AFL-master

AFL-master

- .gitignore
- > travis
  - check_fuzzer_stats.sh
- .travis.yml
- afl-analyze.c
- afl-as.c
- afl-as.h
- afl-cmin
- afl-fuzz.c
- afl-gcc.c
- afl-gotcpu.c
- afl-plot
- afl-showmap.c
- afl-tmin.c
- afl-whatsup
- alloc-inl.h
- android-ashmem.h
- Android.bp
- config.h
- debug.h
- > dictionaries
  - gif.dict
  - html_tags.dict
  - jpeg.dict
  - js.dict
  - json.dict
  - pdf.dict
  - perl.dict
  - png.dict
  - regexp.dict
  - sql.dict
  - tiff.dict
  - webp.dict
  - xml.dict
- > experimental
  - > argv_fuzzing
    - argv-fuzz-inl.h
    - test.c
  - > asan_cgroups
    - limit_memory.sh
  - > bash_shellshock
    - shellshock-fuzz.diff
  - > canvas_harness
    - canvas_harness.html
  - > clang_asm_normalize
    - as
  - > crash_triage
    - triage_crashes.sh
  - > distributed_fuzzing
    - sync_script.sh
  - > libpng_no_checksum
    - libpng-nocrc.patch
  - > persistent_demo
    - persistent_demo.c
  - > post_library

post_library.so.c
post_library_png.so.c
hash.h
> libdislocator
libdislocator.so.c
Makefile
> libtokencap
libtokencap.so.c
Makefile
> llvm_mode
afl-clang-fast.c
afl-llvm-pass.so.cc
afl-llvm-rt.o.c
Makefile
Makefile
> qemu_mode
build_qemu_support.sh
> patches
afl-qemu-cpu-inl.h
configure.diff
cpu-exec.diff
elfload.diff
memfd.diff
syscall.diff
test-instr.c
test-libfuzzer-target.c
types.h

.gitignore

```
# Binaries produced by "make".
afl-analyze
afl-as
afl-clang
afl-clang++
afl-fuzz
afl-g++
afl-gcc
afl-gotcpu
afl-showmap
afl-tmin
as

# Binaries produced by "make -C llvm_mode"
afl-clang-fast
afl-clang-fast++
afl-llvm-pass.so
afl-llvm-rt-32.o
afl-llvm-rt-64.o
afl-llvm-rt.o
```

> travis

check_fuzzer_stats.sh

```
#!/bin/bash
usage() {
  echo "Usage: $0 -o <out_dir> -k <key> -v <value> [-p <precision>]" 1>&2;
  echo " " 1>&2;
  echo "Checks if a key:value appears in the fuzzer_stats report" 1>&2;
  echo " " 1>&2;
  echo -n "If \"value\" is numeric and \"precision\" is defined, checks if the stat " 1>&2;
  echo "printed by afl is value+/-precision." 1>&2;
  exit 1; }

while getopts "o:k:v:p:" opt;  do
  case "${opt}" in
      o)
        o=${OPTARG}
        ;;
      k)
        k=${OPTARG}
        ;;
      v)
        v=${OPTARG}
        ;;
      p)
        p=${OPTARG}
        ;;
      *)
        usage
        ;;
  esac
done

if [ -z $o ] || [ -z $k ] || [ -z $v ]; then usage; fi

# xargs to trim the surrounding whitespaces
stat_v=$( grep $k "$o"/fuzzer_stats | cut -d ":" -f 2 | xargs )
v=$( echo "$v" | xargs )

if [ -z stat_v ];
  then echo "ERROR: key $k not found in fuzzer_stats." 1>&2
  exit 1
fi

re_percent='^[0-9]+([.][0-9]+)?\%$'
# if the argument is a number in percentage, get rid of the %
if [[  "$v" =~ $re_percent ]]; then v=${v: :-1}; fi
if [[  "$stat_v" =~ $re_percent ]]; then stat_v=${stat_v: :-1}; fi

re_numeric='^[0-9]+([.][0-9]+)?$'
# if the argument is not a number, we check for strict equality
if (! [[  "$v" =~ $re_numeric ]]) || (! [[  "$stat_v" =~ $re ]]);
  then if [ "$v" != "$stat_v" ];
    then echo "ERROR: \"$k:$stat_v\" (should be $v)." 1>&2
    exit 2;
  fi
# checks if the stat reported by afl is in the range
elif [ "$stat_v" -lt $(( v - p )) ] || [ "$stat_v" -gt $(( v + p )) ];
  then echo "ERROR: key $k:$stat_v is out of correct range." 1>&2
  exit 3;
```

```
fi
echo "OK: key $k:$stat_v" 1>&2
```

.travis.yml

```
language: c

env:
  - AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 AFL_NO_UI=1 AFL_STOP_MANUALLY=1
  - AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 AFL_NO_UI=1 AFL_EXIT_WHEN_DONE=1
 # TODO: test AFL_BENCH_UNTIL_CRASH once we have a target that crashes
  - AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1 AFL_NO_UI=1 AFL_BENCH_JUST_ONE=1

before_install:
  - sudo apt update
  - sudo apt install -y libtool libtool-bin automake bison libglib2.0

# TODO: Look into splitting off some builds using a build matrix.
# TODO: Move this all into a bash script so we don't need to write bash in yaml.
script:
  - make
  - ./afl-gcc ./test-instr.c -o test-instr-gcc
  - mkdir seeds
  - echo "" > seeds/nil_seed
  - if [ -z "$AFL_STOP_MANUALLY" ];
    then ./afl-fuzz -i seeds -o out/ -- ./test-instr-gcc;
    else timeout --preserve-status 5s ./afl-fuzz -i seeds -o out/ -- ./test-instr-gcc;
    fi
  - .travis/check_fuzzer_stats.sh -o out -k peak_rss_mb -v 1 -p 3
  - rm -r out/*
  - ./afl-clang ./test-instr.c -o test-instr-clang
  - if [ -z "$AFL_STOP_MANUALLY" ];
    then ./afl-fuzz -i seeds -o out/ -- ./test-instr-clang;
    else timeout --preserve-status 5s ./afl-fuzz -i seeds -o out/ -- ./test-instr-clang;
    fi
  - .travis/check_fuzzer_stats.sh -o out -k peak_rss_mb -v 1 -p 2
  - make clean
  - CC=clang CXX=clang++ make
  - cd llvm_mode
  # TODO: Build with different versions of clang/LLVM since LLVM passes don't
  # have a stable API.
  - CC=clang CXX=clang++ LLVM_CONFIG=llvm-config make
  - cd ..
  - rm -r out/*
  - ./afl-clang-fast ./test-instr.c -o test-instr-clang-fast
  - if [ -z "$AFL_STOP_MANUALLY" ];
    then ./afl-fuzz -i seeds -o out/ -- ./test-instr-clang-fast;
    else timeout --preserve-status 5s ./afl-fuzz -i seeds -o out/ -- ./test-instr-clang-fast;
    fi
  - .travis/check_fuzzer_stats.sh -o out -k peak_rss_mb -v 1 -p 3
  # Test fuzzing libFuzzer targets and trace-pc-guard instrumentation.
  - clang -g -fsanitize-coverage=trace-pc-guard ./test-libfuzzer-target.c -c
  - clang -c -w llvm_mode/afl-llvm-rt.o.c
  - wget https://raw.githubusercontent.com/llvm/llvm-project/main/compiler-
rt/lib/fuzzer/afl/afl_driver.cpp
  - clang++ afl_driver.cpp afl-llvm-rt.o.o test-libfuzzer-target.o -o test-libfuzzer-target
  - timeout --preserve-status 5s ./afl-fuzz -i seeds -o out/ -- ./test-libfuzzer-target
```

```
  - cd qemu_mode
  - ./build_qemu_support.sh
  - cd ..
  - gcc ./test-instr.c -o test-no-instr
  - if [ -z "$AFL_STOP_MANUALLY" ];
    then ./afl-fuzz -Q -i seeds -o out/ -- ./test-no-instr;
    else timeout --preserve-status 5s ./afl-fuzz -Q -i seeds -o out/ -- ./test-no-instr;
    fi
  - .travis/check_fuzzer_stats.sh -o out -k peak_rss_mb -v 12 -p 9
```

