BugBench: Benchmarks for Evaluating Bug Detection Tools

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou and Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana Champaign, Urbana, IL 61801

ABSTRACT

Benchmarking provides an effective way to evaluate different tools. Unfortunately, so far there is no good benchmark suite to systematically evaluate software bug detection tools. As a result, it is difficult to quantitatively compare the strengths and limitations of existing or newly proposed bug detection tools.

In this paper, we share our experience of building a bug benchmark suite called *BugBench*. Specifically, we first summarize the general guidelines on the criteria for selecting representative bug benchmarks, and the metrics for evaluating a bug detection tool. Second, we present a set of buggy applications collected by us, with various types of software bugs. Third, we conduct a preliminary study on the application and bug characteristics in the context of software bug detection. Finally, we evaluate several existing bug detection tools including Purify, Valgrind, and CCured to validate the selection of our benchmarks.

1 Introduction

1.1 Motivation

Software bugs account for more than 40% system failures [20], which makes software bug detection an increasingly important research topic. Recently, many bug detection tools have been proposed, with many more expected to show up in the near future. Facing ever so many tools, programmers need a guidance to select tools that are most suitable for their programs and occurring failures; and researchers also desire a unified evaluation method to demonstrate the strength and weakness of their tools versus others. All these needs strongly motivate a representative, fair and comprehensive evaluation benchmark suite for the purpose of evaluating software bug detection tools.

Benchmark is a standard of measurement or evaluation, and an effective and affordable way of conducting experiments [28]. A good all-community accepted benchmark suite has both technique and sociality impact. In the technical aspect, evaluations with standard benchmarks are more rigorous and convincing; alternative ideas can be compared objectively; problems overlooked in previous research might be manifested in benchmarking. In the social aspect, building benchmark enforces the collaboration within community and help the community to form a common understanding of the problem they are facing [26]. There are many successful benchmark examples, such as SPEC (Standard Performance Evaluation Corporation) benchmarks [27], and TPC series (Transaction Processing Council) [29], both of which have been widely used by the corresponding research and product development communities.

However, in the software bug detection area, there is no widely-accepted benchmark suite to evaluate existing or newly proposed methods. As a result, many previous studies either use synthetic toy-applications or borrow benchmarks (such as SPEC and Siemens) from other research areas. While such evaluation might be appropriate to use for proof of concept, it hardly provides a solid demonstration of the unique strength and shortcomings of the proposed method. Being aware of this problem, some studies [5, 23, 32] use real buggy applications for evaluation, which makes the proposed tools much more convincing. Unfortunately, based on our previous experience of evaluating our own bug detection tools [18, 24, 32, 33], finding real applications with real bugs is a time-consuming process, especially since many bug report databases are not well documented for our purposes, i.e., they only report the symptoms but not the root causes. Furthermore, different tools are evaluated with different applications, making it hard to cross-compare tools with similar functionality.

Besides benchmarking, the evaluation criteria of software bug detection tools are also not standardized. Some work evaluated only the execution overhead using SPEC benchmarks, completely overlooking the bug detection functionality. In contrast, some work [16, 18] did much more thorough evaluation. They not only reported false positives and/or false negatives, but also provided the ranking of reported bugs.

As the research area of software bug detection starts booming with many innovative ideas, the urgency of a unified evaluation method with a standard benchmark suite has been recognized, as indicated by the presence of this workshop. For example, researchers at IBM Haifa [6] advocate building benchmarks for testing and debugging concurrent programs. Similarly, although not formally announced as benchmark, a Java application set, HEDC used in [31], is shared by a few laboratories to compare the effectiveness of data race detection methods.

1.2 Our Work

Benchmark suite building is a long-term, iterative process and needs the cooperation from all over the community. In this paper, we share our experience of building a bug benchmark suite as a vehicle to solicit feedbacks. We plan to release the current collection of buggy applications soon to the research community. Specifically, this paper reports our work on bug benchmark design and collection in the following aspects:

(1) General guidelines on bug benchmark selection criteria and evaluation metrics: By learning from successful benchmarks in other areas and prior unsuccessful bug benchmark trials, we summarize several criteria that we follow when selecting a buggy application into our benchmark suite. In addition, based on previous research experience and literature research in software bug detection, we also summarize a set of quantitative and qualitative metrics for evaluating bug detection tools.

