15-745 Optimizing Compilers
Project Final Report

Symbolic Execution in Difficult Environments

David Renshaw
renshaw@cmu.edu

Soonho Kong
soonhok@cs.cmu.edu

April 28, 2011

1 Introduction

1.1 Problem / Opportunity

Symbolic execution is a technique that can automatically generate high-coverage test cases for programs. The idea is to step through a program while keeping some input data *symbolic*—i.e. not fully specified. Then whenever a branch point is reached, the program state can be forked and both branches can be taken, with the symbolic data appropriately constrained on each branch. When a potentially dangerous operation on symbolic data is encountered, a contraint solver can be invoked used to try to generate concrete inputs that cause the program to go wrong at that operation.

Symbolic execution becomes difficult for programs that interact with an external environment like a filesystem. Although one could achieve "high coverage" by returning arbitrary values from external calls, if these values are not consistent with the behavior of the real environment, any "bugs" found on these branches may be spurious. Thus, for symbolic execution to work in this case—specifically if we want there to be no false positives—then the behavior of these interactions must be modeled. The purpose of this project is to explore what is involved in such an undertaking.

1.2 Approach

Our approach is to write a new filesystem model for KLEE [1], which is a symbolic execution engine built on LLVM.

1.3 Related Work

KLEE has been used to find bugs in GNU coreutils [1]. Zamfir and Candea have extended KLEE to add support for threads, creating a debugger for concurrent programs [4]. They have used their system, called "ESD," to synthesize execution paths for difficult-to-reproduce bugs in open source projects such as SQLite. Sasnauskas et al have extended KLEE with support for distributed programs [2]. They have used their implementation, called "KleeNet," to find interaction-related bugs in the TCP/IP stack of the Contiki operating system for embedded systems. Chipounov et al have integrated KLEE with a virtual machine and a dynamic binary translation engine to create S2E [3], an analysis framework reportedly capable of testing an entire software stack such as Windows.

1.4 Contributions

We identify a strategy for extending the filesystem modelling of KLEE. We implement this strategy and we test its effect on the symbolic execution of a selected set of GNU CoreUtils.

2 Design and Approach

2.1 KLEE's Existing Model for the Environment

To increase code coverage for the program that interacts with a file system, KLEE provides environment models and redirects library calls to the models. The KLEE's file-system models understand the semantics of POSIX system calls such as open, read, write, stat, lseek, ftruncate, and ioctl. The models generate the required constraints for the programs, which enable KLEE's symbolic execution to achieve high coverage.

Example Consider a program which opens a file whose pathname is given as a command-line argument. In general, this kind of program has two parts: (1) routine that processes the opened file and (2) routine that handles I/O errors such as `FileNotFound`. In the symbolic executions of this program, KLEE opens a file by calling `__fd_open` function. This function is KLEE's model function for `open` system call and produces constraints so that KLEE generate test cases which cover both of (1) and (2) parts of the program. These models are defined in `klee/runtime/POSIX` directory.

Components KLEE's models for file-system consists of the following components:

- Symbolic File-system : It is a virtual file-system which is provided to the target program. It maintains a list of symbolic files with other counters such as number of I/O failures. KLEE's POSIX system call functions treat symbolic files as regular files. At the beginning of each symbolic execution, KLEE initializes symbolic file system and use it during the execution.

- Symbolic Environment : It represents virtual environment which provides programs with information about current environment such as `umask` and number of opened file descriptors.

Symbolic file and concrete file KLEE's file-system models distinguish symbolic files and concrete files. A file is concrete if its pathname is a non-symbolic variable. For example, if we have a program statement `f = open("./a.txt", O_RDONLY)`, then file descriptor `f` represents a concrete file because its pathname `"./a.txt"` is non-symbolic, which means fixed. It is important to note that KLEE's file-system models treat concrete file by directly invoking standard POSIX system call. For example, if we have `"./a.txt"` in current directory, then KLEE opens it and proceed with the following code. KLEE does not cover the code which handles the case where `"./a.txt"` does not exist.

A file is symbolic if its pathname is a symbolic variable. For a symbolic file, KLEE's file-system models generate constraints and those constraints create symbolic file-system for each test-case and possibly cover multiple branches of the code.

Parameterized KLEE's file-system models are parameterized. Maximum number of symbolic files and their length are provided as command-line arguments. This help KLEE's users customize their testcase generation to balance running time and coverage.

