BegBunch – Benchmarking for C Bug Detection Tools

Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li,
Simon Long, Erica Mealy, Michael Mounteney and Bernhard Scholz*
Sun Microsystems Laboratories
Brisbane, Australia
{cristina.cifuentes,christian.hoermann,nathan.keynes,lian.li}@sun.com

ABSTRACT

Benchmarks for bug detection tools are still in their infancy. Though in recent years various tools and techniques were introduced, little effort has been spent on creating a benchmark suite and a harness for a consistent quantitative and qualitative performance measurement. For assessing the performance of a bug detection tool and determining which tool is better than another for the type of code to be looked at, the following questions arise: 1) how many bugs are correctly found, 2) what is the tool's average false positive rate, 3) how many bugs are missed by the tool altogether, and 4) does the tool scale.

In this paper we present our contribution to the C bug detection community: two benchmark suites that allow developers and users to evaluate accuracy and scalability of a given tool. The two suites contain buggy, mature open source code; bugs are representative of "real world" bugs. A harness accompanies each benchmark suite to compute automatically qualitative and quantitative performance of a bug detection tool.

BegBunch has been tested to run on the Solaris™, Mac OS X and Linux operating systems. We show the generality of the harness by evaluating it with our own Parfait and three publicly available bug detection tools developed by others.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Experimentation

Keywords

Accuracy, scalability

1. INTRODUCTION

Benchmarking provides an objective and repeatable way to measure properties of a bug detection tool. The key to a good benchmark is the ability to create a common ground for comparison of different bug detection tools or techniques based on real, representative data that measure the qualitative and quantitative performance of the bug detection tool. The compilers community has a long standing history of performance benchmarks, and more recently the Java™ virtual machine community has established its benchmarks.

Benchmarks in the bug detection community have not reached the level of maturity needed to prove useful to a variety of users. A benchmark suite for bug detection tools must be able to answer several questions about the tool's results:

Question 1. How many bugs are correctly reported?
Question 2. How many false reports of bugs are made?
Question 3. How many bugs are missed/not reported?
Question 4. How well does the tool scale?

Not all these questions can be answered by a single suite. In order to measure scalability of a tool, large code distributions with million lines of code are needed to be representative of the real world. However, determining how many bugs are correctly and incorrectly reported, or how many bugs are missed, is infeasible for large code bases, because of practical limitations of finding where *all* bugs in a large program are.

In this paper we present BegBunch, a benchmark suite for C bug detection tools. Our contributions are two suites, the Accuracy and the Scalability suites, and associated harnesses. The Accuracy suite evaluates precision, recall and accuracy against marked-up benchmarks, while the Scalability suite evaluates scalability of a tool against a set of applications. BegBunch has been used as part of regression and system-testing of the Parfait [2] bug detection tool.

2. RELATED WORK

We review the literature with respect to existing efforts on bug detection benchmark suites. Table 1 reports on the number of programs, bugs, lines of code (minimum, maximum and average), harness, language and platform support for existing bug detection benchmarks. In this paper, all reported lines of code are uncommented lines of code as generated using David A. Wheeler's SLOCCount [14] tool. As seen in the table, the bug detection community has focused on two types of (accuracy) benchmarks.

Small benchmarks come in the form of bug kernels and synthetic benchmarks, with sizes from ten to one thousand lines of code. *Bug kernels* are self-contained programs extracted from existing buggy code that expose a particular bug. As such, bug kernels preserve the behaviour of buggy

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

DEFECTS'09, July 19, 2009, Chicago, Illinois, USA.

Size	Benchmark	# Programs	# Bugs	LOC			Harness	Language	Multiplatform
				min	max	avg			
Small	Zitser	14	83	175	1.5K	657	No	C	Yes
	Kratkiewicz	291 x 4	873	6	27	14	No	C	Yes
	SAMATE	375	408	20	1.1K	90	No	C,C++,Java,PHP	Yes
Large	BugBench	10	19	735	692K	149K	No	C	Linux
	Faultbench v0.1	6	11	1,276	25K	7K	No	Java	Yes

Table 1: Characteristics of Existing Bug Detection Benchmark Suites

code. Zitser et. al. [15] extracted 14 bug kernels from 3 security-sensitive applications. Kratkiewicz [6] automatically generated 291 synthetic benchmarks that test 22 different attributes affecting buffer overflow bugs. 4 versions of each benchmark were made available: 3 with different overflow sizes and a correct version of the code. The NIST SAMATE Reference Dataset (SRD) project [11] compiled a suite of synthetic benchmarks, mainly for the C, C++ and Java languages, that have been contributed by various groups.

