AFL 源代码速通笔记

> 本文属于 blog 博文类型，此处用于提醒自己书写的内容规范。

因为认识的师傅们都开始卷 fuzz 了，迫于生活压力，于是也开始看这方面的内容了。由于 AFL 作为一个现在仍然适用且比较经典的 fuzzer，因此笔者也打算从它开始。

而既然本文要开始撰写 AFL 的实现原理，那么姑且在这里立一个小目标：争取在读完 AFL 之后能够自己实现(魔改)一个简单的 fuzzer。(不过也得看看其他的 fuzzer，毕竟 AFL 确实有些年头了，主要还是学学它的设计思路)

行吧，那么差不多就这样。

> 本来，本篇博文叫做 《AFL 源代码阅读笔记》，结果跟着大佬们的笔记去读（sakura 师傅的笔记确实是神中神，本文也有很多地方照搬了师傅的原文，因为说实话我觉得自己也写不到那么详细），囫囵吞枣般速通了，前前后后两天时间这样，但感觉自己尚且没有自己实现的能力，还是比较令人失望的（我怎么这么菜）

---

afl-gcc 原理

首先，一般我们用 afl 去 fuzz 一些项目的时候都需要用 afl-gcc 去代替 gcc 进行编译。先说结论，这一步的目的其实是为了**向代码中插桩**，完成插桩后其实还是**调用原生的 gcc 进行编译**。

```
其实这个描述有些偏颇，插桩其实是 afl-as 负责的，不过在这里，笔者将 afl-gcc 和 afl-as
放到同一节，因此用了这样的表述，下文会具体分析 afl-as 的原理。
```

首先需要说明的是，gcc 对代码的编译流程的分层次的：

> 源代码—>预编译后的源代码—>汇编代码—>机器码—>链接后的二进制文件

其中，从源代码到汇编代码的步骤由 gcc 完成；而汇编代码到机器码的部分由 as 完成。

而 afl-gcc 的源代码如下：

```
int main(int argc, char** argv) {
  .......

  find_as(argv[0]);
  edit_params(argc, argv);
  execvp(cc_params[0], (char**)cc_params);
  FATAL("Oops, failed to execute '%s' - check your PATH", cc_params[0]);
  return 0;
}
```

- find_as：查找 as 这个二进制程序，用 afl-as 替换它
- edit_params：修改参数
- execvp：调用原生 gcc 对代码进行编译

```
static void edit_params(u32 argc, char** argv) {
        ......
#else

    if (!strcmp(name, "afl-g++")) {
      u8* alt_cxx = getenv("AFL_CXX");
      cc_params[0] = alt_cxx ? alt_cxx : (u8*)"g++";
    } else if (!strcmp(name, "afl-gcj")) {
      u8* alt_cc = getenv("AFL_GCJ");
      cc_params[0] = alt_cc ? alt_cc : (u8*)"gcj";
    } else {
      u8* alt_cc = getenv("AFL_CC");
      cc_params[0] = alt_cc ? alt_cc : (u8*)"gcc";
    }

#endif /* __APPLE__ */

  }

  while (--argc) {
    u8* cur = *(++argv);

    if (!strncmp(cur, "-B", 2)) {

      if (!be_quiet) WARNF("-B is already set, overriding");

      if (!cur[2] && argc > 1) { argc--; argv++; }
      continue;

    }
    if (!strcmp(cur, "-integrated-as")) continue;

    if (!strcmp(cur, "-pipe")) continue;

#if defined(__FreeBSD__) && defined(__x86_64__)
    if (!strcmp(cur, "-m32")) m32_set = 1;
#endif

    if (!strcmp(cur, "-fsanitize=address") ||
        !strcmp(cur, "-fsanitize=memory")) asan_set = 1;

    if (strstr(cur, "FORTIFY_SOURCE")) fortify_set = 1;

    cc_params[cc_par_cnt++] = cur;

  }

  cc_params[cc_par_cnt++] = "-B";
  cc_params[cc_par_cnt++] = as_path;

  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";

  }

  if (asan_set) {

    /* Pass this on to afl-as to adjust map density. */

    setenv("AFL_USE_ASAN", "1", 1);

  } else if (getenv("AFL_USE_ASAN")) {

    if (getenv("AFL_USE_MSAN"))
      FATAL("ASAN and MSAN are mutually exclusive");

    if (getenv("AFL_HARDEN"))
      FATAL("ASAN and AFL_HARDEN are mutually exclusive");
    cc_params[cc_par_cnt++] = "-U_FORTIFY_SOURCE";
    cc_params[cc_par_cnt++] = "-fsanitize=address";

  } else if (getenv("AFL_USE_MSAN")) {

    if (getenv("AFL_USE_ASAN"))
      FATAL("ASAN and MSAN are mutually exclusive");

    if (getenv("AFL_HARDEN"))
      FATAL("MSAN and AFL_HARDEN are mutually exclusive");

    cc_params[cc_par_cnt++] = "-U_FORTIFY_SOURCE";
    cc_params[cc_par_cnt++] = "-fsanitize=memory";
  }

  if (!getenv("AFL_DONT_OPTIMIZE")) {

#if defined(__FreeBSD__) && defined(__x86_64__)

    /* On 64-bit FreeBSD systems, clang -g -m32 is broken, but -m32 itself
       works OK. This has nothing to do with us, but let's avoid triggering
       that bug. */

    if (!clang_mode || !m32_set)
      cc_params[cc_par_cnt++] = "-g";
#else
      cc_params[cc_par_cnt++] = "-g";
#endif

    cc_params[cc_par_cnt++] = "-O3";
    cc_params[cc_par_cnt++] = "-funroll-loops";

    /* Two indicators that you're building for fuzzing; one of them is
       AFL-specific, the other is shared with libfuzzer. */

    cc_params[cc_par_cnt++] = "-D__AFL_COMPILER=1";
    cc_params[cc_par_cnt++] = "-
DFUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION=1";

  }

  if (getenv("AFL_NO_BUILTIN")) {

    cc_params[cc_par_cnt++] = "-fno-builtin-strcmp";
    cc_params[cc_par_cnt++] = "-fno-builtin-strncmp";
    cc_params[cc_par_cnt++] = "-fno-builtin-strcasecmp";
    cc_params[cc_par_cnt++] = "-fno-builtin-strncasecmp";
    cc_params[cc_par_cnt++] = "-fno-builtin-memcmp";
    cc_params[cc_par_cnt++] = "-fno-builtin-strstr";
    cc_params[cc_par_cnt++] = "-fno-builtin-strcasestr";

  }
  cc_params[cc_par_cnt] = NULL;
}
```

挺长的，不过逻辑基本上都是重复的，主要做两件事：

- 给 gcc 添加一些额外的参数
- 根据参数设置一些 flag

在完成了汇编以后，接下来会使用 afl-as 对生成的汇编代码进行插桩：

```
int main(int argc, char** argv) {
  ......

  gettimeofday(&tv, &tz);
  rand_seed = tv.tv_sec ^ tv.tv_usec ^ getpid();
  srandom(rand_seed);

  edit_params(argc, argv);

  ......

  if (!just_version) add_instrumentation();

  if (!(pid = fork())) {

    execvp(as_params[0], (char**)as_params);
    FATAL("Oops, failed to execute '%s' - check your PATH", as_params[0]);

  }

  if (pid < 0) PFATAL("fork() failed");

  if (waitpid(pid, &status, 0) <= 0) PFATAL("waitpid() failed");

  if (!getenv("AFL_KEEP_ASSEMBLY")) unlink(modified_file);

  exit(WEXITSTATUS(status));
}
```

在 `afl-as` 中，仍然使用 `edit_params` 编辑和修改参数，并使用 `add_instrumentation` 来对生成的汇编代码进行插桩。完成插桩后，用 fork 生成子进程，并调用原生的 as 进行编译。

插桩逻辑也很朴素：

```
static void add_instrumentation(void) {
       ......

    /* If we're in the right mood for instrumenting, check for function
       names or conditional labels. This is a bit messy, but in essence,
       we want to catch:

         ^main:      - function entry point (always instrumented)
         ^.L0:       - GCC branch label
         ^.LBB0_0:   - clang branch label (but only in clang mode)
         ^\tjnz foo  - conditional branches

       ...but not:

         ^# BB#0:    - clang comments
         ^ # BB#0:   - ditto
         ^.Ltmp0:    - clang non-branch labels
         ^.LC0       - GCC non-branch labels
         ^.LBB0_0:   - ditto (when in GCC mode)
         ^\tjmp foo  - non-conditional jumps

       Additionally, clang and GCC on MacOS X follow a different convention
       with no leading dots on labels, hence the weird maze of #ifdefs
       later on.

     */

    if (skip_intel || skip_app || skip_csect || !instr_ok ||
        line[0] == '#' || line[0] == ' ') 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;

    }

    /* Label of some sort. This may be a branch destination, but we need to
       tread carefully and account for several different formatting
       conventions. */

#ifdef __APPLE__

    /* Apple: L<whatever><digit>: */

    if ((colon_pos = strstr(line, ":"))) {

      if (line[0] == 'L' && isdigit(*(colon_pos - 1))) {

#else

    /* Everybody else: .L<whatever>: */

    if (strstr(line, ":")) {

      if (line[0] == '.') {

#endif /* __APPLE__ */

        /* .L0: or LBB0_0: style jump destination */

#ifdef __APPLE__

        /* Apple: L<num> / LBB<num> */

        if ((isdigit(line[1]) || (clang_mode && !strncmp(line, "LBB", 3)))
            && R(100) < inst_ratio) {

#else
        /* Apple: .L<num> / .LBB<num> */
        if ((isdigit(line[2]) || (clang_mode && !strncmp(line + 1, "LBB",
3)))
            && R(100) < inst_ratio) {
#endif /* __APPLE__ */
```

```
          /* An optimization is possible here by adding the code only if the
             label is mentioned in the code in contexts other than call /
jmp.

             That said, this complicates the code by requiring two-pass
             processing (messy with stdin), and results in a speed gain
             typically under 10%, because compilers are generally pretty
good
             about not generating spurious intra-function jumps.

             We use deferred output chiefly to avoid disrupting
             .Lfunc_begin0-style exception handling calculations (a problem
on
             MacOS X). */

          if (!skip_next_label) instrument_next = 1; else skip_next_label =
0;
        }
      } else {
        /* Function label (always instrumented, deferred mode). */
        instrument_next = 1;
      }
    }
  }

  if (ins_lines)
    fputs(use_64bit ? main_payload_64 : main_payload_32, outf);

  if (input_file) fclose(inf);
  fclose(outf);

  if (!be_quiet) {

    if (!ins_lines) WARNF("No instrumentation targets found%s.",
                          pass_thru ? " (pass-thru mode)" : "");
    else OKF("Instrumented %u locations (%s-bit, %s mode, ratio %u%%).",
             ins_lines, use_64bit ? "64" : "32",
             getenv("AFL_HARDEN") ? "hardened" :
             (sanitizer ? "ASAN/MSAN" : "non-hardened"),
             inst_ratio);

  }
}
```

