Parfait – A Scalable Bug Checker for C Code Tool Demonstration

Cristina Cifuentes
Sun Microsystems Laboratories
Brisbane, Australia
cristina.cifuentes@sun.com

Abstract

Parfait is a bug checker of C code that has been designed to address developers' requirements of scalability (support millions of lines of code in a reasonable amount of time), precision (report few false positives) and reporting of bugs that may be exploitable from a security vulnerability point of view. For large code bases, performance is at stake if the bug checking tool is to be integrated into the software development process, and so is precision, as each false alarm (i.e., false positive) costs developer time to track down. Further, false negatives give a false sense of security to developers and testers, as it is not obvious or clear what other bugs were not reported by the tool.

A common criticism of existing bug checking tools is the lack of reported metrics on the use of the tool. To a developer it is unclear how accurate the tool is, how many bugs it does not find, how many bugs get reported that are not actual bugs, whether the tool understands when a bug has been fixed, and what the performance is for the reported bugs.

In this tool demonstration we show how Parfait fairs in the area of buffer overflow checking against the various requirements of scalability and precision.

1. The Parfait Framework

Parfait was designed to statically find bugs in large code bases [1]. Its design features (1) fast and precise bug checking by reducing the problem space with fast analyses applied first, (2) demand driven analyses, facilitating parallelization on multi-core computer architectures now available, and (3) an optional pass to make the analysis relevant to security vulnerability checking. In this demo, we look at preliminary work in these three aspects.

For better precision in bug checking we use an *ensemble of sound static program analyses*, by exploiting the strengths of the individual analysis. The program analyses in the ensemble should range in complexity and expense, and we define an order among them. For two program analysis P and Q, we express that P is less (time) expensive than Q by writing $P < Q$.

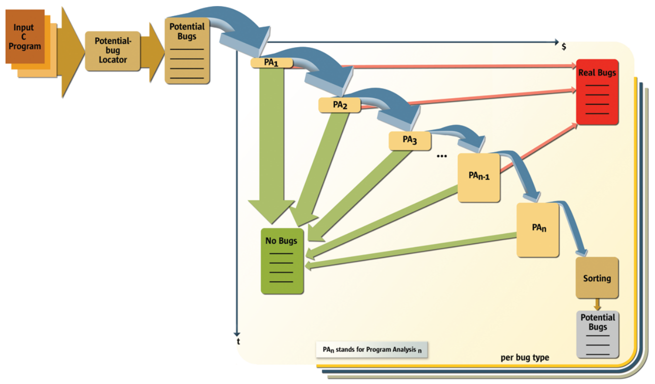

Figure 1. The Parfait Framework

With an ensemble of program analyses, we can state a kind of complexity property for a buggy statement in the program. The complexity property gives an indication of how complex (according to our order) the analysis needs to be for identifying it. The use of an ensemble achieves better precision with smaller runtime overheads, i.e. a buggy statement is detected with the cheapest possible program analysis in the ensemble.

Figure 1 shows our framework. Our algorithm works as follows: First, a worklist for a specific bug (e.g. buffer overflow) is set up and populated with statements that potentially can cause the bug. Second, we iterate over the program analyses in the ensemble in ascending order. With the selected program analysis, we analyze the statements in the worklist. We remove statements from the worklist for which the analysis can prove that there is no bug. For a partially complete analysis [1] , the statement will be removed if the analysis proves that there is a bug and this bug is reported as a real bug. Third, the remaining statements in the work-list are reported as potential bugs and need to be verified by the developer/tester. Heuristics are applied to order the importance of the reported bugs.

To overcome the computational program analysis bottle-neck, we parallelize the algorithm by employing demand driven program analysis instead of traditional forward analyses for the whole program. Demand driven analysis generates a backward slice of a program starting at a particular statement of interest (in our case, a potential buggy statement). The algorithm has two levels of embarrassing parallel problems. The first level is the various worklists for specific bugs (e.g. buffer overflows, string vulnerabilities, integer overflows, etc.) and the second level is for statements in a specific worklist.

2. Evaluation Methodology

As far as we are aware of, there is no established evaluation methodology for bug checking tools. In the research community, a few first attempts have been made to consolidate this problem [6, 2, 3, 4], however no general consensus has yet been reached. Limited results such as *"Tool/Technique X found Y number of bugs."* are frequently presented in the literature. Such statements lack practical use as they fail to address any of the following questions:

- Which kind of bugs were found? (there exists a lot of bugs in any large software project, but which ones are considered relevant to the developer or security auditor?)
- How many bugs were not found? Note that Y could be a small number in comparison with the real total number of bugs in the code.
- How many bugs were reported as bugs and were not actual bugs?
- Does the tool understand when a reported bug has been fixed? (i.e., when the tool is re-run with a fixed bug, is the bug reported again or not?)
- How long did it take to run the tool to find the bugs?

We use a simple scheme that gives a "bug specific" view. For a specific bug class (e.g. buffer overflow bugs, signed/unsigned bugs), we conduct an automated evaluation. The evaluation contains the measurement of functional metrics for each benchmark including false positives and false negatives.

[1]A program analysis is said to be complete if it can report only real bugs in the program (i.e., it does not report on false positives). A program analysis is partially complete if it is complete for a sub-set of statements in the program.

Besides functional metrics we are also interested in the execution time of the tool, i.e., how many lines of code it can process per minute, and how much memory is needed for the execution of the tool. Speed and memory consumption are important to be able to extrapolate the scalability of the system.

As part of our evaluation methodology, we are collecting kernels of sample proprietary and open source buggy source code and annotating it with the bugs that are known to exist in that code. We also use the synthetic benchmarks that have been contributed to the SAMATE project [4].

3. The Tool Demo

In this demo we show how the Parfait framework can be used to detect buffer overflows, using several layered analyses. For each analysis, we show its performance, and the types of buffer overflows that it can find. We also demo preliminary work on a (security) pre-processing filter to Parfait that finds user-input dependencies in a program and scales well to large code bases of millions of lines of code [5]. Finally, we show our benchmarking infrastructure with a combination of examples from the SAMATE project and kernels from existing buggy open source code.

References

[1] C. Cifuentes and B. Scholz. Parfait – designing a scalable bug checker. In *Proceedings of the ACM SIGPLAN Static Analysis Workshop*, 12 June 2008.

[2] K. Kratkiewicz and R. Lippmann. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[3] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bug-Bench: A benchmark for evaluating bug detection tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[4] NIST SAMATE – software assurance metrics and tool evaluation. `http://samate.nist.gov`. Last accessed: January 2007.

[5] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *Proceedings of the Eighth IEEE Working Conference on Source Code Analysis and Manipulation*, September 2008.

[6] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press.