2.2 Limitation: Single Directory with Symbolic Files

KLEE's symbolic file-system is "crude, containing only a single directory with N symbolic files in it."[1] In other words, the file system is flat and symbolic files have pathnames such as `"A"`, `"B"`, and `"C"` without directory hierarchy[1].

2.3 Our Extension

We extend KLEE's file-system models to support directory hierarchy[2] . To achieve this goal, we modify and extend current implementation in the following ways:

1. **Extend Data Structures:** We extend current file-system data-structures to allow our file-system to have directories. A structure `exe_disk_file_t`, which is used to represent symbolic files, is extended to have `is_dir` field which is a flag for directory, and `sym_dfiles` field to contain pointers to the nested entries.

   Note that we limit the number of sub-directories in a directory and the depth of nested directories. This limits the total number of symbolic files and prevents constraints explosion. These arguments are provided

[1]More precisely, the first symbolic file has a pathname `"A"`, the second one has `"B"`, etc.
[2]Our implementation is available at `http://code.google.com/p/cmu15745/`

as command-line arguments to KLEE. A data structure for file-system `exe_file_system_t` keeps those constants which are used in the models.

2. **Directory Construction:** We modify symbolic file creation function `__create_new_dfile` to possibly create a symbolic directory. Whenever we create a symbolic file, we make the field `is_dir` symbolic so that KLEE can have a chance to set this file as a directory. If KLEE sets a file as a directory, then we build files and sub-directories for that directory, recursively.

   Note that we set proper values for the inode number(`st_ino`) and the mode of the file(`st_mode`) so that the directory is treated as a directory in the standard POSIX implementation.

3. **Directory Listing:** We implements directory listing for the symbolic directories. Previously, directory listing function `__fd_getdents` just returns an error when the argument is symbolic. Now it returns a list of files and directories as a contents of the given directory if the argument is a symbolic directory.

4. **Other Functions:** We extend other implemented POSIX functions to support symbolic directories. For example, we should allow to open a directory with POSIX `open` function, but we should not allow to write to a directory with POSIX `write`. For functions in `/runtime/POSIX/fd.c` which implement the behavior of POSIX system call, we add proper routines for the case of symbolic directories.

3 Experimental Setup

We set up experiments to compare our version of modified file-system models with the KLEE's original file-system models.

3.1 Benchmarks

We use the subset of the programs in the `coreutils-6.11` benchmark[3] which depend on a file-system. `coreutils` benchmark is the set of linux basic utilities and was also used as a benchmark in the original KLEE paper[1]. Here is the list of 20 coreutils that are used in our experiments.

basename	cat	chcon	chgrp	chmod	chown	cp
dir	dirname	du	link	ln	ls	mknod
mktemp	mv	rm	rmdir	stat	vdir	

3.2 Arguments to KLEE

After the publication of the KLEE paper (2008), KLEE command-line arguments have been changed and those changes are not well-documented. As a result, it

[3]Available at `http://ftp.gnu.org/gnu/coreutils/coreutils-6.11.tar.gz`.

was difficult to reproduce the result of the KLEE paper with arguments used in the paper. We found the arguments in figure 1 from the thread[4] in the `klee-dev` mailing list and used them in our experiments. The "posix-model" and "sym-dirs" options belong to our extensions of KLEE. We found that, in order to prevent premature abort on some of the inputs, we needed to increase the number of allowed open file descriptors from 1024 per process to 8192 per process.

```
 klee --simplify-sym-indices  \
 --output-module --max-memory=1000 --disable-inlining \
 --optimize --use-forked-stp --use-cex-cache \
 --libc=uclibc --posix-runtime --posix-model=[old|modified] \
--allow-external-sym-calls  --only-output-states-covering-new \
--run-in=<sandbox>  --output-dir=<output_dir> --max-sym-array-size=4096 \
 --max-instruction-time=10. --max-time=1800. --watchdog \
 --max-memory-inhibit=false --max-static-fork-pct=1 \
 --max-static-solve-pct=1  --max-static-cpfork-pct=1 \
 --switch-type=internal --randomize-fork --use-random-path \
 --use-interleaved-covnew-NURS --use-batching-search \
 --batch-instructions 10000 --init-env \
 <program>.bc --sym-args 0 1 10  --sym-args 0 2 2 \
 --sym-files 2 8  --sym-stdout [--sym-dirs 1 1]
```

Figure 1: Command-line arguments for KLEE that we used in our experiments.

