过程报告

1.环境配置&工具介绍

1.1AFL

> AFL（American Fuzzy Lop）是由安全研究员Michał Zalewski开发的一款基于覆盖引导（Coverage-guided）的模糊测试工具，它通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率，增加发现漏洞的概率。其工作流程大致如下：
>
> ①从源码编译程序时进行插桩，以记录代码覆盖率（Code Coverage）；
>
> ②选择一些输入文件，作为初始测试集加入输入队列（queue）；
>
> ③将队列中的文件按一定的策略进行“突变”；
>
> ④如果经过变异文件更新了覆盖范围，则将其保留添加到队列中；
>
> ⑤上述过程会一直循环进行，期间触发了crash的文件会被记录下来。

下载AFL源码（AFL2.57b Ubuntu 20.04）并编译：

```
make
sudo make install
```

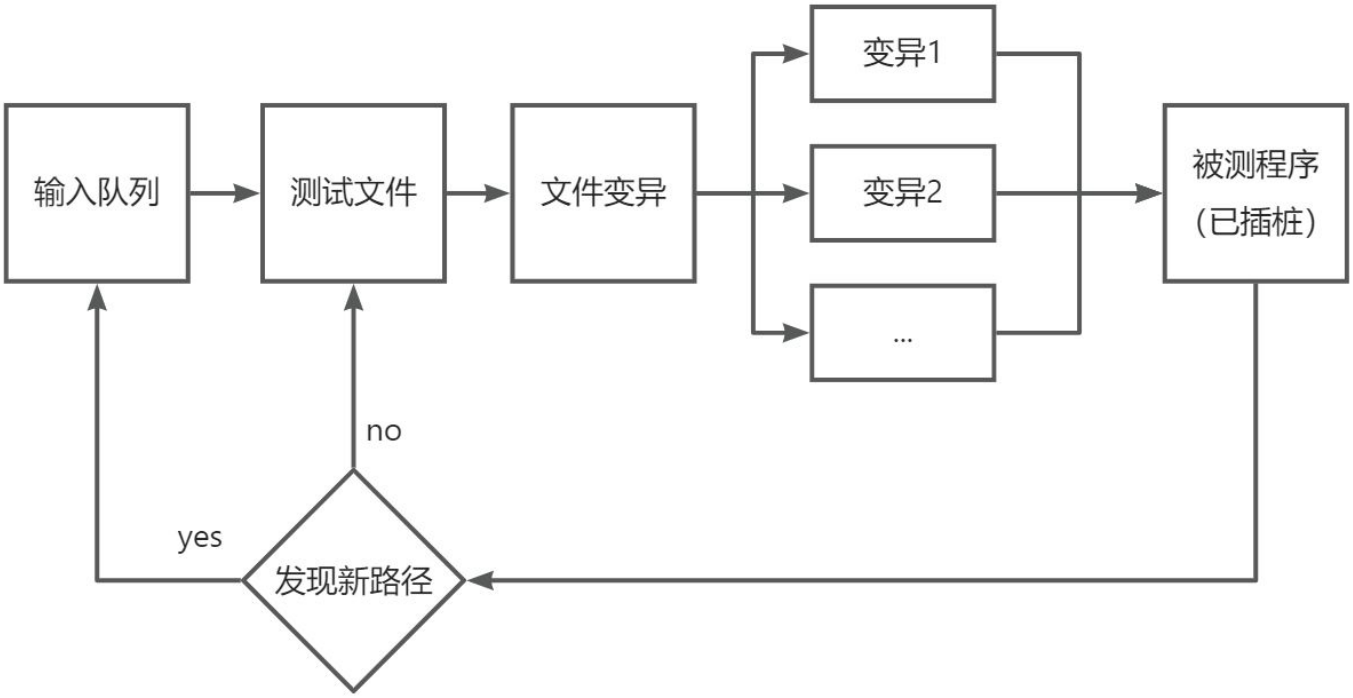
1.2Mull

> Mull在LLVM bitcode的水平上发现并创建程序在内存中的变异。
>
> 所有的变异都被注入到原始程序的代码中。
>
> 每个注入的变异都隐藏在一个condition flag下，使该特定的突变生效。由此产生的程序被编译成一个单一的二进制文件，并被多次执行，每次变异运行一次。
>
> 每次运行时，Mull都会激活一个相应变异的条件，以检查该特定变异的注入如何影响测试的执行。

```
curl -1sLf 'https://dl.cloudsmith.io/public/mull-project/mull-stable/setup.deb.sh' | sudo -E bash
sudo apt-get install mull-12 # Ubuntu 20.04
```

```
yuhaofeng@yuhaofeng:~/桌面$ mull-runner-12 --version
Mull: Practical mutation testing for C and C++
Home: https://github.com/mull-project/mull
