漏洞挖掘技术之 AFL 项目分析

作者：BDomne@看雪论坛

0 引子

AFL（american fuzzy lop）在前 Google 安全研究员 lcamtuf 耗时数年心血的努力下已经日臻完善，俨然成了 fuzz 工具中独树一帜的存在，相关项目可谓竞相学习与借鉴。目前其更新时间停留于 17 年末，即本文所分析的 2.52b 版本，这之后 lcamtuf 大佬的爱好似乎更热衷摆弄工艺品了;b 源码的阅读需要具备基本的 C 语言和 shell 脚本编程功底，因其代码量适当且编程风格良好，亦不失作为 C 语言开源项目学习的好选择。

1 概述

我们知道软件开发周期中有一个测试环节，也就是找 bug，当项目过大时仅靠人工审计很可能会有所遗漏，这时 fuzz 模糊测试就有了用武之地，即借助计算机的算力来辅助完成测试任务。而对于那些已发布的软件再进行同样的测试环节则称为漏洞挖掘，本文的主角 AFL 就是这样的一个 fuzz 利器，当然，更多时候是针对解析器的文件 fuzz，我们先大体介绍下原理，接着将从源码角度剖析其具体实现。

简单来说，AFL 的设计就是在 dumb fuzzer 之上加了一层基于边界（edge）覆盖率的反馈机制，其 fuzz 方案如下图所示。在此过程中 AFL 会维护一个语料库队列 queue，包含了初始测试用例及变异后有新状态产生的测试用例，对应的变异操作分为确定性策略和随机性策略两类，而状态的分析，即边界覆盖率的统计工作，需依赖插桩来完成。

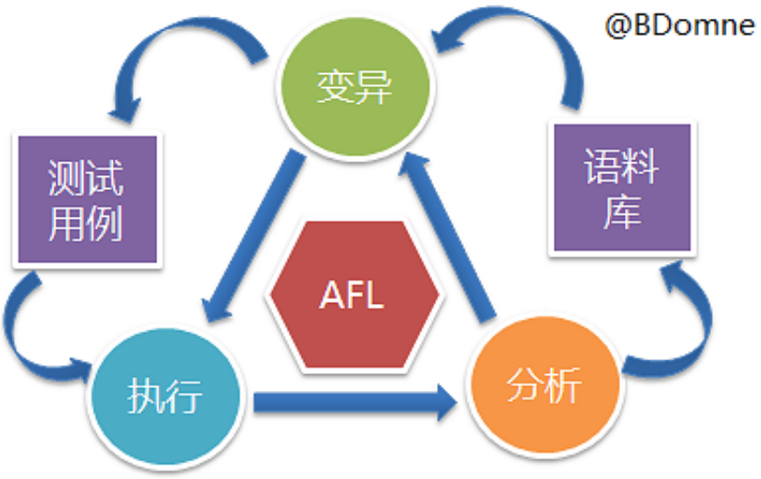


图0 afl-fuzz 的实现方案

这里提下插桩，它指的是通过注入探针代码来实现程序分析的技术。接下来我们重点关注什么是新状态，如图 1 所示，蓝色块代表程序执行过程中的基本块，黄色块代表相应的用于统计的探针代码，因而我们可以完整记录程序的执行路径，即：A -> C -> F -> H -> Z。另外，在 AFL 中为了更方便的描述边界（edge），将源基本块和目的基本块的配对组合称为 tuple，即下图路径中有 4 个 tuple（AC，CF，FH，HZ）。

