ASSIGNMENT 1 FOR CPSC 508

KLEE: UNASSISTED AND AUTOMATIC GENERATION OF HIGH-COVERAGE TESTS FOR COMPLEX SYSTEMS PROGRAMS

January 21, 2019

Student ID: 19115724
University of British Columbia
Electrical and Computer Engineering

Contents

Inroduction

?

I decided to pursue the paper on KLEE, because of a few reasons:

1. I wanted to use this opportunity to explore model checking with respect to systems and C is one of the popular tools available for performing bounded model checking. (I had not worked with this tool before though.)
2. I had read about KLEE in "The PhD Grind" by Philip Guo and hence the name seemed to be quite heard of before to me.
3. The fact that KLEE claimed to have checked the GNU coreutils code and "hidden" found bugs seemed quite fascinating to me.

Chapter 1

Part 1: Critique

The following critique is before I tested the system.

1.1 PROBLEM AND IT'S IMPORTANCE:

The problem being addressed by the authors is three fold. The authors perform ~~a~~ code analysis by trying to provide ~~a~~ high code-coverage. With ~~the~~ high code-coverage success, the authors apply the model checking approach ~~in order~~ to find bugs such as buffer flow on GNU coreutils, busybox and HiSTar, and finally, the authors try to verify the functional correctness of the code. The authors make a strong case for each of these arguments, since there are many pre existing tools such as EXE (the predecessor of KLEE) which use~~a~~ similar approach ~~of~~ symbolic execution, for performing code-analysis. ~~however~~, it usually ends in state-space explosion due to exponential time. Also, runing the exisiting tools fails to provide ~~a~~ complete coverage of the code that the developers are trying to test.

overflow

the reference is vague

1.2 HYPOTHESIS:

The authors hypothesis ~~is~~ that even the most well tested code~~s~~ and systems can be buggy, despite the existance of ~~sophisticated~~ testing mechanisms; and formal verification is one of the approaches ~~which~~ can be used to find those corner cases.

that

symbolic exec.?
EXE?
all approaches?

as well as Busybox, etc

which is

1.3 Experimental Setup/Methodology design:

The authors used COREUTILS version 6.10 run on Fedora Core 7 with SELinux on a Pentium ~~machine. The~~ authors use a symbolic execution of the code a formal approach first proposed by et. al James [1], which means that instead of running the code with concrete values, they ~~decide to~~ run the code on symbolic values, after converting it to LLVM assembly language. Each symbolic path is traced and evaluated and in case of a bug, the path is terminated and a test case is generated with concrete values by compiling the unmodified code(with gcc etc.) by using buggy values obtained during symbolic execution.

is that true?

So what is their contribution

The methodology design is quite straighforward, since they are not coming up with an entirely new approach for code-analysis. ~~The~~ An symbolic execution ~~methodology~~ is the most common approach used by most ~~of~~ m model checkers to find bugs. However, the fact that they first make use of LLVM to compile their C code to LLVM assembly code, is ~~very~~ m strong, S ~~Since~~ this gives them an opportunity to use LLVM compatible tools ~~in order~~ to set up their measurement setup for evaluating the code coverage. Also, this makes KLEE accessible to all the users who want to check their C codes for bugs, without ~~an~~ a hassle, since LLVM is freely available.

I personally think that the research execution of handling the challenges they faced in order to avoid state space explosion is where the crux of the paper lies.

YES!

1.4 Research Execution:

The optimizations applied in order to avoid explosions and decrease the time are very interesting. The methodology seems to be followed very systematically since their main goals are more code coverage in less time.

1.4.1 Strong points:

1. The copy-on-write approach drastically decreases per-state-memory requirements, which means a new path is forked only on changes hence, we can trace the paths back in order to get to the root of the bug.

2. Also, the query optimizations approaches applied before it reaches the constraint solver STP is a smart tactic to decrease the amount of time taken for analysis since STP recieves the simplest set of queries.

3. The constraint independence technique which means that the constraints do not overlap in terms of the memory they reference is a very strong approach; as this ensures constraint isolation from one another. The modelling seems very strong because of the isolation aspect.

was that available in 2008?