Large benchmarks in the form of complete program distributions were the focus of attention of BugBench [9] and Faultbench v0.1 [4], with sizes varying from the low to high thousands of lines of code. BugBench is composed of 17 programs including 19 bugs that were known to exist in these programs. Faultbench provides a suite of 6 Java language programs with 11 bugs.

Our Approach

Table 1 points out the main shortcomings of existing bug detection benchmarks; namely,

- few bugs relative to the size of the programs,
- lack of a harness to run, validate and report data, and
- portability issues in some cases.

In other words, existing bug benchmarks are not general, portable or reusable; all key properties to making a benchmark suite useful.

BegBunch addresses these deficiencies by providing two suites to evaluate the qualitative and quantitative performance of bug detection tools and to automate the execution of the bug detection tool, the validation of the results, and the reporting of the performance data. For convenience to the general bug checking community, a third suite of existing synthetic benchmarks was also created to allow for the existing body of synthetic benchmarks to be more accessible and usable. The benchmark suites provide portability across Unix-based systems as they have been tested on the Solaris, Mac OS X and Linux operating systems.

3. BEGBUNCH: METHODOLOGY

BegBunch consists of various suites that allow tool developers and users to measure different aspects of a tool. We borrow terminology commonly used in the information retrieval community [13] and apply it to bugs instead of documents retrieved (or not retrieved):

- *precision* is the ratio of the number of correctly reported bugs to the total number of bugs reported,
- *recall* is the ratio of the number of correctly reported bugs to the total number of bugs (both correctly reported and not reported), and
- *accuracy* is a measure of the ability of the bug detection tool to report correct bugs while at the same time holding back incorrect ones.

		Bug Marked-up		
		Yes	No	
Bug Reported	Yes	TP	FP	Precision
	No	FN	TN	
		Recall		Accuracy

Table 2: Measurements Table

Based on bugs reported by a tool and bugs marked-up in a benchmark suite, Table 2 defines the terms true positive (TP, i.e., correctly reported bugs), false positive (FP, i.e., incorrectly reported bugs), false negative (FN, i.e., missed bugs) and true negative (TN, i.e., potential bugs that are not real bugs).

Precision and recall can be measured using standard equations. Given a bug detection tool, a bug kernel bk and a bug type bt, the tool's precision, p, and recall, r, with respect to bt and bk is defined as:

$$p_{bt,bk} = \begin{cases} \frac{TP_{bt,bk}}{TP_{bt,bk}+FP_{bt,bk}} & \text{if } TP_{bt,bk}+FP_{bt,bk} > 0 \\ 1 & \text{otherwise} \end{cases}$$

$$r_{bt,bk} = \begin{cases} \frac{TP_{bt,bk}}{TP_{bt,bk}+FN_{bt,bk}} & \text{if } TP_{bt,bk}+FN_{bt,bk} > 0 \\ 1 & \text{otherwise} \end{cases}$$

Accuracy can be measured in different ways based on Table 2. Heckmann and Williams [4] measure accuracy taking into account TP, FP, FN and TN. Theoretically, true negatives (TNs) can be measured on a per-bug type basis. For example, for buffer overflows, we can look at the code and determine all locations where a read or a write into a buffer is made. Except for the locations where a real buffer overflow exists, all other locations would be considered TNs. However, in practice, this measure is not intuitive and is hard to comprehend. Instead, we favour the F-measure from statistics and information retrieval [13], which computes accuracy based on precision and recall alone. The F-measure provides various ways of weighting precision and recall, resulting with different Fn scores. We favour the F1 score as it is the harmonic mean of precision and recall.

Given a bug type bt in a bug kernel bk, a tool's accuracy with respect to bt in bk is defined as:

$$accuracy_{bt,bk} = \begin{cases} \frac{2\times p_{bt,bk}\times r_{bt,bk}}{p_{bt,bk}+r_{bt,bk}} & \text{if } p_{bt,bk}+r_{bt,bk} > 0 \\ 0 & \text{otherwise} \end{cases}$$

A *bug benchmark* is said to *test* a bug type if it exposes that bug type or it provides a corrected version of it. The

Category	From	Bug Type	CWE_{id}	# kernels per category	LOC min	LOC max	LOC avg	# bugs intra:inter
Buffer overflow (BO)	Zitser	Buffer overflow (write,read)	120,125	14	90	618	304	5:78
	Lu	Buffer overflow	120	5	16	1,882	417	1:6
	Sun	Buffer overflow (write,read)	120,125	40	36	2,860	496	44:37
Memory/pointer (M/P)	Lu	Double free	415	1	5,807	5,807	5,807	0:1
	Sun	Null pointer dereference	476	2	35	41	38	0:4
Integer overflow (IO)	Sun	Integer overflow	190	3	47	64	58	0:3
Format string (FS)	Sun	Format string	134	2	30	94	62	0:2
Overall				67	16	5,807	481	50:131

Table 3: Characteristics of the Accuracy Suite of Bug Kernels

accuracy for the corrected version of a bug kernel is either 0 or 1 depending on whether the tool (incorrectly) reports a bug in the corrected version or not.

