实验报告

课程名称：\_\_\_软件工程\_\_\_\_\_

实验名称：\_\_\_\_\_\_KLEE\_\_\_\_\_\_

专业班级：\_\_\_软件工程2班\_

学 号：\_\_\_1813075\_\_\_\_\_\_

姓 名：\_\_\_刘茵\_\_\_\_\_\_\_\_\_

2020年 12月 3日

实验四

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 | 自动化测试工具 KLEE 的使用 | | |
| 实验地点 | 泰达五区 | 实验时间 | 2020/12/03 |
| 实验目的和要求 | | | |
| 本实验的目的是：  1）熟悉 Linux 系统使用方式；  2）了解 LLVM 的架构和使用方式；  3）了解 Docker 使用或者 github 的使用方式；  4）了解并掌握基于约束的自动化测试工具 KLEE 的基本原理和使用； | | | |
| 实验环境 | | | |
| 操作系统：Linux version 5.8.15   (Ubuntu 9.3.0-17ubuntu1~20.04)  虚拟机软件：Oracle VM VirtualBox -v 6.1.16 | | | |
| 实验过程 | | | |
| 1. **利用 docker 镜像（http://klee.github.io/docker/ ）的方式安装 KLEE；** 2. 安装docker，配置国内源   [Ubuntu · Docker -- 从入门到实践 (gitee.io)](https://docker_practice.gitee.io/zh-cn/install/ubuntu.html)  卸载旧版本：  sudo apt purge docker-ce  sudo apt autoremove            配置国内源：      {  "registry-mirrors": [  "https://t5xf3iww.mirror.aliyuncs.com",  "https://hub-mirror.c.163.com",  "https://mirror.baidubce.com"  ]  }   1. 使用docker安装klee   docker pull klee/klee     1. 创建基本画像(永久)   docker run **-ti** **--name**=klee **--ulimit**='stack=-1:-1' klee/klee     1. 进入基本画像     画像提供了 clang 和 LLVM 工具，分别做编译 c 文件和将 c 文件转换字节码 文件的工作。   1. **使用 KLEE 完成 First tutorial**   新建的画像中有 klee\_build 和 klee\_src 两个文件夹，klee\_src 包含了构建 KLEE 的源码，klee\_build 是 klee\_src 构建的工程。klee\_src 中含有examples 文件夹，其中包含了四个初始文件夹，分别是get\_sign,islower,regexp,sort，每个文件夹里面包含一个同名 c 文件，是官方给出的测试代码例子。  测试一个判断正负数的程序，打开前文提到的 get\_sign.c 文件，可以看到测试函数 get\_sign 和 main 函数。其中 klee\_make\_symbolic 是 KLEE 工具 自带的测试函数，通过自定义的变量，不断产生值赋给 a，以此完成自动生成样例功能。     * 编译该 c 文件:   clang **-I** ../../include **-emit-llvm** **-c** **-g** **-O0** **-Xclang** **-disable-O0-optnone** get\_sign.c   * 同目录下生成了一个 get\_sign.bc 字节码文件,然后进行测试:      * 列出当前目录所有文件:   其中 klee-out-0 是本次测试结果,klee-last 是最新测试结果,每次测试  后覆盖。klee-last 中包含最新测试的缺陷说明和测试样例等文件。 打开  error 文件查看具体缺陷。     * 重播实验：   $ export LD\_LIBRARY\_PATH=path-to-klee-build-dir/lib/:$LD\_LIBRARY\_PATH  $ gcc **-I** ../../include **-L** path-to-klee-build-dir/lib/ get\_sign.c **-lkleeRuntest**  path-to-klee-build-dir替换为/home/klee/klee-build     1. **使用 KLEE 完成 Second tutorial**   测试一个简单的正则表达式的匹配函数。该实例的源程序在klee\_src/examples/regexp/regexp.c。   * 编译该 c 文件:      * 使用 llvm tools 验证:      * 用 KLEE 执行代码:其中参数 --only-output-states-covering-new 是用于限定输出,缺省情况下,KLEE 会遍历所有的路径,对于有些程序就需要花费很多时间,或者无限时间,所以 KLEE 可以通过 ctrl+c 终止运行,也可以通过设置如下的参数,对运行和路径进行限制。   -max-time=seconds: 指定最大运行时间.  -max-forks=N: 在 N 符号分支后停止,并且运行剩余路径到终止  -max-memory=N: 用于指定最大可以消耗的内存,单位为 Mbytes     * 错误报告:当 KLEE 在执行程序时发现错误,那么它会生成一个 test case   来展示该错误,并把相关信息写入文件 testN,类型 TYPE 为 err 的文件  中。       * 改变测试的方法: 在符号化后,在 buffer 后面增加串结束符号‘\0’.   Vim>>/klee\_src/examples/regexp/Regexp.c       1. **使用 KLEE 对 Coreutils 进行自动化测试**  * 利用 gcov 构造 CoreUtils: 下载 coreutils(下载压缩包到ubuntu的softtest文件夹并解压，通过docker容器共享文件夹),截图用的是版本 6.10。gcov(GCC Coveage)是一个测试代码覆盖率的工具,配合 Gcc 共同实现对C/C++文件的语句覆盖和分支覆盖进行测试。   coreutils-6.10$ mkdir obj-gcov  coreutils-6.10$ cd obj-gcov  obj-gcov$ ../configure **--disable-nls** CFLAGS="-g -fprofile-arcs -ftest-coverage"  ... verify that configure worked ...  obj-gcov$ make  obj-gcov$ make **-C** src arch hostname  ... verify that make worked ...        成功完成后可以进入 src 目录运行编译生成的代码:     * 在运行中会生成.gcda 文件,包含对程序运行情况的统计。也可以用 gcov   工具生成可阅读的格式。     * 利用 LLVM 构造 Coreutils。这里使用 klee-gcc 进行编译。   下载wllvm，并将环境变量设置为clang       * 使用 LLVM 构建核心实用程序，依次运行：   coreutils-6.11$ mkdir obj-llvm  coreutils-6.11$ cd obj-llvm  obj-llvm$ CC=wllvm ../configure **--disable-nls** CFLAGS="-g -O1 -Xclang -disable-llvm-passes -D\_\_NO\_STRING\_INLINES -D\_FORTIFY\_SOURCE=0 -U\_\_OPTIMIZE\_\_"  obj-llvm$ make  obj-llvm$ make **-C** src arch hostname     * 成功后,可以在 src 文件夹下面看到每一个命令会有三个文件,例如 cat会有 cat、cat.bc、cat.o。运行一个 cat 程序:      * 利用 KLEE 解释执行。除了可以利用上述方式直接执行(实际上也是利用llvm 解释执行),还可以利用 klee 执行。基本格式是:klee klee 的参数要运行的程序 程序参数。运行正常的程序链接的是 C 库,在这里的例子中是直接运行 LLVM bitcode,为了提高效率,需要为程序制定外部调用所用的函数,这里利用 --libc=uclibc 指定所要使用的库; --posix-runtime 则是指定 posix runtime 为运行的操作系统基础。        * 使用 KLEE 作为翻译      * 设置--sym-args 3,也就是设置参数符号化的最大字符个数为 3。        * 利用 klee-stats 可以看到 KLEE 的一个运行情况总结。      * 其中 ICov 表示 LLVM 指令覆盖情况,BCov 表示分支的覆盖情况。这个值的   分母包括所调用库中的相关代码,所以看起来覆盖率很低,当然也可以在KLEE 中设置--optimize 参数改善这一情况,KLEE 会去掉一些 Dead code。  设置了--optimize 后运行结果如下:       * 重播klee生成的测试用例      * 使用ktest-tool查看单个文件      * klee-replay.ktest.ktest可用于读取文件并调用本机应用程序，自动传递复制 KLEE 遵循的路径所需的数据。      1. **复现论文 Figure 7 的结果。coreutils-6.10 版本和 coreutils-6.11 版本。