# Program Analysis for Bug Detection using Parfait

## Invited Talk

Cristina Cifuentes Nathan Keynes Lian Li Bernhard Scholz*

Sun Microsystems Laboratories
Brisbane, Australia
{cristina.cifuentes,nathan.keynes,lian.li}@sun.com

## Abstract

The goal of the Parfait project is to find bugs in C source code in a scalable and precise way. To this end, Parfait was designed as a framework with layers of sound program analyses, multiple layers per bug type, to identify bugs in a program more quickly and accurately.

Parfait also aims to identify security bugs, i.e., bugs that may be exploited by a malicious user. To this end, an optional preprocessing step is available to reduce the scope of potential bugs of interest.

To evaluate Parfait's precision and recall, we have developed BegBunch, a bug benchmarking suite that contains existing synthetic benchmarks and samples of bugs ("bug kernels") taken from open source code.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.8 [*Software Engineering*]: Metrics

***General Terms*** Verification

***Keywords*** Constant propagation, Partial evaluation, Symbolic analysis

## 1. Introduction

Recently, sophisticated bug checking tools that use a variety of program analyses and verification methods have become available, with greater precision than that obtained by tools such as lint (Johnson 1978). Current tools check for bugs using an abstraction or a model of the program, either one procedure/method at a time (i.e., intra-procedurally) or globally (i.e., inter-procedurally). However, the noise level of these tools is still high in practice: each reported bug needs to be manually verified to determine if indeed it is a bug (i.e., a true positive) or not a bug (i.e., a false positive), resulting in much time and effort wasted in validating the results of the tool. *Precision* (the number of true positives divided by the total number of reported bugs) is an issue.

Despite the speed of today's processors, advanced program analyses and verification methods remain too computationally expensive to be able to analyse large code bases in a timely fashion. A tool that requires more than a week to run over a large code base cannot be integrated into the day-to-day software development process. *Scalability* is an issue.

Users of bug checking tools may use filters to deal with false positives, but they have no way of knowing about the tool's false negative rate (i.e., what bugs exist in the code, but are not reported by the tool). Lack of this information may lead to a false sense of security. *Recall* (the number of true positives divided by the total number of bugs in the code) is an issue.

Software that is online constantly is especially vulnerable, as (well-known) attacks to the software may uncover an unknown bug and exploit it to bring down the software, obtain unauthorised access, etc. *Security* is an issue.

## 2. Parfait Design

The Parfait design addresses the requirements of precision, scalability, and recall (Cifuentes and Scholz 2008) by defining an extensible framework composed of layers of program analyses.

For greater precision and scalability in bug checking, the design employs an ensemble of sound static program analyses that vary in complexity and expense. Analyses are ordered from least to more (time) expensive, ensuring that each buggy statement is detected with the cheapest program analysis capable of detecting it, effectively achieving higher precision with smaller runtime overheads.

For recall, Parfait compiles a sound list of potentially buggy statements per bug type for the framework to process. Each sound analysis can then determine one of three things about a potential buggy statement: it can confirm it is a bug, it can reject it as a non-bug, or it cannot tell either way. There are three lists attached to these three results: real bugs, no bugs, and potential bugs. After the ensemble of analyses is run, the user can see the real bugs list (which has high precision) and the potential bugs list (to get an idea of the recall rate; these results need to be manually verified by the user).

To overcome the computational program analysis bottleneck, we employ demand driven program analysis instead of traditional whole program analyses. The demand driven analyses effectively generate a slice of a program starting at each potential buggy statement, limiting the scope to a small and relevant subset of the code.

### 2.1 Layers of Program Analyses – An Example

To detect buffer read/write overruns (i.e., buffer overflows and read outside array bounds), three analyses have been implemented in Parfait: constant bounds checks, partial evaluation, and symbolic analysis; all intra-procedural at present.

* While on sabbatical leave from The University of Sydney, scholz@it.usyd.edu.au


*PEPM'09,* January 19–20, 2009, Savannah, Georgia, USA.
ACM 978-1-60558-327-3/09/01.

For constant bounds checks we perform constant propagation and folding (Aho et al. 1986) to trivially check array references with constant indexes against valid array bounds.

