An Evaluation of Bug Finding Approaches

James Coakley[1], Ronald Fenelus[2], Kehontas Rowe[3]

[1]University of South Florida

[2]Florida International University

[3]Mills College

Abstract

Bugs in software, such as buffer overflows, can often become entry points for breaches in software security. These bugs are often difficult to find manually. They can also be extremely costly to fix after software is released. To alleviate this there are automated bug finding tools. Several different bug finding methods have been developed from existing approaches. We test several implementations of these methods using the automated tools including bounded model checking (CBMC), static analysis (Coverity) with BugBench, and compare and evaluate the results.

Introduction

Software bugs are a very common entry point for security exploits. Integer overflows, buffer overflows, pointer errors, and any bug that affects memory can enable malicious users to force a program to run arbitrary code.

Part of the problem with bugs is that they are quite difficult in practice to find manually. Due to their complexity nearly all software has bugs of various types. While these bugs may be very difficult to encounter within the intended use of the program malicious attackers may be able to craft inputs that exploit these bugs in a way that is potentially damaging. To compound this issue, bugs are much harder and more costly to fix once software has been deployed.

One proposed solution to this problem is to create bug-checking tools to automatically detect and report bugs in software. Several different approaches have been taken to achieve this goal of automated bug finding. The two approaches we examine in this paper are Bounded Model Checking (BMC) and Abstract Static Analysis (ASA). BMC comes from hardware verification, and ASA comes from program analysis from the perspective of software testing. While these tools have several similarities in how they operate, they all employ theorem provers. For example, they have historical and implementational differences. We examine but do not analyze KLEE a Dynamic Symbolic Execution tool as grounds for continued work.

The goal of this paper is to help increase knowledge on the effectiveness of these tools. We attempt to do this by testing each of the approaches, in the form of software tools, on a neutral software testing suite, BugBench [2]. We use CBMC, a bounded model checker; and

Coverity, a static analysis tool; Klee, a dynamic symbolic executor, as representatives of their corresponding approaches.

This paper is organized as follows: In section two we offer brief explanations of the approaches each tool uses as well as specific points on each of the tools. In section three we describe our method of evaluation as well as outline the current results of the project. In section four we discuss results as well as describe how the project means to expand in the future. Finally, in section five we conclude the paper.

2. Overview of Bug Finding Approaches

Coverity's Static Analysis software was created by and is constantly monitored by a professional team of engineers, and is licensed out to other companies to test their own software. Klee is rather successful implementations of Dynamic Analysis, and was used to find bugs that manual testing has not found in 15 years. CBMC is considered the best Bounded Model Checker, and is used within other verification and bug finding approaches. In the following sections we give brief descriptions of BMC, ASA, and DCE as well as interesting specifics on the tools we use, CBMC, Coverity, and Klee respectively.

2.1 CBMC

Bounded model checking is a verification technique which is able to, in theory; verify properties of programs within a bounded number of steps [4]. The approach is to formulate the relevant parts of the program as a boolean formula. This formula can then be combined with the negations of asserted claims conjunctively and fed to a theorem prover to see if there exists a model which satisfies the resulting formula. If the formula is unsatisfiable then properties which are represented by the assertions are verified up to the specified bound. [3] If the claims represent restrictions on the program that ensure certain bugs do not occur then a verified program is bug-free within the bound.

2.2 Coverity

In Static Analysis, the program code is not executed; it is analyzed to build a structure that is passed to the analysis algorithm. Coverity is a proprietary Abstract Static Analyzer. Coverity analyzes the build sequence of the code under test, and flags dangerous operations. From here, the static analyzer is initiated with different options to analyze the output of the build sequence. [5]

2.3 KLEE

Klee substitutes program inputs with symbolic values and replaces program operations with ones that manipulate symbolic values. When program execution branches based on a

symbolic value, KLEE follows both branches and keeps track of a set of constraints for each branch. This set of constraints, or path conditions, must be satisfied in order for execution to continue on that path. When a path terminates or hits a bug, a test case can be generated by solving the path condition of that path. Assuming deterministic code, this test case can be given to the unmodified code as input, and the program will follow the same path. This method is great for getting a high coverage of executable lines of code. [1]

3 Evaluation

3.1 Approach

We ran seven of the programs (polymorph, ncompress, gzip, man, bc, squid, cvs) from the BugBench testing suite through the tools. We created assertions based on the several bugs found in the documentations for BugBench. We test buffer overflows on global variables, the stack, and the heap. Tests were run on a hex core 2.93 GHz Intel Xeon processors with 12 MB of cache memory and 32 GB of main memory.

	Lines of code (Thousand)	Number of Known Bugs
polymorph	722	2
ncompress	1939	1
gzip	8189	1
man	4614	1
bc	17.0	3
squid	93.5	1
cvs	114.5	1

For CBMC, we test with two different options. The first option is whether or not to include the CProver library. The CProver library acts as a replacement for the c standard library source code which is not necessarily available at test time. The second option is the amount of loop unwinding to be performed. We operate with both one and two loop unwindings.

For Coverity, we started out our analysis with a default run enabling all of the checker options to see how many bugs we were able to find. We were able to identify most of the bugs within the BugBench benchmarks on these initial runs. To enhance the results, default options within the static analyzer were tweaked to force a more in-depth and precise analysis. The exact options are presented and described in section 3.2. These options allowed us to verify one more bug than in the default run.