afl-analyze.c

```
/*
  Copyright 2016 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - file format analyzer
   -----------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   A nifty utility that grabs an input file and takes a stab at explaining
   its structure by observing how changes to it affect the execution path.

   If the output scrolls past the edge of the screen, pipe it to 'less -r'.

*/

#define AFL_MAIN
#include "android-ashmem.h"

#include "config.h"
#include "types.h"
#include "debug.h"
#include "alloc-inl.h"
#include "hash.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <dirent.h>
```

```
#include <fcntl.h>
#include <ctype.h>

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/resource.h>

static s32 child_pid;                 /* PID of the tested program         */

static u8* trace_bits;                /* SHM with instrumentation bitmap   */

static u8 *in_file,                   /* Analyzer input test case          */
          *prog_in,                   /* Targeted program input file       */
          *target_path,               /* Path to target binary             */
          *doc_path;                  /* Path to docs                      */

static u8 *in_data;                   /* Input data for analysis           */

static u32 in_len,                    /* Input data length                 */
           orig_cksum,                /* Original checksum                 */
           total_execs,               /* Total number of execs             */
           exec_hangs,                /* Total number of hangs             */
           exec_tmout = EXEC_TIMEOUT; /* Exec timeout (ms)                 */

static u64 mem_limit = MEM_LIMIT;     /* Memory limit (MB)                 */

static s32 shm_id,                    /* ID of the SHM region              */
           dev_null_fd = -1;          /* FD to /dev/null                   */

static u8  edges_only,                /* Ignore hit counts?                */
           use_hex_offsets,           /* Show hex offsets?                 */
           use_stdin = 1;             /* Use stdin for program input?      */

static volatile u8
           stop_soon,                 /* Ctrl-C pressed?                   */
           child_timed_out;           /* Child timed out?                  */


/* Constants used for describing byte behavior. */

#define RESP_NONE       0x00          /* Changing byte is a no-op.         */
#define RESP_MINOR      0x01          /* Some changes have no effect.      */
#define RESP_VARIABLE   0x02          /* Changes produce variable paths.   */
#define RESP_FIXED      0x03          /* Changes produce fixed patterns.   */

#define RESP_LEN        0x04          /* Potential length field            */
#define RESP_CKSUM      0x05          /* Potential checksum                */
#define RESP_SUSPECT    0x06          /* Potential "suspect" blob          */


/* Classify tuple counts. This is a slow & naive version, but good enough here. */

static u8 count_class_lookup[256] = {

  [0]           = 0,
  [1]           = 1,
  [2]           = 2,
```

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

};

static void classify_counts(u8* mem) {

  u32 i = MAP_SIZE;

  if (edges_only) {

    while (i--) {
      if (*mem) *mem = 1;
      mem++;
    }

  } else {

    while (i--) {
      *mem = count_class_lookup[*mem];
      mem++;
    }

  }

}


/* See if any bytes are set in the bitmap. */

static inline u8 anything_set(void) {

  u32* ptr = (u32*)trace_bits;
  u32  i   = (MAP_SIZE >> 2);

  while (i--) if (*(ptr++)) return 1;

  return 0;

}


/* Get rid of shared memory and temp files (atexit handler). */

static void remove_shm(void) {

  unlink(prog_in); /* Ignore errors */
  shmctl(shm_id, IPC_RMID, NULL);

}


/* Configure shared memory. */

static void setup_shm(void) {
```

```
  u8* shm_str;

  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);

  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");

}


/* Read initial file. */

static void read_initial_file(void) {

  struct stat st;
  s32 fd = open(in_file, O_RDONLY);

  if (fd < 0) PFATAL("Unable to open '%s'", in_file);

  if (fstat(fd, &st) || !st.st_size)
    FATAL("Zero-sized input file.");

  if (st.st_size >= TMIN_MAX_FILE)
    FATAL("Input file is too large (%u MB max)", TMIN_MAX_FILE / 1024 / 1024);

  in_len  = st.st_size;
  in_data = ck_alloc_nozero(in_len);

  ck_read(fd, in_data, in_len, in_file);

  close(fd);

  OKF("Read %u byte%s from '%s'.", in_len, in_len == 1 ? "" : "s", in_file);

}


/* Write output file. */

static s32 write_to_file(u8* path, u8* mem, u32 len) {

  s32 ret;

  unlink(path); /* Ignore errors */

  ret = open(path, O_RDWR | O_CREAT | O_EXCL, 0600);

  if (ret < 0) PFATAL("Unable to create '%s'", path);
```

```
  ck_write(ret, mem, len, path);

  lseek(ret, 0, SEEK_SET);

  return ret;

}


/* Handle timeout signal. */

static void handle_timeout(int sig) {

  child_timed_out = 1;
  if (child_pid > 0) kill(child_pid, SIGKILL);

}


/* Execute target application. Returns exec checksum, or 0 if program
   times out. */

static u32 run_target(char** argv, u8* mem, u32 len, u8 first_run) {

  static struct itimerval it;
  int status = 0;

  s32 prog_in_fd;
  u32 cksum;

  memset(trace_bits, 0, MAP_SIZE);
  MEM_BARRIER();

  prog_in_fd = write_to_file(prog_in, mem, len);

  child_pid = fork();