3.3 Experiment Environment

We run the experiments on a Dell Optiplex 960 running Ubuntu 10.04. the machine has 3.8 GB of RAM and an Intel Core 2 Quad processor which has 4 cores at 2.83GHz each.

We set the sandbox directory—the concrete directory in which KLEE runs its input program—to be a directory originally containing about 10 files and 5 subdirectories. We did not clean up this directory between runs. Over the course of the experiements, this directory accumulated about 50 junk files as the programs being tested wrote to concrete files. These new files have jibberish names such as "BXCLo" and almost all have no contents. We believe that these files had negligible effect on the runs. Moreover, we ran the corresponding runs for modified and unmodified KLEE in parallel, so for any given program being tested, the comparison should be fair.

[4]`http://keeda.stanford.edu/pipermail/klee-dev/2011-February/000572.html`

4 Experimental Evaluation

4.1 Metric

We have the following evaluation metrics:

- Instruction coverage (ICOV) & Branch coverage (BCOV) : Our goal is to increase test coverage. Instruction coverage is a percentage of instructions covered by generated test cases. We have instruction coverage instead of line coverage because KLEE runs on top of the LLVM.

  Branch coverage is another metric of code coverage. It counts number of branches taken by generated test cases and divides it by the number of total branches.

- Number of generate tese cases : One of goals of symbolic executions is to minimize the number of generate test cases while maintaining high code coverages.

- Time : We want to minimize the total time which is used for generating test cases. In our experiments, we set timeout with 30 minutes. We collect test cases generated until then.

4.2 Result and Discussion

Table 1 shows our experimental results.

4.2.1 Time

Our modified file-system increases the running time of test-case generation for a few cases. `dirname` program is an example. Previously, it took 33.63 seconds to finish, but we have time-out with the modified implementation. However, we have increases in both of instruction coverage (+1.70, 37.96 → 39.66) and branch coverage (+0.84, 25.84 → 26.68). So, this increased running time is spend well.

4.2.2 Coverage

Figure 2 and 3 show the coverage improvements. Except four programs, we have increases in coverage up to 5.88 percentage point (ICOV, `stat`) and 3.76 percentage point (BCOV, `chown`). Note that three programs that we have decrease in coverage (`du`, `ln`, and `ls`), we already have time-out. We think those coverage decreases are due to increased search space. Especially, we have coverage increases for the cases `cat`, `basename`, and `dirname` where we had time-out with the original implementation.

The results of symbolic execution are very sensitive to the input parameters. We would like to know if and how our extension is actually an improvement. We have been using the KCachegrind tool to try to determine the nature of the

Path	KLEE				Modified KLEE			
	Time(s)	ICov(%)	BCov(%)	# Tests	Time(s)	ICov(%)	BCov(%)	# Tests
basename	37.01	38.42	26.36	42	78.11	40.08	27.18	43
cat	1802.75	40.04	28.54	50	1801.53	41.96	29.87	45
chcon	1807.20	30.36	20.69	53	1805.60	31.48	21.32	55
chgrp	1809.60	45.52	31.65	66	1810.47	46.95	33.02	79
chmod	1806.58	39.77	27.88	95	1809	40.68	28.43	97
chown	1810.09	37.74	27.83	73	1810.76	43.60	31.59	91
cp	1805.76	38.22	29.14	107	1807.70	39.22	30.06	113
dir	1285.78	38.49	28.28	70	1859.12	37.50	27.07	116
dirname	33.63	37.96	25.84	31	65.87	39.66	26.68	38
du	1804.39	42.96	30.02	79	1812.92	39.59	27.67	75
link	1806.60	46.78	32.71	48	1805.33	48.08	33.20	41
ln	1805.71	51.72	38.85	101	1809.98	49.26	38.22	105
ls	1802.66	39.84	29.66	82	1804.09	37.96	27.69	60
mknod	1807.83	47.79	33.04	73	1809.24	49.10	33.70	75
mktemp	1806.13	48.18	35.20	65	1807.79	50.25	37.17	74
mv	1806.55	36.18	26.43	92	1811.07	38.22	27.91	106
rm	1805.71	32.50	23.70	49	1809.11	33.66	24.89	58
rmdir	1805.65	38.69	26.71	40	1807.31	40.55	28.58	60
stat	1804.43	37.59	25.97	64	1820.02	43.47	29.40	79
vdir	1097.58	37.08	26.83	33	973.63	37.74	27.17	37

Table 1: Experimental Results: ICov(%) : Instruction(line) coverage, BCov(%) : Branch coverage, # Tests : Number of generated tests.

