# AFL代码分析

## afl-gcc

1. 调用find_as，在本目录下寻找afl-as，确保其存在
2. 调用edit_params，调整参数

- 检查使用的是否是afl-clang　（注意：未处理的异常不会调用 abort()，因此在使用 Java 二进制文件时，需要修改 afl-fuzz 以将非零退出代码与崩溃条件等同起来。）
- 若不是，判断用的是否是afl-g++,afl-gcj等，总之，就是判断此时用的是哪个工具
- 进入while循环，将参数复制给cc_params
- 根据环境变量等信息调整cc_params
- 调用execvp，根据刚才准备好的参数cc_params，调用相应的执行编译器

## afl-as

从main开始

1. 调用edit_params，调整要传递给as的参数
2. 调用add_instrumentation()来插桩

- 打开输入文件
- 进入大while循环，进行插桩
  - 进行条件判断，满足条件，则进行插桩，然后插入原本的代码，pass_thru模式下，跳过本行
  - 由于只对.text节进行处理，所以进行一定的筛选
    - 检测.code，检测语法更改，检测`__asm__`块
  - 在 main函数，GCC branch label，clang branch label，conditional branches处插入指令
    - 除了jnz等以外，其他的一些标签之类的，有概率插入
- 插入main_payload
- 输出提示信息等

3. fork出子进程来执行as，进行汇编
4. 等待子进程结束

## 总结一下

这里，afl-gcc将c代码编译成汇编代码，同时指定了编译器的搜索路径，编译器默认优先使用该路径中的汇编器和链接器，即`afl-as`，**实际的插桩工作发生在汇编的时候。**

---

## afl-fuzz

1. 从main函数开始看，首先进入while循环，读取在控制台输入的命令中的选项

```
  while ((opt = getopt(argc, argv, "+i:o:f:m:b:t:T:dnCB:S:M:x:QV")) > 0)

    switch (opt) {
            ...
    }
```

2. 设置信号处理函数

```
setup_signal_handlers();
```

3. 检查ASAN选项（ASAN用于检测内存错误）

```
check_asan_opts();
```

4. 存储命令行参数

```
save_cmdline(argc, argv);
```

5. 获取环境相关的信息等
6. 建立共享内存

```
setup_shm();//初始化了virgin_bits   trace_bits 两者分别用于存储未被覆盖的边和各个边的覆盖率（通过哈希得到下标）
```

进入查看：

- 调用memset，为virgin_bits分配空间
- 将virgin_bits virgin_tmout virgin_crash 初始化为全1，表示此时没有任何区域是已经被触及的、已经超时过的和已经触发崩溃的
- 分配共享内存，并将其连接到当前进程的地址空间，这样，便生成了trace_bits（有检测位图的共享内存 大小64kb）

7. 加载目录，读取测试用例等
8. 开始Fuzz

```
perform_dry_run(use_argv);
```

进入此函数查看：

- 主体是一个大的while循环
- 分配use_mem，将测试用例读入其中
- 对队列中的每一个测试用例，进行校准，之后针对不同的校准错误进行处理。校准时调用：

  ```
  res = calibrate_case(argv, q, use_mem, 0, 1);
  ```

  进入此函数查看：

  - 首先进行一系列初始化工作
  - 确定运行次数 若设定了快速校准，则运行3次，否则8次
  - 若没有forkserver，则建立forkserver

```
init_forkserver(argv);
```

- 进入查看
  - 进行fork
  - 子进程就是forkserver，它会执行被测程序

    ```
    execv(target_path, argv);
    ```

  - 父进程（Fuzzer）进行等待（`setitimer()`），之后检查管道，得到子进程（forkserver）"hello"的回复后返回，否则处理错误
  - 这里要注意，到这里，子进程（forkserver）去运行了被测程序，同时，应当注意，之前对其插了桩，插入的代码在afl-as.h中。首先运行`__afl_maybe_log()`，这个函数会将"hello"（即上一步中父进程（fuzzer）要等的回复）写入管道，告诉父进程（fuzzer）一切正常，然后进入`__afl_fork_wait_loop`，读取管道，直到fuzzer通知其进行fork，fork后，子进程（真正运行测试用例的进程）跳到`__afl_fork_resume`，关闭无用的管道，继续执行，父进程（forkserver）则继续作为forkserver，这里注意！！！有个坑，AT&T汇编中，