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

  if (!child_pid) {

    struct rlimit r;

    if (dup2(use_stdin ? prog_in_fd : dev_null_fd, 0) < 0 ||
        dup2(dev_null_fd, 1) < 0 ||
        dup2(dev_null_fd, 2) < 0) {

      *(u32*)trace_bits = EXEC_FAIL_SIG;
      PFATAL("dup2() failed");

    }

    close(dev_null_fd);
    close(prog_in_fd);

    if (mem_limit) {

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

#ifdef RLIMIT_AS
```

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

#else

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

#endif /* ^RLIMIT_AS */

    }

    r.rlim_max = r.rlim_cur = 0;
    setrlimit(RLIMIT_CORE, &r); /* Ignore errors */

    execv(target_path, argv);

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

  }

  close(prog_in_fd);

  /* Configure timeout, wait for child, cancel timeout. */

  child_timed_out = 0;
  it.it_value.tv_sec = (exec_tmout / 1000);
  it.it_value.tv_usec = (exec_tmout % 1000) * 1000;

  setitimer(ITIMER_REAL, &it, NULL);

  if (waitpid(child_pid, &status, 0) <= 0) FATAL("waitpid() failed");

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

  setitimer(ITIMER_REAL, &it, NULL);

  MEM_BARRIER();

  /* Clean up bitmap, analyze exit condition, etc. */

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

  classify_counts(trace_bits);
  total_execs++;

  if (stop_soon) {
    SAYF(cRST cLRD "\n+++ Analysis aborted by user +++\n" cRST);
    exit(1);
  }

  /* Always discard inputs that time out. */

  if (child_timed_out) {

    exec_hangs++;
    return 0;
```

```
  }

  cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

  /* We don't actually care if the target is crashing or not,
     except that when it does, the checksum should be different. */

  if (WIFSIGNALED(status) ||
      (WIFEXITED(status) && WEXITSTATUS(status) == MSAN_ERROR) ||
      (WIFEXITED(status) && WEXITSTATUS(status))) {

    cksum ^= 0xffffffff;

  }

  if (first_run) orig_cksum = cksum;

  return cksum;

}


#ifdef USE_COLOR

/* Helper function to display a human-readable character. */

static void show_char(u8 val) {

  switch (val) {

    case 0 ... 32:
    case 127 ... 255: SAYF("#%02x", val); break;

    default: SAYF(" %c ", val);

  }

}


/* Show the legend */

static void show_legend(void) {

  SAYF("    " cLGR bgGRA " 01 " cRST " - no-op block              "
               cBLK bgLGN " 01 " cRST " - suspected length field\n"
       "    " cBRI bgGRA " 01 " cRST " - superficial content      "
               cBLK bgYEL " 01 " cRST " - suspected cksum or magic int\n"
       "    " cBLK bgCYA " 01 " cRST " - critical stream          "
               cBLK bgLRD " 01 " cRST " - suspected checksummed block\n"
       "    " cBLK bgMGN " 01 " cRST " - \"magic value\" section\n\n");

}

#endif /* USE_COLOR */


/* Interpret and report a pattern in the input file. */
```

```
static void dump_hex(u8* buf, u32 len, u8* b_data) {

  u32 i;

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

#ifdef USE_COLOR
    u32 rlen = 1, off;
#else
    u32 rlen = 1;
#endif /* ^USE_COLOR */

    u8  rtype = b_data[i] & 0x0f;

    /* Look ahead to determine the length of run. */

    while (i + rlen < len && (b_data[i] >> 7) == (b_data[i + rlen] >> 7)) {

      if (rtype < (b_data[i + rlen] & 0x0f)) rtype = b_data[i + rlen] & 0x0f;
      rlen++;

    }

    /* Try to do some further classification based on length & value. */

    if (rtype == RESP_FIXED) {

      switch (rlen) {

        case 2: {

            u16 val = *(u16*)(in_data + i);

            /* Small integers may be length fields. */

            if (val && (val <= in_len || SWAP16(val) <= in_len)) {
              rtype = RESP_LEN;
              break;
            }

            /* Uniform integers may be checksums. */

            if (val && abs(in_data[i] - in_data[i + 1]) > 32) {
              rtype = RESP_CKSUM;
              break;
            }

            break;

          }

        case 4: {

            u32 val = *(u32*)(in_data + i);

            /* Small integers may be length fields. */

            if (val && (val <= in_len || SWAP32(val) <= in_len)) {
              rtype = RESP_LEN;
              break;
```

```
            }

            /* Uniform integers may be checksums. */

            if (val && (in_data[i] >> 7 != in_data[i + 1] >> 7 ||
                in_data[i] >> 7 != in_data[i + 2] >> 7 ||
                in_data[i] >> 7 != in_data[i + 3] >> 7)) {
              rtype = RESP_CKSUM;
              break;
            }

            break;

          }

        case 1: case 3: case 5 ... MAX_AUTO_EXTRA - 1: break;

        default: rtype = RESP_SUSPECT;

      }

    }

    /* Print out the entire run. */

#ifdef USE_COLOR

    for (off = 0; off < rlen; off++) {

      /* Every 16 digits, display offset. */

      if (!((i + off) % 16)) {

        if (off) SAYF(cRST cLCY ">");

        if (use_hex_offsets)
          SAYF(cRST cGRA "%s[%06x] " cRST, (i + off) ? "\n" : "", i + off);
        else
          SAYF(cRST cGRA "%s[%06u] " cRST, (i + off) ? "\n" : "", i + off);

      }

      switch (rtype) {

        case RESP_NONE:     SAYF(cLGR bgGRA); break;
        case RESP_MINOR:    SAYF(cBRI bgGRA); break;
        case RESP_VARIABLE: SAYF(cBLK bgCYA); break;
        case RESP_FIXED:    SAYF(cBLK bgMGN); break;
        case RESP_LEN:      SAYF(cBLK bgLGN); break;
        case RESP_CKSUM:    SAYF(cBLK bgYEL); break;
        case RESP_SUSPECT:  SAYF(cBLK bgLRD); break;

      }

      show_char(in_data[i + off]);

      if (off != rlen - 1 && (i + off + 1) % 16) SAYF(" "); else SAYF(cRST " ");

    }
```

```
#else

    if (use_hex_offsets)
      SAYF("    Offset %x, length %u: ", i, rlen);
    else
      SAYF("    Offset %u, length %u: ", i, rlen);

    switch (rtype) {