简单来说就是一个循环读取每行汇编代码，并对特定的汇编代码进行插桩：

- 首先需要保证代码位于 `text` 内存段
- 如果是 `main` 函数或分支跳转指令则进行插桩
- 如果是注释或强制跳转指令则不插桩

插桩的具体代码保存在 `afl-as.h` 中，在最后一节中笔者会另外介绍，这里我们可以暂时忽略它的实现细节继续往下。

afl-fuzz

按照顺序，现在程序是编译好了，接下来就要用 `afl-fuzz` 对它进行模糊测试了。

一般来说，我们会用 `afl-fuzz -i input -o output -- programe` 启动 fuzzer，对应的，`afl-fuzz.c` 中的前半部分都在做参数解析的工作：

```c
int main(int argc, char** argv) {
  ......

  gettimeofday(&tv, &tz);
  srandom(tv.tv_sec ^ tv.tv_usec ^ getpid());

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

    switch (opt) {

      case 'i': /* input dir */

        if (in_dir) FATAL("Multiple -i options not supported");
        in_dir = optarg;

        if (!strcmp(in_dir, "-")) in_place_resume = 1;

        break;

      case 'o': /* output dir */

        if (out_dir) FATAL("Multiple -o options not supported");
        out_dir = optarg;
        break;

        ......

      case 'V': /* Show version number */

        /* Version number has been printed already, just quit. */
        exit(0);
      default:
        usage(argv[0]);
    }
```

这部分我们大致看一下就行了，主要的关注点自然不在参数解析部分。

```
int main(int argc, char** argv) {
  ......

  setup_signal_handlers();
  check_asan_opts();

  if (sync_id) fix_up_sync();

  if (!strcmp(in_dir, out_dir))
    FATAL("Input and output directories can't be the same");

  if (dumb_mode) {

    if (crash_mode) FATAL("-C and -n are mutually exclusive");
    if (qemu_mode)  FATAL("-Q and -n are mutually exclusive");

  }

  if (getenv("AFL_NO_FORKSRV"))    no_forkserver    = 1;
  if (getenv("AFL_NO_CPU_RED"))    no_cpu_meter_red = 1;
  if (getenv("AFL_NO_ARITH"))      no_arith         = 1;
  if (getenv("AFL_SHUFFLE_QUEUE")) shuffle_queue    = 1;
  if (getenv("AFL_FAST_CAL"))      fast_cal         = 1;

  if (getenv("AFL_HANG_TMOUT")) {
    hang_tmout = atoi(getenv("AFL_HANG_TMOUT"));
    if (!hang_tmout) FATAL("Invalid value of AFL_HANG_TMOUT");
  }

  if (dumb_mode == 2 && no_forkserver)
    FATAL("AFL_DUMB_FORKSRV and AFL_NO_FORKSRV are mutually exclusive");

  if (getenv("AFL_PRELOAD")) {
    setenv("LD_PRELOAD", getenv("AFL_PRELOAD"), 1);
    setenv("DYLD_INSERT_LIBRARIES", getenv("AFL_PRELOAD"), 1);
  }

  if (getenv("AFL_LD_PRELOAD"))
    FATAL("Use AFL_PRELOAD instead of AFL_LD_PRELOAD");
```

```c
  save_cmdline(argc, argv);

  fix_up_banner(argv[optind]);

  check_if_tty();

  get_core_count();

#ifdef HAVE_AFFINITY
  bind_to_free_cpu();
#endif /* HAVE_AFFINITY */

  check_crash_handling();
  check_cpu_governor();

  setup_post();
  setup_shm();
  init_count_class16();

  setup_dirs_fds();
  read_testcases();
  load_auto();

  pivot_inputs();

  if (extras_dir) load_extras(extras_dir);

  if (!timeout_given) find_timeout();

  detect_file_args(argv + optind + 1);

  if (!out_file) setup_stdio_file();

  check_binary(argv[optind]);

  start_time = get_cur_time();

  if (qemu_mode)
    use_argv = get_qemu_argv(argv[0], argv + optind, argc - optind);
  else
    use_argv = argv + optind;

  perform_dry_run(use_argv);

  cull_queue();
```

```
show_init_stats();

seek_to = find_start_position();

write_stats_file(0, 0, 0);
save_auto();

if (stop_soon) goto stop_fuzzing;

/* Woop woop woop */

if (!not_on_tty) {
  sleep(4);
  start_time += 4000;
  if (stop_soon) goto stop_fuzzing;
}

while (1) {

  u8 skipped_fuzz;

  cull_queue();

  if (!queue_cur) {

    queue_cycle++;
    current_entry     = 0;
    cur_skipped_paths = 0;
    queue_cur         = queue;

    while (seek_to) {
      current_entry++;
      seek_to--;
      queue_cur = queue_cur->next;
    }

    show_stats();

    if (not_on_tty) {
      ACTF("Entering queue cycle %llu.", queue_cycle);
      fflush(stdout);
    }

    /* If we had a full queue cycle with no new finds, try
       recombination strategies next. */
      if (queued_paths == prev_queued) {

        if (use_splicing) cycles_wo_finds++; else use_splicing = 1;

      } else cycles_wo_finds = 0;

      prev_queued = queued_paths;

      if (sync_id && queue_cycle == 1 && getenv("AFL_IMPORT_FIRST"))
        sync_fuzzers(use_argv);

    }

    skipped_fuzz = fuzz_one(use_argv);

    if (!stop_soon && sync_id && !skipped_fuzz) {

      if (!(sync_interval_cnt++ % SYNC_INTERVAL))
        sync_fuzzers(use_argv);

    }

    if (!stop_soon && exit_1) stop_soon = 2;

    if (stop_soon) break;

    queue_cur = queue_cur->next;
    current_entry++;

  }

  if (queue_cur) show_stats();

  /* If we stopped programmatically, we kill the forkserver and the current
runner.
     If we stopped manually, this is done by the signal handler. */
  if (stop_soon == 2) {
      if (child_pid > 0) kill(child_pid, SIGKILL);
      if (forksrv_pid > 0) kill(forksrv_pid, SIGKILL);
  }
  /* Now that we've killed the forkserver, we wait for it to be able to get
rusage stats. */
  if (waitpid(forksrv_pid, NULL, 0) <= 0) {
    WARNF("error waitpid\n");
  }
  write_bitmap();
  write_stats_file(0, 0, 0);
  save_auto();

stop_fuzzing:

  SAYF(CURSOR_SHOW cLRD "\n\n+++ Testing aborted %s +++\n" cRST,
       stop_soon == 2 ? "programmatically" : "by user");

  /* Running for more than 30 minutes but still doing first cycle? */

  if (queue_cycle == 1 && get_cur_time() - start_time > 30 * 60 * 1000) {

    SAYF("\n" cYEL "[!] " cRST
           "Stopped during the first cycle, results may be incomplete.\n"
           "    (For info on resuming, see %s/README.)\n", doc_path);

  }

  fclose(plot_file);
  destroy_queue();
  destroy_extras();
  ck_free(target_path);
  ck_free(sync_id);

  alloc_report();

  OKF("We're done here. Have a nice day!\n");

  exit(0);
}
```

setup_signal_handlers

设置一些信号处理函数，比如说退出信号时要主动释放子进程、窗口大小调整时要跟踪变化等。

check_asan_opts

读取环境变量ASAN_OPTIONS和MSAN_OPTIONS，做一些检查

fix_up_sync

略

save_cmdline

保存当前的命令

fix_up_banner

创建一个 banner

check_if_tty

检查是否在tty终端上面运行。

get_core_count

计数logical CPU cores。

check_crash_handling

检查崩溃处理函数，确保崩溃后不会进入程序。

```
  s32 fd = open("/proc/sys/kernel/core_pattern", O_RDONLY);
  u8  fchar;
  if (fd < 0) return;
  ACTF("Checking core_pattern...");
  if (read(fd, &fchar, 1) == 1 && fchar == '|') {
    SAYF("\n" cLRD "[-] " cRST
         "Hmm, your system is configured to send core dump notifications to
an\n"
         "    external utility. This will cause issues: there will be an
extended delay\n"
         "    between stumbling upon a crash and having this information
relayed to the\n"
         "    fuzzer via the standard waitpid() API.\n\n"

         "    To avoid having crashes misinterpreted as timeouts, please log
in as root\n"
         "    and temporarily modify /proc/sys/kernel/core_pattern, like
so:\n\n"

         "    echo core >/proc/sys/kernel/core_pattern\n");

    if (!getenv("AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES"))
      FATAL("Pipe at the beginning of 'core_pattern'");
  }
  close(fd);
```

笔者在 Ubuntu20 上跑 AFL 就会遇到这个问题，因为在默认情况下，系统会将崩溃信息通过管道发送给外部程序，由于这会影响到效率，因此通过 `echo core >/proc/sys/kernel/core_pattern` 修改保存崩溃信息的方式，将它保存为本地文件。

check_cpu_governor

检查 `cpu` 的调节器，来使得 `cpu` 可以处于高效的运行状态。

setup_post

如果用户指定了环境变量 `AFL_POST_LIBRARY` ，那么就会从对应的路径下加载动态库并加载 `afl_postprocess` 函数并保存在 `post_handler` 中。

setup_shm

```
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 somebody is asking us to fuzz instrumented binaries in dumb mode,
     we don't want them to detect instrumentation, since we won't be sending
     fork server commands. This should be replaced with better auto-
detection
     later on, perhaps? */
  if (!dumb_mode) setenv(SHM_ENV_VAR, shm_str, 1);
  ck_free(shm_str);
  trace_bits = shmat(shm_id, NULL, 0);
  if (trace_bits == (void *)-1) PFATAL("shmat() failed");
}
```

初始化 `virgin_bits` 数组用于保存后续模糊测试中覆盖的路径，`virgin_tmout` 保存超时的路径，`virgin_crash` 保存崩溃的路径。

同时建立共享内存 `trace_bits`，该变量用于储存样例运行时的路径。

同时将共享内存的唯一标识符 `shm_id` 转为字符串后保存在环境变量 `SHM_ENV_VAR` 中。

init_count_class16

初始化 `count_class_lookup16` 数组，帮助快速归类统计路径覆盖的数量。

setup_dirs_fds

创建输出目录。

read_testcases

读取测试样例。

```
static void read_testcases(void) {

  struct dirent **nl;
  s32 nl_cnt;
  u32 i;
  u8* fn;

  /* Auto-detect non-in-place resumption attempts. */

  fn = alloc_printf("%s/queue", in_dir);
  if (!access(fn, F_OK)) in_dir = fn; else ck_free(fn);

  ACTF("Scanning '%s'...", in_dir);

  /* We use scandir() + alphasort() rather than readdir() because otherwise,
     the ordering  of test cases would vary somewhat randomly and would be
     difficult to control. */

  nl_cnt = scandir(in_dir, &nl, NULL, alphasort);

  if (nl_cnt < 0) {

    if (errno == ENOENT || errno == ENOTDIR)

      SAYF("\n" cLRD "[-] " cRST
           "The input directory does not seem to be valid - try again. The
fuzzer needs\n"
           "    one or more test case to start with - ideally, a small file
under 1 kB\n"
           "    or so. The cases must be stored as regular files directly in
the input\n"
           "    directory.\n");

    PFATAL("Unable to open '%s'", in_dir);

  }

  if (shuffle_queue && nl_cnt > 1) {
    ACTF("Shuffling queue...");
    shuffle_ptrs((void**)nl, nl_cnt);

  }

  for (i = 0; i < nl_cnt; i++) {

    struct stat st;

    u8* fn = alloc_printf("%s/%s", in_dir, nl[i]->d_name);
    u8* dfn = alloc_printf("%s/.state/deterministic_done/%s", in_dir, nl[i]-
>d_name);

    u8  passed_det = 0;

    free(nl[i]); /* not tracked */

    if (lstat(fn, &st) || access(fn, R_OK))
      PFATAL("Unable to access '%s'", fn);

    /* This also takes care of . and .. */

    if (!S_ISREG(st.st_mode) || !st.st_size || strstr(fn,
"/README.testcases")) {

      ck_free(fn);
      ck_free(dfn);
      continue;

    }

    if (st.st_size > MAX_FILE)
      FATAL("Test case '%s' is too big (%s, limit is %s)", fn,
            DMS(st.st_size), DMS(MAX_FILE));

    /* Check for metadata that indicates that deterministic fuzzing
       is complete for this entry. We don't want to repeat deterministic
       fuzzing when resuming aborted scans, because it would be pointless
       and probably very time-consuming. */

    if (!access(dfn, F_OK)) passed_det = 1;
    ck_free(dfn);

    add_to_queue(fn, st.st_size, passed_det);
  }
```