    ```
    "  call fork\n"
    "\n"
    "  cmpl $0, %eax\n"
    "  jl   __afl_die\n"
    "  je   __afl_fork_resume\n"
    ```

    这一段的意思是，%eax<0时（fork失败），转向die，=0时（子进程），转向resume，这里cmp的含义和x86中的不一样
  - 父进程（forkserver）继续进入`__afl_fork_wait_loop`

- 若校验和不为0，则拷贝trace_bits到first_trace,并检查virgin_bits中是否有新情况
- 进入for循环，循环次数为3或8次（由前面确定）
  - 调用write_to_testcase，将修改后的use_mem写入测试用例
  - 调用run_target，通知forkserver准备fork并fuzz（将trace_bits设为0，将信息写入管道，读取状态管道）
    - 注意！！！这里要结合插桩的代码进行理解，一些fuzz的逻辑隐含在插入被测程序的代码中（如fork出子进程，等待父进程指令，运行过程中更新trace_bits（在trampoline_fmt_64进行）等）
  - 看是否出现了新情况

    ```
    cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);
    ```

  - 跟之前的校验和（q->exec_cksum）作比较，若不等，则是出现了新情况
    - q->exec_cksum若为0，说明是第一次执行，继续进行初始化操作，即令其等于cksum，并将trace_bits复制到first_trace中
    - 若不为0，如果first_trace的某一位和trace_bits的不一样，就说明该位发生了变化，将var_bytes的相应位置置1，并让这个用例多运行几次，即将次数调成40
- 之后，进行一定的处理工作（计算运行时间等）

    - 更新位图的得分（update_bitmap_score）目的是找到top_rated，即最小的、消耗最短时间到达该边的测试用例，进而找到这些“最优”用例组成的集合——favorables，通过这个集合中的用例，就能通过最小的空间、时间代价来到达所有的边
        - 对于每个trace_bits，若被置了值
            - 若相应位置的top_rated不为空，说明有竞争者，比较运行时间x用例大小的值，此值较小的为更优，若top_rated不是较小的，则将他的引用数减一，减一后为0，就将他释放
            - 若top_rated为空，则将这个用例作为top_rated，将其引用数加一
- 如果刚才校准测试用例没有出错，就会返回FAULT_NONE，此时，会执行check_map_coverage()，检查覆盖范围
- 处理错误并返回

9. 调用 cull_queue()

   它遍历 top_rated[] 条目，然后顺序地获取先前未见过的字节 (temp_v) 的获胜者并将它们标记为喜欢的条目，至少直到下一次运行。

   在所有模糊测试步骤中，喜欢的条目会获得更多的运行时间。

进入查看：

- 将temp_v置为全1
- 将队列所有queue_entry的favored的属性置为零
- 对于每一条边，若它有top_rated，且没有在temp_v中（说明此边已经被到达过），删除top_rated中属于当前条目的所有位，同时，将当前queue_entry设置为favored
- 进行遍历，若当前queue_entry不是favored，就将他标注为冗余的，这样，就达到了筛选queue的目标