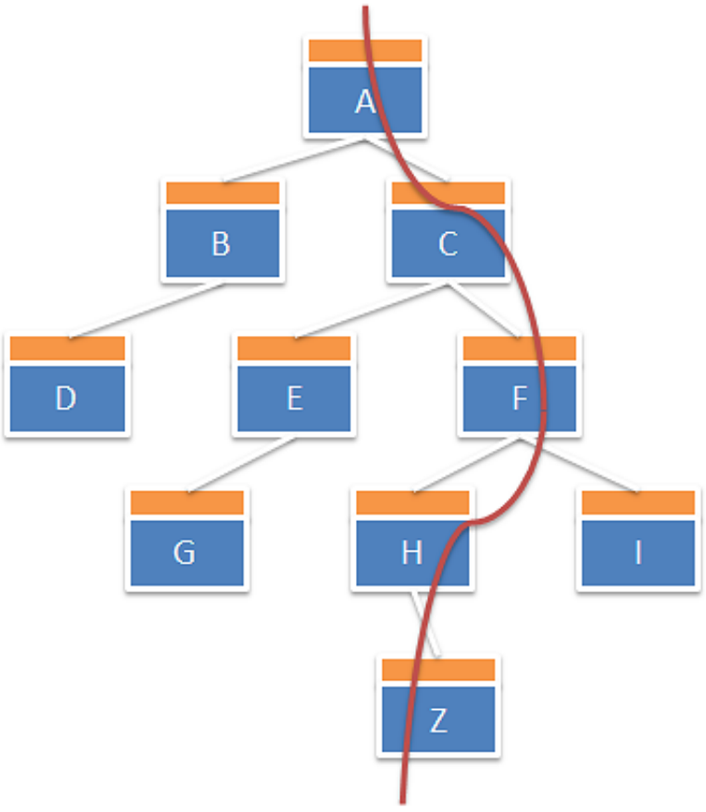


图1 程序执行路径

很自然，通过记录 tuple 信息就可以统计边界覆盖率了，AFL 通过一个 bitmap 来记录这些信息，不过 bitmap 是以 byte 而非 bit 作为单位，因为程序还要记录 tuple 的命中数，而命中数又被划分成如下 8 个组别：

```
1，2，3，4–7，8–15，16–31，32–127，128+
```

当有新的 tuple 出现或已有 tuple 中出现新的命中组则视为产生新状态，相应的测试用例将被归入到语料库中。可以看到，AFL 所做的仅是挑选新状态，对基本块间的关联性未去深究。此外，AFL 之所以不采用基本块覆盖率，是因为漏洞往往是由未知的或非正常的状态转换导致的，故边界覆盖率更合适些。

借助上述遗传算法，fuzz 过程中目标程序的边界覆盖率将不断提高，我们也就有了更大概率去发现漏洞。

最后，为方便读者能对 AFL 项目代码有个整体的认识，这里给出简要的文件目录解析：

```
【插桩模块】
afl-as.c, afl-as.h, afl-gcc.c：普通模式插桩，gcc 和 clang 均适用；
llvm_mode：llvm 模式插桩，仅 clang 适用；
qemu_mode：qemu 模式插桩，针对二进制文件；

【fuzzer 模块】
afl-fuzz.c：fuzzer 实现代码；

【其它辅助模块】
libdislocator：简单的内存错误检测工具；
libtokencap：语法关键字提取并生成字典文件；
afl-analyze.c：对测试用例的字段进行分析；
afl-cmin：对 fuzzing 中用到的语料库进行精简操作；
afl-tmin.c：对 fuzzing 中用到的测试用例进行最小化操作；
afl-gotcpu.c：统计 CPU 占用率；
afl-plot：绘制报告图表；
afl-showmap.c：打印目标程序单轮 fuzz 后的 tuple 信息；
afl-whatsup：各并行例程 fuzzing 结果总计；

alloc-inl.h：定义带检测功能的内存分配和释放操作；
config.h：配置信息的定义；
debug.h：跟提示信息相关的宏定义；
hash.h：哈希函数的实现定义；
types.h：部分类型及宏的定义；
test-instr.c：用作测试的目标程序；

docs：项目相关的说明文档；
experimental：一些新特性的试验研究。
```

2 插桩模块

我们先来讨论与插桩有关的源码。在 AFL 中一共有三种不同模式的插桩操作，如下图，其中普通模式和 llvm 模式是针对目标程序提供源码的情况，显然相较汇编级的普通模式插桩，编译级的 llvm 模式插桩包含更多优化，在性能上会更佳些，而对仅提供二进制文件的目标程序则需借助 qemu 模式，其性能是最低的。本小节主要介绍普通模式下的插桩，另外两种模式与之相似，所插入代码从用途上都是分为两类：1）记录目标程序执行过程中的 tuple 信息，需保证在每个基本块上都有插入；2）必要的初始操作以及维护一个 forkserver。