**      1. **paste -d\\ abcdefghijklmnopqrstuvwxyz**   klee --libc=uclibc --posix-runtime ./paste.bc -d\\ abcdefghijklmnopqrstuvwxyz   * coreutils-6.10 版本      * coreutils-6.11 版本      1. **pr -e t2.txt**   klee --libc=uclibc --posix-runtime ./pr.bc -e t2.txt   * coreutils-6.10 版本      * coreutils-6.11 版本      1. **tac -r t3.txt t3.txt**   klee --libc=uclibc --posix-runtime ./tac.bc -r t3.txt t3.txt   * coreutils-6.10 版本      * coreutils-6.11 版本      1. **mkdir -Z a b**   klee --libc=uclibc --posix-runtime ./mkdir.bc -Z a b   * coreutils-6.10 版本      * coreutils-6.11 版本      1. **mkfifo -Z a b**   klee --libc=uclibc --posix-runtime ./mkfifo.bc -Z a b   * coreutils-6.10 版本      * coreutils-6.11版本      1. **mknod -Z a b p**   klee --libc=uclibc --posix-runtime ./mknod.bc -Z a b p   * coreutils-6.10 版本      * coreutils-6.11 版本（错误类型更明确）      1. **md5sum -c t1.txt**   klee --libc=uclibc --posix-runtime ./md5sum.bc -c t1.txt   * coreutils-6.10 版本      * coreutils-6.11 版本      1. **ptx -F\\ abcdefghijklmnopqrstuvwxyz**   klee --libc=uclibc --posix-runtime ./ptx.bc -F\\ abcdefghijklmnopqrstuvwxyz   * coreutils-6.10 版本      * coreutils-6.11 版本     “ptx -F\\ abcdefghijklmnopqrstuvwxyz ”在6.10中报错, 未能完成, 但在 6.11中顺利完成;   1. **ptx x t4.txt**   klee --libc=uclibc --posix-runtime ./ptx.bc x t4.txt   * coreutils-6.10 版本      * coreutils-6.11 版本(无变化)      1. **seq -f %0 1**  * coreutils-6.10 版本      * coreutils-6.11 版本（错误类型更明确）     可以看到新版本 11 中,所有的缺陷都已经被修复。   1. **自己编写一个包含缺陷的程序（要求至少 50 行代码，且包含不少于 5个缺陷），使用 KLEE 进行测试。**  * 源码   #include <stdio.h>  #include <stdlib.h>  void kleeTest(int *a*)  {      int arr[10];      int d[10];      for (int i = 0; i < 10; i++)      {*//赋初始值*          arr[i] = i;      }      if (*a* < -50)      {*//求余分母为 0*          for (int i = 0; i < 10; i++)          {              int num = i;              d[i] = arr[i] % num;          }      }      else if (*a* < -25)      {*//除法分母为 0*          for (int i = 0; i <= 10; i++)          {              int num = i;              d[i] = arr[i] / num;          }      }      else if (*a* < 0)      {*//数组越界*          for (int i = 0; i <= 11; i++)          {              arr[i] = i;          }      }      else if (*a* < 25)      {*//空指针*          int \*a = NULL;          int b = \*a + 1;      }      else if (*a* < 50)      {*//内存泄漏*          free(arr);      }  }  int main()  {      int n;      klee\_make\_symbolic(&n, sizeof(n), "n");      kleeTest(n);      return 0;  }  使用klee自动检测 | | | |
| 心得体会 | | | |
| KLEE 是一款开源的自动软件测试工具，基于 LLVM 编译底层基础，能够自动生成测试样例检测软件缺陷。  与 junit 不同的地方在于，KLEE 能对 c 程序生成字节码.bc 文件，并自动生成各类缺陷，不需要再自己编写。因而使用 KLEE 进行软件测试是比较轻松的方式。 | | | |