10. 保存一系列信息，显示一系列信息
11. 进入主循环

- 挑选用例（cull_queue()）
- 若当前用例的指针为空，说明已经遍历了一遍，进行一些处理，从头开始
- 用fuzz_one对用例进行fuzz，具体来讲，就是先校准用例（`calibrate_case`，在这个函数内真正运行用例），修剪用例（防止过大），对用例进行评分，对用例进行变异
    - 如果是待定首选路径，若不是favored的，或者是已经fuzz过的，就有可能直接返回（99%的概率）
    - 如果不是favored的，同时正在排队的用例有10个以上，
        - 若已经fuzz过一轮，但是本用例仍未被fuzz过，那么有75%的概率直接返回
        - 其他情况下有95%的概率直接返回
    - 在 IGNORE_FINDS 模式下，跳过任何不在初始数据集中的条目。
    - 将用例映射到内存中
    - 若校准失败，则重新校准（次数限制在`CAL_CHANCES`以内）
    - 若没修剪过，则对用例进行修剪（调用trim_case()）
        - 找到最小的大于len的二的整数次幂(len_p2)
        - 初步确定要删去的步长

```
remove_len = MAX(len_p2 / TRIM_START_STEPS, TRIM_MIN_BYTES) ;
//TRIM_START_STEPS 16
//TRIM_MIN_BYTES 4
```

        - 进行修剪直到步数太高或步距太小（小于1/1024）
            - 进入循环
                - 进行删减
                - 调用run_target，如果出错，就不保留删减的结果，立刻返回
                    - 如果没出错，同时trace_bits的哈希值没有变，也就是说删掉这些内容后，用例覆盖路径的情况没有改变，就说明这次的删减是正确的，保存trace_bits到clean_trace中，继续进行修剪
            - 将修改写入原来的用例，之后对用例进行评分（update_bitmap_score()）
    - 对用例进行评分，以调整havoc fuzz的长度
    - 进行一系列位反转等调整操作，调整后进行fuzz，若出错，则跳出
12. 写入virgin_bits，写入状态文件及其他信息，释放资源，退出

# qemu-mode

qemu-mode文件夹内最开始有build_qemu_support.sh，paches，paches内有5个diff文件，用于patch下载好的qemu

## build_qemu_support.sh

1. 首先，进行一系列检查，如系统类型是否是Linux等
2. 下载qemu
3. 解压
4. 检查cpu_target
5. 用paches中的diff文件对qemu内文件进行patch
6. 编译
7. 将qemu拷贝为afl目录下的afl-qemu-trace，并测试是否生效

## patch分析

1. elfload.diff，加入寻找入口点和start，end的代码
2. cpu_exec.diff 加入头文件和宏定义

```
+    AFL_QEMU_CPU_SNIPPET2;
...
             if (!tb) {
                 /* if no translated code available, then translate it now
*/
                 tb = tb_gen_code(cpu, pc, cs_base, flags, 0);
+                AFL_QEMU_CPU_SNIPPET1;
             }

             mmap_unlock();
```

这两个宏在afl-qemu-cpu-ini.h中定义，AFL_QEMU_CPU_SNIPPET1，通知父进程遇到了一个尚未翻译的新块，并告诉它也在自己的上下文中进行翻译，从而减小下一次fork时的开销

AFL_QEMU_CPU_SNIPPET2，就是每次qemu执行一个基本块的时候，afl去check一下这个基本块的地址是不是目标程序的entry_point。如果是，就启动forkerver。如果不是entry_point的话就看一下是不是覆盖了新的分支。如果是的话就记录。

3. configure.diff 更改包含的文件
4. memfd.diff 更改包含的文件
5. syscall.diff 更改处理错误的代码

总的来看，用于实现AFL主要逻辑的patch就是cpu_exec.diff，在其中完成了记录覆盖情况的逻辑

# 最后总结

进行fuzz时，若有源代码，则先用afl-gcc对其进行编译，afl-gcc只是一层gcc的封装，同时将汇编器改为afl-as，在汇编的时候，afl-as对代码进行插桩，插入记录覆盖和最后计算覆盖率的代码，之后用afl-fuzz进行测试，这期间会fork出forkserver，由它来通过接受fuzzer的消息，fork出进程来真正执行测试用例。

若没有源代码，则用qemu模式进行fuzz，afl-fuzz主函数内会判断是否使用了qemu模式，若使用了，会在之后调用get_qemu_argv调整传给qemu的命令，之后此命令作为use_argv传入perform_dry_run等函数，运行qemu进行测试