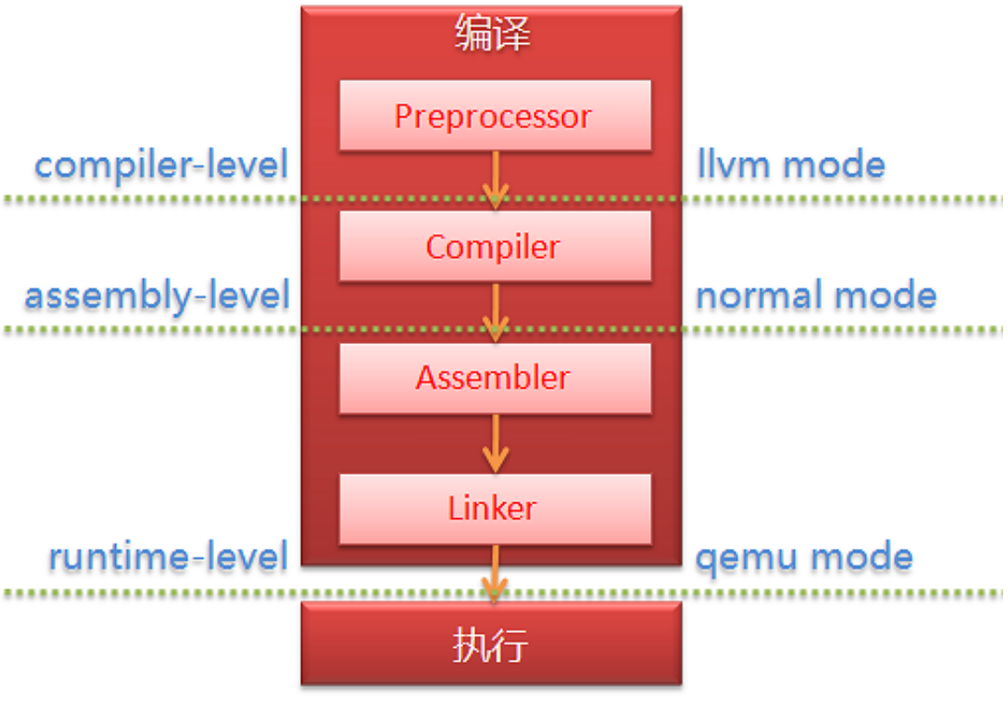


图2 AFL 中用到的插桩技术

2.1 普通模式

我们知道源代码编译实际上分为预处理、编译、汇编和链接四个不同阶段，而 gcc 中的 `-B prefix` 选项可用于指定各阶段执行文件的优先查找目录。故普通模式下 AFL 采用的插桩思路是先对 gcc 做一层简单封装，这些操作中有一项是设定 `-B prefix` 选项并指向伪造的（即封装后的）as 汇编器所在目录：

```
/*file: afl-gcc.c  function: edit_params*/

cc_params[cc_par_cnt++] = "-B";     //cc_params, Parameters passed to the real CC
cc_params[cc_par_cnt++] = as_path; //Path to the AFL 'as' wrapper

if (clang_mode)
  cc_params[cc_par_cnt++] = "-no-integrated-as";

if (getenv("AFL_HARDEN")) {

  cc_params[cc_par_cnt++] = "-fstack-protector-all";

  if (!fortify_set)
    cc_params[cc_par_cnt++] = "-D_FORTIFY_SOURCE=2";

}
```

图3 设置 as 汇编器优先查找目录

接着流程自然将交由封装后的 as 汇编器来处理前面生成的汇编文件，亦即插入指令后再交由真正的 as 汇编器处理，这里将会查找汇编文件中的 .text 节区并在各控制转移指令处插入跳板 `trampoline_fmt_32` / `trampoline_fmt_64`：

```
/*file: afl-as.c   function: add_instrumentation*/
    /* All right, this is where the actual fun begins. For one, we only want to
       instrument the .text section. So, let's keep track of that in processed
       files - and let's set instr_ok accordingly. */

    if (line[0] == '\t' && line[1] == '.') {

      ......

      if (!strncmp(line + 2, "text\n", 5) ||
          !strncmp(line + 2, "section\t.text", 13) ||
          !strncmp(line + 2, "section\t__TEXT,__text", 21) ||
          !strncmp(line + 2, "section __TEXT,__text", 21)) {
        instr_ok = 1;
        continue;
      }

      ......

    }

    ......

    /* Conditional branch instruction (jnz, etc). We append the instrumentation
       right after the branch (to instrument the not-taken path) and at the
       branch destination label (handled later on). */

    if (line[0] == '\t') {

      if (line[1] == 'j' && line[2] != 'm' && R(100) < inst_ratio) {

        fprintf(outf, use_64bit ? trampoline_fmt_64 : trampoline_fmt_32,
                R(MAP_SIZE));

        ins_lines++;

      }

      continue;

    }
```

图4 向汇编文件插入探针指令

分析可知跳板的作用是跳转到具体的实现部分 `main_payload_32` / `main_payload_64`，此部分指令只会插入一次。鉴于插入代码同时支持 32 位和 64 位程序，为了不赘述我们只看 `main_payload_32` 部分，其中一个分支是用于记录目标程序执行过程中的 tuple 信息，相关计算公式及实现代码如下：

```
shm_trace_map[cur_loc ^ prev_loc]++

shm_trace_map：记录 tuple 信息的 bitmap
cur_loc：目的基本块位置    prev_loc：源基本块位置
```

```
/*file: afl-as.h*/
  "__afl_store:\n"
  "\n"
  "  /* Calculate and store hit for the code location specified in ecx. */\n"
  "\n"
#ifndef COVERAGE_ONLY
  "  movl __afl_prev_loc, %edi\n"
  "  xorl %ecx, %edi\n"
  "  shrl $1, %ecx\n"
  "  movl %ecx, __afl_prev_loc\n"
#else
  "  movl %ecx, %edi\n"
#endif /* ^!COVERAGE_ONLY */
  "\n"
#ifdef SKIP_COUNTS
  "  orb  $1, (%edx, %edi, 1)\n"
#else
  "  incb (%edx, %edi, 1)\n"
#endif /* ^SKIP_COUNTS */
  "\n"
```