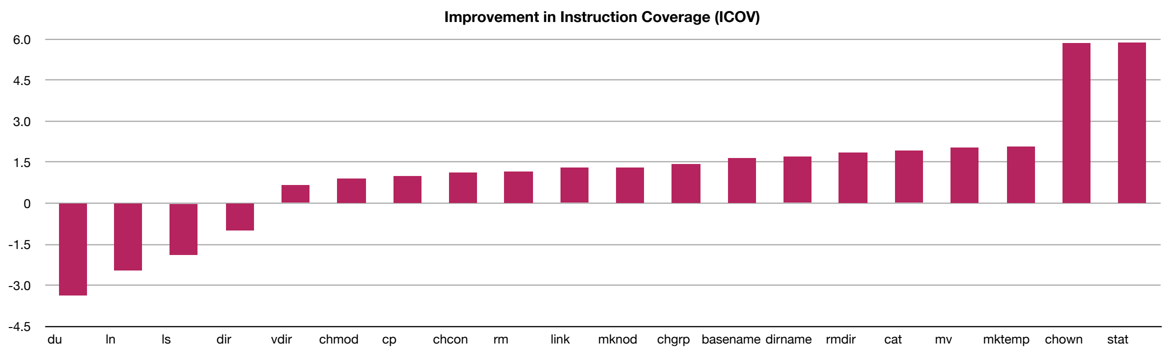

Figure 2: Improvement in Instruction Coverage(ICOV)

coverage increases. KCachegrind lets the user examine coverage on a line-by-line basis. Unfortunately, it is hard to understand the main differences that we see. For example, for `stat`, the modified version of KLEE gets considerablt better coverage. The only significant way that this manifests itself in terms of the code from the `stat.c` file is that the `usage` function is not covered as well

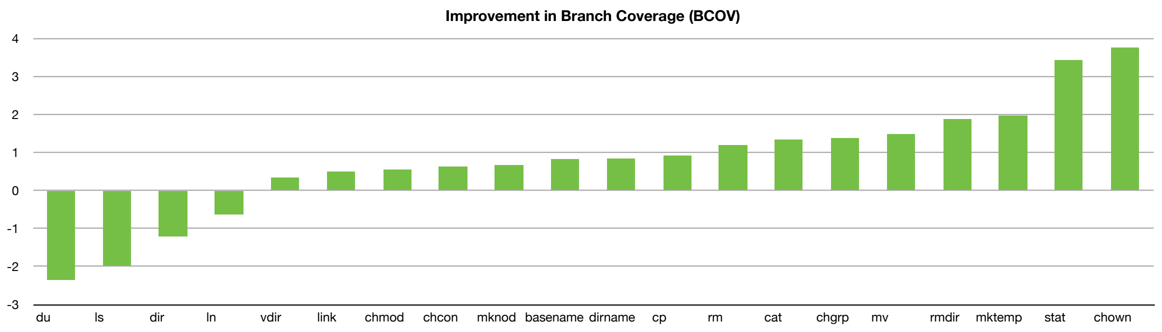

Figure 3: Improvement in Branch Coverage(BCOV)

by unmodified KLEE. The filesystem model should not affect the execution of a help message, so this discrepancy is puzzling.

4.2.3 Number of test cases

We observed about 10% increase in number of generated test-cases on average. Because higher code coverage inevitably causes larger number of test-cases, we think that this increase is natural consequences.

5 Surprises and Lessons Learned

Our primary mistake with this project was in underestimating how much time it would take for us to understand the parts of KLEE that we would be interacting with. We ended with very little time to implement our intended changes.

Also, we learned that small changes in the setup of symbolic execution can have large and hard to analyze effects on the results of the execution.

6 Conclusions and Future Work

The next step in this project would be to understand better where our coverage increases are coming from, and, using that knowledge, to further improve our model. It would be interesting to try the modified version of KLEE on other input programs, such as the BUSYBOX utilities tested in [1].

7 Distribution of Total Credit

We have designed the framework together. Soonho has done the implementation part, and David has done the experiments and analysis part. The credit distribution is 50% - 50%.

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[2] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment, 2010.

[3] Volodymyr Kuznetsov Vitaly Chipounov and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[4] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2010.