(2) A C/C++ bug benchmark suite *BugBench*: By far, we have collected 17 C/C++ applications for our *BugBench* and we are still looking for more applications to enrich the suite. All of the applications are from the open source community and contain various software defects including buffer overflows, stack smashing, double frees, uninitialized reads, memory leaks, data races, atomic violations, semantic bugs, etc. Some of these buggy applications have been used by our previous work [18, 24, 32, 33], and also forwarded by us to a few other research groups at UCSD, Purdue, etc in their studies [7, 22].

(3) A preliminary study of benchmark and bug characteristics: We have studied the characteristics of several benchmarks that contain memory-related bugs, including memory access frequencies, malloc frequencies, crash latencies (the distance from the root cause to the manifestation point), etc., which would affect the overhead and bug-detection capability of a dynamic bug detection tool. *To our best knowledge, ours is one of the first in studying buggy application characteristics in the context of software bug detection.*

(4) A preliminary evaluation of several existing tools: To validate our selection of benchmarks and characteristics, we have conducted a preliminary evaluation using several existing tools including Purify [12], Valgrind [25] and CCured [23]. Our preliminary results show that our benchmarks can effectively differentiate the strengths and limitations of these tools.

2 Lessons from Prior Work

2.1 Successful Benchmark in Other Areas

SPEC (Standard Performance Evaluation Cooperative) was founded by several major computer vendors in order to "*provide the industry with a realistic yardstick to measure the performance of advanced computer systems*" [3]. To achieve this purpose, SPEC has very strict application selection process. First, candidates are picked from those that have significant use in their fields, e.g. gcc from compiler field, and weather prediction from scientific computation field. Then, candidates are checked for their clarity and portability over different architecture platforms. Qualified candidates will be analyzed for detailed dynamic characteristics, such as instruction mix, memory usage, etc. Based on these characteristics, SPEC committee decides whether there are enough diversity and little redundancy in the benchmark suite. After several iterations of the above steps, a SPEC-benchmark is finally announced.

TPC (Transaction Processing Council) was founded in the middle 80's to satisfy the demand of comparing numerous database management systems. TPC benchmark shares some common properties as that in SPEC, i.e. representative, diverse, and portable, etc. Take TPC-C (an OLTP benchmark) as an example[17]. To be representative, TPC-C uses five real-world popular transactions: new order, payment, delivery, order status, and stock level. In terms of diversity, these transactions cover almost all important database operations. In addition, TPC-C has a comprehensive evaluation metric set. It adopts two standard metrics: new-order transaction rate and price/performance, together with additional tests for ACID properties, e.g. whether the database can recover from failure. All these contribute to the great success of TPC-C benchmark.

2.2 Prior Benchmarks in Software Engineering and Bug Detection Areas

Recently, much progress has been made on benchmarking in software engineering-related areas. *CppETS* [26] is a benchmark suite in reverse engineering for evaluating "factor extractors". It provides a collection of C++ programs, each associated with a question file. Evaluated tools will answer the questions based on their factor extracting results and get points from their answers. The final score from all test programs indicates the performance of this tool. This benchmark suite is a good vehicle to objectively evaluate and compare factor extractors.

The benchmark suites more related to bug detection are Siemens benchmark suite [11] and PEST benchmark suite [15] for software testing. In these benchmark suites, each application is associated with some buggy versions. Better testing tools can distinguish more buggy versions from correct ones. Although these benchmark suites provide a relatively large bug pool, most bugs are semantic bugs. There is almost no memory-related bugs and definitely no multi-threading bugs. Furthermore, the benchmark applications are very small (some are less than 100 line of code), hence cannot represent real bug detection scenarios and can hardly be used to measure time overhead. Therefore, they are not suitable for serving as bug detection benchmarks.