# demo测试

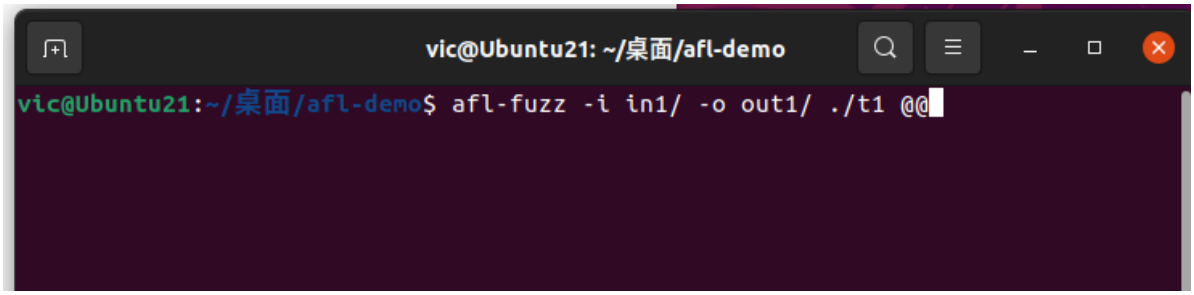
编写测试用代码

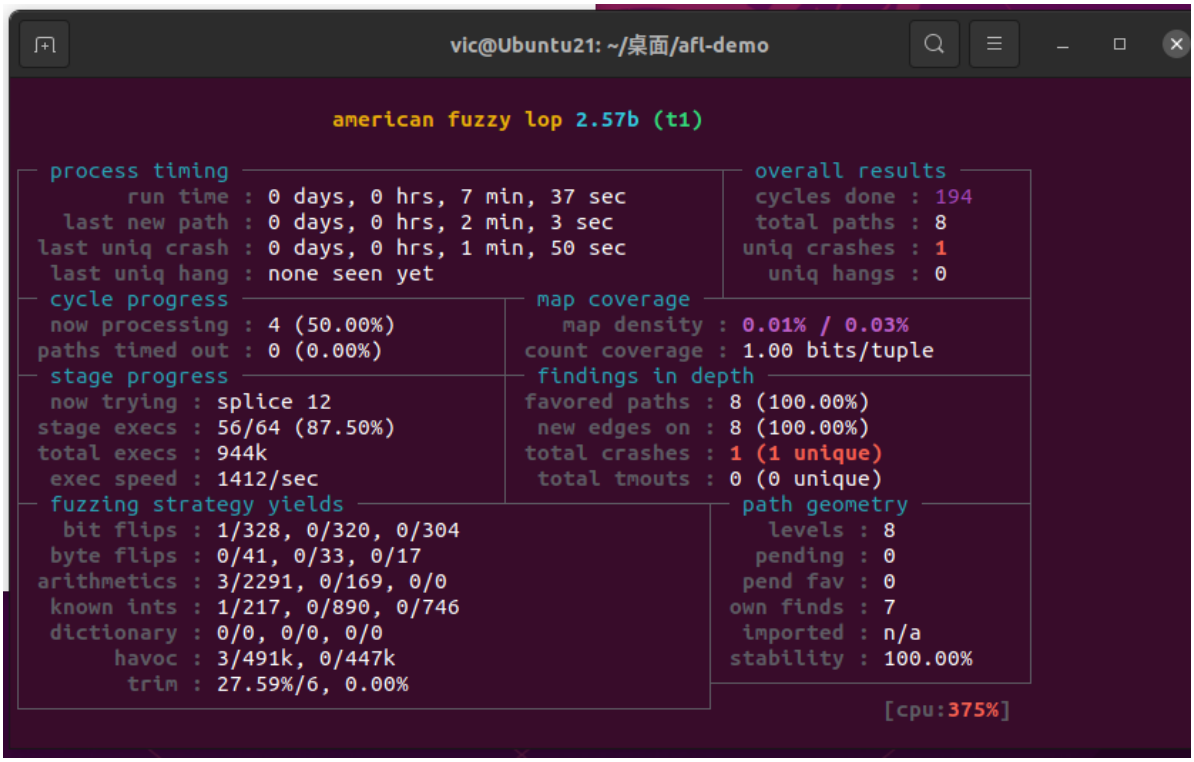
```
#include <stdio.h>

#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;
}
```