      case RESP_NONE:     SAYF("no-op block\n"); break;
      case RESP_MINOR:    SAYF("superficial content\n"); break;
      case RESP_VARIABLE: SAYF("critical stream\n"); break;
      case RESP_FIXED:    SAYF("\"magic value\" section\n"); break;
      case RESP_LEN:      SAYF("suspected length field\n"); break;
      case RESP_CKSUM:    SAYF("suspected cksum or magic int\n"); break;
      case RESP_SUSPECT:  SAYF("suspected checksummed block\n"); break;

    }

#endif /* ^USE_COLOR */

    i += rlen - 1;

  }

#ifdef USE_COLOR
  SAYF(cRST "\n");
#endif /* USE_COLOR */

}



/* Actually analyze! */

static void analyze(char** argv) {

  u32 i;
  u32 boring_len = 0, prev_xff = 0, prev_x01 = 0, prev_s10 = 0, prev_a10 = 0;

  u8* b_data = ck_alloc(in_len + 1);
  u8  seq_byte = 0;

  b_data[in_len] = 0xff; /* Intentional terminator. */

  ACTF("Analyzing input file (this may take a while)...\n");

#ifdef USE_COLOR
  show_legend();
#endif /* USE_COLOR */

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

    u32 xor_ff, xor_01, sub_10, add_10;
    u8  xff_orig, x01_orig, s10_orig, a10_orig;

    /* Perform walking byte adjustments across the file. We perform four
       operations designed to elicit some response from the underlying
       code. */
```

```
  in_data[i] ^= 0xff;
  xor_ff = run_target(argv, in_data, in_len, 0);

  in_data[i] ^= 0xfe;
  xor_01 = run_target(argv, in_data, in_len, 0);

  in_data[i] = (in_data[i] ^ 0x01) - 0x10;
  sub_10 = run_target(argv, in_data, in_len, 0);

  in_data[i] += 0x20;
  add_10 = run_target(argv, in_data, in_len, 0);
  in_data[i] -= 0x10;

  /* Classify current behavior. */

  xff_orig = (xor_ff == orig_cksum);
  x01_orig = (xor_01 == orig_cksum);
  s10_orig = (sub_10 == orig_cksum);
  a10_orig = (add_10 == orig_cksum);

  if (xff_orig && x01_orig && s10_orig && a10_orig) {

    b_data[i] = RESP_NONE;
    boring_len++;

  } else if (xff_orig || x01_orig || s10_orig || a10_orig) {

    b_data[i] = RESP_MINOR;
    boring_len++;

  } else if (xor_ff == xor_01 && xor_ff == sub_10 && xor_ff == add_10) {

    b_data[i] = RESP_FIXED;

  } else b_data[i] = RESP_VARIABLE;

  /* When all checksums change, flip most significant bit of b_data. */

  if (prev_xff != xor_ff && prev_x01 != xor_01 &&
      prev_s10 != sub_10 && prev_a10 != add_10) seq_byte ^= 0x80;

  b_data[i] |= seq_byte;

  prev_xff = xor_ff;
  prev_x01 = xor_01;
  prev_s10 = sub_10;
  prev_a10 = add_10;

}

dump_hex(in_data, in_len, b_data);

SAYF("\n");

OKF("Analysis complete. Interesting bits: %0.02f%% of the input file.",
    100.0 - ((double)boring_len * 100) / in_len);

if (exec_hangs)
  WARNF(cLRD "Encountered %u timeouts - results may be skewed." cRST,
        exec_hangs);
```

```
  ck_free(b_data);

}



/* Handle Ctrl-C and the like. */

static void handle_stop_sig(int sig) {

  stop_soon = 1;

  if (child_pid > 0) kill(child_pid, SIGKILL);

}


/* Do basic preparations - persistent fds, filenames, etc. */

static void set_up_environment(void) {

  u8* x;

  dev_null_fd = open("/dev/null", O_RDWR);
  if (dev_null_fd < 0) PFATAL("Unable to open /dev/null");

  if (!prog_in) {

    u8* use_dir = ".";

    if (access(use_dir, R_OK | W_OK | X_OK)) {

      use_dir = getenv("TMPDIR");
      if (!use_dir) use_dir = "/tmp";

    }

    prog_in = alloc_printf("%s/.afl-analyze-temp-%u", use_dir, getpid());

  }

  /* Set sane defaults... */

  x = getenv("ASAN_OPTIONS");

  if (x) {

    if (!strstr(x, "abort_on_error=1"))
      FATAL("Custom ASAN_OPTIONS set without abort_on_error=1 - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom ASAN_OPTIONS set without symbolize=0 - please fix!");

  }

  x = getenv("MSAN_OPTIONS");

  if (x) {
```

```
    if (!strstr(x, "exit_code=" STRINGIFY(MSAN_ERROR)))
      FATAL("Custom MSAN_OPTIONS set without exit_code="
            STRINGIFY(MSAN_ERROR) " - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom MSAN_OPTIONS set without symbolize=0 - please fix!");

  }

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

  setenv("MSAN_OPTIONS", "exit_code=" STRINGIFY(MSAN_ERROR) ":"
                         "symbolize=0:"
                         "abort_on_error=1:"
                         "allocator_may_return_null=1:"
                         "msan_track_origins=0", 0);

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

}


/* Setup signal handlers, duh. */

static void setup_signal_handlers(void) {

  struct sigaction sa;

  sa.sa_handler   = NULL;
  sa.sa_flags     = SA_RESTART;
  sa.sa_sigaction = NULL;

  sigemptyset(&sa.sa_mask);

  /* Various ways of saying "stop". */

  sa.sa_handler = handle_stop_sig;
  sigaction(SIGHUP, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);

  /* Exec timeout notifications. */

  sa.sa_handler = handle_timeout;
  sigaction(SIGALRM, &sa, NULL);

}


/* Detect @@ in args. */

static void detect_file_args(char** argv) {

  u32 i = 0;
```

```
  u8* cwd = getcwd(NULL, 0);

  if (!cwd) PFATAL("getcwd() failed");

  while (argv[i]) {

    u8* aa_loc = strstr(argv[i], "@@");

    if (aa_loc) {

      u8 *aa_subst, *n_arg;

      /* Be sure that we're always using fully-qualified paths. */

      if (prog_in[0] == '/') aa_subst = prog_in;
      else aa_subst = alloc_printf("%s/%s", cwd, prog_in);

      /* Construct a replacement argv value. */

      *aa_loc = 0;
      n_arg = alloc_printf("%s%s%s", argv[i], aa_subst, aa_loc + 2);
      argv[i] = n_arg;
      *aa_loc = '@';

      if (prog_in[0] != '/') ck_free(aa_subst);

    }

    i++;

  }

  free(cwd); /* not tracked */

}


/* Display usage hints. */

static void usage(u8* argv0) {

  SAYF("\n%s [ options ] -- /path/to/target_app [ ... ]\n\n"

       "Required parameters:\n\n"

       "  -i file       - input test case to be analyzed by the tool\n"

       "Execution control settings:\n\n"