In the bug detection community, there is not much work done in benchmarking. Recently, researchers in IBM Haifa [14] propose building multithreading program benchmarks. However, their efforts are unsuccessful as also acknowledged in their following paper [6], because they rely on students to purposely generate buggy programs instead of using real ones.

3 Benchmarking Guideline

3.1 Classification of Software Bugs

In order to build good bug benchmarks, we first need to classify software bugs. There are different ways to classify bugs [1, 15], in this section we make classification based on different challenges the bug exposes to the detection tools. Since our benchmark suite cannot cover all bug types, in the following we only list the bug types that are most security critical and most common. They are also the design focus of most bug detection tools.

Memory related bugs Memory related bugs are caused by improper handling of memory objects. These bugs are often exploited to launch security attack. Based on US-CERT vulnerability Notes Database [30], they contribute the most to all reported vulnerabilities since 1991. Memory-related bugs can be further classified into: (1) Buffer overflow: Illegal access beyond the buffer boundary. (2) Stack smashing: Illegally overwrite the function return address. (3) Memory leak: Dynamically allocated memory have no reference to it, hence can never be freed. (4) Uninitialized read: Read memory data before it is initialized. The reading result is illegal. (5) Double free: One memory location freed twice.

Concurrent bugs Concurrent bugs are those that happen only in multi-threading (or multi-processes) environment. They are caused by ill-synchronized operations from multiple threads. Concurrent bugs can be further divided into following groups: (1) Data race bugs: Conflicting accesses from concurrent threads touch the shared data in arbitrary order. (2) Atomicity-related bugs: A bunch of operations from one thread is unexpectedly interrupted by conflicting operations from other threads. (3) Deadlock: In resource sharing, one or more processes permanently wait for some resources and can never proceed any more.

An important property of concurrent bugs is un-deterministic, which makes them hard to be reproduced. Such temporal sensitivity adds extra difficulty to bug detection.

Semantic bugs A big family of software bugs are semantic bugs, i.e. bugs that are inconsistent with the original design and the programmers' intention. We often need semantic information to detect these bugs.

3.2 Classification of Bug Detection Tools

Different tools detect bugs using different methods. A good benchmark suite should be able to demonstrate the strength and weakness of each tool. Therefore, in this section, we study the classification of bug detection tools, by taking a few tools as examples and classifying them by two criteria in Table 1.

	Static	**Dynamic**	**Model Checking**
Programming-rule based tools	PREfix [2] RacerX [4]	Purify [12] Valgrind [25]	VeriSoft[9] JPFinder[13] CMC[21]
Statistic-rule based tools	CP-Miner [18] D. Engler's [5]	DIDUCE [10] AccMon [32] Liblit's [19]	
Annotation-based	ESC/Java [8]		

Table 1: Classification of a few detection tools

As shown in Table 1, one way to classify tools is based on the rules they use to detect bugs. Most detection tools hold some "rules" in mind: code violating the rules is reported as bug. *Programming-rule-based* tools use rules that should be followed in programming, such as "array pointer cannot move out-of-bound". *Statistic-rule-based* approaches learn statistically correct rules (invariants) from successful runs in training phase. *Annotation-based* tools use programmer-written annotations to check semantic bugs.

We can also divide tools into *static*, *dynamic* and *model checking*. *Static* tools detect bugs by static analysis, without requiring code execution. *Dynamic* tools are used during execution, analyzing run-time information to detect bugs on-the-fly. They add runtime overhead but are more accurate. *Model checking* is a formal verification method. It was usually grouped into static detection tools. However, recently people also use model checking during program execution.