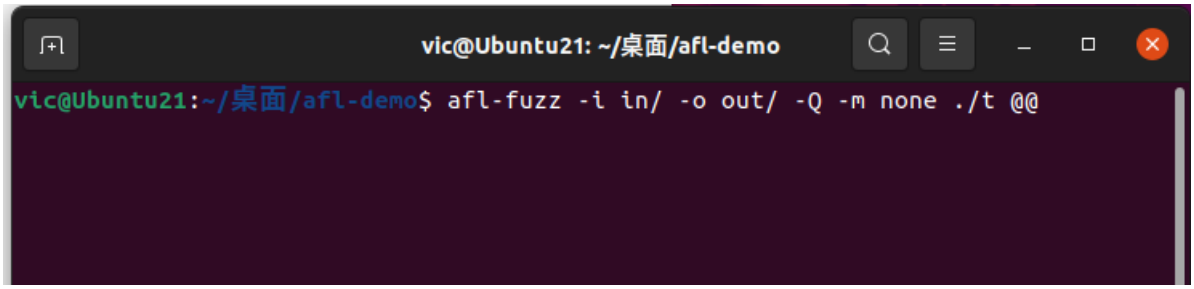
用afl-gcc和gcc分别编译一次，输出为t1和t，用普通模式和qemu模式进行测试