图5 记录 tuple 信息

而另一分支在进行初始操作之外，主要作用还是维护 forkserver，它会将已经初始化好的目标进程，例如暂停在 main 函数入口，按需再 fork 出一个子进程交予 fuzzer 进行测试，即 fuzzer 是被 fuzz 进程的父父进程，借助 copy-on-write 特性，此方法可以极大的提高 fuzz 效率。forkserver 和 fuzzer 间是通过 pipe 管道进行通信的，除了控制命令，fork 成功后的 PID 以及 waitpid 返回状态都是借由此方式传递：

```
/*file: afl-as.h*/
"  /* Once woken up, create a clone of our process. */\n"
"\n"
"  call fork\n"
"\n"
"  cmpl $0, %eax\n"
"  jl   __afl_die\n"
"  je   __afl_fork_resume\n"
"\n"
"  /* In parent process: write PID to pipe, then wait for child. */\n"
"\n"
"  movl  %eax, __afl_fork_pid\n"
"\n"
"  pushl $4              /* length    */\n"
"  pushl $__afl_fork_pid /* data      */\n"
"  pushl $" STRINGIFY((FORKSRV_FD + 1)) "      /* file desc */\n"
"  call  write\n"
"  addl  $12, %esp\n"
"\n"
"  pushl $0             /* no flags  */\n"
"  pushl $__afl_temp    /* status    */\n"
"  pushl __afl_fork_pid /* PID       */\n"
"  call  waitpid\n"
"  addl  $12, %esp\n"
"\n"
"  cmpl  $0, %eax\n"
"  jle   __afl_die\n"
"\n"
"  /* Relay wait status to pipe, then loop back. */\n"
"\n"
"  pushl $4          /* length    */\n"
"  pushl $__afl_temp /* data      */\n"
"  pushl $" STRINGIFY((FORKSRV_FD + 1)) "  /* file desc */\n"
"  call  write\n"
"  addl  $12, %esp\n"
"\n"
"  jmp __afl_fork_wait_loop\n"
"\n"
```

图6 forkserver 和 fuzzer 间的交互

因此，如果我们使用 afl-gcc 来代替 gcc 进行源码编译，那么得到的目标程序将被插入相关代码。

2.2 llvm 模式

该模式之所以能进行编译级插桩主要还是得益于 LLVM 优秀的架构设计，简言之，代码首先由编译器前端 clang 处理后得到中间代码 IR，再经过各 pass 工作节点的优化和转换，最终交给编译器后端生成机器码：

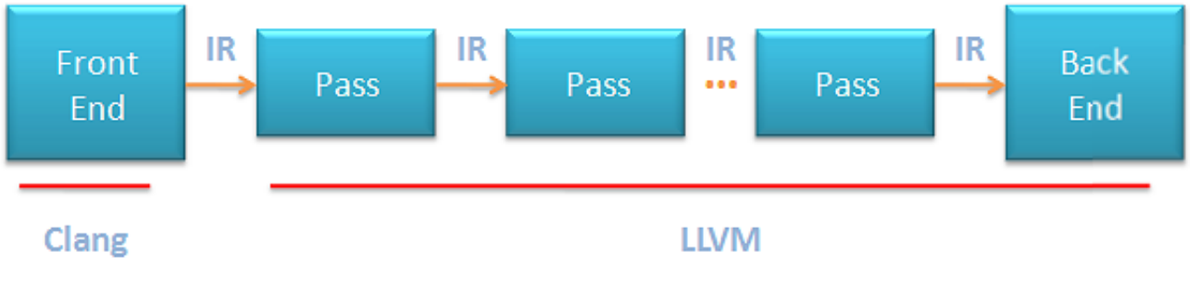


图7 LLVM 架构设计

AFL 的插桩思路是通过编写 pass 来实现 tuple 信息的记录，在此过程中会对每一基本块都插入探针，具体代码在 afl-llvm-pass.so.cc 文件。而初始化和 forkserver 操作则通过链接完成，afl-llvm-rt.o.c 文件实现了 afl-as.h 中 main_payload 的这部分功能：

```
/*file: afl-clang-fast.c   function: edit_params*/
 /* we want to call a function from the runtime .o file. */

 cc_params[cc_par_cnt++] = "-D__AFL_INIT()="
   "do { static volatile char *_A __attribute__((used)); "
   " _A = (char*)\"" DEFER_SIG "\"; "
   "__attribute__((visibility(\"default\"))) "
   "void _I(void) __asm__(\"__afl_manual_init\"); "
   "_I(); } while (0)";

 if (maybe_linking) {

   if (x_set) {
     cc_params[cc_par_cnt++] = "-x";
     cc_params[cc_par_cnt++] = "none";
   }

   switch (bit_mode) {

     case 0:
       cc_params[cc_par_cnt++] = alloc_printf("%s/afl-llvm-rt.o", obj_path);
       break;
```