1.4.2 Weak Points:

coreutils is a huge collection of programs

1. However, one interesting point to note here is that STP (The simple theorem prover) is a ~~very~~ basic theorem prover and does not support non-linear arithmetic. Hence, one small issue we can notice here, it can only verify simple programs, with simple mathematical models. Using a stronger theorem prover such as Z3 would have given it more range.

2. It is modelled specifically to the needs of GNU coreutils, which was later applied to Busybox which is similar to coreutils but is smaller version of coreutils. The other systems tested are a part of a similar domain, hence extending it to a much wider range of applications could have been considered.

1.5 Results:

This paper really shines in the different sets of analyis results that it produces. The fact that they used one tool to get so much information about a code is impressive. The first set of results, compare the queries execution timings with and without optimizations. The strongest results seem to be the ones seem to be the ones produced as a part of their code coverage analysis. They first ~~find~~ use gcov, which is an open source tool, ~~in order~~ to obtain the executable lines of codes of coreutils. Building on this, the authors produce the next set of results where they use the symbolic execution approach to find memory overflow bugs, which can lead to crashing of a system. The final set of results are checking the code equivalence using KLEE by including annotations. However, this is one set of result that did not particularly satisfy me since this approach seems a little incomplete to me because they are using bounded values to show equivalence of two codes. (explained further in weak point 3.)

#of

1.5.1 Strong points:

1. The memory overflow bug found in pr, which is not an obvious one is very motivating and really shows the strength of formal methods.

2. For further analysis the ~~authors~~ compare random tests with manual tests and KLEE tool. I particularly found it ~~very~~ interesting because usually fuzzy testing approaches ~~are found to~~ produce ~~a~~ wider range of results. However, since coreutils is a strongly maintained organization, manual testing triumphed. code base

3. They tested ~~out~~ their tool to ~~perform similar~~ analysis on Busybox and minix, however, I have a slight skepticisim that there must be quite some overlap between the applications. why does that matter?

1.5.2 Weak Points:

1. While presenting the results for Table 2, they have specifically mentioned that KLEE does not cover the library codes, whereas the developer tests might be ~~covering~~ and that can be a possible reason for low coverage of coreutils and busybox (it's a possibility and that has not been mentioned).

2. Another specific point to be noted is that there different types of code-coverage such as function coverage, statement coverage, edge coverage, condition coverage etc. However, here the focus seems to be entirely on statement coverage. They do point out somewhere that to remove the bias they take into consideration the branch coverage, which shows that absolute coverage decreases. Hence, how much is the high coverage is really helping us.

3. For showing the complete equivalence of two snippets of code there are two ways, either evaluate it for every possible set of input there is available (which is not really possible since its NP-complete), or mathematically prove by introducing a bisimulation invarient for the two programs f and f' and describing f and f' in a mathematical form using relational or arithmetic theories. However, the model checker KLEE ~~checks~~ ing for equivalence based on the symbolic values which pass through it, though covers a wider range than test suites does not gaurantee perfect equivalence.

1.6 THINGS I MIGHT HAVE DONE DIFFERENTLY:

not in 2008...

1. I would have used a stronger theorem prover such as Z3 or CVC4.

2. Since experimenting on coreutils and getting data is what probably led to the numerous number of optimizations, I would have tried selecting a wider range of applications to generalize klee bug catching techniques since it will not be tools with much bigger code base such as Snort.

1.7 LESSONS LEARNT:

:)

1. KLEE is a direct successor of EXE, and they called it KLEE instead of calling it EXE 2.

2. The result tables such as code coverage show really impressive results, however, the nitty gritty details such as the absolute values of coverage and type of code-coverage seem to be mentioned in one line in the paper. So never believe the result tables immediately.

3. The paper despite even in 2019 shows a very high level of sophistication which is almost 11 years after it's inception. This shows how well evaluated the system was.

4. The paper is so well written that I did not require much external resources in order to understand it.

5. The research papers takeaway can be explained in a sentence, for example the main idea of this paper was high code coverage in less time. Every other result was a consequence of the code coverage aspect.

how to achieve

Chapter 2

Part 2: Reproducing Results

The KLEE paper provides a numerous number of results which can be reproduced, which initially got me really excited as honestly according to my understanding KLEE being an automated model checking tool, once installed, reproducing results becomes one command away.

nee nee nee . . .