3.3 Benchmark Selection Criteria

Based on the study in section 2 and 3.1, 3.2, we summarize following bug detection benchmark selection criteria. **(1) Representative:** The applications in our benchmark suite should be able to represent real buggy applications. That means: First, the application should be real, implemented by experienced programmers instead of novices. It is also desirable if the application has significant use in practice. Second, the bug should also be real, naturally generated, not purposely injected. **(2) Diverse:** In order to cover a wide range of real cases, the applications in benchmark should be diverse in the state space of some important characteristics, including bug types; some dynamic execution characteristics, such as heap and stack usage, the frequency of dynamic allocations, memory access properties, pointer dereference frequency, etc; and the complexity of bugs and applications, including the bug's crash latency, the application's code size and data structure complexity, etc. Some of these characteristics will be discussed in detail in section 4.2. **(3) Portable:** The benchmark should be able to evaluate tools designed on different architecture platforms, so it is better to choose hardware-independent applications. **(4) Accessible:** Benchmark suites are most useful when everybody can easily access them and use them in evaluation. Obviously, proprietary applications can not meet this requirement, so we only consider open source code to build our benchmark. **(5) Fair:** The benchmark should not bias toward any detection tool. Applying above criteria, we can easily see that benchmarks like SPEC, Siemens are not suitable in our context: many SPEC applications are not buggy at all and Siemens benchmarks are not diverse enough in code size, bug types and other characteristics.

In addition to the above five criteria designed for selecting applications into the bug benchmark suite, application inputs also need careful selection. A good input set should contain both correct inputs and bug-triggering inputs. Bug-triggering inputs will expose the bug and correct inputs can be used to calculate false positives and enable the overhead measurement in both buggy runs and correct runs. Additionally, a set of correct inputs can also be used to unify the training phase of invariant-based tools.

3.4 Evaluation Metrics

The effectiveness of a bug detection tool has many aspects. A complete evaluation and comparison should base on a set of metrics that reflect the most important factors. As shown in Table 2, our metric set is composed of four groups of metrics, each representing an important aspect of bug detection.

Most metrics can be measured quantitatively. Even for some traditionally subjective metric, such as "pinpoint root cause", we can measure it quantitatively by the distance from the bug root cause to the bug detection position in terms of dynamic and/or static instruction numbers (we call it *Detection Latency*). Some metrics, such as manual effort and new hardware reliance, will be measured qualitatively.

Functionality Metrics	**Overhead Metrics**
Bug Detection False Positive	Time Overhead
Bug Detection False Negative	Space Overhead
Easy to Use Metrics	Static Analysis Time
Reliance on Manual Effort	Training Overhead
Reliance on New Hardware	Dynamic Detection Overhead
Helpful to Users Metrics	
Bug Report Ranking	
Pinpoint Root Cause?	

Table 2: Evaluation metric set

We should also notice that, the same metric may have different meanings for different types of tools. That is the reason that we list three different types of overhead together with the time and space overhead metrics. We will only measure static analysis time for static tools; measure both training and dynamic detection overhead for *statistical-rule-based* tools and measure only dynamic detection overhead for most *programming-rule-based* tools. The comparison among tools of the same category is more appropriate for some metrics. When comparing tools of different categories, we should keep the differences in mind.

4 Benchmark

4.1 Benchmark Suite

Based on the criteria in section 3.3, we have collected 17 buggy C/C++ programs from open source repositories. These programs contain various bugs including 13 memory-related bugs, 4 concurrent bugs and 2 semantic bugs [1]. We have also prepared different test cases, both bug-triggering and non-triggering ones, for each application. We are still in the process of collecting more buggy applications.

Table 3 shows that all applications are real open-source applications with real bugs and most of them have significant use in their domains. They have different code sizes and have covered most important bug types.

As we can see from the table, the benchmark suite for memory-related bugs is already semi-complete. We will conduct more detailed analysis for them in the following sections. Other types of bugs, however, are incomplete yet. Enriching *BugBench* with more applications on other types of bugs and more analysis on large applications remains as our future work.

4.2 Preliminary Characteristics Analysis

An important criterion for a good benchmark suite is its diversity on important characteristics, as we described in section 3.3. In this section, we focus on a subset of our benchmarks (memory-related bug applications) and analyze their characteristics that would affect *dynamic* memory bug detection tools.