图8 clang 编译参数设置

2.3 qemu 模式

最后提下 qemu 模式，AFL 在 DBI（Dynamic Binary Instrumentation）框架上选择的是对其性能支持较好的 QEMU，而非 DynamoRIO 或 PIN。对于该模式下的插桩，AFL 的思路是直接利用 QEMU 内置的跟踪功能，并通过 patch 源码的方式来实现 afl-as.h 中的操作：

```
/*file: afl-qemu-cpu-inl.h*/
/* This snippet kicks in when the instruction pointer is positioned at
   _start and does the usual forkserver stuff, not very different from
   regular instrumentation injected via afl-as.h. */

#define AFL_QEMU_CPU_SNIPPET2 do { \
    if(itb->pc == afl_entry_point) { \
      afl_setup(); \
      afl_forkserver(cpu); \
    } \
    afl_maybe_log(itb->pc); \
  } while (0)
```

图9 patch 源码实现 afl-as.h 中的操作

3 fuzzer 模块

我们继续往下，本小节将讨论 AFL 中最为重要的部分，即 fuzzer 实现，其主要作用是通过不断改变测试用例来影响目

标程序的执行路径，当然，这里所采用的都是经过实践检验确是行之有效的方法策略，相关代码都在 afl-fuzz.c 文件，我们将围绕初始化、fuzzing 策略和语料库更迭来展开分析。

3.1 初始化

起始阶段 fuzzer 会进行一系列的准备工作，为记录插桩得到的目标程序执行路径，即 tuple 信息，这里通过名为 `trace_bits` 和 `virgin_bits` 的 bitmap 来分别记录当前的 tuple 信息及总的 tuple 信息，其中 `trace_bits` 位于共享内存上，能方便进程间的通信。另外，还分别通过名为 `virgin_tmout` 和 `virgin_crash` 的 bitmap 来记录 fuzz 过程中出现的所有目标程序 timeout 及 crash 时的 tuple 信息，它们的设置如下：

```
/*file: afl-fuzz.c*/
/* Configure shared memory and virgin_bits. This is called at startup. */
EXP_ST void setup_shm(void) {

  u8* shm_str;

  if (!in_bitmap) memset(virgin_bits, 255, MAP_SIZE);
  memset(virgin_tmout, 255, MAP_SIZE);
  memset(virgin_crash, 255, MAP_SIZE);

  shm_id = shmget(IPC_PRIVATE, MAP_SIZE, IPC_CREAT | IPC_EXCL | 0600);
  if (shm_id < 0) PFATAL("shmget() failed");

  atexit(remove_shm);

  shm_str = alloc_printf("%d", shm_id);
  if (!dumb_mode) setenv(SHM_ENV_VAR, shm_str, 1);
  ck_free(shm_str);

  trace_bits = shmat(shm_id, NULL, 0);
  if (!trace_bits) PFATAL("shmat() failed");
}
```

图10 记录 tuple 信息的 bitmap 设置

此阶段还有一个需要特别提的操作是 forkserver 上线，由 `init_forkserver` 函数来完成，也就是运行 afl-as.h 文件 `main_payload` 中维护 forkserver 的分支，这样一来 `run_target` 函数只需关注和 forkserver 的交互即可，而不必每次都重新创建一个目标进程，其中， `fsrv_ctl_fd` 管道用于写， `fsrv_st_fd` 管道用来读：

```
/*file: afl-fuzz.c   function: run_target*/
  /* If we're running in "dumb" mode, we can't rely on the fork server
     logic compiled into the target program, so we will just keep calling
     execve(). */

  if (dumb_mode == 1 || no_forkserver) {
    ......
  } else {
    s32 res;

    /* In non-dumb mode, we have the fork server up and running, so simply
       tell it to have at it, and then read back PID. */

    if ((res = write(fsrv_ctl_fd, &prev_timed_out, 4)) != 4) {
      if (stop_soon) return 0;
      RPFATAL(res, "Unable to request new process from fork server (OOM?)");

    }

    if ((res = read(fsrv_st_fd, &child_pid, 4)) != 4) {
      if (stop_soon) return 0;
      RPFATAL(res, "Unable to request new process from fork server (OOM?)");
    }

    if (child_pid <= 0) FATAL("Fork server is misbehaving (OOM?)");
  }
```

图11 forkserver 上线监听

此外，余下的关键初始操作大致可分为目录处理和前期检测两个类别，我们给出相关函数的简析：

```
【目录处理相关的操作】
//读取输入的各测试用例，然后加入语料库队列
static void read_testcases(void);
//将输入的各测试用例重命名后硬链接到输出目录
static void pivot_inputs(void);
//输出目录的设置
EXP_ST void setup_dirs_fds(void);
//载入手动提供的字典
static void load_extras(u8* dir);
//载入自动生成的字典
static void load_auto(void);

【检测相关的操作】
//测试用例的评估
static u8 calibrate_case(char** argv, struct queue_entry* q,
    u8* use_mem, u32 handicap, u8 from_queue);
//确认目标程序是否能正常执行
static void perform_dry_run(char** argv);
//检查目标程序是否存在，是否为脚本，是否正常插桩
EXP_ST void check_binary(u8* fname);
//给出初始态的统计信息
static void show_init_stats(void);
```