For statements with statically-computable dependencies on an array index, we use partial evaluation (Futamura 1971) techniques to specialize a slice of code around the buggy statement with the constant inputs, and execute the slice in an interpreter to determine whether the index is within valid bounds or not.

For other statements we use symbolic analysis (Cheatham et al. 1979) to compute the symbolic range of variables. The symbolic range of an array index variable will then be compared with the array bounds to determine whether or not the array reference is out of bounds.

In tests over the 6-million line Solaris™ Operating System ON (Operating system and Networking) consolidation code base, these analyses have found more than 300 buffer read/write overrun bugs in under 30 minutes on an AMD Opteron 2.8 GHz with 16 GB of memory. These bugs have been manually confirmed to be real bugs and are being submitted into the bug tracking system for Solaris.

Full support for the C language requires support for the C library; an ISO standard that specifies well-defined semantics for the various functions in the library (WG14 2005). In Parfait, we make use of pre-conditions to support checking of C library function semantics. Post-conditions will be added in the future.

## 3. Security Vulnerabilities

For security vulnerabilities (defined as those that can be exploited by an attacker), we have developed a pre-processing analysis pass over the Parfait framework to reduce the full set of potential buggy statements gathered by Parfait, to just those statements that are reachable from user input. This user-input dependence analysis over the whole program is mapped onto a reachability graph: a point in the graph is either reachable or not from external input (Scholz et al. 2008).

More complex definitions of security vulnerability require further layers of analyses to be put in place over the Parfait framework, to be able to detect vulnerabilities such as denial of service, weak passwords, etc.

## 4. BegBunch – Quantifying Precision and Recall

To compare bug checking tools, it is necessary to measure objectively their level of correctness. To do this, we have compiled two sets of sample bugged source (benchmark suites), and written a framework to run the tool on the samples (Cifuentes et al. 2009).

The suite is composed of synthetic (e.g., (Kratkiewicz and Lippmann 2005), (NIST 2006)) and real bug kernels extracted from open source projects (Zitser et al. 2004; Lu et al. 2005; Cifuentes et al. 2009). Each benchmark is annotated with the type and locations of bugs exposed in the benchmark. Given a bug checking tool's output, BegBunch's harness computes how many bugs were reported correctly (computing the precision rate), and how many bugs were missed (computing the recall rate). Currently, for buffer read/write overruns from the BegBunch suite, Parfait obtains 100% precision and 75% recall rates.

## Acknowledgments

Several students have contributed to the Parfait effort while on internships or as part of student projects with our group; we thank them all for their contribution: Erica Mealy, Chenyi Zhang, Simon Long, David Gwynne, and Jimmy Ti. We also thank Michael Mounteney who contracted with us to make BegBunch a reality. And, of course, we thank the LLVM (Lattner and Adve 2004) team for their infrastructure, upon which Parfait is built.

Thanks also to Scott Rotondo for providing requirements for a bug-checking tool for the Solaris OS, Sharon Liu and Chok Poh for providing requirements for the native part of the JDK™ software, Liang Chen for suggesting bug-checking as an interesting area to work in, and Douglas Walls for historical information on bug checking at Sun.

## References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley, 1986.

T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Eng.*, 5(4):402–417, 1979.

Cristina Cifuentes and Bernhard Scholz. Parfait – designing a scalable bug checker. In *Proceedings of the ACM SIGPLAN Static Analysis Workshop*, pages 4–11, 12 June 2008.

Cristina Cifuentes, Bernhard Scholz, Michael Mounteney, Erica Mealy, Nathan Keynes, and Lian Li. BegBunch: A benchmarker for C-source bug detection tools. Submitted for publication, January 2009.

Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1971.

S.C. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1978.

Kendra Kratkiewicz and Richard 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.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: A benchmark for evaluating bug detection tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

NIST. National Institute of Standards and Technology SAMATE Reference Dataset (SRD) project. `http://samate.nist.gov/SRD`, January 2006.

Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. In *Proceedings of the Eighth IEEE Working Conference on Source Code Analysis and Manipulation*, pages 25–34, 28-29 September 2008.

WG14. *ISO C 99 Standard - TC2*. ISO/IEC Working Group 14, 9899:TC2 edition, May 2005.

Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 97–106. ACM Press, 2004.