       "  -f file       - input file read by the tested program (stdin)\n"
       "  -t msec       - timeout for each run (%u ms)\n"
       "  -m megs       - memory limit for child process (%u MB)\n"
       "  -Q            - use binary-only instrumentation (QEMU mode)\n\n"

       "Analysis settings:\n\n"

       "  -e            - look for edge coverage only, ignore hit counts\n\n"

       "Other stuff:\n\n"
```

```
       "  -V            - show version number and exit\n\n"

       "For additional tips, please consult %s/README.\n\n",

       argv0, EXEC_TIMEOUT, MEM_LIMIT, doc_path);

  exit(1);

}


/* Find binary. */

static void find_binary(u8* fname) {

  u8* env_path = 0;
  struct stat st;

  if (strchr(fname, '/') || !(env_path = getenv("PATH"))) {

    target_path = ck_strdup(fname);

    if (stat(target_path, &st) || !S_ISREG(st.st_mode) ||
        !(st.st_mode & 0111) || st.st_size < 4)
      FATAL("Program '%s' not found or not executable", fname);

  } else {

    while (env_path) {

      u8 *cur_elem, *delim = strchr(env_path, ':');

      if (delim) {

        cur_elem = ck_alloc(delim - env_path + 1);
        memcpy(cur_elem, env_path, delim - env_path);
        delim++;

      } else cur_elem = ck_strdup(env_path);

      env_path = delim;

      if (cur_elem[0])
        target_path = alloc_printf("%s/%s", cur_elem, fname);
      else
        target_path = ck_strdup(fname);

      ck_free(cur_elem);

      if (!stat(target_path, &st) && S_ISREG(st.st_mode) &&
          (st.st_mode & 0111) && st.st_size >= 4) break;

      ck_free(target_path);
      target_path = 0;

    }

    if (!target_path) FATAL("Program '%s' not found or not executable", fname);

  }
```

```
}


/* Fix up argv for QEMU. */

static char** get_qemu_argv(u8* own_loc, char** argv, int argc) {

  char** new_argv = ck_alloc(sizeof(char*) * (argc + 4));
  u8 *tmp, *cp, *rsl, *own_copy;

  /* Workaround for a QEMU stability glitch. */

  setenv("QEMU_LOG", "nochain", 1);

  memcpy(new_argv + 3, argv + 1, sizeof(char*) * argc);

  /* Now we need to actually find qemu for argv[0]. */

  new_argv[2] = target_path;
  new_argv[1] = "--";

  tmp = getenv("AFL_PATH");

  if (tmp) {

    cp = alloc_printf("%s/afl-qemu-trace", tmp);

    if (access(cp, X_OK))
      FATAL("Unable to find '%s'", tmp);

    target_path = new_argv[0] = cp;
    return new_argv;

  }

  own_copy = ck_strdup(own_loc);
  rsl = strrchr(own_copy, '/');

  if (rsl) {

    *rsl = 0;

    cp = alloc_printf("%s/afl-qemu-trace", own_copy);
    ck_free(own_copy);

    if (!access(cp, X_OK)) {

      target_path = new_argv[0] = cp;
      return new_argv;

    }

  } else ck_free(own_copy);

  if (!access(BIN_PATH "/afl-qemu-trace", X_OK)) {

    target_path = new_argv[0] = BIN_PATH "/afl-qemu-trace";
    return new_argv;
```

```
  }

  FATAL("Unable to find 'afl-qemu-trace'.");

}


/* Main entry point */

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

  s32 opt;
  u8  mem_limit_given = 0, timeout_given = 0, qemu_mode = 0;
  char** use_argv;

  doc_path = access(DOC_PATH, F_OK) ? "docs" : DOC_PATH;

  SAYF(cCYA "afl-analyze " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  while ((opt = getopt(argc,argv,"+i:f:m:t:eQV")) > 0)

    switch (opt) {

      case 'i':

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

      case 'f':

        if (prog_in) FATAL("Multiple -f options not supported");
        use_stdin = 0;
        prog_in   = optarg;
        break;

      case 'e':

        if (edges_only) FATAL("Multiple -e options not supported");
        edges_only = 1;
        break;

      case 'm': {

          u8 suffix = 'M';

          if (mem_limit_given) FATAL("Multiple -m options not supported");
          mem_limit_given = 1;

          if (!strcmp(optarg, "none")) {

            mem_limit = 0;
            break;

          }

          if (sscanf(optarg, "%llu%c", &mem_limit, &suffix) < 1 ||
              optarg[0] == '-') FATAL("Bad syntax used for -m");

          switch (suffix) {
```

```
          case 'T': mem_limit *= 1024 * 1024; break;
          case 'G': mem_limit *= 1024; break;
          case 'k': mem_limit /= 1024; break;
          case 'M': break;

          default:  FATAL("Unsupported suffix or bad syntax for -m");

        }

        if (mem_limit < 5) FATAL("Dangerously low value of -m");

        if (sizeof(rlim_t) == 4 && mem_limit > 2000)
          FATAL("Value of -m out of range on 32-bit systems");

      }

      break;

    case 't':

      if (timeout_given) FATAL("Multiple -t options not supported");
      timeout_given = 1;

      exec_tmout = atoi(optarg);

      if (exec_tmout < 10 || optarg[0] == '-')
        FATAL("Dangerously low value of -t");

      break;

    case 'Q':

      if (qemu_mode) FATAL("Multiple -Q options not supported");
      if (!mem_limit_given) mem_limit = MEM_LIMIT_QEMU;

      qemu_mode = 1;
      break;

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

      /* Version number has been printed already, just quit. */
      exit(0);

    default:

      usage(argv[0]);

  }

if (optind == argc || !in_file) usage(argv[0]);

use_hex_offsets = !!getenv("AFL_ANALYZE_HEX");

setup_shm();
setup_signal_handlers();

set_up_environment();

find_binary(argv[optind]);
```

```
  detect_file_args(argv + optind);

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

  SAYF("\n");

  read_initial_file();

  ACTF("Performing dry run (mem limit = %llu MB, timeout = %u ms%s)...",
       mem_limit, exec_tmout, edges_only ? ", edges only" : "");

  run_target(use_argv, in_data, in_len, 1);

  if (child_timed_out)
    FATAL("Target binary times out (adjusting -t may help).");

  if (!anything_set()) FATAL("No instrumentation detected.");

  analyze(use_argv);

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

  exit(0);

}
```