3.2 fuzzing 策略

接着我们来看 fuzzing 策略，它在很大程度上关乎一个 fuzzer 的成败，因此变异引擎的设计颇显关键。为了得到更高的代码覆盖率，我们对测试用例施加的变化既不能过大也不能太小，同时还要平衡好各方面的效率，如何拿捏好这个度更像一门艺术。

在 AFL 中用到的 fuzzing 策略分为两类，即确定性策略和随机性策略。其中确定性策略又分为位翻转、字节翻转、算术加减、整数替换、字典替换和字典插入，目的是为了生成更多简洁有效的测试用例，不过由于此类策略比较耗时，因此测试用例只会进行一轮这样的操作。而如果一个测试用例在执行完随机性策略后仍未产生新状态，则会将其与另一测试用例随机拼接后再次交由随机性策略处理：

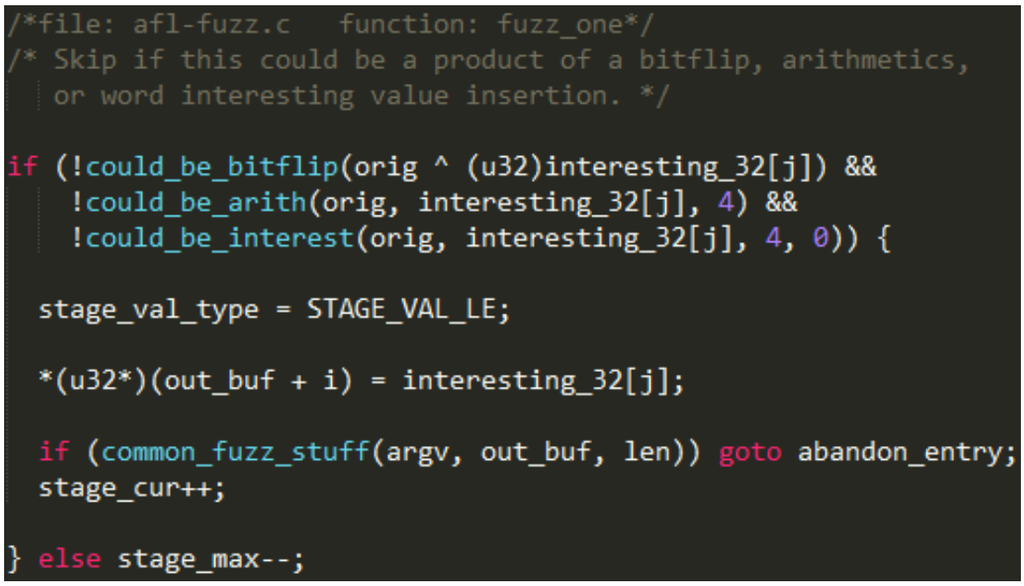
```
/*file: afl-fuzz.c   function: fuzz_one*/
  /* Skip right away if -d is given, if we have done deterministic fuzzing on
     this entry ourselves (was_fuzzed), or if it has gone through deterministic
     testing in earlier, resumed runs (passed_det). */
  if (skip_deterministic || queue_cur->was_fuzzed || queue_cur->passed_det)
    goto havoc_stage;
  ......
  /*********************************************
   * SIMPLE BITFLIP (+dictionary construction) *
   *********************************************/
  ......
  /************************
   * ARITHMETIC INC/DEC *
   ************************/
  ......
  /************************
   * INTERESTING VALUES *
   ************************/
  ......
  /**********************
   * DICTIONARY STUFF *
   **********************/
  ......
  /* If we made this to here without jumping to havoc_stage or abandon_entry,
     we're properly done with deterministic steps and can mark it as such
     in the .state/ directory. */
  if (!queue_cur->passed_det) mark_as_det_done(queue_cur);

  /****************
   * RANDOM HAVOC *
   ****************/
havoc_stage:
  ......
  /*************
   * SPLICING *
   *************/
```

图12 变异引擎的 fuzzing 策略

对于各变异操作，基本的处理流程如下：

```
fuzz_one //测试用例变异过程
      common_fuzz_stuff //变异完成后的通用处理步骤
            write_to_testcase //将变异后的内容写入测试文件
            run_target //运行目标进程
            save_if_interesting //判断是否保存该测试用例
                  has_new_bits //判断该测试用例是否产生新状态
```

