南開大學
Nankai University

计算机学院

软件安全实验报告

实验七：AFL 模糊测试实验

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 1 日

目录

1 实验名称

AFL 模糊测试实验

2 实验要求

根据课本 7.4.5 章节，复现 AFL 在 KALI 下的安装、应用，查阅资料理解覆盖引导和文件变异的概念和含义。

3 实验过程

3.1 AFL 安装

首先我们创建一个 demo 文件夹，然后在这个文件夹里打开终端。接着我们进行 AFL 的安装，在终端中输入以下命令：

```
wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
```

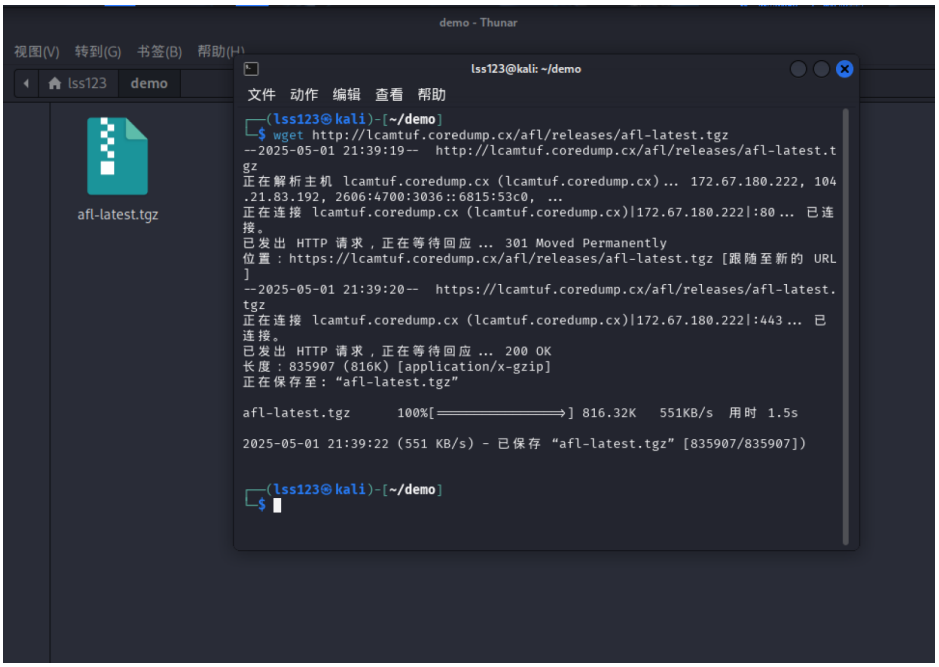

图 3.1: Enter Caption

然后进行解压，并进入 afl 目录，然后再使用 make 指令编译 afl。

```
tar xvf afl-latest.tgz
cd afl-2.52b
make
```

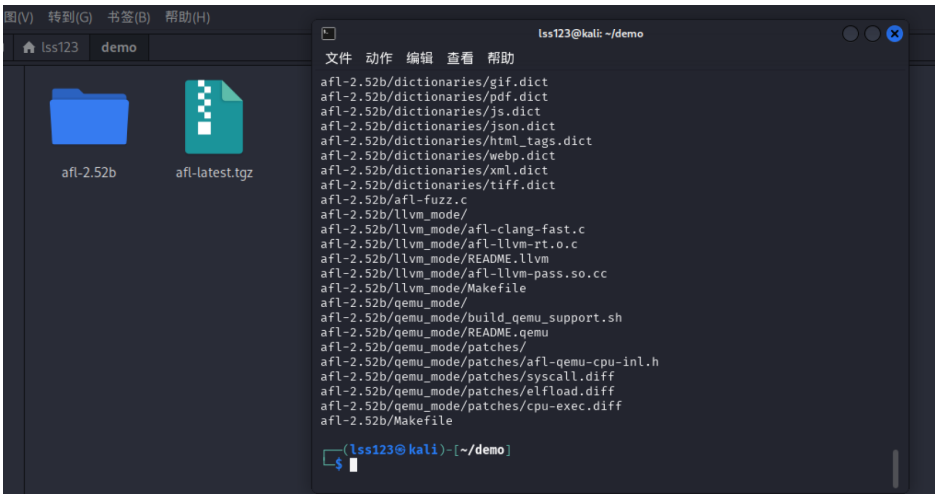

图 3.2

但是我发现在 make 时报错了。在查阅了相关资料后，我发现这个错误的原因应该是由于我们的 kali 系统与官网上 afl 的系统不同，kali 虚拟机是基于 arm 架构的，但 afl 尝试编译的是 x86 架构的代码。