2.1 Experiment trying to reproduce:

I try to reproduce two of the results which I feel are the crux and reproducing them can lead to successfully reproducing all the results of the paper.

1. The results I try to reproduce ~~is~~ are similar to the ones shown in Table 2, which shows the code coverage produced by coreutils and the corresponding graph of the results in Figure 5. ~~The~~ the Table 2 compares the overall coverage, without including the library by running the KLEE tests and the Development tests. Based on the results they plot the graph in Figure 5.
2. However, in the paper, they show only results for overall coverage. On running the tool KLEE we can see that it produces ICov and BCov as two seperate coverages.
3. Before producing the coverage code, I however attempt to re produce the first example stated in the paper, which is 170 line MINIX tr code. The latest version of the tr code seemed to consist of much higher number lines of code. (285 lines)

did it in 2008?,

2.2 EXPERIMENTAL SETUP

2.2.1 Version of the system:

1. Coreutils version v 6.11 (This is the version used) for reproducing results
2. Docker container with only KLEE installed
3. KLEE -version = KLEE 1.4.0.0

— unsurprising but excellent description

2.2.2 Installation of KLEE:

From this point onwards, every step that I took was like a battle. Winning the initial battle did not mean you win the war.

1. Installing KLEE has two approaches, one with manual work and the second one also with another type of manual work, which is using the docker container.
2. In order to use KLEE, it supports a particular version number of LLVM, so depending on the KLEE version, it is necessary to make sure that the LLVM version is compatible.
3. IN order to perform code coverage tools such as gcov were used. Usually such tools come installed in any Linux type machine that we use, however, in case of a docker container, even the simplest text-editor does not come installed.
4. A script that came really handy during the setting up of basic LInux functionalities in a docker container [2]
5. After the docker setup, I assumed that the klee is ready to be shipped and used and the first result I tried to reproduce is the very first example which is explained in the paper MINIX tr tool.

2.3 ASSUMPTIONS FOR EXPERIMENTS:

1. The authors do not provide an exact look into the external tools they use to evaluate the code coverage. This is mentioned two times in one short sentence that they make use of gcov.
2. Since this paper was written back in 2008, the STP version is also not mentioned by the authors.

3. The 89 coreutils applications to be tested on are not mentioned by the author in the paper.

4. The authors explicitely mention that the tool ran on most of the codes without any modification and one such exception was the sort code. However, the authors do not mention what changes they make before running KLEE on it.

5. It is very difficult to reproduce the results since the exact commands used by KLEE back then have been removed from the current versions of KLEE.

6. Since KLEE is currently a a maintained software by the open source community, a small help was available from the mailing lists and the online tutorials [3].

7. The authors did not mention the symbolic arguments used for running KLEE on such tools. From a high level understanding of the tools, the authors tweak the symbolic arguments to obtain higher code coverage.

2.4 LIST OF TOOLS USED:

Since I decided to use the docker container for evaluating and running on codes, it was not a happy sight for me as I had to install the most basic editor in nano.

1. gcov - code coverage

2. LLVM - assembly code for sumbolic execution

3. gcc - compiling with gcc after symbolic values have been obtained

4. kcachegrind - visualize KLEE's progress

5. zcov - for visualizing coverage results.

Chapter 3

Critique Addendum

3.1 RESULTS COMPARISON AND REPRODUCIBILITY

3.1.1 MINIX 'tr' Results

1. In order to reproduce the results obtained as a part of the first example presented in I took the MINIX v3.1.0 tr code - 170 lines (exactly as mentioned in the paper), compiled it using the commands mentioned in the paper.

2. The compilation failed and resulted in 12 errors and 14 warnings. Looking at the errors, it mostly seemed related to the system calls, as I presume that the system calls protoypes must have been updated in the past few years.

3. I tried fixing a few of the errors in vain.

4. Now instead of taking the v3.1.0 tr code I decided to take the latest tr code which is a 285 line code. I compiled the code. Though there were ~~lesser~~ fewer errors, there were still errors but the compilation was successful.

5. However, running KLEE on the compiled code did not produce the results as expected by the paper because in v3.1.0 there might have been a buffer overflow error which seems to have been fixed in the current version.