afl-as.c

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - wrapper for GNU as
   ---------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   The sole purpose of this wrapper is to preprocess assembly files generated
   by GCC / clang and inject the instrumentation bits included from afl-as.h. It
   is automatically invoked by the toolchain when compiling programs using
   afl-gcc / afl-clang.
```

```
   Note that it's an explicit non-goal to instrument hand-written assembly,
   be it in separate .s files or in __asm__ blocks. The only aspiration this
   utility has right now is to be able to skip them gracefully and allow the
   compilation process to continue.

   That said, see experimental/clang_asm_normalize/ for a solution that may
   allow clang users to make things work even with hand-crafted assembly. Just
   note that there is no equivalent for GCC.

*/

#define AFL_MAIN

#include "config.h"
#include "types.h"
#include "debug.h"
#include "alloc-inl.h"

#include "afl-as.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <fcntl.h>

#include <sys/wait.h>
#include <sys/time.h>

static u8** as_params;          /* Parameters passed to the real 'as'   */

static u8*  input_file;         /* Originally specified input file      */
static u8*  modified_file;      /* Instrumented file for the real 'as'  */

static u8   be_quiet,           /* Quiet mode (no stderr output)        */
            clang_mode,         /* Running in clang mode?               */
            pass_thru,          /* Just pass data through?              */
            just_version,       /* Just show version?                   */
            sanitizer;          /* Using ASAN / MSAN                    */

static u32  inst_ratio = 100,   /* Instrumentation probability (%)      */
            as_par_cnt = 1;     /* Number of params to 'as'             */

/* If we don't find --32 or --64 in the command line, default to
   instrumentation for whichever mode we were compiled with. This is not
   perfect, but should do the trick for almost all use cases. */

#ifdef WORD_SIZE_64

static u8   use_64bit = 1;

#else

static u8   use_64bit = 0;

#ifdef __APPLE__
#  error "Sorry, 32-bit Apple platforms are not supported."
```

```
#endif /* __APPLE__ */

#endif /* ^WORD_SIZE_64 */


/* Examine and modify parameters to pass to 'as'. Note that the file name
   is always the last parameter passed by GCC, so we exploit this property
   to keep the code simple. */

static void edit_params(int argc, char** argv) {

  u8 *tmp_dir = getenv("TMPDIR"), *afl_as = getenv("AFL_AS");
  u32 i;

#ifdef __APPLE__

  u8 use_clang_as = 0;

  /* On MacOS X, the Xcode cctool 'as' driver is a bit stale and does not work
     with the code generated by newer versions of clang that are hand-built
     by the user. See the thread here: http://goo.gl/HBWDtn.

     To work around this, when using clang and running without AFL_AS
     specified, we will actually call 'clang -c' instead of 'as -q' to
     compile the assembly file.

     The tools aren't cmdline-compatible, but at least for now, we can
     seemingly get away with this by making only very minor tweaks. Thanks
     to Nico Weber for the idea. */

  if (clang_mode && !afl_as) {

    use_clang_as = 1;

    afl_as = getenv("AFL_CC");
    if (!afl_as) afl_as = getenv("AFL_CXX");
    if (!afl_as) afl_as = "clang";

  }

#endif /* __APPLE__ */

  /* Although this is not documented, GCC also uses TEMP and TMP when TMPDIR
     is not set. We need to check these non-standard variables to properly
     handle the pass_thru logic later on. */

  if (!tmp_dir) tmp_dir = getenv("TEMP");
  if (!tmp_dir) tmp_dir = getenv("TMP");
  if (!tmp_dir) tmp_dir = "/tmp";

  as_params = ck_alloc((argc + 32) * sizeof(u8*));

  as_params[0] = afl_as ? afl_as : (u8*)"as";

  as_params[argc] = 0;

  for (i = 1; i < argc - 1; i++) {

    if (!strcmp(argv[i], "--64")) use_64bit = 1;
    else if (!strcmp(argv[i], "--32")) use_64bit = 0;
```

```
#ifdef __APPLE__

    /* The Apple case is a bit different... */

    if (!strcmp(argv[i], "-arch") && i + 1 < argc) {

      if (!strcmp(argv[i + 1], "x86_64")) use_64bit = 1;
      else if (!strcmp(argv[i + 1], "i386"))
        FATAL("Sorry, 32-bit Apple platforms are not supported.");

    }

    /* Strip options that set the preference for a particular upstream
       assembler in Xcode. */

    if (clang_mode && (!strcmp(argv[i], "-q") || !strcmp(argv[i], "-Q")))
      continue;

#endif /* __APPLE__ */

    as_params[as_par_cnt++] = argv[i];

  }

#ifdef __APPLE__

  /* When calling clang as the upstream assembler, append -c -x assembler
     and hope for the best. */

  if (use_clang_as) {

    as_params[as_par_cnt++] = "-c";
    as_params[as_par_cnt++] = "-x";
    as_params[as_par_cnt++] = "assembler";

  }

#endif /* __APPLE__ */

  input_file = argv[argc - 1];

  if (input_file[0] == '-') {

    if (!strcmp(input_file + 1, "-version")) {
      just_version = 1;
      modified_file = input_file;
      goto wrap_things_up;
    }

    if (input_file[1]) FATAL("Incorrect use (not called through afl-gcc?)");
      else input_file = NULL;

  } else {

    /* Check if this looks like a standard invocation as a part of an attempt
       to compile a program, rather than using gcc on an ad-hoc .s file in
       a format we may not understand. This works around an issue compiling
       NSS. */
```

```
    if (strncmp(input_file, tmp_dir, strlen(tmp_dir)) &&
        strncmp(input_file, "/var/tmp/", 9) &&
        strncmp(input_file, "/tmp/", 5)) pass_thru = 1;

  }

  modified_file = alloc_printf("%s/.afl-%u-%u.s", tmp_dir, getpid(),
                               (u32)time(NULL));

wrap_things_up:

  as_params[as_par_cnt++] = modified_file;
  as_params[as_par_cnt]   = NULL;

}


/* Process input file, generate modified_file. Insert instrumentation in all
   the appropriate places. */

static void add_instrumentation(void) {

  static u8 line[MAX_LINE];

  FILE* inf;
  FILE* outf;
  s32 outfd;
  u32 ins_lines = 0;

  u8  instr_ok = 0, skip_csect = 0, skip_next_label = 0,
      skip_intel = 0, skip_app = 0, instrument_next = 0;

#ifdef __APPLE__

  u8* colon_pos;