Given a suite of n bug benchmarks with b benchmarks that test one or more instances of bug type bt; $1 <= b <= n$; the overall accuracy of a tool over bt is the average of the individual benchmark results over the set b:

$$Accuracy_{bt} = \sum_{bm=1}^{b} accuracy_{bt,bm} \quad (1)$$

Questions 1-3 can be answered by measuring the true positive, false positive and false negative rates of a bug detection tool against a given benchmark suite. These raw data allow us to compute precision, recall and accuracy on a per bug type and bug benchmark basis. The overall accuracy figure, i.e., equation (1), provides an indication of the tool's accuracy with respect to a given bug type.

3.1 The Accuracy Suite

The aim of the Accuracy suite is to measure precision, recall and accuracy of a bug detection tool against a set of bug kernels extracted from existing buggy open source applications. The suite is intended to be representative of existing bugs in C/C++ code bases.

Bug Category	'04	'05	'06	Total	%
Buffer overflow	152	195	209	556	71.2%
Memory/pointer bug	13	15	7	35	4.5%
Integer overflow	34	25	61	120	15.4%
Format string	21	29	19	69	8.9%

Table 4: Summary of MITRE's OS vendor data for the years 2004-2006

To determine the ratio of bugs to be part of the benchmark suite, we used MITRE's data on reported vulnerabilities in operating system code for the years 2004-2006 [1] and internal feedback from Sun's product groups as to the most common types and category of bugs in systems code. This narrowed the set of bug types of interest to four main categories, as summarized in Table 4: buffer overflows, i.e., read and write accesses to an array outside its bounds; memory/pointer bugs, i.e., memory leak and double-free bugs; integer overflow, i.e., over or under flow of an integer; and format string, i.e., vulnerabilities arising as a consequence of tainted format strings.

We extracted bug kernels from existing open source applications such as the OpenSolaris™ operating system, the MySQL™ server, etc., by inspecting bug tracking systems and mailing lists, aiming to ensure there was a breakdown between easy, medium and hard to find bugs (measured by lines of code in the bug kernel and complexity of the code as represented by whether the bug relies on intra- or inter-procedural information). As pointed out in other forums, a benchmark suite that only holds hard cases is not very useful in practice [12] as it does not allow for meaningful comparison. Each bug kernel was tested with the gcc compiler to reproduce the bug in 3 different platforms: Solaris, Mac OS X and Linux, to ensure generality and completeness of the code.

The resulting Accuracy suite is the combination of our work, 47 bug kernels, combined with the 14 bug kernels from Zitser et. al. [15], and 6 bug kernels extracted from BugBench [9] that mapped to the above classification. Table 3 summarizes the data in our benchmark suite. For each bug category, we list the bug types that belong to the category along with their unique Common Weakness Enumeration (CWE) ID [10], the number of kernels, the code size based on uncommented lines of code, and the ratio of intra- vs. inter-procedural bugs. The table also summarizes the total number of bugs in the suite: 67 kernels in 4 bug categories with a total 181 bugs and a 50:131 intra:inter-procedural ratio.

3.2 The Scalability Suite

The aim of the Scalability suite is to measure how well a bug detection tool scales, i.e., answer Question 4. This suite is composed of open source applications that include a distribution of small to large code sizes in various domains.

Application	LOC	Domain
MySQL 5.0.51a	886,718	Database
Perl 5.8.4	428,595	Scripting language
Asterisk 1.6.0.3	290,079	Telephone PBX
MEME 4.0.0	216,976	Biotech
Sendmail 8.12.3	87,751	Systems
Tcl 8.0.5	70,377	Scripting language
OpenSSH 3.6.1p1	46,191	Systems
WU-ftpd 2.6.0	17,564	Systems

Table 5: Characteristics of the Scalability Suite

Table 5 shows the applications that belong to the Scalability suite. For each application, the number of uncommented lines of C/C++ code and the application's domain are reported. The suite ranges from systems-level code, to virtual machines, databases and general-purpose applications.

