1 INTRODUCTION

In this coursework, I have accomplished several subtasks according to the coursework requirements. To do so, I have utilized the required symbolic execution tool KLEE to perform testing on the suggested project GNU Coreutils. The following sections provide a description of how I have configured the tool, measured the line coverage and how I have improved the coverage.

2 CONFIGURATION

In order to use KLEE, it is required to successfully install the recommanded packages and libraries into the host machines. KLEE official website [2] provides various guidelines on how to install and configure KLEE locally in linux based machine. Moreover, detailed instructions are also mentioned for using KLEE as a docker container. KLEE developer community releases up-to-date docker containers with additional features and optimization. Therefore, I decided to use the KLEE latest docker container [3] in my local machine to complete the coursework.

First, I installed docker and then I pulled the KLEE docker image in my local machine. The container includes pre-installed packages for running some basic KLEE commands on sample source files. However before applying KLEE commands on the complete project, I needed to install some recommanded utility packages and also configured the container for advanced usages. I installed *wllvm* package and configured it inside the docker container so that a whole project can be built by *llvm*. After that I checked the installation of all the required packages such as *gcov*, *clang*, *llvm* and *wllvm*, inside the container to execute KLEE on GNU Coreutils project.

3 MEASURING LINE COVERAGE

In this section, I described how I measured the line coverage obtained by the KLEE generated test cases for two programs namely ***chmod*** and ***mv*** chosen from the Coreutils project. According to the provided tutorial I used the Coreutils-6.11 version in this coursework. Now, before generating the symbolic test case and line coverage for each of the selected program, it is required to compile the project with ***gcov*** and ***llvm***. The following steps were performed for measuring line coverage for both programs.

Step-1 First the complete Coreutils project was built with ***gcov***
Step-2 Next the project was also built with ***llvm***.
Step-3 Then, symbolic data was introduced to the selected programs **chmod** and ***mv*** .
Step-4 After that KLEE generates and runs the test cases based on the symbolic data
Step-5 KLEE's internal statistics was viewed to know how the test cases were produced
Step-6 Finally selected programs were run with **gcov** to measure the coverage obtained by the KLEE generated test cases

The following subsection provides the detailed explanation of the executed commands for the program ***chmod*** and ***mv***.

3.1 chmod

chmod is a utility program that helps to change the file mode bits according to the given mode. It is a very well-known program and can be found in the Coreutils project. As previously mentioned, first the complete Coreutils project was build with ***gcov*** and ***llvm***. Then, I run the ***klee*** command to introduce symbolic data during running the bitcode version of the program ***chmod***. The command indicates, KLEE is run with ***uclibc*** and ***POSIX*** as runtime support symbolic argument of 3 characters. After running this command KLEE generated 49999036 instruction and the number of completed and generated paths is 5386. Now to view the KLEE's internal statistic, ***klee-stats*** command was run. Here the ***klee-last*** refers to a directory that contains the recently generated test cases produced by KLEE.

```
~/coreutils-6.11/obj-llvm/src$ klee --libc=uclibc --posix-runtime ./chmod.bc --sym-arg 3
~/coreutils-6.11/obj-llvm/src$ klee-stats klee-last
```

Table 1 represents the internal statistic of KLEE where ***ICov*** refers to the percentage of LLVM instructions which were covered. ***BCov*** also shows the percentage of branches that were covered. The command is executed within 3 minutes. Now to measure how much line coverage can be obtained by the KLEE generated test cases, I run *gcov* with KLEE generated test cases. To do so, I utilized the ***klee-replay*** command to run the test cases with *gcov*. The following command was run to executed the KLEE generated test cases with gcov. It is observed that for the program ***chmod*** **(File '../../src/chmod.c')** the coverage was 33.14% **(Lines executed:33.14% of 175)**.

```
~/coreutils-6.11/obj-gcov/src$ klee-replay ./chmod ../../obj-llvm/src/klee-last/*.ktest
~/coreutils-6.11/obj-gcov/src$ gcov chmod
   File '../../src/chmod.c'
   Lines executed:33.14% of 175
```

Table 1: KLEE Internal Statistic for chmod

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	49999036	177.44	17.42	12.79	34259	11.76

3.2 mv

mv is a simple command that is used to move or rename file and directory. Similar to ***chmod***, it is also widely used utility program as a part of Coreutils project. As I previously built the Coreutils project with *gcov* and *llvm*, now for the program ***mv***, I just simply run ***klee*** command with ***uclibc*** and ***POSIX*** as runtime support including the symbolic argument of 3 characters.

```
~/coreutils-6.11/obj-gcov/src$ klee --libc=uclibc --posix-runtime ./mv.bc --sym-arg 3
~/coreutils-6.11/obj-gcov/src$ klee-stats klee-last
```

Again to view the KLEEE's internal statisic, I again run the ***klee-stats*** command. It is found that in total 55147168 instructions were done by KLEE and the number of completed and generated paths is 5344. And the command executed within 3 minutes. **Table 2** demonstrates the internal statics of KLEE during running the ***mv*** program. The LLVM instruction coverage is **ICov=14.79%** and the branch coverage **BCov=10.21%**.