- 首先获取输入样例的文件夹路径 `in_dir`
- 扫描 `in_dir`，如果目录下文件的数量少于等于 0 则报错
- 如果设置了 `shuffle_queue` 就打乱顺序
- 遍历所有文件名，保存在 `fn` 中
- 过滤掉 `.` 和 `..` 这样的路径
- 如果文件的大小超过了 `MAX_FILE` 则终止
- `add_to_queue`

add_to_queue

```
static void add_to_queue(u8* fname, u32 len, u8 passed_det) {

  struct queue_entry* q = ck_alloc(sizeof(struct queue_entry));
  q->fname        = fname;
  q->len          = len;
  q->depth        = cur_depth + 1;
  q->passed_det   = passed_det;

  if (q->depth > max_depth) max_depth = q->depth;

  if (queue_top) {

    queue_top->next = q;
    queue_top = q;

  } else q_prev100 = queue = queue_top = q;

  queued_paths++;
  pending_not_fuzzed++;

  cycles_wo_finds = 0;

  /* Set next_100 pointer for every 100th element (index 0, 100, etc) to
allow faster iteration. */
  if ((queued_paths - 1) % 100 == 0 && queued_paths > 1) {

    q_prev100->next_100 = q;
    q_prev100 = q;

  }
```

```
    last_path_time = get_cur_time();
  }
```

`afl-fuzz` 维护一个 `queue_entry` 的链表，该链表用来保存测试样例，每次调用 `add_to_queue` 都会将新样例储存到链表头部。

另外还有一个 `q_prev100` 也是 `queue_entry` 的链表，但它每 100 个测试样例保存一次。

load_auto

尝试在输入目录下寻找自动生成的字典文件，调用 `maybe_add_auto` 将相应的字典加入到全局变量 `a_extras` 中，用于后续字典模式的变异当中。

pivot_inputs

在输出文件夹中创建与输入样例间的硬链接，称之为 `orignal` 。

load_extras

如果指定了 `-x` 参数（字典模式），加载对应的字典到全局变量`extras`当中，用于后续字典模式的变异当中。

find_timeout

如果指定了 `resuming_fuzz` ，即从输出目录当中恢复模糊测试状态，会从之前的模糊测试状态 `fuzzer_stats` 文件中计算中 `timeout` 值，保存在 `exec_tmout` 中。

detect_file_args

检测输入的命令行中是否包含`@@`参数，如果包含的话需要将 `@@` 替换成目录文件 `"%s/.cur_input"`, `out_dir` ，使得模糊测试目标程序的命令完整；同时将目录文件 `"%s/.cur_input"` 路径保存在 `out_file` 当中，后续变异的内容保存在该文件路径中，用于运行测试目标文件。

setup_stdio_file

如果目标程序的输入不是来源于文件而是来源于标准输入的话，则将目录文件 `"%s/.cur_input"` 文件打开保存在 `out_fd` 文件句柄中，后续将标准输入重定向到该文件中；结合 `detect_file_args` 函数实现了将变异的内容保存在 `"%s/.cur_input"` 文件中，运行目标测试文件并进行模糊测试。

check_binary

检查二进制文件是否合法。

perform_dry_run

将每个测试样例作为输入去运行目标程序，检查程序是否能够正常工作：

```
static void perform_dry_run(char** argv) {

  struct queue_entry* q = queue;
  u32 cal_failures = 0;
  u8* skip_crashes = getenv("AFL_SKIP_CRASHES");

  while (q) {

    u8* use_mem;
    u8  res;
    s32 fd;

    u8* fn = strrchr(q->fname, '/') + 1;

    ACTF("Attempting dry run with '%s'...", fn);

    fd = open(q->fname, O_RDONLY);
    if (fd < 0) PFATAL("Unable to open '%s'", q->fname);

    use_mem = ck_alloc_nozero(q->len);

    if (read(fd, use_mem, q->len) != q->len)
      FATAL("Short read from '%s'", q->fname);

    close(fd);

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

    if (stop_soon) return;

    if (res == crash_mode || res == FAULT_NOBITS)
      SAYF(cGRA "    len = %u, map size = %u, exec speed = %llu us\n" cRST,
           q->len, q->bitmap_size, q->exec_us);

        ......

    if (q->var_behavior) WARNF("Instrumentation output varies across
runs.");

    q = q->next;
  }

  if (cal_failures) {

    if (cal_failures == queued_paths)
      FATAL("All test cases time out%s, giving up!",
            skip_crashes ? " or crash" : "");

    WARNF("Skipped %u test cases (%0.02f%%) due to timeouts%s.",
cal_failures,
          ((double)cal_failures) * 100 / queued_paths,
          skip_crashes ? " or crashes" : "");

    if (cal_failures * 5 > queued_paths)
      WARNF(cLRD "High percentage of rejected test cases, check settings!");

  }
  OKF("All test cases processed.");
}
```

对每个测试样例使用 `calibrate_case` 进行测试，并返回运行结果，然后处理其中异常的情况，比如说程序崩溃或运行超时等。

calibrate_case

```
static u8 calibrate_case(char** argv, struct queue_entry* q, u8* use_mem,
                         u32 handicap, u8 from_queue) {

  static u8 first_trace[MAP_SIZE];

  u8  fault = 0, new_bits = 0, var_detected = 0, hnb = 0,
      first_run = (q->exec_cksum == 0);

  u64 start_us, stop_us;

  s32 old_sc = stage_cur, old_sm = stage_max;
  u32 use_tmout = exec_tmout;
  u8* old_sn = stage_name;

  /* Be a bit more generous about timeouts when resuming sessions, or when
     trying to calibrate already-added finds. This helps avoid trouble due
     to intermittent latency. */

  if (!from_queue || resuming_fuzz)
```

```
    use_tmout = MAX(exec_tmout + CAL_TMOUT_ADD,
                    exec_tmout * CAL_TMOUT_PERC / 100);

  q->cal_failed++;

  stage_name = "calibration";
  stage_max  = fast_cal ? 3 : CAL_CYCLES;

  /* Make sure the forkserver is up before we do anything, and let's not
     count its spin-up time toward binary calibration. */

  if (dumb_mode != 1 && !no_forkserver && !forksrv_pid)
    init_forkserver(argv);

  if (q->exec_cksum) {

    memcpy(first_trace, trace_bits, MAP_SIZE);
    hnb = has_new_bits(virgin_bits);
    if (hnb > new_bits) new_bits = hnb;

  }

  start_us = get_cur_time_us();

  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {

    u32 cksum;

    if (!first_run && !(stage_cur % stats_update_freq)) show_stats();

    write_to_testcase(use_mem, q->len);

    fault = run_target(argv, use_tmout);

    /* stop_soon is set by the handler for Ctrl+C. When it's pressed,
       we want to bail out quickly. */

    if (stop_soon || fault != crash_mode) goto abort_calibration;

    if (!dumb_mode && !stage_cur && !count_bytes(trace_bits)) {
      fault = FAULT_NOINST;
      goto abort_calibration;
    }

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

```
      if (q->exec_cksum != cksum) {

        hnb = has_new_bits(virgin_bits);
        if (hnb > new_bits) new_bits = hnb;

        if (q->exec_cksum) {

          u32 i;

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

            if (!var_bytes[i] && first_trace[i] != trace_bits[i]) {

              var_bytes[i] = 1;
              stage_max    = CAL_CYCLES_LONG;

            }

          }

          var_detected = 1;

        } else {

          q->exec_cksum = cksum;
          memcpy(first_trace, trace_bits, MAP_SIZE);

        }

      }

    }

    stop_us = get_cur_time_us();

    total_cal_us     += stop_us - start_us;
    total_cal_cycles += stage_max;

    /* OK, let's collect some stats about the performance of this test case.
       This is used for fuzzing air time calculations in calculate_score(). */

    q->exec_us     = (stop_us - start_us) / stage_max;
    q->bitmap_size = count_bytes(trace_bits);
    q->handicap    = handicap;
    q->cal_failed  = 0;
```

```
  total_bitmap_size += q->bitmap_size;
  total_bitmap_entries++;

  update_bitmap_score(q);

  /* If this case didn't result in new output from the instrumentation, tell
     parent. This is a non-critical problem, but something to warn the user
     about. */

  if (!dumb_mode && first_run && !fault && !new_bits) fault = FAULT_NOBITS;

abort_calibration:

  if (new_bits == 2 && !q->has_new_cov) {
    q->has_new_cov = 1;
    queued_with_cov++;
  }

  /* Mark variable paths. */

  if (var_detected) {

    var_byte_count = count_bytes(var_bytes);

    if (!q->var_behavior) {
      mark_as_variable(q);
      queued_variable++;
    }

  }

  stage_name = old_sn;
  stage_cur  = old_sc;
  stage_max  = old_sm;

  if (!first_run) show_stats();

  return fault;

}
```

该函数用以对样例进行测试，在后续的测试过程中也会反复调用。此处，其主要的工作是：

- 判断样例是否是首次运行，记录在 `first_run`
- 设置超时阈值 `use_tmout`

- 调用 `init_forkserver` 初始化 `fork server`
- 多次运行测试样例，记录数据

init_forkserver

fork server 是 AFL 中一个重要的机制。

`afl-fuzz` 主动建立一个子进程为 `fork server`，而模糊测试则是通过 `fork server` 调用 fork 建立子进程来进行测试。

> 参考在源代码注释中的这篇文章可以有更加深入的理解：https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

之所以需要设计它，笔者在这里给出一个比较概括的理由：

一般来说，如果我们想要测试输入样例，就需要用 `fork+execve` 去执行相关的二进制程序，但是执行程序是需要加载代码、动态库、符号解析等各种耗时的行为，这会让 AFL 不够效率。

但是这个过程其实是存在浪费的，可以注意到，如果我们要对相同的二进制程序进行多次不同的输入样本进行测试，那按照原本的操作，我们应该多次执行 `fork+execve` ，而浪费就出现在这，因为我们明明已经加载好了一切，却又要因此重复加载释放。

因此 `fork server` 的设计主要就是为了解决这个浪费。它通过向代码中进行插桩的方式，使得在二进制程序中去建立一个 `fork server` （对，它实际上是由目标程序去建立的），然后这个 `fork server` 会在完成一切初始化后，停止在某一个地方（往往设定在 `main` 函数）等待 fuzzer 去喊开始执行。

一旦 fuzzer 喊了开始，就会由这个 `fork server` 去调用 `fork` 然后往下执行。而我们知道，`fork` 由于写时复制的机制存在，它其实并没有过多的开销，可以完全继承原有的所有上下文信息，从而避开了多次 `execve` 的加载开销。

摘抄一段这部分插桩的内容：

```
__afl_forkserver:

  /* Phone home and tell the parent that we're OK. */

  pushl $4          /* length    */
  pushl $__afl_temp /* data      */
  pushl $199        /* file desc */
  call  write
  addl  $12, %esp
```

```
__afl_fork_wait_loop:

  /* Wait for parent by reading from the pipe. This will block until
     the parent sends us something. Abort if read fails. */

  pushl $4          /* length    */
  pushl $__afl_temp /* data      */
  pushl $198        /* file desc */
  call  read
  addl  $12, %esp

  cmpl  $4, %eax
  jne   __afl_die

  /* Once woken up, create a clone of our process. */

  call fork

  cmpl $0, %eax
  jl   __afl_die
  je   __afl_fork_resume

  /* In parent process: write PID to pipe, then wait for child.
     Parent will handle timeouts and SIGKILL the child as needed. */

  movl  %eax, __afl_fork_pid

  pushl $4              /* length    */
  pushl $__afl_fork_pid /* data      */
  pushl $199            /* file desc */
  call  write
  addl  $12, %esp

  pushl $2             /* WUNTRACED */
  pushl $__afl_temp    /* status    */
  pushl __afl_fork_pid /* PID       */
  call  waitpid
  addl  $12, %esp

  cmpl  $0, %eax
  jle   __afl_die

  /* Relay wait status to pipe, then loop back. */

  pushl $4          /* length    */
  pushl $__afl_temp /* data      */
  pushl $199          /* file desc */
  call  write
  addl  $12, %esp

  jmp __afl_fork_wait_loop

__afl_fork_resume:

  /* In child process: close fds, resume execution. */

  pushl $198
  call  close

  pushl $199
  call  close

  addl  $8, %esp
  ret
```

`fork server` 主要是通过管道和 afl-fuzz 中的 `fork server` 进行通信的，但他们其实不做过多的事情，往往只是通知一下程序运行的状态。因为真正的反馈信息，包括路径的发现等这部分功能是通过共享内存去实现的，它们不需要用 `fork server` 这种效率较低的方案去记录数据。

剩下的就是关闭一些不需要的文件或管道了，代码姑且贴在这里，以备未来有需要时可以现查：

```
EXP_ST void init_forkserver(char** argv) {

  static struct itimerval it;
  int st_pipe[2], ctl_pipe[2];
  int status;
  s32 rlen;

  ACTF("Spinning up the fork server...");

  if (pipe(st_pipe) || pipe(ctl_pipe)) PFATAL("pipe() failed");

  forksrv_pid = fork();

  if (forksrv_pid < 0) PFATAL("fork() failed");

  if (!forksrv_pid) {

    struct rlimit r;

    /* Umpf. On OpenBSD, the default fd limit for root users is set to
```

```
       soft 128. Let's try to fix that... */

    if (!getrlimit(RLIMIT_NOFILE, &r) && r.rlim_cur < FORKSRV_FD + 2) {

      r.rlim_cur = FORKSRV_FD + 2;
      setrlimit(RLIMIT_NOFILE, &r); /* Ignore errors */

    }

    if (mem_limit) {

      r.rlim_max = r.rlim_cur = ((rlim_t)mem_limit) << 20;

#ifdef RLIMIT_AS

      setrlimit(RLIMIT_AS, &r); /* Ignore errors */

#else

      /* This takes care of OpenBSD, which doesn't have RLIMIT_AS, but
         according to reliable sources, RLIMIT_DATA covers anonymous
         maps - so we should be getting good protection against OOM bugs. */

      setrlimit(RLIMIT_DATA, &r); /* Ignore errors */

#endif /* ^RLIMIT_AS */


    }

    /* Dumping cores is slow and can lead to anomalies if SIGKILL is
delivered
       before the dump is complete. */

    r.rlim_max = r.rlim_cur = 0;

    setrlimit(RLIMIT_CORE, &r); /* Ignore errors */

    /* Isolate the process and configure standard descriptors. If out_file
is
       specified, stdin is /dev/null; otherwise, out_fd is cloned instead.
*/

    setsid();

    dup2(dev_null_fd, 1);
    dup2(dev_null_fd, 2);

    if (out_file) {

      dup2(dev_null_fd, 0);

    } else {

      dup2(out_fd, 0);
      close(out_fd);

    }

    /* Set up control and status pipes, close the unneeded original fds. */

    if (dup2(ctl_pipe[0], FORKSRV_FD) < 0) PFATAL("dup2() failed");
    if (dup2(st_pipe[1], FORKSRV_FD + 1) < 0) PFATAL("dup2() failed");

    close(ctl_pipe[0]);
    close(ctl_pipe[1]);
    close(st_pipe[0]);
    close(st_pipe[1]);

    close(out_dir_fd);
    close(dev_null_fd);
    close(dev_urandom_fd);
    close(fileno(plot_file));

    /* This should improve performance a bit, since it stops the linker from
       doing extra work post-fork(). */

    if (!getenv("LD_BIND_LAZY")) setenv("LD_BIND_NOW", "1", 0);

    /* Set sane defaults for ASAN if nothing else specified. */

    setenv("ASAN_OPTIONS", "abort_on_error=1:"
                           "detect_leaks=0:"
                           "symbolize=0:"
                           "allocator_may_return_null=1", 0);

    /* MSAN is tricky, because it doesn't support abort_on_error=1 at this
       point. So, we do this in a very hacky way. */

    setenv("MSAN_OPTIONS", "exit_code=" STRINGIFY(MSAN_ERROR) ":"
                           "symbolize=0:"
                           "abort_on_error=1:"
```

```
                             "allocator_may_return_null=1:"
                             "msan_track_origins=0", 0);

    execv(target_path, argv);

    /* Use a distinctive bitmap signature to tell the parent about execv()
       falling through. */

    *(u32*)trace_bits = EXEC_FAIL_SIG;
    exit(0);

  }

  /* Close the unneeded endpoints. */

  close(ctl_pipe[0]);
  close(st_pipe[1]);

  fsrv_ctl_fd = ctl_pipe[1];
  fsrv_st_fd  = st_pipe[0];

  /* Wait for the fork server to come up, but don't wait too long. */

  it.it_value.tv_sec = ((exec_tmout * FORK_WAIT_MULT) / 1000);
  it.it_value.tv_usec = ((exec_tmout * FORK_WAIT_MULT) % 1000) * 1000;

  setitimer(ITIMER_REAL, &it, NULL);

  rlen = read(fsrv_st_fd, &status, 4);

  it.it_value.tv_sec = 0;
  it.it_value.tv_usec = 0;

  setitimer(ITIMER_REAL, &it, NULL);

  /* If we have a four-byte "hello" message from the server, we're all set.
     Otherwise, try to figure out what went wrong. */

  if (rlen == 4) {
    OKF("All right - fork server is up.");
    return;
  }

  if (child_timed_out)
    FATAL("Timeout while initializing fork server (adjusting -t may help)");
  if (waitpid(forksrv_pid, &status, 0) <= 0)
    PFATAL("waitpid() failed");

  if (WIFSIGNALED(status)) {

    if (mem_limit && mem_limit < 500 && uses_asan) {

      SAYF("\n" cLRD "[-] " cRST
           "Whoops, the target binary crashed suddenly, before receiving any
input\n"
           "    from the fuzzer! Since it seems to be built with ASAN and
you have a\n"
           "    restrictive memory limit configured, this is expected;
please read\n"
           "    %s/notes_for_asan.txt for help.\n", doc_path);

    } else if (!mem_limit) {

      SAYF("\n" cLRD "[-] " cRST
           "Whoops, the target binary crashed suddenly, before receiving any
input\n"
           "    from the fuzzer! There are several probable
explanations:\n\n"

           "    - The binary is just buggy and explodes entirely on its own.
If so, you\n"
           "      need to fix the underlying problem or find a better
replacement.\n\n"

#ifdef __APPLE__

           "    - On MacOS X, the semantics of fork() syscalls are non-
standard and may\n"
           "      break afl-fuzz performance optimizations when running
platform-specific\n"
           "      targets. To fix this, set AFL_NO_FORKSRV=1 in the
environment.\n\n"

#endif /* __APPLE__ */

           "    - Less likely, there is a horrible bug in the fuzzer. If
other options\n"
           "      fail, poke <lcamtuf@coredump.cx> for troubleshooting
tips.\n");

    } else {
      SAYF("\n" cLRD "[-] " cRST
           "Whoops, the target binary crashed suddenly, before receiving any
input\n"
           "    from the fuzzer! There are several probable
explanations:\n\n"

           "    - The current memory limit (%s) is too restrictive, causing
the\n"
           "      target to hit an OOM condition in the dynamic linker. Try
bumping up\n"
           "      the limit with the -m setting in the command line. A
simple way confirm\n"
           "      this diagnosis would be:\n\n"

#ifdef RLIMIT_AS
           "      ( ulimit -Sv $[%llu << 10]; /path/to/fuzzed_app )\n\n"
#else
           "      ( ulimit -Sd $[%llu << 10]; /path/to/fuzzed_app )\n\n"
#endif /* ^RLIMIT_AS */

           "      Tip: you can use http://jwilk.net/software/recidivm to
quickly\n"
           "      estimate the required amount of virtual memory for the
binary.\n\n"

           "    - The binary is just buggy and explodes entirely on its own.
If so, you\n"
           "      need to fix the underlying problem or find a better
replacement.\n\n"

#ifdef __APPLE__

           "    - On MacOS X, the semantics of fork() syscalls are non-
standard and may\n"
           "      break afl-fuzz performance optimizations when running
platform-specific\n"
           "      targets. To fix this, set AFL_NO_FORKSRV=1 in the
environment.\n\n"

#endif /* __APPLE__ */

           "    - Less likely, there is a horrible bug in the fuzzer. If
other options\n"
           "      fail, poke <lcamtuf@coredump.cx> for troubleshooting
tips.\n",
           DMS(mem_limit << 20), mem_limit - 1);

    }

    FATAL("Fork server crashed with signal %d", WTERMSIG(status));

  }

  if (*(u32*)trace_bits == EXEC_FAIL_SIG)
    FATAL("Unable to execute target application ('%s')", argv[0]);

  if (mem_limit && mem_limit < 500 && uses_asan) {

    SAYF("\n" cLRD "[-] " cRST
           "Hmm, looks like the target binary terminated before we could
complete a\n"
           "    handshake with the injected code. Since it seems to be built
with ASAN and\n"
           "    you have a restrictive memory limit configured, this is
expected; please\n"
           "    read %s/notes_for_asan.txt for help.\n", doc_path);

  } else if (!mem_limit) {

    SAYF("\n" cLRD "[-] " cRST
         "Hmm, looks like the target binary terminated before we could
complete a\n"
         "    handshake with the injected code. Perhaps there is a horrible
bug in the\n"
         "    fuzzer. Poke <lcamtuf@coredump.cx> for troubleshooting
tips.\n");

  } else {

    SAYF("\n" cLRD "[-] " cRST
         "Hmm, looks like the target binary terminated before we could
complete a\n"
         "    handshake with the injected code. There are %s probable
explanations:\n\n"

         "%s"
         "    - The current memory limit (%s) is too restrictive, causing an
OOM\n"
         "      fault in the dynamic linker. This can be fixed with the -m
option. A\n"
         "      simple way to confirm the diagnosis may be:\n\n"
```

```
#ifdef RLIMIT_AS
         "      ( ulimit -Sv $[%llu << 10]; /path/to/fuzzed_app )\n\n"
#else
         "      ( ulimit -Sd $[%llu << 10]; /path/to/fuzzed_app )\n\n"
#endif /* ^RLIMIT_AS */

         "      Tip: you can use http://jwilk.net/software/recidivm to
quickly\n"
         "      estimate the required amount of virtual memory for the
binary.\n\n"

         "    - Less likely, there is a horrible bug in the fuzzer. If other
options\n"
         "      fail, poke <lcamtuf@coredump.cx> for troubleshooting
tips.\n",
         getenv(DEFER_ENV_VAR) ? "three" : "two",
         getenv(DEFER_ENV_VAR) ?
         "    - You are using deferred forkserver, but __AFL_INIT() is
never\n"
         "      reached before the program terminates.\n\n" : "",
         DMS(mem_limit << 20), mem_limit - 1);

