Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities II (Extended Abstract)

George Chatzieleftheriou, Apostolos Chatzopoulos, and Panagiotis Katsaros

Department of Informatics, Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{gchatzie,aachatzop,katsaros}@csd.auth.gr

Keywords: static analysis, software security, benchmark tests

A large number of tools that automate the process of finding errors in programs has recently emerged in the software development community. Many of them use static analysis as the main method for analyzing and capturing faults in the source code. Static analysis is deployed as an approximation of the programs' runtime behavior with inherent limitations regarding its ability to detect actual code errors. It belongs to the class of computational problems which are undecidable [2]. For any such analysis, the major issues are: (1) the programming language of the source code where the analysis is applied (2) the type of errors to be detected (3) the effectiveness of the analysis and (4) the efficiency of the analysis.

In order to incorporate a static analysis tool for detecting potential code defects in the software development cycle, significant costs are required. Thus, it is important to know if such a tool is effective in finding all types of errors and especially the critical ones for ensuring product quality. It is also a matter of major importance to know how efficient a tool is with respect to the size of the code bases being analyzed. When two or more static analysis tools are compared based on code bases from existing software projects the results are biased: they actually refer to the tools' capability to detect only the defects within the tested code bases, which are characterized by a specific degree of program complexity and size. We believe that empirical studies on open source programs should be completed with evaluation results, which cover systematically the most frequent code defects in a specific software context.

The main focus of our work [1] is on software security and reliability. We have created a versatile test suite, which implements code defects for the C programming language. Our test suite is based on those errors which are more often reported in public catalogs. We have identified major defect categories, such that all examined defects are classified in one of them. Category *General* includes three types of flaws, namely division by zero, use of uninitialized variables and null pointer dereference. The second category, *Integers*, includes integer overflows, sign and truncations errors. Direct overflows, off-by-one errors and unbounded copies appear in categories *Arrays* and *Strings* along with the format string vulnerabilities [8], string truncation and null termination errors. Many frequent C code defects are presented in category *Memory*, such as double free attempts, improperly allocated memory, initialization errors, memory leaks, absence of failure checks and access in previously freed memory. Category *File operation* contains the errors of redundant file closure, omission of file closure, absence of failure check and access in a file that is either, previously closed, not opened or opened with a different mode. Last but not least, category *Concurrency* errors includes deadlocks and time-of-check-time-of-use (TOCTOU) errors.

Our methodology aims to systematically vary analysis requirements in order to detect the mentioned code defects, and assess the static analysis tool effectiveness in a wide range of potential coding complexities. Our publicly available test suite consists of 750 programs for 30 distinct code defects from the mentioned categories. All programs include one line with the tested flaw and another line of code used to check the tools' capability to avoid reporting spurious errors. The test suite was applied to four open-source [3] [4] [5] [6], and two commercial tools [7], whose effectiveness was measured using metrics such as accuracy, precision, recall, specificity and F-measure. Accuracy is the ratio of correct classifications over the total number of observations. Precision is the ratio of the number of true positives over the number of reported errors. Recall is the ratio of the number of true positives over the number of actual errors. Specificity is the ratio of the number of true negatives over the sum of true negatives and false positives. The F-measure provides an aggregate measure for precision and recall, two metrics that are characterized by an intrinsic tradeoff. We also measured the tools' efficiency in terms of running time and peak memory usage.

We have evaluated the tools' effectiveness based on a wide range of C constructs and different conditions of language semantics under which the defects may arise. Each defect is reproduced in many different programs, which are used to assess the default configuration of the static analysis tools with respect to their path sensitivity, context sensitivity and alias analysis capabilities. The test programs for analyzing the tools' efficiency were automatically generated such that for each case of different program size between 1000 and 7000 lines of code, three programs with different analysis sensitivity requirements are considered, namely path sensitivity, context-sensitivity and alias analysis.

The main outcome from test driving the referenced static analysis tools showed that only one open-source tool can compete and in fact was found superior over the commercial ones, in terms of precision. On the other hand, the tested commercial tools had a higher recall compared to all tested open source tools. This finding shows that their analyses are designed and configured, such that they are able to detect as many defects as possible with slightly lower precision than the tool described in [6]. However, the higher precision of the open-source tool is accompanied by a significant cost in analysis efficiency: for test programs with 7000 lines of code the average analysis running time was more than two times the average running time of the commercial tools.

Our methodology can be easily extended towards diverse quality contexts and software domains, and can be enriched for tool comparisons for other programming languages. As an interesting scenario, we consider its application for validating runtime safety of applications for a mobile computing platform. Such a type of validation is often a formal requirement for the distribution of applications through internet-wide markets and the procedure usually requires certification based on platform-specific security needs.

The results show how the evaluated tools compete in terms of important tradeoffs between analysis effectiveness and efficiency, as well as between precision and recall. The degree of extensibility and customization that each tool offers to the user should also be taken into account. In the last few years, the theory and the technology of static program analysis is rapidly developed and the market's driving forces call for new ways to balance the discussed tradeoffs between analysis effectiveness and efficiency. For this reason, we believe that there is an undeniable need to regularly repeat and publish every few years systematic studies such the one reported in [1].

Acknowledgment

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

References

1. George Chatzieleftheriou and Panagiotis Katsaros. 2011. *Test-Driving Static Analysis Tools in Search of C Code vulnerabilities.* In Proc. Of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW '11).
2. William Landi. 1992. *Undecidability of static analysis.* ACM Lett. Program. Lang. Syst. 1,4 (December 1992), 323-337.
3. David Evans and David Larochelle. 2002. *Improving Security Using Extensible Lightweight Static Analysis.* IEEE Softw. 19, 1 (January 2002), 42-51.
4. Gerard J. Holzmann 2002. *Static source code checking for user-defined properties.* Proc. IDPT. Vol. 2 2002.
5. *Cppcheck A Tool for static C/C++ static code analysis.* Available: http://sourceforge.net/apps/mediawiki/cppcheck
6. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. *Frama-C: a software analysis perspective.* In Proceedings of the 10th international conference on Software Engineering and Formal Methods (SEFM'12), George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer-Verlag, Berlin, Heidelberg, 233-247.
7. *Parasoft C++ Test*, Available: http://www.parasoft.com/
8. Aleph One. *Smashing the stack for fun and profit.* Phrack magazine 7.49 (1996): 14-16.