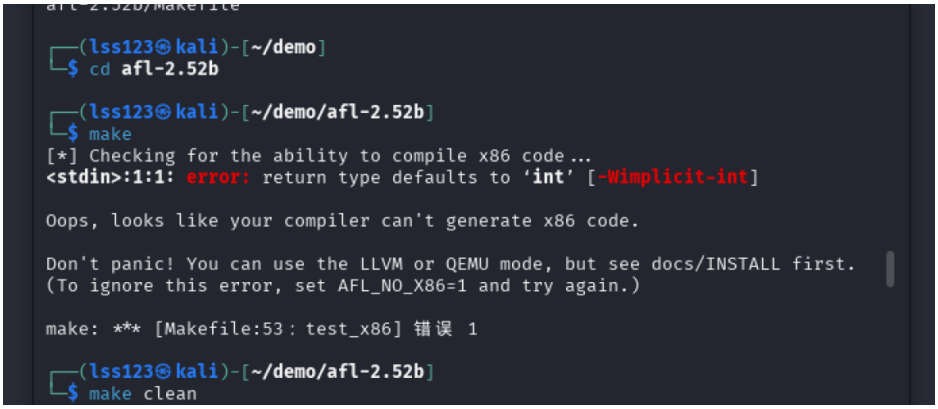

图 3.3

在询问了 deepseek 之后，我找到了一种能够成功安装的途径。我们通过设置 AFL_NO_X86=1, 强制跳过 x86 检查, 然后再进行编译就不会报错。

```
AFL_NO_X86=1 make
sudo AFL_NO_X86=1 make install
```