Dynamic memory allocation and memory access behaviors are the most important characteristics that have significant impact on the overheads of dynamic memory-related bug detection tools. This is because many memory-related bug detection tools intercept memory allocation functions and monitor most memory accesses. In table 4, we use frequency and size to represent dynamic allocation properties. As we can see, in 8 applications, the memory allocation frequency ranges from 0 to 769 per Million Instructions and the size ranges from 0 to 6.0 MBytes. Such large range of memory allocation behaviors will lead to different overheads in dynamic bug detection tools, such as Valgrind and Purify. In general, the more frequent of memory allocation, the larger

[1] some applications contain more bugs than we describe in Table 3.

Name	Program	Source	Description	Line of Code	Bug Type
NCOM	ncompress-4.2.4	Red Hat Linux	file (de)compression	1.9K	Stack Smash
POLY	polymorph-0.4.0	GNU	file system ”unixier” (Win32 to Unix filename converter)	0.7K	Stack Smash & Global Buffer Overflow
GZIP	gzip-1.2.4	GNU	file (de)compression	8.2K	Global Buffer Overflow
MAN	man-1.5h1	Red Hat Linux	documentation tools	4.7K	Global Buffer Overflow
GO	099.go	SPEC95	game playing (Artificial Intelligent)	29.6K	Global Buffer Overflow
COMP	129.compress	SPEC95	file compression	2.0K	Global Buffer Overflow
BC	bc-1.06	GNU	interactive algebraic language	17.0K	Heap Buffer Overflow
SQUD	squid-2.3	Squid	web proxy cache server	93.5K	Heap Buffer Overflow
CALB	cachelib	UIUC	cache management library	6.6K	Uninitialized Read
CVS	cvs-1.11.4	GNU	version control	114.5K	Double Free
YPSV	ypserv-2.2	Linux NIS	NIS server	11.4K	Memory Leak
PFTP	proftpd-1.2.9	ProFTPD	ftp server	68.9K	Memory Leak
SQUD2	squid-2.4	Squid	web proxy cache	104.6K	Memory Leak
HTPD1	httpd-2.0.49	Apache	HTTP server	224K	Data Race
MSQL1	msql-4.1.1	MySQL	database	1028K	Data Race
MSQL2	msql-3.23.56	MySQL	database	514K	Atomicity
MSQL3	msql-4.1.1	MySQL	database	1028K	Atomicity
PSQL	postgresql-7.4.2	PostgreSQL	database	559K	Semantic Bug
HTPD2	httpd2.0.49	Apache	HTTP server	224K	Semantic Bug

Table 3: Benchmark suite

Name	Malloc Freq. (# per MInst)	Allocated Memory Size	Heap Usage vs. Stack Usage	Memory Access (# per Inst)	Memory Read vs. Write	Symptom	Crash Latency (# of Inst)
NCOM	0.003	8B	0.4% vs. 99.5%	0.848	78.4% vs. 21.6%	No Crash	NA
POLY	7.14	10272B	23.9% vs. 76.0%	0.479	72.6% vs. 27.4%	Varies on Input*	9040K*
GZIP	0	0B	0.0% vs. 100%	0.688	80.1% vs. 19.9%	Crash	15K
MAN	480	175064B	85.1% vs. 14.8%	0.519	70.9% vs. 20.1%	Crash	29500K
GO	0.006	364B	1.6% vs. 98.3%	0.622	82.7% vs. 17.3%	No Crash	NA
COMP	0	0B	0.0% vs. 100%	0.653	79.1% vs. 20.9%	No Crash	NA
BC	769	58951B	76.6% vs. 23.2%	0.554	71.4% vs. 28.6%	Crash	189K
SQUD	138	5981371B	99.0% vs. 0.9%	0.504	54.2% vs. 45.8%	Crash	0

Table 4: Overview of the applications and their characteristics (*:The crash latency is based on the input that will cause the crash.)

overhead would be imposed by such tools. To reflect the memory access behavior, we use access frequency, read/write ratio and heap/stack usage ratio. Intuitively, access frequency directly influences the dynamic memory bug detection overhead: the more frequent memory accesses, the larger the checking overhead. Some tools use different policies to check read and write accesses and some tools differentiate stack and heap access, so all these ratios are important to understand the overhead. In table 4, the access frequencies of our benchmark applications change from 0.479 to 0.848 access per instruction and heap usage ratio from 0 to 99.0%. Both show a good coverage. Only the read/write ratio seems not to change much within all 8 applications, which indicates the need to further improve our benchmark suite based on this.

