results and quality of the results. Moreover, the case study involved Red Lizard Software's own tool *Goanna* [2] as well as the popular open source tool *Cppcheck*[3].

In the following we give an introduction to the tools and environments used, explain the results we obtained and give a summary of our observations that might be helpful for others in similar circumstances.

2 Experimental Setup and Evaluation

Tools. We choose two static analysis tools for our case study representing two different classes of software checking tools: *Cppcheck*, a free GPL licensed tool for checking generic problems in C/C++ code including memory leaks, out of bounds arrays, and null pointer dereferences. Cppcheck aims at having zero false positives. The second tool, *Goanna*, is a commercial static analysis tool developed around software verification techniques such as model checking, SMT solving and abstract interpretation. Its aims at deep analysis that is scalable to large code bases.

The purpose of this evaluation is not to decide on a better tool, but to explore differences and strengths of each tool. Moreover, the tools where chosen based on their availability to the authors and the good reputation of Cppcheck. While Goanna is representing the commercial tool space, we would not expect fundamental differences from similar commercial tools. Further comparisons can be found in the NIST SAMATE program [3] and earlier evaluations [4].

Android Kernel. As a test bed the Android Goldfish 3.4 kernel was chosen. This is a generic kernel for software emulation of various hardware platforms of interest to Red Lizard Sofwtare's customers. The Goldfish kernel contains the essence of the recent Android releases KitKat and Lollipop and has around $740k$ lines of code.

Configuration. The analysis was run using Goanna 3.4.1 and Cppcheck 1.68. We focused on the security relevant checks for each tool including buffer overflows, null pointer issues or tainted data. This resulted in 27 specific categories for Cppcheck and 46 for Goanna. All experiments were run on a quadcore Dell PowerEdge 1950 2.66GHz with 16 GB of RAM running Ubuntu 12.10.

2.1 Evaluation Results

Installation and Configuration. Both Cppcheck and Goanna have been straight forward to install. Cppcheck comes as a drop-in binary and Goanna has an installer file. Configuring Cppcheck required running all check with subsequent filtering results as there is no method to select individual checks. For Goanna the defaults of the security package were selected.

[3] `http://cppcheck.sourceforge.net/`

Running the Analysis. Goanna parses the source code and does handle includes, preprocessing and macro expansion. It gets the necessary information after monitoring the native build process for relevant compiler and linker calls. In contrast, Cppcheck does not fully parse the code, but scans the actual text of the source files. As a result it be run on any file or from a directory. We used the Goanna facility to determine the relevant set of files in the build and passed those files to Cppcheck. The overall run time for Cppcheck on the Android Goldfish 3.4 kernel was just under 10 minutes, while the same run took 75 minutes for Goanna. Compilation itself is slightly over 8 minutes.

Quantitative Results. Goanna reported 279 potential issues from 14 categories out of the total 46 security checks[4]. Cppcheck reported 37 potential issues in 3 categories (`Null Pointer` *warnings* and *errors*, and `Memory Leaks`) for the same code base. Additionally we made use of Cppchecks ability to run over any file even if not in the build to include all driver files for all platforms in a second experiment. For the latter Cppcheck reported 755 issues in 10 categories.

We manually evaluated how many of the reported issues are false positives, i.e., where the tool spuriously warns, to determine the actual true positive (TP) rate. We used a random sample size of 20 for each category of each tool. Only 4 categories in of each tool had more than 20 warnings. Only 1 category more than 100 warning and was as such under-sampled. We did not have any means to determine the false negative rate, i.e., the number of issues that are in the code and are missed.

Firstly, for the 37 issues Cppcheck detected in the actual build the `Null Pointer` errors had a 67% TP rate, while the weaker `Null Pointer` warnings had a 21% TP rate. The `Memory Leak` issues were all false positives. Only about one third of all reported issues were true positives. However, for the second experiment scanning all remaining kernel files Cppcheck had a much better TP rate of 76% when averaging out over the reported 10 categories. Only only 2 categories had less than 50% TP rates and 6 categories at TP rates of 90% or higher.

Secondly, for Goanna's 279 reported issues in 14 categories the lowest TP rate was of any category was 67% (two of three bugs were correct) and 11 categories had a TP rate of 90% or higher. Averaged out over the 14 categories Goanna's TP rate was 94%. We were advised by Red Lizard Software, however, that this TP rate was better than what usually should be expected from such a tool.

Thirdly, we examined the overlap between Goanna's 279 reported issues and Cppcheck's 37. Only a combined 23 issues were in files, where both tools reported a warning. For those files Cppcheck reported 11 warnings with 9 false positives and Goanna 12 warnings with 3 false positives. No two of the same false positives were reported by both tools, but all true positives of Cppcheck in the overlap were also correctly reported by Goanna. Outside the files where both tools reported and issue there were 8 TPs by Cppcheck not reported by Goanna, the remaining Goanna TPs were not reported by Cppcheck.

[4] all raw data available at `http://www.cse.unsw.edu.au/~rhuuck/fm15`

Qualitative Results. Generally, the cause of bugs differed a lot between the two tools. Cppcheck's bugs tend to be patterns such as a constant 0 that is used later as a divisor or an explicit Null pointer being referenced close by. The Goanna bugs tended to be much deeper involving computation of data or passing of values between functions. At the same time it took the evaluator much longer to determine the truth of a Cppcheck warning as the tool outputs a line number with a rather simple message. Goanna additionally provided a compact trace through the program with explanatory text, making an assessment easier. One of the easier to rectify drawbacks of Cppcheck was the absence of a reference manual, leading to some steep learning curve and a degree of guesswork for causes of claimed errors.

3 Lessons

There are a few key observation from our case study: Both tools where simple to setup, had reasonable sets of security checks and had no problems scaling to the Android kernel as such. Also, both tools had medium to very low rates of false positives rates making them applicable in practice. We suspect that the Android code's maturity makes it easier to understand for tools as well.

Major differences are: The Cppcheck issues were much shallower than the Goanna issues, although still relevant. A cause is Cppcheck's absence of full parsing and reliance on pattern matching instead. This has the advantage though of easy applicability and the ability to scan all files even outside the build. In contrast, Goanna analyzes what is being build, which has its pros and cons.

As a summary, we believe that Cppcheck is a useful first line of defense with very low run time overhead. Its key role is quick scanning of code under development. Goanna or similar commercial tools appear much more comprehensive in its analysis suitable for higher levels of assurance, while also requiring slightly more run time.

While we were able to asses the check results for both tools, we do not know the implications of the bugs we found. Currently, we are in the process of feeding our findings back to both the Android developers as well as Red Lizard Software and its customers.

References

1. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer (1999)
2. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. TASE '07, Washington, DC, USA, IEEE Computer Society (2007) 45–56
3. Okun, V., Delaitre, A., Black, P.E.: Report on the Third Static Analysis Tool Exposition (SATE 2010) . Technical report, NIST (2011) Special Report 500-283.
4. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci. **217** (July 2008) 5–21