  }

  FATAL("Fork server handshake failed");

}
```

cull_queue

将运行过的种子根据运行的效果进行排序，后续模糊测试根据排序的结果来挑选样例进行模糊测试。

show_init_stats

初始化 `UI` 。

find_start_position

如果是恢复运行，则调用该函数来寻找到对应的样例的位置。

write_stats_file

更新统计信息文件以进行无人值守的监视。

save_auto

保存自动提取的 `token` ，用于后续字典模式的 `fuzz` 。

afl-fuzz 主循环

- 首先调用 `cull_queue` 来优化队列
- 如果 `queue_cur` 为空，代表所有queue都被执行完一轮
  - 设置queue_cycle计数器加一，即代表所有queue被完整执行了多少轮。
  - 设置current_entry为0，和queue_cur为queue首元素，开始新一轮fuzz。
  - 如果是resume fuzz情况，则先检查seek_to是否为空，如果不为空，就从seek_to指定的queue项开始执行。
  - 刷新展示界面`show_stats`
  - 如果在一轮执行之后的queue里的case数，和执行之前一样，代表在完整的一轮执行里都没有发现任何一个新的case
    - 如果use_splicing为1，就设置cycles_wo_finds计数器加1
    - 否则，设置use_splicing为1，代表我们接下来要通过splice重组queue里的case。
- 执行`skipped_fuzz = fuzz_one(use_argv)`来对queue_cur进行一次测试
  - 注意fuzz_one并不一定真的执行当前queue_cur，它是有一定策略的，如果不执行，就直接返回1，否则返回0
- 如果skipped_fuzz为0，且存在sync_id
  - sync_interval_cnt计数器加一，如果其结果是SYNC_INTERVAL(默认是5)的倍数，就进行一次sync
- `queue_cur = queue_cur->next;current_entry++;`，开始测试下一个queue

```
while (1) {

  u8 skipped_fuzz;

  cull_queue();

  if (!queue_cur) {

    queue_cycle++;
    current_entry     = 0;
    cur_skipped_paths = 0;
    queue_cur         = queue;

    while (seek_to) {
      current_entry++;
      seek_to--;
      queue_cur = queue_cur->next;
    }

    show_stats();

    if (not_on_tty) {
      ACTF("Entering queue cycle %llu.", queue_cycle);
      fflush(stdout);
    }

    /* If we had a full queue cycle with no new finds, try
       recombination strategies next. */

    if (queued_paths == prev_queued) {

      if (use_splicing) cycles_wo_finds++; else use_splicing = 1;

    } else cycles_wo_finds = 0;

    prev_queued = queued_paths;

    if (sync_id && queue_cycle == 1 && getenv("AFL_IMPORT_FIRST"))
      sync_fuzzers(use_argv);

  }

  skipped_fuzz = fuzz_one(use_argv);

  if (!stop_soon && sync_id && !skipped_fuzz) {

    if (!(sync_interval_cnt++ % SYNC_INTERVAL))
      sync_fuzzers(use_argv);

  }

  if (!stop_soon && exit_1) stop_soon = 2;

  if (stop_soon) break;

  queue_cur = queue_cur->next;
  current_entry++;

}
```

fuzz_one

从测试样例的队列中取出 `current_entry` 进行测试，成功则返回 0 ，否则返回 1。这里主要是对该函数主要内容进行记录，不做细节的代码分析。

- 打开 `queue_cur` 并映射到 `orig_in` 和 `in_buf`
- 分配len大小的内存，并初始化为全 0，然后将地址赋值给 `out_buf`

CALIBRATION 阶段

- 若 `queue_cur->cal_failed < CAL_CHANCES` 且 `queue_cur->cal_failed >0` ，则调用 `calibrate_case`

TRIMMING 阶段

- 如果样例没经过该阶段，那么就调用 `trim_case` 修剪样例
- 将修剪后的结果重新放入 `out_buf`

缩减的思路是这样的：如果对一个样本进行缩减后，它所覆盖的路径并未发生变化，那么就说明缩减的这部分内容是可有可无的，因此可以删除。

具体策略如下：

- 如果这个case的大小len小于5字节，就直接返回
- 设定 `stage_name` 为 `tmp` ，该变量仅用来标识本次缩减所使用的策略
- 计算 `len_p2` ，其值是大于等于 `q->len` 的第一个2的幂次。
- 取 `len_p2/16` 为 `remove_len` 作为起始步长。
- 进入循环，终止条件为 `remove_len` 小于终止步长 `len_p2/1024` , 每轮循环步长会除2。
    - 初始化一些必要数据后，再次进入循环，这次是按照当前设定的步长对样本进行遍历
    - 用 `run_target` 运行样例， `trim_execs` 计数器加一
    - 对比路径是否变化
        - 若无变化
            - 则从 `q->len` 中减去 `remove_len` 个字节，并由此重新计算出一个 `len_p2` ，这里注意一下 `while (remove_len >= MAX(len_p2 / TRIM_END_STEPS, TRIM_MIN_BYTES))`
            - 将 `in_buf+remove_pos+remove_len` 到最后的字节，前移到 `in_buf+remove_pos` 处，等于删除了 `remove_pos` 向后的 `remove_len` 个字节。
            - 如果 `needs_write` 为 0，则设置其为 1，并保存当前 `trace_bits` 到 `clean_trace` 中。
        - 如有变化
            - `remove_pos` 加上 `remove_len`
- 如果needs_write为1

- 删除原来的 `q->fname` ，创建一个新的 `q->fname` ，将 `in_buf` 里的内容写入，然后用 `clean_trace` 恢复 `trace_bits` 的值。
- 进行一次 `update_bitmap_score`

```
static u8 trim_case(char** argv, struct queue_entry* q, u8* in_buf) {

  static u8 tmp[64];
  static u8 clean_trace[MAP_SIZE];

  u8  needs_write = 0, fault = 0;
  u32 trim_exec = 0;
  u32 remove_len;
  u32 len_p2;

  /* Although the trimmer will be less useful when variable behavior is
     detected, it will still work to some extent, so we don't check for
     this. */

  if (q->len < 5) return 0;

  stage_name = tmp;
  bytes_trim_in += q->len;

  /* Select initial chunk len, starting with large steps. */

  len_p2 = next_p2(q->len);

  remove_len = MAX(len_p2 / TRIM_START_STEPS, TRIM_MIN_BYTES);

  /* Continue until the number of steps gets too high or the stepover
     gets too small. */

  while (remove_len >= MAX(len_p2 / TRIM_END_STEPS, TRIM_MIN_BYTES)) {

    u32 remove_pos = remove_len;

    sprintf(tmp, "trim %s/%s", DI(remove_len), DI(remove_len));

    stage_cur = 0;
    stage_max = q->len / remove_len;

    while (remove_pos < q->len) {

      u32 trim_avail = MIN(remove_len, q->len - remove_pos);
      u32 cksum;
      write_with_gap(in_buf, q->len, remove_pos, trim_avail);

      fault = run_target(argv, exec_tmout);
      trim_execs++;

      if (stop_soon || fault == FAULT_ERROR) goto abort_trimming;

      /* Note that we don't keep track of crashes or hangs here; maybe TODO?
*/

      cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

      /* If the deletion had no impact on the trace, make it permanent. This
         isn't perfect for variable-path inputs, but we're just making a
         best-effort pass, so it's not a big deal if we end up with false
         negatives every now and then. */

      if (cksum == q->exec_cksum) {

        u32 move_tail = q->len - remove_pos - trim_avail;

        q->len -= trim_avail;
        len_p2  = next_p2(q->len);

        memmove(in_buf + remove_pos, in_buf + remove_pos + trim_avail,
                move_tail);

        /* Let's save a clean trace, which will be needed by
           update_bitmap_score once we're done with the trimming stuff. */

        if (!needs_write) {

          needs_write = 1;
          memcpy(clean_trace, trace_bits, MAP_SIZE);

        }

      } else remove_pos += remove_len;

      /* Since this can be slow, update the screen every now and then. */

      if (!(trim_exec++ % stats_update_freq)) show_stats();
      stage_cur++;

    }
```

```
      remove_len >>= 1;

    }

    /* If we have made changes to in_buf, we also need to update the on-disk
       version of the test case. */

    if (needs_write) {

      s32 fd;

      unlink(q->fname); /* ignore errors */

      fd = open(q->fname, O_WRONLY | O_CREAT | O_EXCL, 0600);

      if (fd < 0) PFATAL("Unable to create '%s'", q->fname);

      ck_write(fd, in_buf, q->len, q->fname);
      close(fd);

      memcpy(trace_bits, clean_trace, MAP_SIZE);
      update_bitmap_score(q);

    }

  abort_trimming:

    bytes_trim_out += q->len;
    return fault;

  }
```

PERFORMANCE SCORE 阶段

- perf_score = `calculate_score(queue_cur)`
- 如果 `skip_deterministic` 为1，或者 `queue_cur` 被 fuzz 过，或者 `queue_cur` 的 `passed_det` 为1，则跳转去 `havoc_stage` 阶段
- 设置doing_det为 1

SIMPLE BITFLIP 阶段

这个阶段读起来感觉比较抽象。首先定义了这么一个宏：

```
#define FLIP_BIT(_ar, _b) do { \
    u8* _arf = (u8*)(_ar); \
    u32 _bf = (_b); \
    _arf[(_bf) >> 3] ^= (128 >> ((_bf) & 7)); \
  } while (0)
```

这个宏的操作是对一个 bit 进行反转。

而接下来首先有一个循环：

```
  stage_short = "flip1";
  stage_max   = len << 3;
  stage_name  = "bitflip 1/1";

  stage_val_type = STAGE_VAL_NONE;

  orig_hit_cnt = queued_paths + unique_crashes;

  prev_cksum = queue_cur->exec_cksum;
  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {
    stage_cur_byte = stage_cur >> 3;
    FLIP_BIT(out_buf, stage_cur);
    if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
    FLIP_BIT(out_buf, stage_cur);
    ......
```

`stage_max` 是输入的总 bit 数，然后分别对每个 bit 进行翻转后用 `common_fuzz_stuff` 进行测试，然后再将其翻转回来。

而如果对某个字节的最后一个 bit 翻转后测试，发现路径并未增加，就能够将其认为是一个 `token` 。

- token默认最小是3，最大是32,每次发现新token时，通过`maybe_add_auto`添加到`a_extras`数组里。
- `stage_finds[STAGE_FLIP1]`的值加上在整个FLIP_BIT中新发现的路径和Crash总和
- `stage_cycles[STAGE_FLIP1]`的值加上在整个FLIP_BIT中执行的target次数`stage_max`
- 设置stage_name为`bitflip 2/1`,原理和之前一样，只是这次是连续翻转相邻的两位。

然后在后面的一个循环中又做类似的事，但每次会翻转两个 bit：

```
  stage_name  = "bitflip 2/1";
  stage_short = "flip2";
  stage_max   = (len << 3) - 1;
  orig_hit_cnt = new_hit_cnt;

  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {

    stage_cur_byte = stage_cur >> 3;

    FLIP_BIT(out_buf, stage_cur);
    FLIP_BIT(out_buf, stage_cur + 1);

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

    FLIP_BIT(out_buf, stage_cur);
    FLIP_BIT(out_buf, stage_cur + 1);