Table 2: KLEE Internal Statistic for mv

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	55147168	187.54	14.79	10.21	41973	10.67

Now, for measuring how much coverage is done by the KLEE generated test cases, again I run the ***klee-replay*** command and with the help of ***gcov***, it is found that the line coverage for the program **mv** (**File '../../src/mv.c**) is 52.38% (**Lines executed:52.38% of 168**).

```
~/coreutils-6.11/obj-gcov/src$ klee-replay ./mv ../../obj-llvm/src/klee-last/*.ktest
~/coreutils-6.11/obj-gcov/src$ gcov mv
   File '../../src/mv.c'
   Lines executed:52.38% of 168
```

4 IMPROVING LINE COVERAGE

In this section, I discuss how I have increased the coverage for the selected two programs **chmod** and **mv**. To enhance the coverage, there are two ways. I can utilize, KLEE's provided command line arguments and flags or I may modify the existing source code of the selected programs from the Coreutils project. However, if I modify the existing source code, I need to execute the regression test suit of Coreutils project to ensure that the modification will not affect the functionality of the programs. It seems difficult for me as I don't have expertize in C/C++ programming language. Therefore, I have decided to use the KLEE's provided command line arguments and flags to improve the coverage. The following subsections describe the coverage improvements of the two program **chmod** and **mv**.

4.1 chmod

To improved the coverage, I introduced multiple flags with the **klee** command. The first flag I used is *–optimize*. This will tell KLEE to run the LLVM optimization passes on the bitcode module before executing and thus it will remove any dead code. Then *–max-instruction-time=30s* indicates that it will spend 30 seconds for executing any instruction. Next, *max-time=20min* helps to run klee path exploration upto 20 minutes. Similarly *–libc=uclibc* and *–posix-runtime* represent that KLEE will use ***uclibc*** and ***POSIX*** as the runtime support library.

```
:~/coreutils-6.11/obj-llvm/src$ klee --optimize --max-instruction-time=30s --max-time=20min
    --libc=uclibc --posix-runtime ./chmod.bc --sym-args 1 3 5  --sym-files 1 10 --sym-stdin 2
    --sym-stdout

:~/coreutils-6.11/obj-llvm/src$ klee-stats klee-last
```

Now I add some augments that add symbolic values into the program. For example *–sym-args 1 3 5* , tells KLEE to use zero to three command line arguments where the first one character long and the others 3 and 5 characters long. Then the options *–sym-files 1 10* says to use standard one input file holding 10 bytes of symbolic data. Next options *–sym-stdin 2* imposes KLEE to use standard input with 2 symbolic

Table 3: KLEE Internal Statistic for chmod (Improved Coverage)

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	13010333	1205.02	33.34	22.05	21763	48.20

data. And finally, *–sym-stdout* tells to make symbolic output. After running this *klee* command, KLEE explored paths and produced test cases. To view KLEE's internal statistics, I run the *klee-stats* command. **Table 3** demonstrates that statistics. It is analysed that the instruction coverage and branch coverage have improved a lot.

To view how much coverage has improved with the newly KLEE generated test cases, I again run these test cases with *gcov* for the program *chmod*. The coverage significantly improve from 33.14% to 72.57%. It implies that the provided options and flags have positively impacted the keel generated test cases that gain high coverage compared to the previously generated test cases.

```
:~/coreutils-6.11/obj-gcov/src$ klee-replay ./chmod ../../obj-llvm/src/klee-last/*.ktest
:~/coreutils-6.11/obj-gcov/src$ gcov chmod
   File '../../src/chmod.c'
   Lines executed:72.57% of 175
```

4.2 mv

Similar to *chmod*, to improve the coverage of *mv*, I added some options and flags in the *klee* command. The first option is *–optimize* that is already discussed in the previous section. Next options *–max-instruction-time=20s* and *–max-time=20min* represent that any instruction would be executed up-to 20 seconds and the *klee* command run up-to 20 minutes. Then I add new option called *–search=random-path*. It tells *klee* to maintain a binary tree recording the program path followed for all active states. Here the leaves of the tree are the current states and the internal nodes are places where execution forked [1]. As already discussed *–libc=uclibc* and *–posix-runtime* is used to get ***uclibc*** and ***POSIX*** as the runtime support library.

```
:~/coreutils-6.11/obj-llvm/src$ klee --optimize --max-instruction-time=20s --max-time=20min
    --search=random-path --libc=uclibc --posix-runtime ./mv.bc --sym-args 1 2 4  --sym-files
    1 10 --sym-stdin 2 --sym-stdout

:~/coreutils-6.11/obj-llvm/src$ klee-stats klee-last
```

Next I introduced some augments that initiate symbolic values into the program *mv*. The first argument *–sym-args 1 2 4* , tell KLEE to use zero to three command line arguments where the first one character long and the others 2 and 4 characters long. Next *–sym-files 1 10* indicates to utilize standard one input file holding 10 bytes of symbolic data. Then argument *–sym-stdin 2* imposes KLEE to use standard input with 2 symbolic data. The last , *–sym-stdout* tells to make symbolic output. Then I run this command and also run *klee-state* to view KLEEs's internal statistic. It is observed that both the instruction and branch coverage have improved as shown in **Table 4**.

Table 4: KLEE Internal Statistic for mv (Improved Coverage)

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	37683201	1240.19	29.24	20.63	29011	60.63

In order to know,how much coverage improvement has been achieved, I run the newly KLEE generated test cases with *gcov* for the program *mv*. The coverage improvement is strongly positive as it increase to 81.55% from 52.38%. This implies the options and flags in the *klee* command have strong influence in the newly generated test cases and responsible for this enhanced coverage.

```
:~/coreutils-6.11/obj-gcov/src$ klee-replay ./mv ../../obj-llvm/src/klee-last/*.ktest
:~/coreutils-6.11/obj-gcov/src$ gcov mv
   File '../../src/mv.c'
   Lines executed:81.55% of 168
```

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[2] KLEE. 2020. KLEE. http://klee.github.io/. [Online], [Accessed: 2020-14-02].

[3] klee docker. 2020. klee-docker. https://hub.docker.com/r/klee/klee/. [Online], [Accessed: 2020-14-02].