Docs: https://mull.readthedocs.io
Support: https://mull.readthedocs.io/en/latest/Support.html
Version: 0.19.0
Commit: 3c193a0
Date: 27 Aug 2022
LLVM: 12.0.0
```

2.AFL生成测试用例



afl-fuzzing过程

2.1afl-g++编译源程序

指定编译器为afl-g++，afl会自动为源程序进行插桩（cmake为例）：

```
cmake .. -DCMAKE_CXX_COMPILER=afl-g++
```

2.2准备种子语料库

创建输入文件夹in和输出文件夹out

在输入文件夹in中放入种子文件（根据具体程序选择合适的用例）

2.3.1fuzzing并记录结果(对于库项目项目 以libxml2为例)

首先cd到libxml2路径下执行如下命令

```
./autogen.sh
--prefix=/home/zeref/AFl-fuzz_project/afl-training/challenges/libxml2/build/
--with-python-install-dir=/home/zeref/AFl-fuzz_project/afl-training/challenges/libxml2/build/
CC=afl-clang-fast
```

> 这里的--prefix指定安装目录，包括libc和include头文件那些
>
> 这里的--with-python-install-dir指定安装python的目录

随后执行AFL_USE_ASAN=1 make -j 4

> -j 4，是开四个进程进行处理
>
> 这里的AFL_USE_ASAN=1，是指开启ASAN辅助，这个玩意是基于clang的一个内存错误检测器，可以检测到常见的内存漏洞，如栈溢出，堆溢出，double free，uaf等等
>
> 由于afl呢是基于崩溃报错来反馈漏洞的，但很多时候，少量的字节 堆溢出是不会引起崩溃报错的，这样就需要额外开启ASAN来辅助挖掘漏洞