#endif /* __APPLE__ */

  if (input_file) {

    inf = fopen(input_file, "r");
    if (!inf) PFATAL("Unable to read '%s'", input_file);

  } else inf = stdin;

  outfd = open(modified_file, O_WRONLY | O_EXCL | O_CREAT, 0600);

  if (outfd < 0) PFATAL("Unable to write to '%s'", modified_file);

  outf = fdopen(outfd, "w");

  if (!outf) PFATAL("fdopen() failed");

  while (fgets(line, MAX_LINE, inf)) {

    /* In some cases, we want to defer writing the instrumentation trampoline
       until after all the labels, macros, comments, etc. If we're in this
       mode, and if the line starts with a tab followed by a character, dump
       the trampoline now. */
```

```
if (!pass_thru && !skip_intel && !skip_app && !skip_csect && instr_ok &&
    instrument_next && line[0] == '\t' && isalpha(line[1])) {

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

  instrument_next = 0;
  ins_lines++;

}

/* Output the actual line, call it a day in pass-thru mode. */

fputs(line, outf);

if (pass_thru) continue;

/* 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] == '.') {

  /* OpenBSD puts jump tables directly inline with the code, which is
     a bit annoying. They use a specific format of p2align directives
     around them, so we use that as a signal. */

  if (!clang_mode && instr_ok && !strncmp(line + 2, "p2align ", 8) &&
      isdigit(line[10]) && line[11] == '\n') skip_next_label = 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;
  }

  if (!strncmp(line + 2, "section\t", 8) ||
      !strncmp(line + 2, "section ", 8) ||
      !strncmp(line + 2, "bss\n", 4) ||
      !strncmp(line + 2, "data\n", 5)) {
    instr_ok = 0;
    continue;
  }

}

/* Detect off-flavor assembly (rare, happens in gdb). When this is
   encountered, we set skip_csect until the opposite directive is
   seen, and we do not instrument. */

if (strstr(line, ".code")) {

  if (strstr(line, ".code32")) skip_csect = use_64bit;
  if (strstr(line, ".code64")) skip_csect = !use_64bit;

}

/* Detect syntax changes, as could happen with hand-written assembly.
```

```
   Skip Intel blocks, resume instrumentation when back to AT&T. */

if (strstr(line, ".intel_syntax")) skip_intel = 1;
if (strstr(line, ".att_syntax")) skip_intel = 0;

/* Detect and skip ad-hoc __asm__ blocks, likewise skipping them. */

if (line[0] == '#' || line[1] == '#') {

  if (strstr(line, "#APP")) skip_app = 1;
  if (strstr(line, "#NO_APP")) skip_app = 0;

}

/* 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);

  }

}


/* Main entry point */

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

  s32 pid;
  u32 rand_seed;
  int status;
  u8* inst_ratio_str = getenv("AFL_INST_RATIO");

  struct timeval tv;
  struct timezone tz;

  clang_mode = !!getenv(CLANG_ENV_VAR);

  if (isatty(2) && !getenv("AFL_QUIET")) {

    SAYF(cCYA "afl-as " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  } else be_quiet = 1;

  if (argc < 2) {

    SAYF("\n"
         "This is a helper application for afl-fuzz. It is a wrapper around GNU 'as',\n"
         "executed by the toolchain whenever using afl-gcc or afl-clang. You probably\n"
         "don't want to run this program directly.\n\n"

         "Rarely, when dealing with extremely complex projects, it may be advisable to\n"
         "set AFL_INST_RATIO to a value less than 100 in order to reduce the odds of\n"
         "instrumenting every discovered branch.\n\n");

    exit(1);
```

```
  }

  gettimeofday(&tv, &tz);

  rand_seed = tv.tv_sec ^ tv.tv_usec ^ getpid();

  srandom(rand_seed);

  edit_params(argc, argv);

  if (inst_ratio_str) {

    if (sscanf(inst_ratio_str, "%u", &inst_ratio) != 1 || inst_ratio > 100)
      FATAL("Bad value of AFL_INST_RATIO (must be between 0 and 100)");

  }

  if (getenv(AS_LOOP_ENV_VAR))
    FATAL("Endless loop when calling 'as' (remove '.' from your PATH)");

  setenv(AS_LOOP_ENV_VAR, "1", 1);

  /* When compiling with ASAN, we don't have a particularly elegant way to skip
     ASAN-specific branches. But we can probabilistically compensate for
     that... */

  if (getenv("AFL_USE_ASAN") || getenv("AFL_USE_MSAN")) {
    sanitizer = 1;
    inst_ratio /= 3;
  }

  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));