  }
```

- 然后保存结果到`stage_finds[STAGE_FLIP2]和stage_cycles[STAGE_FLIP2]`里。
- 同理，设置stage_name为`bitflip 4/1`，翻转连续的四位并记录。
- 构建 `Effector map`
  - 进入 `bitflip 8/8` 的阶段，这个阶段就是对每个字节的所有 bit 都进行翻转，然后用 `common_fuzz_stuff` 进行测试
  - 如果其造成执行路径与原始路径不一致，就将该byte在 `effector map` 中标记为1，即“有效”的，否则标记为 0，即“无效”的。
  - 这样做的逻辑是：如果一个byte完全翻转，都无法带来执行路径的变化，那么这个byte很有可能是属于”data”，而非”metadata”（例如size, flag等），对整个fuzzing的意义不大。所以，在随后的一些变异中，会参考effector map，跳过那些“无效”的byte，从而节省了执行资源。

然后进入 `bitflip 16/8` 部分，按对每两个字节进行一次翻转然后测试：

```
  for (i = 0; i < len - 1; i++) {

    /* Let's consult the effector map... */

    if (!eff_map[EFF_APOS(i)] && !eff_map[EFF_APOS(i + 1)]) {
      stage_max--;
      continue;
    }

    stage_cur_byte = i;

    *(u16*)(out_buf + i) ^= 0xFFFF;

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

    *(u16*)(out_buf + i) ^= 0xFFFF;

  }
```

- 这里要注意在翻转之前会先检查eff_map里对应于这两个字节的标志是否为0，如果为0，则这两个字节是无效的数据，stage_max减一，然后开始变异下一个字。
- common_fuzz_stuff执行变异后的结果，然后还原。

最后是 `bitflip 32/8` 阶段，每 4 个字节进行翻转然后测试：

```
  stage_name  = "bitflip 32/8";
  stage_short = "flip32";
  stage_cur   = 0;
  stage_max   = len - 3;

  orig_hit_cnt = new_hit_cnt;

  for (i = 0; i < len - 3; i++) {

    /* Let's consult the effector map... */
    if (!eff_map[EFF_APOS(i)] && !eff_map[EFF_APOS(i + 1)] &&
        !eff_map[EFF_APOS(i + 2)] && !eff_map[EFF_APOS(i + 3)]) {
      stage_max--;
      continue;
    }

    stage_cur_byte = i;

    *(u32*)(out_buf + i) ^= 0xFFFFFFFF;

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

    *(u32*)(out_buf + i) ^= 0xFFFFFFFF;
  }
```

- 在每次翻转之前会检查eff_map里对应于这四个字节的标志是否为0，如果是0，则这两个字节是无效的数据，stage_max减一，然后开始变异下一组双字。

ARITHMETIC INC/DEC 阶段

- arith 8/8，每次对8个bit进行加减运算，按照每8个 bit 的步长从头开始，即对文件的每个 byte 进行整数加减变异
- arith 16/8，每次对16个bit进行加减运算，按照每8个bit的步长从头开始，即对文件的每个 word进行整数加减变异
- arith 32/8，每次对32个bit进行加减运算，按照每8个bit的步长从头开始，即对文件的每个 dword进行整数加减变异
- 加减变异的上限，在 `config.h` 中的宏 `ARITH_MAX` 定义，默认为 35。所以，对目标整数会进行+1, +2, ..., +35, -1, -2, ..., -35 的变异。特别地，由于整数存在大端序和小端序两种表示方式，AFL会贴心地对这两种整数表示方式都进行变异。
- 此外，AFL 还会智能地跳过某些 `arithmetic` 变异。第一种情况就是前面提到的 effector map： 如果一个整数的所有 bytes 都被判断为“无效”，那么就跳过对整数的变异。第二种情况是之前 bitflip 已经生成过的变异：如果加/减某个数后，其效果与之前的某种bitflip相同，那么这次变异肯定在上一个阶段已经执行过了，此次便不会再执行。

此处展示 arith 8/8 部分代码：

```
  stage_name  = "arith 8/8";
  stage_short = "arith8";
  stage_cur   = 0;
  stage_max   = 2 * len * ARITH_MAX;

  stage_val_type = STAGE_VAL_LE;

  orig_hit_cnt = new_hit_cnt;

  for (i = 0; i < len; i++) {

    u8 orig = out_buf[i];

    /* Let's consult the effector map... */

    if (!eff_map[EFF_APOS(i)]) {
      stage_max -= 2 * ARITH_MAX;
      continue;
    }

    stage_cur_byte = i;

    for (j = 1; j <= ARITH_MAX; j++) {

      u8 r = orig ^ (orig + j);

      /* Do arithmetic operations only if the result couldn't be a product
         of a bitflip. */
```

```
        if (!could_be_bitflip(r)) {

          stage_cur_val = j;
          out_buf[i] = orig + j;

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

        } else stage_max--;

        r =  orig ^ (orig - j);

        if (!could_be_bitflip(r)) {

          stage_cur_val = -j;
          out_buf[i] = orig - j;

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

        } else stage_max--;

        out_buf[i] = orig;

      }

    }

    new_hit_cnt = queued_paths + unique_crashes;

    stage_finds[STAGE_ARITH8]  += new_hit_cnt - orig_hit_cnt;
    stage_cycles[STAGE_ARITH8] += stage_max;
```

INTERESTING VALUES 阶段

- interest 8/8，每次对8个bit进替换，按照每8个bit的步长从头开始，即对文件的每个byte进行替换
- interest 16/8，每次对16个bit进替换，按照每8个bit的步长从头开始，即对文件的每个word进行替换
- interest 32/8，每次对32个bit进替换，按照每8个bit的步长从头开始，即对文件的每个dword进行替换
- 而用于替换的 `interesting values` 是AFL预设的一些比较特殊的数,这些数的定义在config.h文件中：

```
static s8  interesting_8[]  = { INTERESTING_8 };
static s16 interesting_16[] = { INTERESTING_8, INTERESTING_16 };
static s32 interesting_32[] = { INTERESTING_8, INTERESTING_16,
INTERESTING_32 };
```

- 同样，`effector map` 仍然会用于判断是否需要变异；此外，如果某个 `interesting value`，是可以通过 `bitflip` 或者 `arithmetic` 变异达到，那么这样的重复性变异也是会跳过的。

此处给出 `interest 8/8` 部分代码：

```
  stage_name  = "interest 8/8";
  stage_short = "int8";
  stage_cur   = 0;
  stage_max   = len * sizeof(interesting_8);

  stage_val_type = STAGE_VAL_LE;

  orig_hit_cnt = new_hit_cnt;

  /* Setting 8-bit integers. */

  for (i = 0; i < len; i++) {

    u8 orig = out_buf[i];

    /* Let's consult the effector map... */

    if (!eff_map[EFF_APOS(i)]) {
      stage_max -= sizeof(interesting_8);
      continue;
    }

    stage_cur_byte = i;

    for (j = 0; j < sizeof(interesting_8); j++) {

      /* Skip if the value could be a product of bitflips or arithmetics. */

      if (could_be_bitflip(orig ^ (u8)interesting_8[j]) ||
          could_be_arith(orig, (u8)interesting_8[j], 1)) {
        stage_max--;
        continue;
      }
      stage_cur_val = interesting_8[j];
      out_buf[i] = interesting_8[j];

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

      out_buf[i] = orig;
      stage_cur++;

    }

  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_INTEREST8]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_INTEREST8] += stage_max;
```

DICTIONARY STUFF 阶段

- 通过 `-x` 选项指定一个词典，如果没有则跳过前两个阶段
- user extras(over),从头开始,将用户提供的tokens依次替换到原文件中,stage_max为 `extras_cnt * len`
- user extras(insert),从头开始,将用户提供的tokens依次插入到原文件中,stage_max为 `extras_cnt * len`
- 如果在之前的分析中提取到了 tokens，则进入 `auto extras` 阶段
- auto extras(over),从头开始,将自动检测的tokens依次替换到原文件中, `stage_max` 为 `MIN(a_extras_cnt, USE_AUTO_EXTRAS) * len`

此处给出 `auto extras (over)` 部分的源代码：

```
  if (!a_extras_cnt) goto skip_extras;

  stage_name  = "auto extras (over)";
  stage_short = "ext_AO";
  stage_cur   = 0;
  stage_max   = MIN(a_extras_cnt, USE_AUTO_EXTRAS) * len;

  stage_val_type = STAGE_VAL_NONE;

  orig_hit_cnt = new_hit_cnt;

  for (i = 0; i < len; i++) {

    u32 last_len = 0;
```

```
      stage_cur_byte = i;

      for (j = 0; j < MIN(a_extras_cnt, USE_AUTO_EXTRAS); j++) {

        /* See the comment in the earlier code; extras are sorted by size. */

        if (a_extras[j].len > len - i ||
            !memcmp(a_extras[j].data, out_buf + i, a_extras[j].len) ||
            !memchr(eff_map + EFF_APOS(i), 1, EFF_SPAN_ALEN(i,
a_extras[j].len))) {

          stage_max--;
          continue;

        }

        last_len = a_extras[j].len;
        memcpy(out_buf + i, a_extras[j].data, last_len);

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

        stage_cur++;

      }

      /* Restore all the clobbered memory. */
      memcpy(out_buf + i, in_buf + i, last_len);

    }

    new_hit_cnt = queued_paths + unique_crashes;

    stage_finds[STAGE_EXTRAS_AO]  += new_hit_cnt - orig_hit_cnt;
    stage_cycles[STAGE_EXTRAS_AO] += stage_max;
```

RANDOM HAVOC 阶段

该部分使用一个巨大的 switch ，通过随机数进行跳转，并在每个分支中使用随机数来完成随机性的行为：

- 首先指定出变换的此处上限 `use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2))`
- 然后进入循环，生成一个随机数去选择下列中的某一个情况来对样例进行变换
  - 随机选取某个bit进行翻转
  - 随机选取某个byte，将其设置为随机的interesting value

- 随机选取某个word，并随机选取大、小端序，将其设置为随机的interesting value
- 随机选取某个dword，并随机选取大、小端序，将其设置为随机的interesting value
- 随机选取某个byte，对其减去一个随机数
- 随机选取某个byte，对其加上一个随机数
- 随机选取某个word，并随机选取大、小端序，对其减去一个随机数
- 随机选取某个word，并随机选取大、小端序，对其加上一个随机数
- 随机选取某个dword，并随机选取大、小端序，对其减去一个随机数
- 随机选取某个dword，并随机选取大、小端序，对其加上一个随机数
- 随机选取某个byte，将其设置为随机数
- 随机删除一段bytes
- 随机选取一个位置，插入一段随机长度的内容，其中75%的概率是插入原文中随机位置的内容，25%的概率是插入一段随机选取的数
- 随机选取一个位置，替换为一段随机长度的内容，其中75%的概率是替换成原文中随机位置的内容，25%的概率是替换成一段随机选取的数
- 随机选取一个位置，用随机选取的token（用户提供的或自动生成的）替换
- 随机选取一个位置，用随机选取的token（用户提供的或自动生成的）插入

- 然后调用 `common_fuzz_stuff` 进行测试
- 重复上述过程 `stage_max` 次

```
  for (stage_cur = 0; stage_cur < stage_max; stage_cur++) {

    u32 use_stacking = 1 << (1 + UR(HAVOC_STACK_POW2));

    stage_cur_val = use_stacking;

    for (i = 0; i < use_stacking; i++) {

      switch (UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0))) {

        case 0:

          /* Flip a single bit somewhere. Spooky! */

          FLIP_BIT(out_buf, UR(temp_len << 3));
          break;

        case 1:

          /* Set byte to interesting value. */

          out_buf[UR(temp_len)] = interesting_8[UR(sizeof(interesting_8))];
          break;
        case 2:

          /* Set word to interesting value, randomly choosing endian. */

          if (temp_len < 2) break;

          if (UR(2)) {

            *(u16*)(out_buf + UR(temp_len - 1)) =
              interesting_16[UR(sizeof(interesting_16) >> 1)];

          } else {

            *(u16*)(out_buf + UR(temp_len - 1)) = SWAP16(
              interesting_16[UR(sizeof(interesting_16) >> 1)]);

          }

          break;

        case 3:

          /* Set dword to interesting value, randomly choosing endian. */

          if (temp_len < 4) break;

          if (UR(2)) {

            *(u32*)(out_buf + UR(temp_len - 3)) =
              interesting_32[UR(sizeof(interesting_32) >> 2)];

          } else {

            *(u32*)(out_buf + UR(temp_len - 3)) = SWAP32(
              interesting_32[UR(sizeof(interesting_32) >> 2)]);

          }

          break;

        case 4:

          /* Randomly subtract from byte. */

          out_buf[UR(temp_len)] -= 1 + UR(ARITH_MAX);
```

```
          break;

        case 5:

          /* Randomly add to byte. */

          out_buf[UR(temp_len)] += 1 + UR(ARITH_MAX);
          break;

        case 6:

          /* Randomly subtract from word, random endian. */

          if (temp_len < 2) break;

          if (UR(2)) {

            u32 pos = UR(temp_len - 1);

            *(u16*)(out_buf + pos) -= 1 + UR(ARITH_MAX);

          } else {

            u32 pos = UR(temp_len - 1);
            u16 num = 1 + UR(ARITH_MAX);

            *(u16*)(out_buf + pos) =
              SWAP16(SWAP16(*(u16*)(out_buf + pos)) - num);

          }

          break;

        case 7:

          /* Randomly add to word, random endian. */

          if (temp_len < 2) break;

          if (UR(2)) {

            u32 pos = UR(temp_len - 1);

            *(u16*)(out_buf + pos) += 1 + UR(ARITH_MAX);

          } else {
```

```
          u32 pos = UR(temp_len - 1);
          u16 num = 1 + UR(ARITH_MAX);

          *(u16*)(out_buf + pos) =
            SWAP16(SWAP16(*(u16*)(out_buf + pos)) + num);

        }

        break;

      case 8:

        /* Randomly subtract from dword, random endian. */

        if (temp_len < 4) break;

        if (UR(2)) {

          u32 pos = UR(temp_len - 3);

          *(u32*)(out_buf + pos) -= 1 + UR(ARITH_MAX);

        } else {

          u32 pos = UR(temp_len - 3);
          u32 num = 1 + UR(ARITH_MAX);

          *(u32*)(out_buf + pos) =
            SWAP32(SWAP32(*(u32*)(out_buf + pos)) - num);

        }

        break;

      case 9:

        /* Randomly add to dword, random endian. */

        if (temp_len < 4) break;

        if (UR(2)) {

          u32 pos = UR(temp_len - 3);

          *(u32*)(out_buf + pos) += 1 + UR(ARITH_MAX);
        } else {

          u32 pos = UR(temp_len - 3);
          u32 num = 1 + UR(ARITH_MAX);

          *(u32*)(out_buf + pos) =
            SWAP32(SWAP32(*(u32*)(out_buf + pos)) + num);

        }

        break;

      case 10:

        /* Just set a random byte to a random value. Because,
           why not. We use XOR with 1-255 to eliminate the
           possibility of a no-op. */

        out_buf[UR(temp_len)] ^= 1 + UR(255);
        break;

      case 11 ... 12: {

          /* Delete bytes. We're making this a bit more likely
             than insertion (the next option) in hopes of keeping
             files reasonably small. */

          u32 del_from, del_len;

          if (temp_len < 2) break;

          /* Don't delete too much. */

          del_len = choose_block_len(temp_len - 1);

          del_from = UR(temp_len - del_len + 1);

          memmove(out_buf + del_from, out_buf + del_from + del_len,
                  temp_len - del_from - del_len);

          temp_len -= del_len;

          break;

        }
        case 13:

          if (temp_len + HAVOC_BLK_XL < MAX_FILE) {

            /* Clone bytes (75%) or insert a block of constant bytes (25%).
*/

            u8  actually_clone = UR(4);
            u32 clone_from, clone_to, clone_len;
            u8* new_buf;

            if (actually_clone) {

              clone_len  = choose_block_len(temp_len);
              clone_from = UR(temp_len - clone_len + 1);

            } else {

              clone_len = choose_block_len(HAVOC_BLK_XL);
              clone_from = 0;

            }

            clone_to   = UR(temp_len);

            new_buf = ck_alloc_nozero(temp_len + clone_len);

            /* Head */

            memcpy(new_buf, out_buf, clone_to);

            /* Inserted part */

            if (actually_clone)
              memcpy(new_buf + clone_to, out_buf + clone_from, clone_len);
            else
              memset(new_buf + clone_to,
                     UR(2) ? UR(256) : out_buf[UR(temp_len)], clone_len);

            /* Tail */
            memcpy(new_buf + clone_to + clone_len, out_buf + clone_to,
                   temp_len - clone_to);

            ck_free(out_buf);
            out_buf = new_buf;
            temp_len += clone_len;

          }

          break;

        case 14: {

            /* Overwrite bytes with a randomly selected chunk (75%) or fixed
               bytes (25%). */

            u32 copy_from, copy_to, copy_len;

            if (temp_len < 2) break;

            copy_len  = choose_block_len(temp_len - 1);

            copy_from = UR(temp_len - copy_len + 1);
            copy_to   = UR(temp_len - copy_len + 1);

            if (UR(4)) {

              if (copy_from != copy_to)
                memmove(out_buf + copy_to, out_buf + copy_from, copy_len);

            } else memset(out_buf + copy_to,
                          UR(2) ? UR(256) : out_buf[UR(temp_len)],
copy_len);

            break;

          }

        /* Values 15 and 16 can be selected only if there are any extras
           present in the dictionaries. */

        case 15: {

            /* Overwrite bytes with an extra. */

            if (!extras_cnt || (a_extras_cnt && UR(2))) {

              /* No user-specified extras or odds in our favor. Let's use an
                 auto-detected one. */

              u32 use_extra = UR(a_extras_cnt);
              u32 extra_len = a_extras[use_extra].len;
              u32 insert_at;

              if (extra_len > temp_len) break;

              insert_at = UR(temp_len - extra_len + 1);
              memcpy(out_buf + insert_at, a_extras[use_extra].data,
extra_len);

            } else {

              /* No auto extras or odds in our favor. Use the dictionary. */

              u32 use_extra = UR(extras_cnt);
              u32 extra_len = extras[use_extra].len;
              u32 insert_at;

              if (extra_len > temp_len) break;

              insert_at = UR(temp_len - extra_len + 1);
              memcpy(out_buf + insert_at, extras[use_extra].data,
extra_len);

            }

            break;

          }

        case 16: {

            u32 use_extra, extra_len, insert_at = UR(temp_len + 1);
            u8* new_buf;

            /* Insert an extra. Do the same dice-rolling stuff as for the
               previous case. */

            if (!extras_cnt || (a_extras_cnt && UR(2))) {

              use_extra = UR(a_extras_cnt);
              extra_len = a_extras[use_extra].len;

              if (temp_len + extra_len >= MAX_FILE) break;

              new_buf = ck_alloc_nozero(temp_len + extra_len);
              /* Head */
              memcpy(new_buf, out_buf, insert_at);

              /* Inserted part */
              memcpy(new_buf + insert_at, a_extras[use_extra].data,
extra_len);

            } else {

              use_extra = UR(extras_cnt);
              extra_len = extras[use_extra].len;

              if (temp_len + extra_len >= MAX_FILE) break;

              new_buf = ck_alloc_nozero(temp_len + extra_len);

              /* Head */
              memcpy(new_buf, out_buf, insert_at);

              /* Inserted part */
              memcpy(new_buf + insert_at, extras[use_extra].data,
extra_len);

            }

            /* Tail */
            memcpy(new_buf + insert_at + extra_len, out_buf + insert_at,
                   temp_len - insert_at);

            ck_free(out_buf);
            out_buf   = new_buf;
            temp_len += extra_len;

            break;

          }

      }

    }

    if (common_fuzz_stuff(argv, out_buf, temp_len))
      goto abandon_entry;

    /* out_buf might have been mangled a bit, so let's restore it to its
       original size and shape. */
    if (temp_len < len) out_buf = ck_realloc(out_buf, len);
    temp_len = len;
    memcpy(out_buf, in_buf, len);

    /* If we're finding new stuff, let's run for a bit longer, limits
       permitting. */

    if (queued_paths != havoc_queued) {

      if (perf_score <= HAVOC_MAX_MULT * 100) {
        stage_max  *= 2;
        perf_score *= 2;
      }

      havoc_queued = queued_paths;

    }

  }
```

SPLICING 阶段

最后一个阶段，它会随机选择出另外一个输入样例，然后对当前的输入样例和另外一个样例都选择出合适的偏移量，然后从该处将他们拼接起来，然后重新进入到 `RANDOM HAVOC` 阶段。

```
#ifndef IGNORE_FINDS

  /************
   * SPLICING *
   ************/

  /* This is a last-resort strategy triggered by a full round with no
findings.
     It takes the current input file, randomly selects another input, and
     splices them together at some offset, then relies on the havoc
     code to mutate that blob. */

retry_splicing:

  if (use_splicing && splice_cycle++ < SPLICE_CYCLES &&
      queued_paths > 1 && queue_cur->len > 1) {

    struct queue_entry* target;
    u32 tid, split_at;
    u8* new_buf;
    s32 f_diff, l_diff;

    /* First of all, if we've modified in_buf for havoc, let's clean that
       up... */

    if (in_buf != orig_in) {
      ck_free(in_buf);
      in_buf = orig_in;
      len = queue_cur->len;
    }

    /* Pick a random queue entry and seek to it. Don't splice with yourself.
*/

    do { tid = UR(queued_paths); } while (tid == current_entry);

    splicing_with = tid;
    target = queue;

    while (tid >= 100) { target = target->next_100; tid -= 100; }
    while (tid--) target = target->next;

    /* Make sure that the target has a reasonable length. */

    while (target && (target->len < 2 || target == queue_cur)) {
      target = target->next;
      splicing_with++;
    }

    if (!target) goto retry_splicing;

    /* Read the testcase into a new buffer. */

    fd = open(target->fname, O_RDONLY);

    if (fd < 0) PFATAL("Unable to open '%s'", target->fname);

    new_buf = ck_alloc_nozero(target->len);

    ck_read(fd, new_buf, target->len, target->fname);

    close(fd);

    /* Find a suitable splicing location, somewhere between the first and
       the last differing byte. Bail out if the difference is just a single
       byte or so. */

    locate_diffs(in_buf, new_buf, MIN(len, target->len), &f_diff, &l_diff);

    if (f_diff < 0 || l_diff < 2 || f_diff == l_diff) {
      ck_free(new_buf);
      goto retry_splicing;
    }

    /* Split somewhere between the first and last differing byte. */

    split_at = f_diff + UR(l_diff - f_diff);

    /* Do the thing. */

    len = target->len;
    memcpy(new_buf, in_buf, split_at);
    in_buf = new_buf;

    ck_free(out_buf);
    out_buf = ck_alloc_nozero(len);
    memcpy(out_buf, in_buf, len);

    goto havoc_stage;