最后sudo make -j 4 install，安装过程中可能会报一些小错误，但不用管它，我们只需要用到安装好的libc库和include文件头

随后（最重要）需要编写一个harness.c文件，用来调用库文件中的内容。（如下所示）

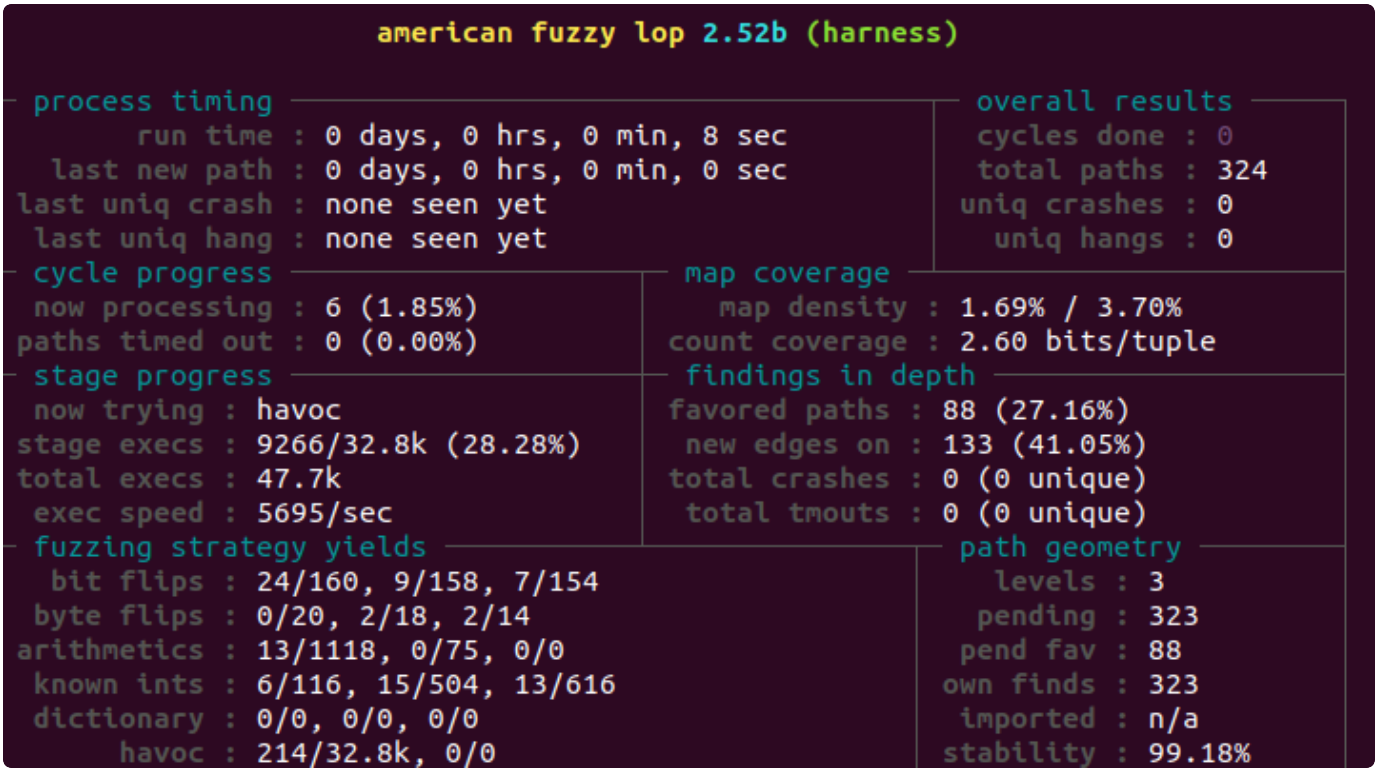
```c
#include "libxml/parser.h"
#include "libxml/tree.h"

int main(int argc, char **argv) {
    if (argc != 2){
        return(1);
    }

    xmlInitParser();
    while (__AFL_LOOP(1000)) {
        xmlDocPtr doc = xmlReadFile(argv[1], NULL, 0);
        if (doc != NULL) {
            xmlFreeDoc(doc);
        }
    }
    xmlCleanupParser();

    return(0);
}
```

> 我们将1000传递给了afl_loop()函数。这就相当于AFL在启动一个新进程前，要fuzz1000个测试用例。

AFL_USE_ASAN=1 afl-clang-fast ./harness.c -I ./build/include/libxml2/ -L ./build/lib -lxml2 -lz -lm -g -o harness 用此指令进行动态编译