而在确定性策略 fuzzing 过程中，如果发现后续的变异操作已经在前面进行过了，为避免重复将会跳过该操作，如果没有重复则由 `common_fuzz_stuff` 函数将变异后的内容写入测试文件并运行目标程序：



```
/*file: afl-fuzz.c   function: fuzz_one*/
/* Skip if this could be a product of a bitflip, arithmetics,
   or word interesting value insertion. */

if (!could_be_bitflip(orig ^ (u32)interesting_32[j]) &&
    !could_be_arith(orig, interesting_32[j], 4) &&
    !could_be_interest(orig, interesting_32[j], 4, 0)) {

  stage_val_type = STAGE_VAL_LE;

  *(u32*)(out_buf + i) = interesting_32[j];

  if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
  stage_cur++;

} else stage_max--;
```

图13 跳过重复的变异操作

3.3 语料库更迭

下面谈谈语料库的更迭，随着 fuzzing 的深入，目标程序会产生越来越多的执行路径，相应测试用例能否加入语料库 queue 队列取决于 `trace_bits` 是否出现新状态。我们已经知道 fuzzer 会通过 `virgin_bits` 来记录目标程序的总 tuple 信息，每个 tuple 对应一个字节，初始值为 0xFF，即各比特位均为 1。如此设计自是有用意，可否记得之前提过每个 tuple 都分为 8 个命中组，固每个比特位可对应一个命中组，若无此类新状态出现其值仍为 1，若出现其值则置 0。这也是为何执行完目标程序后要调用 `classify_counts` 函数对此次记录的各 tuple 命中数进行归组：

```c
/*file: afl-fuzz.c*/
/* Destructively classify execution counts in a trace. This is used as a
   preprocessing step for any newly acquired traces. Called on every exec,
   must be fast. */

static const u8 count_class_lookup8[256] = {

  [0]           = 0,
  [1]           = 1,
  [2]           = 2,
  [3]           = 4,
  [4 ... 7]     = 8,
  [8 ... 15]    = 16,
  [16 ... 31]   = 32,
  [32 ... 127]  = 64,
  [128 ... 255] = 128

};
```

图14 按 tuple 的命中数归组

这样只需将执行完归组操作后的 `trace_bits` 和 `virgin_bits` 做个比对就能判断出是否有新状态产生，同时，采用位运算也是为了提高代码执行的速率：

```c
/*file: afl-fuzz.c*/
/* Check if the current execution path brings anything new to the table.
   Update virgin bits to reflect the finds. Returns 1 if the only change is
   the hit-count for a particular tuple; 2 if there are new tuples seen. */

static inline u8 has_new_bits(u8* virgin_map) {

  u32* current = (u32*)trace_bits;
  u32* virgin  = (u32*)virgin_map;
  u32  i = (MAP_SIZE >> 2);
  u8   ret = 0;

  while (i--) {

    if (unlikely(*current) && unlikely(*current & *virgin)) {

      if (likely(ret < 2)) {

        u8* cur = (u8*)current;
        u8* vir = (u8*)virgin;

        if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff) ||
            (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff)) ret = 2;
        else ret = 1;

      }

      *virgin &= ~*current;
    }

    current++;
    virgin++;
  }
```

图15 判断是否有新状态产生

另外，在 fuzzer 中维护着一个覆盖了当前所有 tuple 状态的语料库子集，这些测试用例被标记为 “favored”，它们在性能上相对较优，因此能获得更大的 fuzz 概率，即 `fuzz_one` 函数开始部分所做的判断。要得到这样的集合需要借助一个称为 `top_rated` 的数组，它保存有各 tuple 状态当前相应的最优测试用例，当有新状态产生时，我们判断其是否较 `top_rated` 中已有的成员更优：

```
/*file: afl-fuzz.c   function: update_bitmap_score*/
  /* For every byte set in trace_bits[], see if there is a previous winner,
     and how it compares to us. */

  for (i = 0; i < MAP_SIZE; i++)

    if (trace_bits[i]) {

       if (top_rated[i]) {

         /* Faster-executing or smaller test cases are favored. */

         if (fav_factor > top_rated[i]->exec_us * top_rated[i]->len) continue;

         /* Looks like we're going to win. Decrease ref count for the
            previous winner, discard its trace_bits[] if necessary. */

         if (!--top_rated[i]->tc_ref) {
           ck_free(top_rated[i]->trace_mini);
           top_rated[i]->trace_mini = 0;
         }

       }

       /* Insert ourselves as the new winner. */

       top_rated[i] = q;
       q->tc_ref++;
```

图16 判断是否更新 top_rated 数组

如果 `top_rated` 数组被更新，那么之后会调用 `cull_queue` 函数，它会顺序遍历 `top_rated` 数组中的测试用例，直到选出的测试用例集合能完全覆盖当前所有的 tuple 状态，即完成语料库中 “favored” 集合的更新。