4. THE HARNESS

In order to measure accuracy and scalability, we developed a couple of harnesses that compute these figures based on the

output of a bug detection tool. The harnesses are written in Python and are extensible; they can support various bug detection tools.

4.1 The Accuracy Harness

Each bug that appears in the benchmark suite is annotated with an XML-like markup on the same line that *exposes* the bug, surrounding the relevant part of the statement containing the bug. The following syntax is used:

```
‘/* <bug’ [‘interproc’] <bug-type>‘> */’ <C/C++ block>
‘/* </bug> */’
```

where the default is for bugs to be intra-procedural unless otherwise stated. For example, to markup a `memset` statement that has a null pointer dereference that is based on an argument to the function, we add the following markup surrounding the `memset` at fault:

```
/* <bug interproc null-pointer-deref> */ memset(
  a->password, 0x55, strlen(a->password)) /* </bug> */;
```

Where a bug is exposed may be a matter of discussion. Given a bug kernel, it contains all the code of the application except for standard C libraries. A bug is exposed at the location where the error is first observed if the program was to be run. In the case of an intra-procedural bug, the bug is exposed within the procedure. In the case of an inter-procedural bug, the bug is exposed within the called procedure. In the case of bugs due to calls into C library functions, the bug is exposed at the call site, not within the library code, as the tool does not have access to the code in order to analyze it.

The Accuracy harness checks a bug detection tool's output against the marked up bugs in the suite and computes how many bug reports were correct (i.e., true positives), how many were incorrect (i.e., false positives) and how many were missed (i.e., false negatives), on a per bug type and benchmark basis. It then applies equation (1) to compute overall accuracy on a per-bug type basis.

4.2 The Scalability Harness

The Scalability harness allows for configuration of the various benchmarks and computes the time it takes to build the code, along with the extra overhead time to run the bug detection tool to analyze the code. Both these data are plotted, allowing users to see how well the tool scales over a range of small to large applications. The plot also gives an idea of how much time the bug detection tool takes to run beyond standard build time.

5. EVALUATION

We tested the extensibility of the BegBunch v0.3 harness with publicly available bug detection tools that support C and/or C++ code: Parfait [2] v0.2.1 (our own tool), Splint [3] v3.1.2, the Clang Static Analyzer [8] v0.175 and UNO [5] v2.13.

5.1 The Accuracy Suite

For each tool, a Python class that parses the output of the tool was written. On average, 100 lines of Python code were written for the abovementioned tools. Time was spent understanding the output produced by the various tools and trying to ensure that the reported data are representative of the tool. However, in some cases, some of the tools either did not support some of the bug categories in BegBunch or expected annotations in the source code of the benchmark suite in order to be more precise (e.g., Splint). We did not add annotations to the source code and used Splint with the strict mode.

Tool	Type	# TP	# FP	# FN	Accuracy
Parfait	BO	53	0	118	41.8%
Splint	BO	49	359	122	16.8%
Splint	NPD	0	1	4	0%
Clang	NPD	1	0	3	25.0%
UNO	BO	2	2	169	1.7%

Table 6: Evaluation of C bug detection tools against the Accuracy suite

Table 6 provides the results of our evaluation against the Accuracy suite. For each tool, data are reported against a bug type, summarizing the number of correctly reported bugs (TP), incorrectly reported bugs (FP) and missed bugs (FN), along with the accuracy rate. The bug types reported by some of these tools are: buffer overflow (BO) and null pointer dereference (NPD).

The table is meant to show the extensibility of the BegBunch harness rather than provide comparative data between the various tools. We realize that tools are written for different purposes and are at different levels of maturity. Further, bugs reported by other tools may use a different definition of where a bug is *exposed*.

5.2 The Scalability Suite

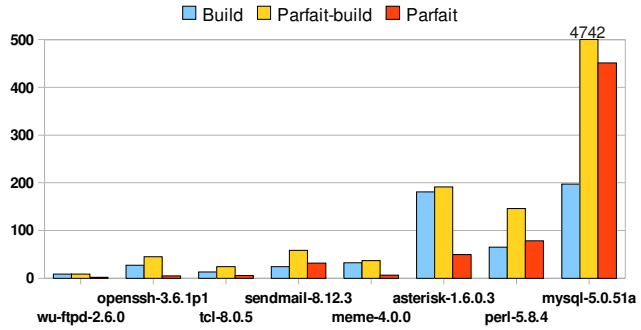

Figure 1: Time plot for Parfait running over the Scalability suite

The scalability of Parfait was measured on an AMD Opteron 2.8 GHz with 16 GB of memory. Three times (in seconds) are reported in Figure 1: the build time, i.e., the time to compile and build the C/C++ files (using gcc with the Solaris operating system), the Parfait build time, i.e., the time to compile and build the C/C++ files using the LLVM [7] frontend, and the Parfait analysis time.