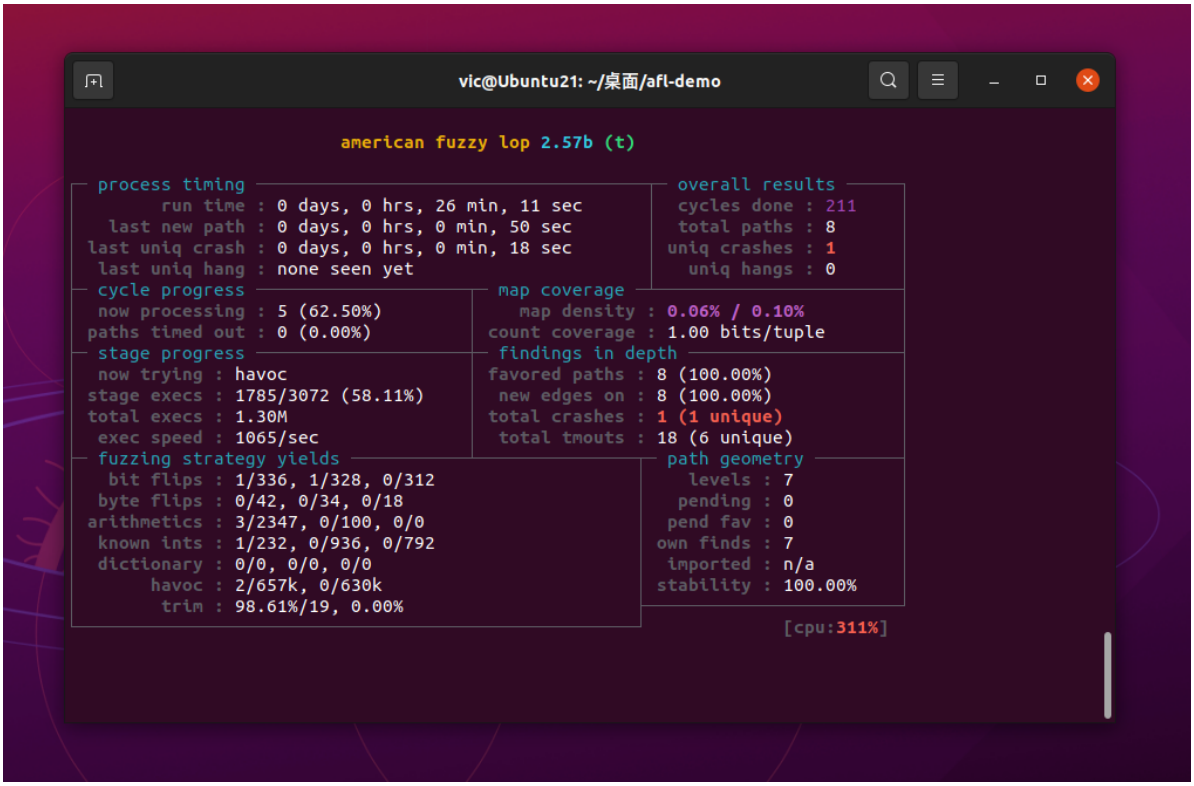
普通模式输出结果：


vic@Ubuntu21: ~/桌面/afl-demo
vic@Ubuntu21:~/桌面/afl-demo$ afl-fuzz -i in1/ -o out1/ ./t1 @@


vic@Ubuntu21: ~/桌面/afl-demo
american fuzzy lop 2.57b (t1)
process timing
run time : 0 days, 0 hrs, 7 min, 37 sec
last new path : 0 days, 0 hrs, 2 min, 3 sec
last uniq crash : 0 days, 0 hrs, 1 min, 50 sec
last uniq hang : none seen yet
overall results
cycles done : 194
total paths : 8
uniq crashes : 1
uniq hangs : 0
cycle progress
now processing : 4 (50.00%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.01% / 0.03%
count coverage : 1.00 bits/tuple
stage progress
now trying : splice 12
stage execs : 56/64 (87.50%)
total execs : 944k
exec speed : 1412/sec
findings in depth
favored paths : 8 (100.00%)
new edges on : 8 (100.00%)
total crashes : 1 (1 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 1/328, 0/320, 0/304
byte flips : 0/41, 0/33, 0/17
arithmetics : 3/2291, 0/169, 0/0
known ints : 1/217, 0/890, 0/746
dictionary : 0/0, 0/0, 0/0
havoc : 3/491k, 0/447k
trim : 27.59%/6, 0.00%
path geometry
levels : 8
pending : 0
pend fav : 0
own finds : 7
imported : n/a
stability : 100.00%
[cpu:375%]

qemu模式输出结果：


vic@Ubuntu21: ~/桌面/afl-demo
vic@Ubuntu21:~/桌面/afl-demo$ afl-fuzz -i in/ -o out/ -Q -m none ./t @@


vic@Ubuntu21: ~/桌面/afl-demo
american fuzzy lop 2.57b (t)
process timing
run time : 0 days, 0 hrs, 26 min, 11 sec
last new path : 0 days, 0 hrs, 0 min, 50 sec
last uniq crash : 0 days, 0 hrs, 0 min, 18 sec
last uniq hang : none seen yet
overall results
cycles done : 211
total paths : 8
uniq crashes : 1
uniq hangs : 0
cycle progress
now processing : 5 (62.50%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.06% / 0.10%
count coverage : 1.00 bits/tuple
stage progress
now trying : havoc
stage execs : 1785/3072 (58.11%)
total execs : 1.30M
exec speed : 1065/sec
findings in depth
favored paths : 8 (100.00%)
new edges on : 8 (100.00%)
total crashes : 1 (1 unique)
total tmouts : 18 (6 unique)
fuzzing strategy yields
bit flips : 1/336, 1/328, 0/312
byte flips : 0/42, 0/34, 0/18
arithmetics : 3/2347, 0/100, 0/0
known ints : 1/232, 0/936, 0/792
dictionary : 0/0, 0/0, 0/0
havoc : 2/657k, 0/630k
trim : 98.61%/19, 0.00%
path geometry
levels : 7
pending : 0
pend fav : 0
own finds : 7
imported : n/a
stability : 100.00%
[cpu:311%]