随后使用afl-fuzz进行模糊测试

```c
afl-fuzz -m none -i in -o out ./harness @@
```



参考资料：https://github.com/brahmstaedt/libxml2-fuzzing

https://github.com/mykter/afl-training

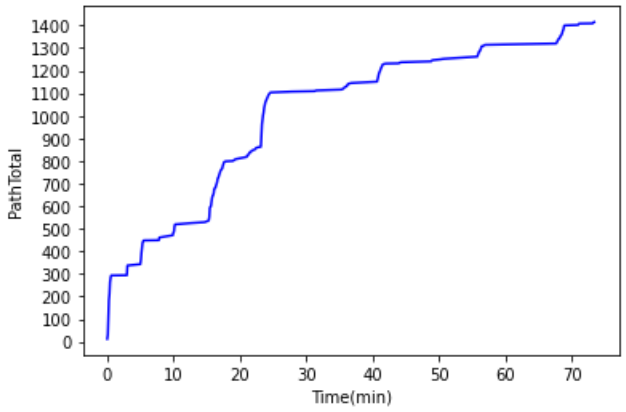
2.3.2fuzzing并记录结果(对于已经有可以执行程序的项目来说)

```
1 afl-fuzz -i ./in -o ./out [program name] @@
```

```
[!] WARNING: Some test cases are huge (62.7 kB) - see /usr/local/share/doc/afl/perf_tips.txt!
[+] Here are some useful stats:

    Test case count : 12 favored, 0 variable, 12 total
       Bitmap range : 1213 to 5151 bits (average: 3395.17 bits)
        Exec timing : 1702 to 7695 us (average: 4034 us)

[*] No -t option specified, so I'll use exec timeout of 40 ms.
[+] All set and ready to roll!


                     american fuzzy lop 2.57b (exiv2)

┌─ process timing ─────────────────────┬─ overall results ─────┐
│        run time : 0 days, 1 hrs, 6 min, 19 sec │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 5 min, 56 sec │  total paths : 1316   │
│ last uniq crash : none seen yet      │ uniq crashes : 0      │
│  last uniq hang : none seen yet      │   uniq hangs : 0      │
├─ cycle progress ─────┬─ map coverage ─┴───────────────────────┘
│  now processing : 4 (0.30%)          │    map density : 5.18% / 14.43%
│ paths timed out : 0 (0.00%)          │ count coverage : 2.36 bits/tuple
├─ stage progress ─────┼─ findings in depth ────────────────────┐
│  now trying : interest 16/8          │ favored paths : 331 (25.15%)          │
│ stage execs : 8679/290k (2.99%)      │  new edges on : 469 (35.64%)          │
│ total execs : 2.13M                  │ total crashes : 0 (0 unique)          │
│  exec speed : 594.1/sec              │  total tmouts : 13 (9 unique)         │
├─ fuzzing strategy yields ────────────┴─ path geometry ──────┐
│   bit flips : 366/397k, 124/397k, 73/397k │    levels : 2                 │
│  byte flips : 8/49.6k, 9/4275, 6/4425     │   pending : 1313              │
│ arithmetics : 293/234k, 14/228k, 0/130k   │  pend fav : 331               │
│  known ints : 10/13.8k, 23/51.5k, 23/111k │ own finds : 1304              │
│  dictionary : 0/0, 0/0, 5/10.1k           │  imported : n/a               │
│       havoc : 350/73.7k, 0/0              │ stability : 100.00%           │
│        trim : 26.16%/7178, 91.54%         └───────────────────────────────┘
                                                            [cpu:316%]
```