Obviously, the bug complexity may directly determines the false negatives of bug detection tools. In addition, the more difficult to detect, the more benefits a bug detection tool can provide the programmers. While it is possible to use many ways to measure complexity (which we will do in the future), we use the symptom and crash latency to measure this property. Crash latency is the latency from the root cause of a bug to the place where the application finally crashes due to the propagation of this bug. If the crash latency is very short, for example, right at the root cause, even without any detection tool, programmers may be able to catch the bug immediately based on the crash position. On the other hand, if the bug does not manifest until a long chain of error propagation, detecting the bug root cause would be much more challenging for both programmers and all bug detection tools. As shown in Table 4, the bugs in our benchmarks manifest in different ways: crash or silent errors. For applications that will crash, their crash latency varies from zero latency to 29 million instructions.

5 Preliminary Evaluation

In order to validate the selection of our bug benchmark suite, in this section, we use *BugBench* to evaluate 3 popular bug detection tools: Valgrind [25], Purify [12] and CCured [23]. All these three are designed to detect memory-related bugs, so we choose 8 memory-relate buggy applications from our benchmarks. The evaluation results are shown in Table 5.

In terms of time overhead, among the three tools, CCured always has the lowest overhead, because it conducted static analysis beforehand. Purify and Valgrind have similar magnitude of overhead. Since Valgrind is implemented based on instruction emulation and Purify is based on binary code instrumentation, we do not compare the overheads of these two tools. Instead, we show how an application's characteristics affect one tool's overhead. Since our bug benchmark suite shows a wide range of characteristics, the overheads imposed by these tools also vary from more than 100 times of overhead to less than 20% overhead. For example, the application BC has the largest overhead in both Valgrind and Purify, as high as 119 times. The reason is that BC has very high memory allocation frequency, as shown in Table 4. On the other hand, POLY has very small overhead due to its smallest memory access frequency as well as its small allocation frequency.

In terms of bug detection functionality, CCured successfully catches all the bugs in our applications and also successfully points out the root cause in most cases. Both Valgrind and Purify fail to catch the bugs in NCOM and COMP. The former is a stack buffer overflow and the latter is a one-byte global buffer overflow. Valgrind also fails to catch another global buffer overflow in GO and has long detection latencies in the other three global buffer over-

	Catch Bug?			False Positive			Pinpoint The Root Cause (Detection Latency(KInst)[1])			Overhead			Easy to Use		
	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured
NCOM	No	No	Yes	0	0	0	N/A	N/A	Yes	6.44X	13.5X	18.5%	Easiest	Easy	Moderate
POLY	Vary[2]	Yes	Yes	0	0	0	No(9040K)[2]	Yes	Yes	11.0X	27.5%	4.03%	Easiest	Easy	Moderate
GZIP	Yes	Yes	Yes	0	0	0	No(15K)	Yes	Yes	20.5X	46.1X	3.71X	Easiest	Easy	Moderate
MAN	Yes	Yes	Yes	0	0	0	No(29500K)	Yes	Yes	115.6X	7.36X	68.7%	Easiest	Easy	Hard
GO	No	Yes	Yes	0	0	0	N/A	Yes	Yes	87.5X	36.7X	1.69X	Easiest	Easy	Moderate
COMP	No	No	Yes	0	0	0	N/A	N/A	Yes	29.2X	40.6X	1.73X	Easiest	Easy	Moderate
BC	Yes	Yes	Yes	0	0	0	Yes	Yes	Yes	119X	76.0X	1.35X	Easiest	Easy	Hardest
SQUD	Yes	Yes	N/A[3]	0	0	N/A[3]	Yes	Yes	N/A[3]	24.21X	18.26X	N/A[3]	Easiest	Easy	Hardest

Table 5: Evaluation of memory bug detection tools. (1: Detection latency is only applicable when fail to pinpoint the root cause; 2: Valgrind's detection result varies on inputs. Here we use the input by which Valgrind fails to pinpoint root cause; 3: We fail to apply CCured on Squid)

flow applications: POLY, GZIP and MAN. The results indicate that Valgrind and Purify handle heap objects much better than they do on stack and global objects.