6. However, I could not reproduce the result on v3.1.0 of the code because of technical errors.

7. "A user can start checking many real programs with KLEE in seconds: KLEE typically requires no source modifications or manual work. " - Exact quote from the paper. I

So you got no results?

DISAGREE.

3.1.2 Coreutils Results

The second sets of results that I try to reproduce are the results presented in Table 2 of the paper - finding the code coverage for the tools in coreutils. [4]

This time I was better-prepared and more familiar with the tool and hence expected positive results. But my docker installation failed me.

1. The KLEE paper analysed their results on coreutils v6.10. Hence, not making a wider change, I decided to conduct the experiments on v6.11.

2. Since this result focused more on the code covergae aspect rather than the bug-finding aspect, I realized that the version number should not really matter. Why? Well, because after these bugs were found, they would have been fixed in the later releases and v6.10 is a decade old, so using KLEE for bug catching on newer versions will not give us the exact results produced during v6.10.

3. Compiling the code was not an issue in this case. However, KLEE uses gcov which is a statement by statement coverage analysis. Gcov adds annotations to the source code in order to instrument the code.

4. Life is not so fruitful. SInce nothing was going downhill for a while, it was time. gcov is not a pre-installed package in docker as it comes with gcc. So checking the compatible versions I manually installed gcov and gcc. Without these tools getting the code coverage becomes tough as this tool keeps track of the number of execution paths.

5. Another set of problems, in order for gcc and gcov to run, they both should have the same version number which should be compatible with the llvm version number we use.

6. Finally after sorting these issues. I was able to run one tool from coreutils and I decided to test it on cat and echo.

7. The results in the paper do not specifically list out code coverage for every individual 89 tools, however they try to list it in categories.

8. For the cat.gcov we can see that there is 24.27 percent coverage.

and what did you get

9. It will take a lot of time to reproduce even a single KLEE table. The reason is that they have conducted a series of symbolic executions on the 89 tools by using some means of trail and error till it obtains better results that the normal tools code coverage.

3.2 GENERAL DISCUSSION REGARDING REPRODUCIBILITY

√B
√B
√B

1. The biggest lesson here is, research based tools are built to work only on the system setup by the authors.

2. After spending time working on the installation, I assumed (a horrible assumption) that all that is left now is to run the automated testing tool on coreutils. I could not have been more wrong.

3. It is very difficult to reproduce research results. Despite the fact that the authors mention that KLEE runs on unmodified codes for coreutils except one, there is a huge painful process before that which the authors slyly do not mention.

4. You might think that the grunt work is done, once you have setup the environent (which I believe is the toughest), the results are right there again that is not true. In order to produce the results as shown in table you need help of several external tools which help tracking down and calculating the code coverage.

5. Also, since coreutils is a huge software, we use w-llvm (whole llvm), which again is not mentioned by the authors.

3.3 FINAL THOUGHTS

1. This made me realize the time and effort that goes into systems research. I honestly was not aware that it requires so much work and effort.

2. When I first read the paper, I was super impressed with the claims of using it and getting results in seconds and I even managed to get the installation sorted in 2 hours. However, everything went downhill from there. It looked more like an oversell of the product. Every setup we are doing, every tool we are manually setting up.

3. SInce KLEE has so many dependencies, they should provide a packaged version so that we can easily install it on Linux machines. They do provide a docker container called klee, which however lacks the basic functionalities of a Linux system.

4. However, when I realized it took me so much effort to test on one coreutils tool, the fact that they tested it on 89 tools, with different symbolic inputs was very very impressive. It shows effort, a lot of effort and time. So despite the pain, I only have respect for the creators of KLEE for having patience and time.

5. Also, the impressive part about the paper for me was, using techniques such as copy on write to prevent state space explosion which it retains despite the effort taken to run it.

Great write-up. Sorry so painful.

Chapter 4

References

1. The classic original paper is "Symbolic Execution and Program Testing" by James King in Communications of the ACM, 19(7) (July 1976),
2. https://github.com/tatsy/docker-ubuntu-cxx/blob/master/Dockerfile
3. http://klee.github.io/docs/coreutils-experiments/
4. http://klee.github.io/tutorials/testing-coreutils/