We were not able to test Klee effectively enough to find the optimal way to test the BugBench benchmarks. We leave this as a future project, to be completed in the coming weeks, for a future revision of the paper. We are predicting that for many of these large, real-world programs that KLEE will run out of memory, or just simply take too long to find bugs.

3.2 Results

CBMC

Program	Time	Memory	Violation
polymorph	.31 s	21,444 K	1
ncompress	12.4 s	263, 792 K	1
gzip	10 s	221,672 K	1
man*	1692 s	> 32 G	0
bc^	.51 s	34,532 K	0
squid*	>86400 s	15.1 G	0
cvs*	>86400 s	4.2 G	0

Programs marked with an asterisk were not able to complete verification before running out of memory or time. Library issues prevented CBMC from unwinding most of the functions in bc, resulting in an incomplete analysis of the program.

Coverity

	--all –force				Enhanced options *			
	D	Time	Mem	BF	D	Time	Mem	BF
polymorph	30	2.05s	58,624 kB	2 / 2	32	5.14s	113,168 kB	2 / 2
ncompress	22	7.95s	118,912 kB	1 /1	23	19.14s	222,624 kB	1 / 1
gzip	22	20.56s	158,144 kB	1 / 1	19	63.75s	273,072 kB	1 / 1
man	86	12.98s	67,840 kB	0 / 1	92	23.76s	135,504 kB	1 / 1
bc	13	17.35s	224,656 kB	1 / 3	14	22.48s	316,032 kB	1 / 3
squid	147	181.75s	323,470 kB	1 / 1	161	208.24s	326,560 kB	1 / 1
cvs	493	400.06s	748,496 kB	0 / 1	379	552.89s	669,712 kB	0 / 1

D=Total Defects; T=Time; Mem=Maximum Memory Usage; BF=Bugs found

* --all --force --enable-constraint-fpp --enable-fnptr --max-loop 8 --max-mem 16834

These options were enabled to allow the static analyzer to use more time and more memory in order to do a more precise analysis. --enable-constraint-fpp is a False Positive Pruner, and helps in the reduction of possible false positives being output by the analysis. --enable-fnptr keeps better track of function pointers, so as not to miss possible bugs by missing function pointers. --max-loop 8 sets the maximum number of loops to 8, which means Coverity will traverse up to 8 loops even if just a single traversal is necessary. And --max-mem 16384 sets a limit to the amount of memory used during the analysis. Setting max-loop to 128 and then to the maximum possible number found no additional defects.

4. Discussion

The results indicate that CBMC is effective for small code bases but quickly becomes ineffective as the size of a program increases. Along with failing to finish on the larger programs several bugs were missed due to the necessary amount of unwinding to find them. A bug in polymorph and a bug in man require 1024 and 100 unwindings respectively to find. The amount of memory required to successfully verify these programs with the required number of unwindings is much larger than the 32 GB limit we operate under.. The main flaw in CBMC is its lack of scalability to larger programs.

Coverity's static analyzer was best at finding bugs within the BugBench benchmarks, and that is to be expected as it also reports many false positives. Coverity is able to scale well for large portions of code, and completes extensive analysis in reasonable time. There are a few options, which will be explored in a future revision of this paper that we would like to test and see if we can catch more bugs. If there is an option that is better at analyzing memory allocation (specifically freeing memory), we would like to utilize that to see if the bug in cvs is found. Another flaw with Coverity is the total number of defects found. To properly find the bugs, the user has to look through a large number of defects, which may be false positives.

We were unable to get any meaningful results out of KLEE, as the benchmarks that we are testing are mainly buffer overflows with large input lengths which are prohibitive for KLEE. KLEE can test the program with a set of arguments that allow it to terminate within our defined cut-off time, but these will not find the type of bugs we test on in this paper. The time KLEE takes to run can increase exponentially in the size of the symbolic inputs, making testing larger inputs infeasible. This we will leave for further testing in the near future.

5. Conclusion

Based on our results, we can say that Coverity is the best software so far in our testing of these programs. Coverity scales best to large, real-world code that can be too large for many other bug finding approaches. It was able to find the most bugs, and was able to find them within a very reasonable time. Again, each method has its advantages and disadvantages. Coverity's main advantage is that it is able to find most of the documented bugs, and flag all dangerous options in the process. The biggest disadvantage is the very large number of defects and false positives that it reports. CBMC is effective when it is able to run successfully, but this is only possible for a small number of programs. For the programs that were CBMC finished checking, it was able to find bugs without any false positives. We are not able to report results for KLEE, as we have not fully tested KLEE on our BugBench benchmarks. In a future revision of our paper, we will have these results and will have a proper evaluation of KLEE. We also plan to extend our benchmarks to include more programs with more variety in size, complexity, and other types of bugs. This will allow us to better evaluate each of the bug finding approaches, and come to a better conclusion as to each approaches strengths and weaknesses.

[1] Cadar, C., Dunbar, D., Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Program. Stanford University

[2] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In Workshop on the Evaluation of Software Defect Detection Tools, 2005.

[3] Edmund Clarke, Edmund Clarke, Daniel Kroening, Daniel Kroening, Flavio Lerda, and Flavio Lerda. A tool for checking ANSI-c programs. volume 2988 of Lecture Notes in Computer Science, chapter 15, pages 168{176. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. Trans. Comp.-Aided Des. Integ. Cir. Sys. 27, 7 (July 2008), 1165-1178.

[5] Chelf, B., Chou, A. The Next Generation of Static Analysis. Boolean Satisfiability and Path Simulation - A Perfect Match. Coverity

This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422).