Parfait makes use of the LLVM infrastructure, as such files are compiled and linked into LLVM bitcode files. The Parfait analysis time consists of loading the bitcode files into memory, analysing them and producing the results. Using the harness, other tools can measure scalability in the same way.

Bug Category	From	CWE_{ID}	# bugs per category	LOC min	LOC max	LOC avg	# bug benchmarks
Buffer overflow (BO)	Kratkiewicz	120	873	83	106	91	1,691
	SAMATE	120	134	22	1,127	119	
Memory/pointer (M/P)	SAMATE	401,415,416,476, 590	54	29	90	50	
Integer overflow (IO)	SAMATE	190	13	26	77	46	
Format string (FS)	SAMATE	134	9	42	759	130	
Other (O)	SAMATE	73,78,79,89,99,132,195,197,215,243, 252,255,329,366,457,468,636,628	165	20	178	57	
Overall			1,248	20	1,127	88	1,691

Table 7: Characteristics of the Synthetic Suite of Bug Kernels

6. CONCLUSIONS AND EXPERIENCE

Benchmarking of tools reflects the level of maturity reached by a given tool's community. Bug detection tools are reaching maturity whilst benchmarks for bug detection tools are still in their infancy. Benchmarking helps tool developers, but more importantly, it helps users in general to better understand the capabilities of the various tools available against a set of parameters that are relevant to them.

In this paper we present BegBunch v0.3, a benchmark for C bug detection tools that measures accuracy and scalability of a tool. BegBunch's suites and harnesses allow developers and users to determine the state of their tool with respect to bug benchmarks derived from mature open source code.

It took hundreds of hours to put together the BegBunch suites and harnesses. Extraction of bug kernels took, on average, 2 days each. We found that most bug tracking systems do not keep track of which bugs are fixed in a given commit. BegBunch has proven to be useful for our own testing purposes. We hope that once it is open sourced, the community will contribute to increase the types of bugs covered by the suites and improve on it. For more information please refer to `http://research.sun.com/projects/downunder/projects/begbunch`

7. REFERENCES

[1] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. Technical report, The MITRE Corporation, May 2007. Version 1.1.

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

[3] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, January/February 2002.

[4] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proc. of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 41–50, October 2008.

[5] G. J. Holzmann. Static source code checking for user-defined properties. In *Proceedings of 6th World Conference on Integrated Design & Process Technology (IDPT)*, June 2002.

[6] 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.

[7] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, March 2004.

[8] LLVM/Clang Static Analyzer. `http://clang.llvm.org/StaticAnalysis.html`. Last accessed: 1 December 2008.

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

[10] MITRE Corporation. Common Weakness Enumeration. `http://cwe.mitre.org/`, April 2008.

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

[12] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83, Portland, Oregon, 2003. IEEE Computer Society.

[13] C. van Rijsbergen. *Information Retrieval*. Butterworth, 2 edition, 1979.

[14] D. A. Wheeler. More Than A Gigabuck: Estimating GNU/Linux's Size. `http://www.dwheeler.com/sloc/`, 2001. Last accessed: 16 March 2009.

[15] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. of International Symposium on Foundations of Software Engineering*, pages 97–106. ACM Press, 2004.

APPENDIX – The Synthetic Suite

For completeness, we collated the synthetic bug benchmarks from Kratkiewicz [6] and the SAMATE dataset [11] into a Synthetic suite that makes use of the BegBunch harness. Errors in the SAMATE dataset were fixed and reported back to NIST. Most benchmarks in this suite are intra-procedural.

Tool	Type	# TP	# FP	# FN	Accuracy
Parfait	BO	869	0	184	85.7%
Splint	BO	582	380	471	49.4%
Splint	NPD	2	1	7	20.0%
Splint	UAF	8	1	9	47.1%
Splint	UV	10	1	4	69.0%
Clang	NPD	3	1	6	30.0%
Clang	UV	7	0	7	50.0%
UNO	BO	457	5	596	45.2%

Table 8: Evaluation of C bug detection tools against the Synthetic suite

Table 7 summarizes the synthetic bug benchmarks by bug category, where category Other groups all benchmarks that contain types not defined in our Accuracy suite. Table 8 shows the evaluation of various bug detection tools against this suite. The bug types reported by these tools are: buffer overflow (BO), null pointer dereference (NPD), use after free (UAF) and uninitialized variable (UV).