As for POLY, we tried different buggy inputs for Valgrind and the results are interesting: if the buffer is not overflowed significantly, Valgrind will miss it; with moderate overflow, Valgrind catches the bug after a long path of error propagation, not the root cause; only with significant overflow, Valgrind can detect the root cause. The different results are due to POLY's special bug type: first global corruption and later stack corruption.

Although CCured performs much better than Valgrind and Purify in both overhead and functionality evaluation, the tradeoff is its high reliance on manual effort in code preprocessing. As shown in the "Easy to Use" column of Table 5, among all these tools, Valgrind is the easiest to use and requires no re-compilation. Purify is also fairly easy to use, but requires re-compilation. CCured is the most difficult to use. It often requires fairly amount of source code modification. For example, in order to use CCured to check BC, we have worked about 3 to 4 days to study the CCured policy and BC's source code to make it satisfy the CCured's language requirement. Moreover, we fail to apply CCured on a more complicated server application: SQUD.

6 Current Status & Future Work

Our *BugBench* is an ongoing project. We will release these applications together with documents and input sets through our web page soon. We welcome feedbacks to refine our benchmark.

In the future, we plan to extend our work in several dimensions. First, we will enrich the benchmark suite with more applications, more types of bugs based on our selection criteria and characteristic analysis (the characteristics in Table 4 show that some important benchmark design space is not covered yet). We are also in the plan of designing tools to automatically extract bugs from bug databases (e.g. Bugzilla) maintained by programmers, so that we can not only get many real bugs but also gain deeper insight into real large buggy applications. Second, we will evaluate more bug detection tools, which will help us enhance our *BugBench*. Third, we intend to add some supplemental tools, for example, program annotation for static tools, and scheduler and record-replay tools for concurrent bug detection tools.

REFERENCES

[1] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[3] K. M. Dixit. The spec benchmarks. *Parallel Computing*, 17(10-11), 1991.

[4] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, Oct. 2003.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP '01*, pages 57–72, 2001.

[6] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*, 2004.

[7] L. Fei and S. Midkiff. Artemis: Practical runtime monitoring of applications for errors. Technical Report TR-ECE05-02, Purdue University, 2005.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[9] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *ISSTA*, pages 124–133, 1998.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02*, May 2002.

[11] M. J. Harrold and G. Rothermel. Siemens Programs, HR Variants. URL: http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, Jan. 1992.

[13] K. Havelund and J. U. Skakkebæk. Applying model checking in java verification. In *SPIN*, 1999.

[14] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *IPDPS*, 2003.

[15] James R. Lyle, Mary T. Laamanen, and Neva M. Carlson. PEST: Programs to evaluate software testing tools and techniques. URL: www.nist.gov/itl/div897/sqg/pest/pest.html.

[16] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *SIGSOFT '04/FSE-12*, pages 83–93, 2004.

[17] C. Levine. TPC-C: an OLTP benchmark. URL: http://www.tpc.org/information/sessions/sigmod/sigmod97.ppt, 1997.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[20] E. Marcus and H. Stern. Blueprints for high availability. John Willey and Sons, 2000.

[21] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, Dec. 2002.

[22] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA'05*, 2005.

[23] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, Jan. 2002.

[24] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05*, 2005.

[25] J. Seward. Valgrind. URL: http://www.valgrind.org/.

[26] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *ICSE '03*, pages 74–83. IEEE Computer Society, 2003.

[27] Standard Performance Evaluation Corporation. SPEC benchmarks. URL: http://www.spec.org/.

[28] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[29] Transaction Processing Council. TPC benchmarks. URL: http://www.tpc.org/.

[30] US-CERT. US-CERT vulnerability notes database. URL: http://www.kb.cert.org/vuls.

[31] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[32] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *MICRO '04*, Dec. 2004.

[33] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *ISCA '04*, June 2004.