  }
```

结束

- 设置 `ret_val` 的值为 0
- 如果 `queue_cur` 通过了评估，且 `was_fuzzed` 字段是 0，就设置 `queue_cur->was_fuzzed` 为 1，然后 `pending_not_fuzzed` 计数器减一
- 如果 `queue_cur` 是 `favored` , `pending_favored` 计数器减一。

sync_fuzzers

读取其他 sync 文件夹下的 queue 文件，然后保存到自己的 queue 里。

- 打开 `sync_dir` 文件夹
- while循环读取该文件夹下的目录和文件 `while ((sd_ent = readdir(sd)))`
  - 跳过 `.` 开头的文件和 `sync_id` 即我们自己的输出文件夹
  - 读取 `out_dir/.synced/sd_ent->d_name` 文件即 `id_fd` 里的前4个字节到 `min_accept` 里，设置 `next_min_accept` 为 `min_accept` ，这个值代表之前从这个文件夹里读取到的最后一个queue的id。

- 设置 `stage_name` 为 `sprintf(stage_tmp, "sync %u", ++sync_cnt);` ，设置 `stage_cur` 为 0， `stage_max` 为 0
- 循环读取 `sync_dir/sd_ent->d_name/queue` 文件夹里的目录和文件
  - 同样跳过 `.` 开头的文件和标识小于 `min_accept` 的文件，因为这些文件应该已经被 sync 过了。
  - 如果标识 `syncing_case` 大于等于 `next_min_accept` ，就设置 `next_min_accept` 为 `syncing_case + 1`
  - 开始同步这个 case
    - 如果 case 大小为 0 或者大于 `MAX_FILE` (默认是1M),就不进行 sync。
    - 否则 mmap 这个文件到内存内存里，然后 `write_to_testcase(mem, st.st_size)` ,并 `run_target` ,然后通过 `save_if_interesting` 来决定是否要导入这个文件到自己的 queue 里，如果发现了新的 path，就导入。
      - 设置 `syncing_party` 的值为 `sd_ent->d_name`
      - 如果 `save_if_interesting` 返回 1， `queued_imported` 计数器就加 1
  - `stage_cur` 计数器加一，如果 `stage_cur` 是 `stats_update_freq` 的倍数，就刷新一次展示界面。
- 向id_fd写入当前的 `next_min_accept` 值

总结来说，这个函数就是先读取有哪些 fuzzer 文件夹，然后读取其他 fuzzer 文件夹下的 queue 文件夹里的 case，并依次执行，如果发现了新 path，就保存到自己的 queue 文件夹里，而且将最后一个 sync 的 case id 写入到 `.synced/其他fuzzer文件夹名` 文件里，以避免重复运行。

common_fuzz_stuff

因为 fuzz_one 部分过于庞大，而这个函数又不是那么特殊，因此把它拉出来做一个简短的说明。

- 若有 `post_handler` ，那么就对样例调用 `post_handler`
- 将样例写入文件，然后 `run_target` 执行
- 如果执行结果是超时则做如下操作:

```
  if (fault == FAULT_TMOUT) {

    if (subseq_tmouts++ > TMOUT_LIMIT) {
      cur_skipped_paths++;
      return 1;
    }

  } else subseq_tmouts = 0;
```

- 如果发现了新路径，那么保存并增加 `queued_discovered` 计数器
- 更新页面 `show_stats`

```
EXP_ST u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {

  u8 fault;

  if (post_handler) {

    out_buf = post_handler(out_buf, &len);
    if (!out_buf || !len) return 0;

  }

  write_to_testcase(out_buf, len);

  fault = run_target(argv, exec_tmout);

  if (stop_soon) return 1;

  if (fault == FAULT_TMOUT) {

    if (subseq_tmouts++ > TMOUT_LIMIT) {
      cur_skipped_paths++;
      return 1;
    }

  } else subseq_tmouts = 0;

  /* Users can hit us with SIGUSR1 to request the current input
     to be abandoned. */

  if (skip_requested) {

     skip_requested = 0;
     cur_skipped_paths++;
     return 1;

  }

  /* This handles FAULT_ERROR for us: */

  queued_discovered += save_if_interesting(argv, out_buf, len, fault);

  if (!(stage_cur % stats_update_freq) || stage_cur + 1 == stage_max)
```

```
    show_stats();

  return 0;

}
```

save_if_interesting

执行结果是否发现了新路径，决定是否保存或跳过。如果保存了这个 case，则返回 1，否则返回 0。

- 如果没有新的路径发现或者路径命中次数相同，就直接返回0
- 将 case 保存到 `fn = alloc_printf("%s/queue/id:%06u,%s", out_dir, queued_paths, describe_op(hnb))` 文件里
- 将新样本加入队列 `add_to_queue`
- 如果 `hnb` 的值是2，代表发现了新路径，设置刚刚加入到队列里的 queue 的 `has_new_cov` 字段为 1，即 `queue_top->has_new_cov = 1` ，然后 `queued_with_cov` 计数器加一
- 保存hash到其exec_cksum
- 评估这个queue，`calibrate_case(argv, queue_top, mem, queue_cycle - 1, 0)`
- 根据fault结果进入不同的分支
  - 若是出现错误，则直接抛出异常
  - 若是崩溃
    - total_crashes计数器加一
    - 如果unique_crashes大于能保存的最大数量`KEEP_UNIQUE_CRASH`即5000，就直接返回keeping的值
    - 如果不是dumb mode，就`simplify_trace((u64 *) trace_bits)`进行规整
    - 没有发现新的crash路径，就直接返回
    - 否则，代表发现了新的crash路径，unique_crashes计数器加一，并将结果保存到`alloc_printf("%s/crashes/id:%06llu,sig:%02u,%s", out_dir,unique_crashes, kill_signal, describe_op(0))`文件。
    - 更新last_crash_time和last_crash_execs
  - 若是超时
    - `total_tmouts` 计数器加一
    - 如果 `unique_hangs` 的个数超过能保存的最大数量 `KEEP_UNIQUE_HANG` 则返回
    - 若不是 dumb mode，就 `simplify_trace((u64 *) trace_bits)` 进行规整。
    - 没有发现新的超时路径，就直接返回
    - 否则，代表发现了新的超时路径，`unique_tmouts` 计数器加一
    - 若 `hang_tmout` 大于 `exec_tmout` ，则以 `hang_tmout` 为timeout，重新执行一次 `runt_target`

    - 若出现崩溃，就跳转到 `keep_as_crash`
    - 若没有超时则直接返回
    - 否则就使 `unique_hangs` 计数器加一，更新 `last_hang_time` 的值，并保存到 `alloc_printf("%s/hangs/id:%06llu,%s", out_dir, unique_hangs, describe_op(0))` 文件。
- 若是其他情况，则直接返回

插桩与路径发现的记录

其实插桩已经叙述过一部分了，在上文中的 `fork server` 部分，笔者就介绍过该机制就是通过插桩实现的。

但还有一部分内容没有涉及，新路径是如何在发现的同时被通知给 fuzzer 的？

在插桩阶段，我们为每个分支跳转都添加了一小段代码，这里笔者以 64 位的情况进行说明：

```
static const u8* trampoline_fmt_64 =

  "\n"
  "/* --- AFL TRAMPOLINE (64-BIT) --- */\n"
  "\n"
  ".align 4\n"
  "\n"
  "leaq -(128+24)(%%rsp), %%rsp\n"
  "movq %%rdx,  0(%%rsp)\n"
  "movq %%rcx,  8(%%rsp)\n"
  "movq %%rax, 16(%%rsp)\n"
  "movq $0x%08x, %%rcx\n"
  "call __afl_maybe_log\n"
  "movq 16(%%rsp), %%rax\n"
  "movq  8(%%rsp), %%rcx\n"
  "movq  0(%%rsp), %%rdx\n"
  "leaq (128+24)(%%rsp), %%rsp\n"
  "\n"
  "/* --- END --- */\n"
  "\n";
```

它首先保存了一部分将要被破坏的寄存器，然后调用了 `__afl_maybe_log` 来记录路径的发现。该函数同样是由汇编编写的，但我们可以用一些其他工具来反编译它：

```
char __fastcall _afl_maybe_log(__int64 a1, __int64 a2, __int64 a3, __int64
a4)
{
  char v4; // of
  char v5; // al
  __int64 v6; // rdx
  __int64 v7; // rcx
  char *v9; // rax
  int v10; // eax
  void *v11; // rax
  int v12; // edi
  __int64 v13; // rax
  __int64 v14; // rax
  __int64 v15; // [rsp-10h] [rbp-180h]
  char v16; // [rsp+10h] [rbp-160h]
  __int64 v17; // [rsp+18h] [rbp-158h]

  v5 = v4;
  v6 = _afl_area_ptr;
  if ( !_afl_area_ptr )
  {
    if ( _afl_setup_failure )
      return v5 + 127;
    v6 = _afl_global_area_ptr;
    if ( _afl_global_area_ptr )
    {
      _afl_area_ptr = _afl_global_area_ptr;
    }
    else
    {
      v16 = v4;
      v17 = a4;
      v9 = getenv("__AFL_SHM_ID");
      if ( !v9 || (v10 = atoi(v9), v11 = shmat(v10, 0LL, 0), v11 == -1LL) )
      {
        ++_afl_setup_failure;
        v5 = v16;
        return v5 + 127;
      }
      _afl_area_ptr = v11;
      _afl_global_area_ptr = v11;
      v15 = v11;
      if ( write(199, &_afl_temp, 4uLL) == 4 )
      {
        while ( 1 )
        {
          v12 = 198;
          if ( read(198, &_afl_temp, 4uLL) != 4 )
            break;
          LODWORD(v13) = fork();
          if ( v13 < 0 )
            break;
          if ( !v13 )
            goto __afl_fork_resume;
          _afl_fork_pid = v13;
          write(199, &_afl_fork_pid, 4uLL);
          v12 = _afl_fork_pid;
          LODWORD(v14) = waitpid(_afl_fork_pid, &_afl_temp, 0);
          if ( v14 <= 0 )
            break;
          write(199, &_afl_temp, 4uLL);
        }
        _exit(v12);
      }
__afl_fork_resume:
      close(198);
      close(199);
      v6 = v15;
      v5 = v16;
      a4 = v17;
    }
  }
  v7 = _afl_prev_loc ^ a4;
  _afl_prev_loc ^= v7;
  _afl_prev_loc = _afl_prev_loc >> 1;
  ++*(v6 + v7);
  return v5 + 127;
}
```

前面的一大段代码其实都是为了去建立我们在上文所说的“共享内存”，在完成初始化后调用最后这么一小段代码进行记录：

```
  v7 = _afl_prev_loc ^ a4;
  _afl_prev_loc ^= v7;
  _afl_prev_loc = _afl_prev_loc >> 1;
  ++*(v6 + v7);
```

此处 `v6` 即为共享内存的地址，而 `a4` 为 `cur_location` ，因此 `v7=cur_location ^ prev_location` ，它将作为索引，使得共享内存中的对应偏移处的值增加。而在 fuzzer 部分就可以通过检查这块内存来发现是否有新路径被得到了。

另外，`_afl_prev_loc = _afl_prev_loc >> 1;` 的目的是为了避开 `A->A` 和 `B->B` 以及 `A->B` 和 `B->A` 被识别为相同路径的情况。

其他阅读材料

sakuraのAFL源码全注释
https://eternalsakura13.com/2020/08/23/afl/

fuzzer AFL 源码分析
https://tttang.com/user/f1tao

AFL二三事——源码分析
https://paper.seebug.org/1732/#afl-afl-asc

AFL漏洞挖掘技术漫谈（一）：用AFL开始你的第一次Fuzzing
https://paper.seebug.org/841/