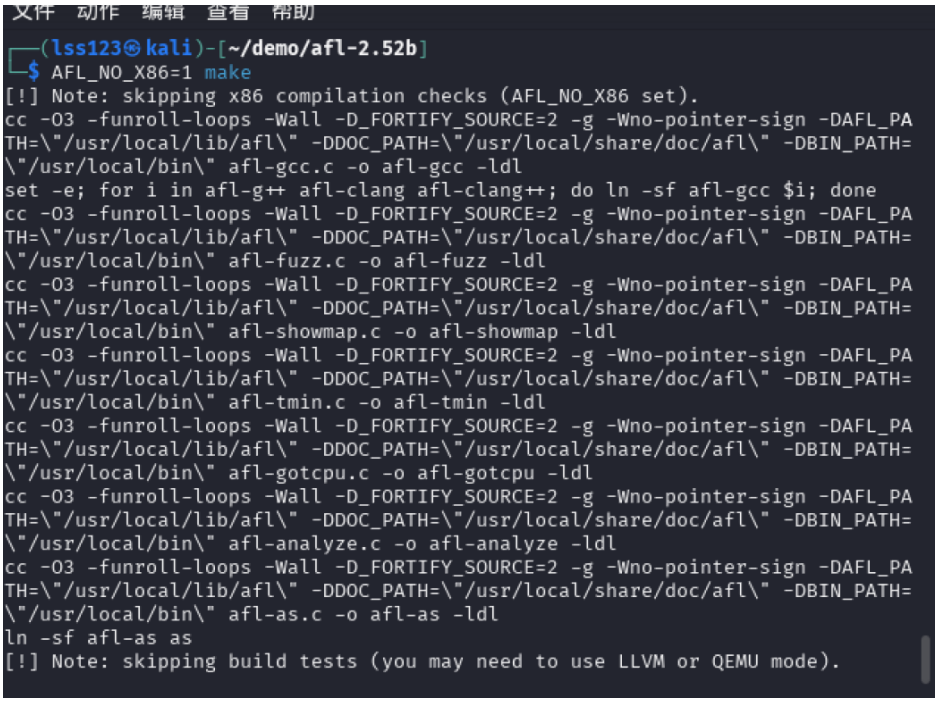

图 3.4

再输入以下命令，确保安装成功。

```
afl-fuzz --version
```

可以看到，afl 的确安装成功。

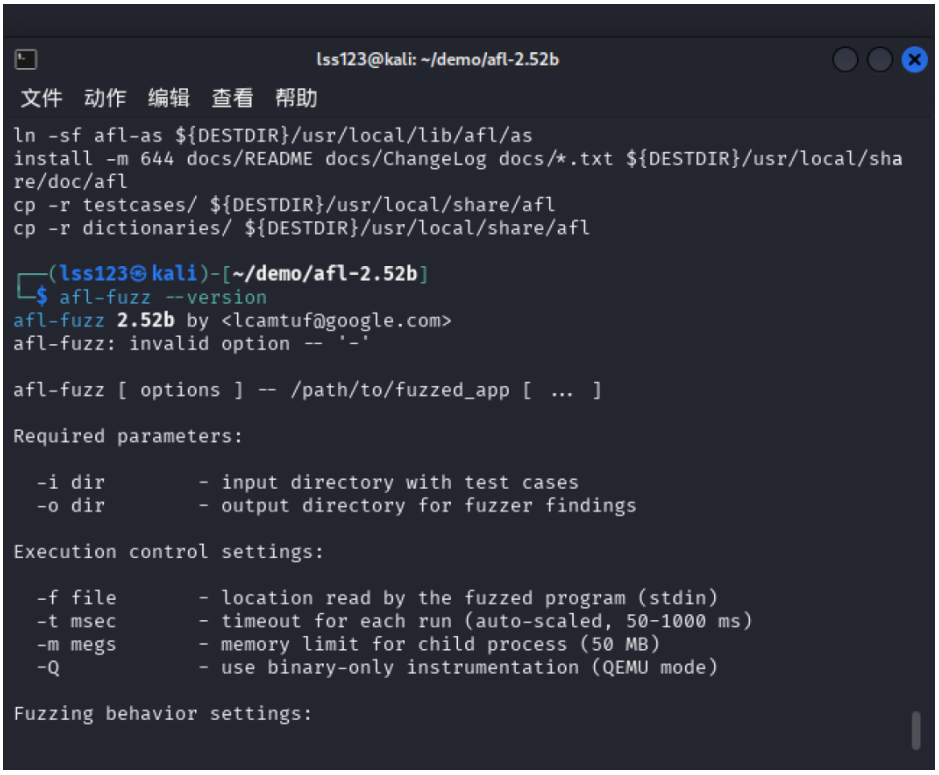

图 3.5

3.2 模糊测试的复现

3.2.1 创建 test.c 测试程序

我们在 demo 文件夹中创建 test.c 文件，并输入以下代码。这段代码的意图就是，如果检测到输入的字符串为 deadbeef，则会触发异常并中断程序。

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  char ptr[20];
  if(argc>1){
           FILE *fp = fopen(argv[1], "r");
           fgets(ptr, sizeof(ptr), fp);
  }
  else{
          fgets(ptr, sizeof(ptr), stdin);
  }
  printf("%s", ptr);
  if(ptr[0] == 'd') {
          if(ptr[1] == 'e') {
                  if(ptr[2] == 'a') {
                          if(ptr[3] == 'd') {
                                  if(ptr[4] == 'b') {
                                          if(ptr[5] == 'e') {
                                                  if(ptr[6] == 'e') {