4 其它辅助模块

到目前为止我们已经讨论了 AFL 中最核心的 fuzzer 模块和实现边界覆盖率统计的插桩模块，那么接下来我们把重点放在辅助模块上，它们的设计初衷都是为了优化提升 fuzzer 的性能，不过有些特性还留有进步的空间。这里挑出其中比较有意思的 4 块内容进行阐述，也就是语料库及测试用例的最小化、并行 fuzzing、语法字典与内存检测工具。

4.1 语料库及测试用例的最小化

首先，为保证 fuzzing 的高效，我们有必要在语料库和测试用例两个层面进行一些精简化操作，前者使得 CPU 能将更多时间片用在较优测试用例的 fuzz 上，而后者使得测试用例在变异环节更有可能触发新状态，相关代码在 afl-cmin 和 afl-tmin.c 文件。

实际上 fuzzer 模块也内置有相关功能，不过出于执行速率的考量没设计的这么复杂。上小节最后提到的 “favored” 集合

即为精简后的语料库子集，afl-cmin 中的思路类似但更复杂且会将冗余的测试用例删除。至于测试用例的最小化，其目的是尽可能的移除测试用例中的数据，同时保持目标程序的执行状态不变，对于不产生 crash 的测试用例我们可基于前面讨论的插桩技术来判断这些简化操作对目标程序的执行路径是否有影响，afl-tmin.c 中所用到的简化操作分为块清'0'、块移除和字符清 '0'：

```
/*file: afl-tmin.c*/
/* Actually minimize! */
static void minimize(char** argv) {

  ......
  /************************
   * BLOCK NORMALIZATION *
   ************************/
  ......
  /******************
   * BLOCK DELETION *
   ******************/
  ......
  /*************************
   * ALPHABET MINIMIZATION *
   *************************/
  ......
  /***************************
   * CHARACTER MINIMIZATION *
   ***************************/
  ......

  SAYF("\n"
       cGRA "     File size reduced by : " cRST "%0.02f%% (to %u byte%s)\n"
       cGRA "    Characters simplified : " cRST "%0.02f%%\n"
       cGRA "     Number of execs done : " cRST "%u\n"
       cGRA "          Fruitless execs : " cRST "path=%u crash=%u hang=%s%u\n\n",
       100 - ((double)in_len) * 100 / orig_len, in_len, in_len == 1 ? "" : "s",
       ((double)(alpha_d_total)) * 100 / (in_len ? in_len : 1),
       total_execs, missed_paths, missed_crashes, missed_hangs ? cLRD : "",
       missed_hangs);
  ......

}
```

图17 测试用例的简化操作

4.2 并行 fuzzing

同时，对 fuzzing 来说，计算资源是很重要的指标，AFL 在设计上支持并行 fuzzing，这就给了我们很大的可操作空间。由于每个 afl-fuzz 例程只占用 CPU 中的单核，因此这里的并行对单台多核 CPU 机器也是支持的，我们可以通过 afl-gotcpu 工具来获取 CPU 的使用情况，进而决定增加或减少 afl-fuzz 例程，并可通过 afl-whatsup 工具统计各并行例程的结果。并行的一大好处是可以实现各例程间的语料库共享，从而加快 fuzz 过程。

4.3 语法字典

而字典对 fuzzer 性能的提升也很关键，我们知道 AFL 变异引擎的确定性策略中包含字典替换和字典插入操作，这使得我们在 fuzz 一些强语法性的文件格式时能获得更高的代码覆盖率，虽然此方法离直接生成符合特定语法格式的测试用例还有距离，但实际效果也还算理想。libtokencap 目录下的代码旨在方便我们生成字典文件，通过对 strcmp()、

memcmp() 这类函数进行类似 hook 的操作我们可提取参数中的语法关键字。

4.4 内存检测工具

此外，为了更好的检测与内存错误相关的 bug，AFL 还引入了对内存检测工具 asan（address sanitizer）的支持，但在 fuzzing 过程中由于内存开销太大，所以实用性还不强。libdislocator 目录下提供了一个更为简单的内存检测工具实现，当然就使用方面仍需改善。

5 结语

本文讨论了 AFL 项目中主要的设计思路，但并不涉及 `experimental` 目录下的那些新特性试验，这方面读者可自行研究。总体来看，AFL 在设计上遵循的是一套简单有效的方式，没做太多复杂的过度优化，能恰当的挠到痒处，而非追求看起来很酷实践后却是不尽如人意的隔靴搔痒。最后，笔者对于漏洞挖掘这块知识尚是初学，行文中如有错误或纰漏之处还望各位多加指正。

6 参考

[1] American fuzzy lop
http://lcamtuf.coredump.cx/afl/
[2] Real world fuzzing
http://pages.cs.wisc.edu/~rist/642-fall-2011/toorcon.pdf
[3] Fuzzing random programs without execve()
https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html
[4] afl-fuzz: making up grammar with a dictionary in hand
https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html