对上述结果进行分析（以上图为例）：fuzzer运行了一个小时，共找到执行路径1316条。

从而可以绘制随时间寻找path数量的图形：



2.4过程中遇到的问题

2.4.1种子选择

在初始的种子的选择上应该尽可能选择更加有效的种子，种子应该尽可能小，以保证执行效率。

根据查询资料，种子的选择可以遵循：

1. 使用项目自身提供的测试用例

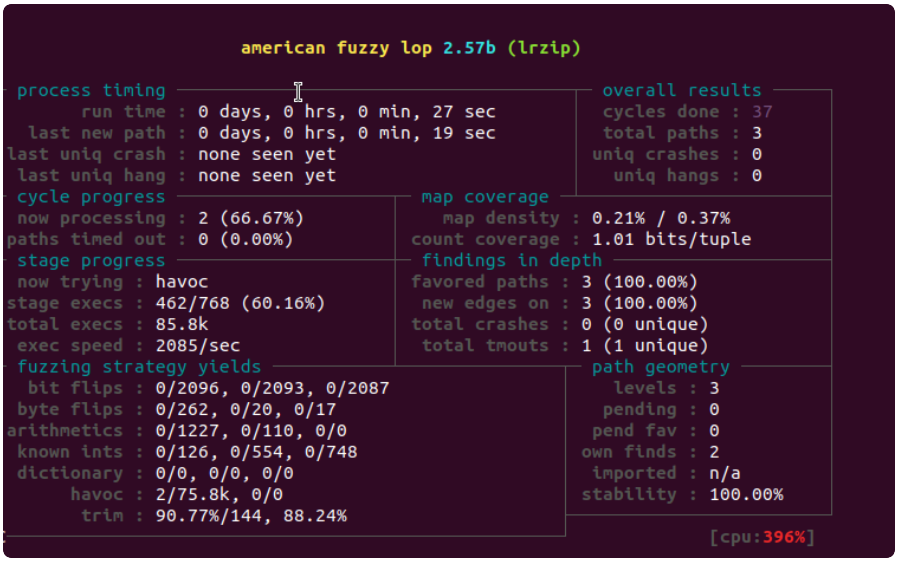
2. 目标程序bug提交页面
3. 使用格式转换器，用从现有的文件格式生成一些不容易找到的文件格式：
4. afl源码的testcases目录下提供了一些测试用例
5. 其他开源的语料库

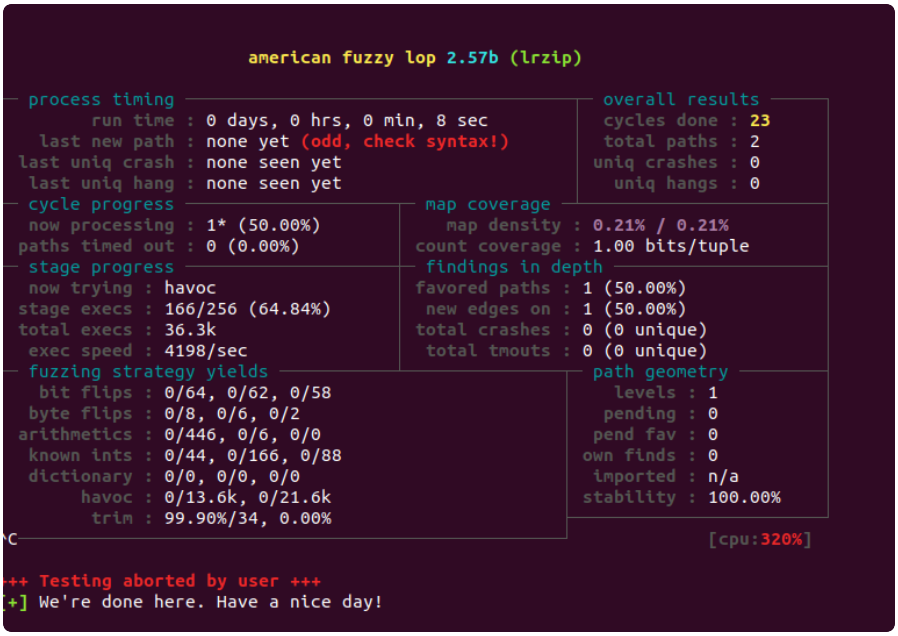
2.4.2syntax error

在输入格式上出现问题，一般出现是由于没有为测试程序指定输入或者输入错误

使用@@来代替输入用例文件，Fuzzer会将其替换为实际执行的文件

※lrzip项目在fuzz时出现不明原因的syntax error，输入文件夹一开始只放入test0时正常运行，再加入test1时产生syntax error，但再删除test1并且清空输出文件夹后（即恢复只有test0的初始状态）仍产生syntax error，如图所示。