                                                          if(ptr[7] == 'f') {
                                                                  abort();
                                                          }
                                                          else    printf("%c",ptr[7]);
                                                  }
                                                  else    printf("%c",ptr[6]);
                                          }
                                          else    printf("%c",ptr[5]);
                                  }
                                  else    printf("%c",ptr[4]);
                          }
                          else    printf("%c",ptr[3]);
                  }
                  else    printf("%c",ptr[2]);
          }
          else    printf("%c",ptr[1]);
  }
  else    printf("%c",ptr[0]);
  return 0;
}
```

然后使用 afl 的编译器编译，可以使模糊测试过程更加高效。我们在终端中输入 afl-gcc -o test test.c 命令，来对 test.c 源文件进行编译，并生成文件 test。

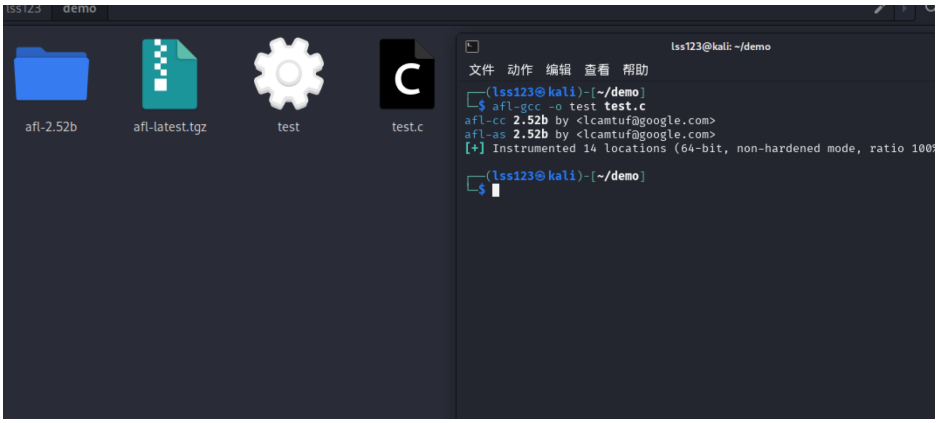

图 3.6

编译后会有插桩符号，使用 readelf -s ./test | grep afl 命令可以对其进行验证。

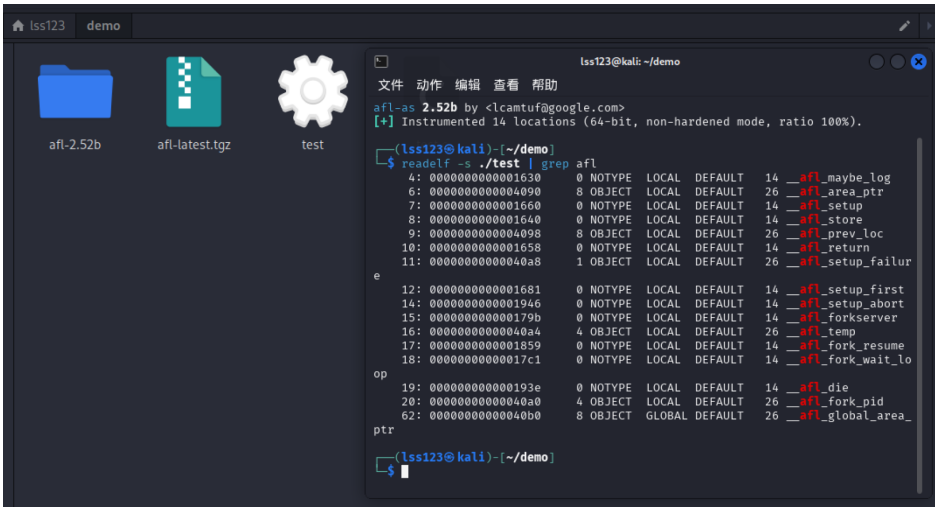

图 3.7

可以看到成功插入了相关的分析代码。

3.2.2 创建测试用例

在这里我们需要创建一个 in 文件夹和一个 out 文件夹，分别用于为模糊测试提供合法输入样例（是基于变异的模糊测试，在此基础上进行变异）和接收相关输出的内容。

命令：mkdir in out

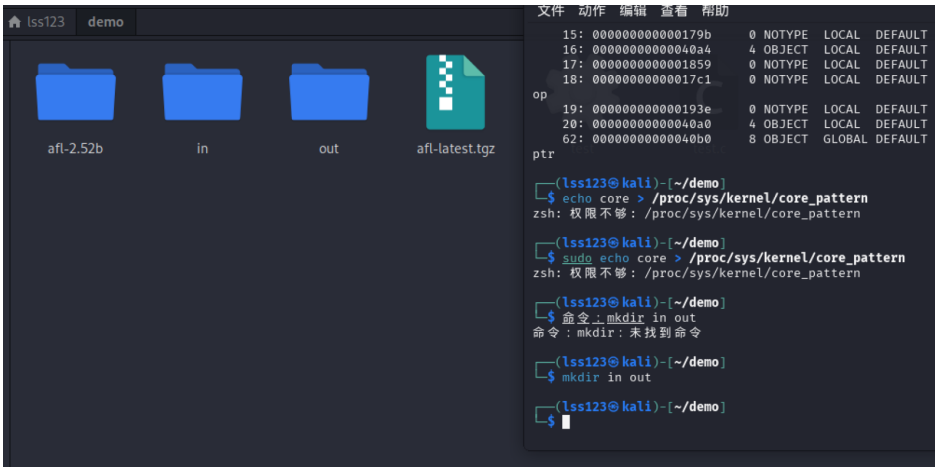

图 3.8

然后，向 in 目录下的 foo 文件中写入字符串 “hello”，作为样例，AFL 会通过这个语料进行变异，构造更多的测试用例。

命令：echo hello> in/foo

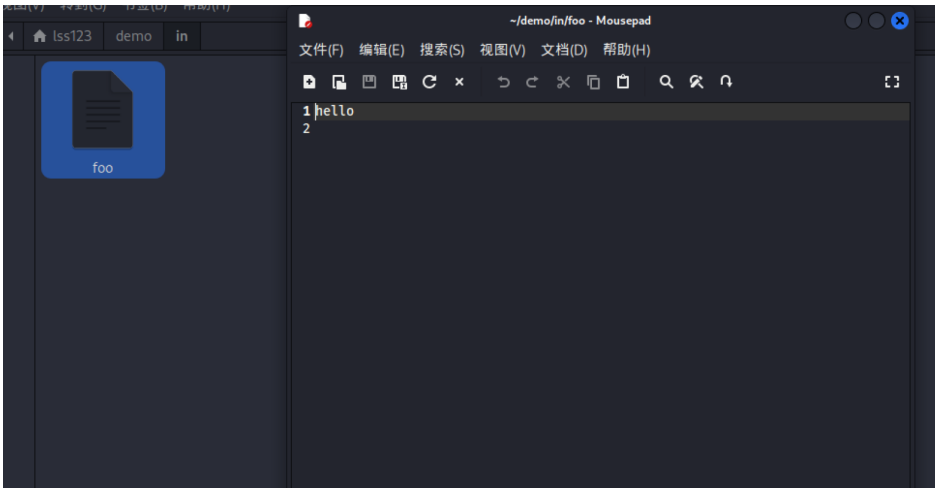

图 3.9

可以看到，确实写入了 hello 字符串。

至此我们完成了对测试用例的创建，接着要进行模糊测试。

3.2.3 启动模糊测试

运行如下命令，开始启动模糊测试：

命令：afl-fuzz -i in -o out – ./test @@

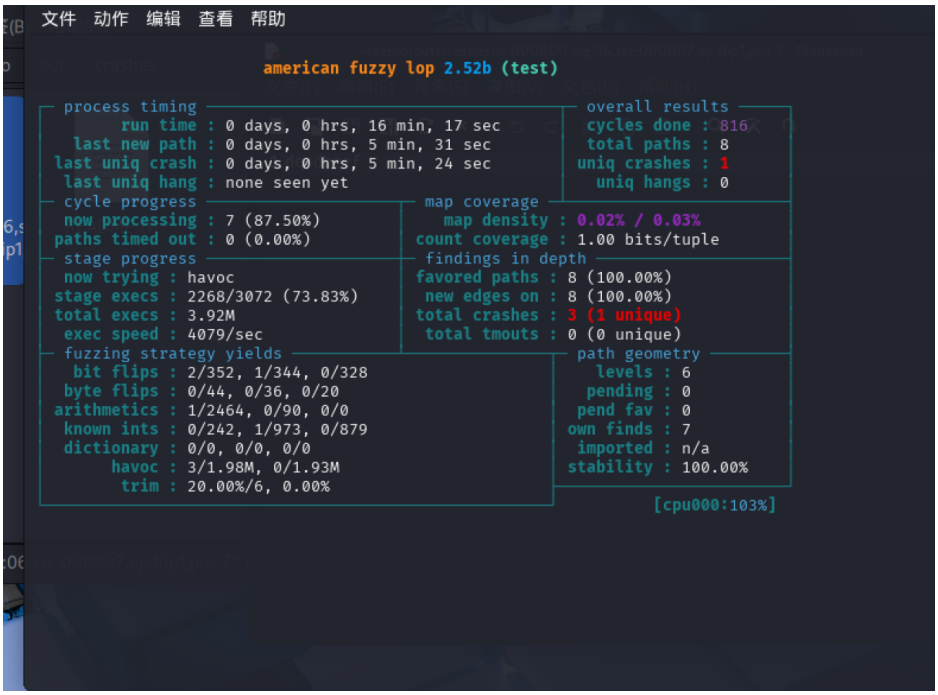

图 3.10

当 ui 界面中 total crashes 中出现了 crash 时，说明程序异常中止，然后我们进入 out 目录，可以在 crashes 文件夹中找到导致本次程序异常中止的输入。