后删除in文件夹后重建同名文件夹，放入测试用例后仍然syntax error；

解决方式是新建了不同名的输入输出文件夹，放入相同测试用例后正常运行，但是报错原因不明。

2.4.3 AFL qemu模式编译错误

在使用AFL fuzz无源码的二进制文件时需要使用qemu mode，直接对二进制文件进行插桩（而非编译时插桩）。（也可以使用dumb mode，但是dumb mode下只是对输入文件随机变化，效果较差）但是AFL的源码自身存在问题，默认情况下qemu_mode无法正常编译。在查询资料后，修改了qemu的build.sh文件，并且重写（或替换）了syscall.diff文件后，重新编译AFL才能正确使用 -Q 选项进行fuzz。

参考解释资料：https://www.codeleading.com/article/61745363753/

另外，qemu模式下性能较差，相同时间下产生的路径显著减少，需要适当延长时间以满足测试要求。

2.4.4 版本不匹配问题

由于部分项目可能为更新的系统设计，所需一些软件包的版本比ubuntu20.04中apt-get获得的最新版本还高，需要手动检查并下载安装相应版本。

例如：zziplib cmake时显示路径有问题（“include could not find load file: CheckCompileFlag”）经查询，该文件为cmake3.19以后新增特性，ubuntu20.04 apt安装最新只能到3.16，手动下载后解决问题。

其他项目中类似问题较多。

2.4.5 库文件没有可运行文件

由于部分项目是库文件，不存在可运行的文件，只有通过自己编写C文件来运行库文件中的一些函数，再通过afl-gcc进行编译，来实现对库文件项目的模糊测试。

参考资料：https://github.com/brahmstaedt/libxml2-fuzzing

https://github.com/mykter/afl-training

3.Mull检测并记录变异体得分

3.1使用Mull编译源程序

3.1.1编写mull.yml配置文件

例如：

```
mutators:
    - cxx_add_to_sub
```

3.1.2编译

将编译器切换为clang++-12并添加编译参数：

```CMake
export CXX=clang++-12
cmake
-DCMAKE_CXX_FLAGS="-O0 -fexperimental-new-pass-manager
-fpass-plugin=/usr/lib/mull-ir-frontend-12
-g -grecord-command-line"
..
```

3.2运行Mull并记录结果

3.2.1基本运行

```
mull-runner-12 [program name]
```

```
yuhaofeng@yuhaofeng:~/桌面/exiv2-0.27.5-Source/build/bin$ mull-runner-12 ./exiv2 -test-program=python3 -- 1.py ./exiv2 ./1.jpg
[warning] Could not find dynamic library: libexiv2.so.27
[warning] Could not find dynamic library: libstdc++.so.6
[warning] Could not find dynamic library: libm.so.6
[warning] Could not find dynamic library: libgcc_s.so.1
[warning] Could not find dynamic library: libc.so.6
[info] Warm up run (threads: 1)
       [################################] 1/1. Finished in 50ms
[info] Filter mutants (threads: 1)
       [################################] 1/1. Finished in 0ms
[info] Baseline run (threads: 1)
       [################################] 1/1. Finished in 26ms
[info] Running mutants (threads: 1)
       [################################] 32/32. Finished in 826ms
[info] Survived mutants (31/32):
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:242:56: warning: Survived: Replaced + with - [cxx_add_to_sub]
               Exiv2::DataBuf   ascii((long)(size * 3 + 1));
                                                      ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:250:46: warning: Survived: Replaced + with - [cxx_add_to_sub]
                      long   count = (start+chunk) < length ? chunk : length - start ;
                                              ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1038:95: warning: Survived: Replaced + with - [cxx_add_to_sub]
                    writePreviewFile(pvMgr.getPreviewImage(pvList[num]), static_cast<int>(num + 1));
                                                                                              ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1045:41: warning: Survived: Replaced + with - [cxx_add_to_sub]
                         << " " << num + 1 << "\n";
                                          ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1048:87: warning: Survived: Replaced + with - [cxx_add_to_sub]
           writePreviewFile(pvMgr.getPreviewImage(pvList[num]), static_cast<int>(num + 1));
                                                                                       ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1660:45: warning: Survived: Replaced + with - [cxx_add_to_sub]
       const long monOverflow = (tm.tm_mon + monthAdjustment_) / 12;
                                           ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1661:32: warning: Survived: Replaced + with - [cxx_add_to_sub]
       tm.tm_mon = (tm.tm_mon + monthAdjustment_) % 12;
                              ^
/home/yuhaofeng/桌面/exiv2-0.27.5-Source/src/actions.cpp:1662:39: warning: Survived: Replaced + with - [cxx_add_to_sub]
       tm.tm_year += yearAdjustment_ + monOverflow;
                                     ^
[info] Mutation score: 3%
[info] Total execution time: 907ms
```