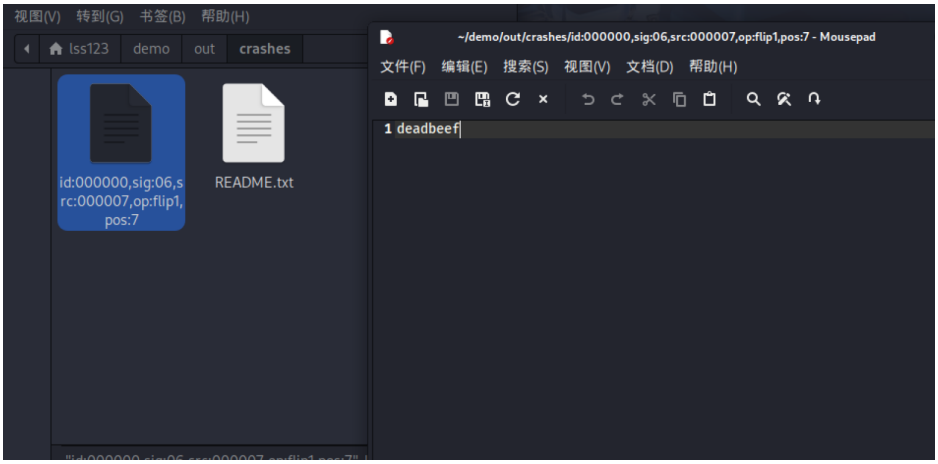

图 3.11

可以看到，输出恰为我们分析得到的 “deadbeef” 字符串，我们完成了对程序异常中断的判断及验证。

3.3 覆盖引导和文件变异的概念及含义

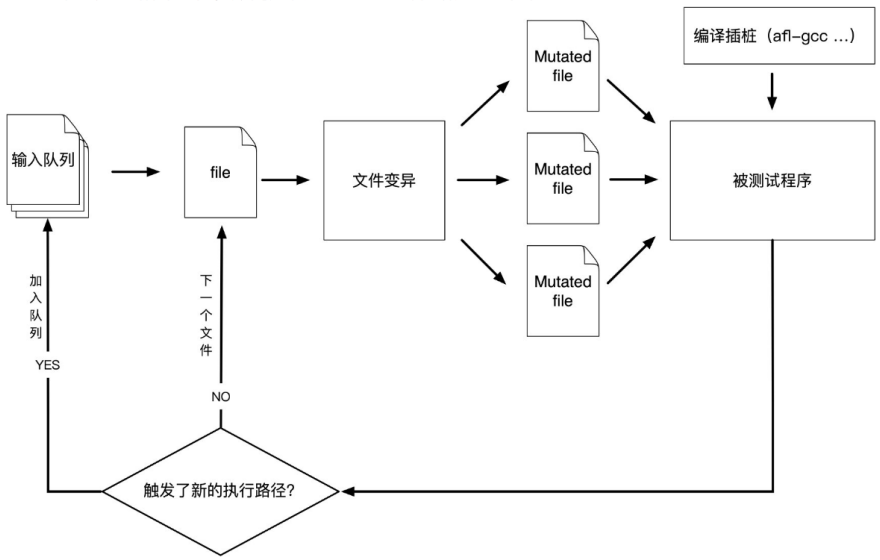

图 3.12

查阅了相关资料，得出 AFL 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。

3.3.1 覆盖引导

覆盖引导（Coverage-Guided）是一种动态分析技术，主要用于软件测试（如模糊测试/Fuzzing）或代码优化中。它的核心思想是通过**代码覆盖率（Code Coverage）**指标来指导测试过程，确保测试用例能够尽可能覆盖更多的代码路径、分支或条件，从而提高测试的有效性。

也就是说，在 AFL 的工作流程中，需要通过覆盖引导技术，基于我们初始给的合法测试样例，不断更新代码覆盖率以生成更有效的测试输入，触发深层代码逻辑或漏洞，实现智能模糊测试。从而使得生成的数据更有针对性，减少了大量无关测试数据的生成，提高效率。

3.3.2 文件变异

文件变异通过对已知有效的输入文件（如 PDF、JPEG、网络协议数据包等）进行**有目的的修改**，生成大量非预期或畸形的输入，观察目标程序是否能正确处理这些输入。若程序崩溃或出现逻辑错误，则可能发现潜在漏洞。

我们从初始的 hello 样例，进而生成其他测试样例的过程，就是基于变异的模糊测试，利用了文件变异。

4 心得体会

1. 学会了如何安装 kali 虚拟机和 afl 程序。

2. 理解了 afl 的运作过程，能够基于 afl 完成对模糊测试的复现。

3. 对模糊测试的概念、原理以及实践都有了更好的理解以及运用。其本质就是一个从初始数据生成大量畸形数据来监测目标程序异常的过程。