记录变异体得分

3.2.2使用mull对AFL生成的测试用例进行测试

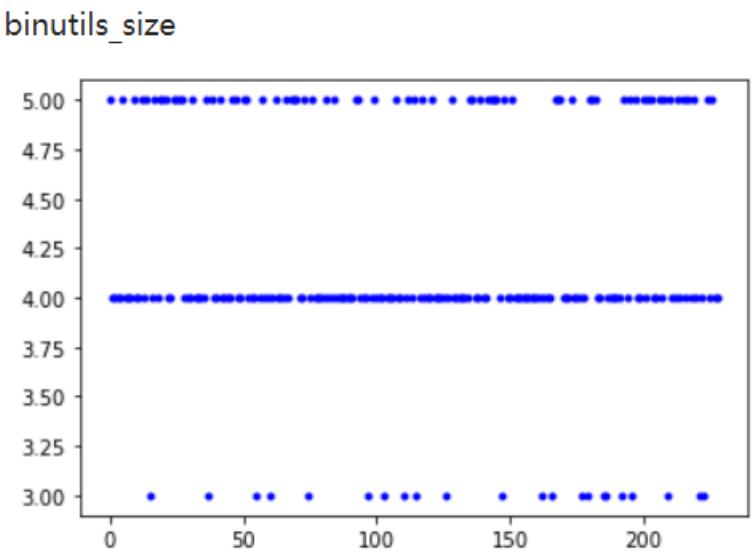
编写脚本遍历所有生成的测试用例放入mull中运行，记录变异体杀死情况，从而检测测试用例的质量

```
#!/bin/bash
#path = $1
#program = $2
#programName = $3

eval "cd $2"

filelist=`ls $1`
for file in $filelist
do
  echo $file

  eval "mull-runner-12 --reporters Elements --report-dir report ./$3 -test-program=python3 -- 1.py ./$3 $1/$file "
done
```

```
bash runMull.sh out/queue . [program name]
```

从而可以绘制出每个测试用例的变异体杀死率：

e.g



3.3过程中遇到的问题

3.3.1插桩失败，即无法生成变异体

没有正确输入编译参数，且编译器应该使用clang++-12

> //CXX compiler
>
> CMAKE_CXX_COMPILER:FILEPATH=/usr/bin/clang++-12

正确设置后即可以正常识别编译参数，重新编译即可

3.3.2无法运行带参数的程序

如果只需要一个input_dir参数，则只需要直接加在后面即可

```
mull-runner-12 [program name] [input_dir]
```

但是如果需要多个参数，则直接跟在后面是无法识别的，所以需要使用脚本来实现（下面以py为例）：

```
import sys
import subprocess

test_executable = sys.argv[1] #程序名
path = sys.argv[2] #路径
subprocess.run([test_executable, "-u","-v","print",path], check=True)
```

使用subprocess将参数跟在参数列表里即可，如上述例子使用了exiv2 -u -v print [path]

然后在运行mull-runner时使用：

```
mull-runner-12 [program name] -test-program=python3 -- 1.py [name] [path]
```

即可

4.总结

过程非常坎坷。

网上的资料也非常有限，所以对于自己写C文件来运行库文件的内容，都是通过网上一些非常有限的资料看到的，慢慢摸索出来。

最后对于每个项目都运行了一遍，跑出了我们自己实验的结果。