}
```

afl-as.h

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
```

```
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - injectable parts
   -------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   Forkserver design by Jann Horn <jannhorn@googlemail.com>

   This file houses the assembly-level instrumentation injected into fuzzed
   programs. The instrumentation stores XORed pairs of data: identifiers of the
   currently executing branch and the one that executed immediately before.

   TL;DR: the instrumentation does shm_trace_map[cur_loc ^ prev_loc]++

   The code is designed for 32-bit and 64-bit x86 systems. Both modes should
   work everywhere except for Apple systems. Apple does relocations differently
   from everybody else, so since their OSes have been 64-bit for a longer while,
   I didn't go through the mental effort of porting the 32-bit code.

   In principle, similar code should be easy to inject into any well-behaved
   binary-only code (e.g., using DynamoRIO). Conditional jumps offer natural
   targets for instrumentation, and should offer comparable probe density.

*/

#ifndef _HAVE_AFL_AS_H
#define _HAVE_AFL_AS_H

#include "config.h"
#include "types.h"

/*
   ------------------
   Performances notes
   ------------------

   Contributions to make this code faster are appreciated! Here are some
   rough notes that may help with the task:

   - Only the trampoline_fmt and the non-setup __afl_maybe_log code paths are
     really worth optimizing; the setup / fork server stuff matters a lot less
     and should be mostly just kept readable.

   - We're aiming for modern CPUs with out-of-order execution and large
     pipelines; the code is mostly follows intuitive, human-readable
     instruction ordering, because "textbook" manual reorderings make no
     substantial difference.
```

```
   - Interestingly, instrumented execution isn't a lot faster if we store a
     variable pointer to the setup, log, or return routine and then do a reg
     call from within trampoline_fmt. It does speed up non-instrumented
     execution quite a bit, though, since that path just becomes
     push-call-ret-pop.

   - There is also not a whole lot to be gained by doing SHM attach at a
     fixed address instead of retrieving __afl_area_ptr. Although it allows us
     to have a shorter log routine inserted for conditional jumps and jump
     labels (for a ~10% perf gain), there is a risk of bumping into other
     allocations created by the program or by tools such as ASAN.

   - popf is *awfully* slow, which is why we're doing the lahf / sahf +
     overflow test trick. Unfortunately, this forces us to taint eax / rax, but
     this dependency on a commonly-used register still beats the alternative of
     using pushf / popf.

     One possible optimization is to avoid touching flags by using a circular
     buffer that stores just a sequence of current locations, with the XOR stuff
     happening offline. Alas, this doesn't seem to have a huge impact:

     https://groups.google.com/d/msg/afl-users/MsajVf4fRLo/2u6t88ntUBIJ

   - Preforking one child a bit sooner, and then waiting for the "go" command
     from within the child, doesn't offer major performance gains; fork() seems
     to be relatively inexpensive these days. Preforking multiple children does
     help, but badly breaks the "~1 core per fuzzer" design, making it harder to
     scale up. Maybe there is some middle ground.

   Perhaps of note: in the 64-bit version for all platforms except for Apple,
   the instrumentation is done slightly differently than on 32-bit, with
   __afl_prev_loc and __afl_area_ptr being local to the object file (.lcomm),
   rather than global (.comm). This is to avoid GOTRELPC lookups in the critical
   code path, which AFAICT, are otherwise unavoidable if we want gcc -shared to
   work; simple relocations between .bss and .text won't work on most 64-bit
   platforms in such a case.

   (Fun fact: on Apple systems, .lcomm can segfault the linker.)

   The side effect is that state transitions are measured in a somewhat
   different way, with previous tuple being recorded separately within the scope
   of every .c file. This should have no impact in any practical sense.

   Another side effect of this design is that getenv() will be called once per
   every .o file when running in non-instrumented mode; and since getenv() tends
   to be optimized in funny ways, we need to be very careful to save every
   oddball register it may touch.

 */

static const u8* trampoline_fmt_32 =

  "\n"
  "/* --- AFL TRAMPOLINE (32-BIT) --- */\n"
  "\n"
  ".align 4\n"
  "\n"
  "leal -16(%%esp), %%esp\n"
  "movl %%edi,  0(%%esp)\n"
  "movl %%edx,  4(%%esp)\n"
```

```
  "movl %%ecx,  8(%%esp)\n"
  "movl %%eax, 12(%%esp)\n"
  "movl $0x%08x, %%ecx\n"
  "call __afl_maybe_log\n"
  "movl 12(%%esp), %%eax\n"
  "movl  8(%%esp), %%ecx\n"
  "movl  4(%%esp), %%edx\n"
  "movl  0(%%esp), %%edi\n"
  "leal 16(%%esp), %%esp\n"
  "\n"
  "/* --- END --- */\n"
  "\n";

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";

static const u8* main_payload_32 =

  "\n"
  "/* --- AFL MAIN PAYLOAD (32-BIT) --- */\n"
  "\n"
  ".text\n"
  ".att_syntax\n"
  ".code32\n"
  ".align 8\n"
  "\n"

  "__afl_maybe_log:\n"
  "\n"
  "  lahf\n"
  "  seto %al\n"
  "\n"
  "  /* Check if SHM region is already mapped. */\n"
  "\n"
  "  movl  __afl_area_ptr, %edx\n"
  "  testl %edx, %edx\n"
  "  je    __afl_setup\n"
  "\n"
  "__afl_store:\n"
  "\n"
  "  /* Calculate and store hit for the code location specified in ecx. There\n"
  "     is a double-XOR way of doing this without tainting another register,\n"
```

```
  "     and we use it on 64-bit systems; but it's slower for 32-bit ones. */\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"
  "__afl_return:\n"
  "\n"
  "  addb $127, %al\n"
  "  sahf\n"
  "  ret\n"
  "\n"
  ".align 8\n"
  "\n"
  "__afl_setup:\n"
  "\n"
  "  /* Do not retry setup if we had previous failures. */\n"
  "\n"
  "  cmpb $0, __afl_setup_failure\n"
  "  jne  __afl_return\n"
  "\n"
  "  /* Map SHM, jumping to __afl_setup_abort if something goes wrong.\n"
  "     We do not save FPU/MMX/SSE registers here, but hopefully, nobody\n"
  "     will notice this early in the game. */\n"
  "\n"
  "  pushl %eax\n"
  "  pushl %ecx\n"
  "\n"
  "  pushl $.AFL_SHM_ENV\n"
  "  call  getenv\n"
  "  addl  $4, %esp\n"
  "\n"
  "  testl %eax, %eax\n"
  "  je    __afl_setup_abort\n"
  "\n"
  "  pushl %eax\n"
  "  call  atoi\n"
  "  addl  $4, %esp\n"
  "\n"
  "  pushl $0          /* shmat flags    */\n"
  "  pushl $0          /* requested addr */\n"
  "  pushl %eax        /* SHM ID         */\n"
  "  call  shmat\n"
  "  addl  $12, %esp\n"
  "\n"
  "  cmpl $-1, %eax\n"
  "  je   __afl_setup_abort\n"
  "\n"
  "  /* Store the address of the SHM region. */\n"
```

```
"\n"
"  movl %eax, __afl_area_ptr\n"
"  movl %eax, %edx\n"
"\n"
"  popl %ecx\n"
"  popl %eax\n"
"\n"
"__afl_forkserver:\n"
"\n"
"  /* Enter the fork server mode to avoid the overhead of execve() calls. */\n"
"\n"
"  pushl %eax\n"
"  pushl %ecx\n"
"  pushl %edx\n"
"\n"
"  /* Phone home and tell the parent that we're OK. (Note that signals with\n"
"     no SA_RESTART will mess it up). If this fails, assume that the fd is\n"
"     closed because we were execve()d from an instrumented binary, or because\n"
"     the parent doesn't want to use the fork server. */\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"
"  cmpl  $4, %eax\n"
"  jne   __afl_fork_resume\n"
"\n"
"__afl_fork_wait_loop:\n"
"\n"
"  /* Wait for parent by reading from the pipe. Abort if read fails. */\n"
"\n"
"  pushl $4          /* length    */\n"
"  pushl $__afl_temp /* data      */\n"
"  pushl $" STRINGIFY(FORKSRV_FD) "        /* file desc */\n"
"  call  read\n"
"  addl  $12, %esp\n"
"\n"
"  cmpl  $4, %eax\n"
"  jne   __afl_die\n"
"\n"
"  /* Once woken up, create a clone of our process. This is an excellent use\n"
"     case for syscall(__NR_clone, 0, CLONE_PARENT), but glibc boneheadedly\n"
"     caches getpid() results and offers no way to update the value, breaking\n"
"     abort(), raise(), and a bunch of other things :-( */\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"
  "__afl_fork_resume:\n"
  "\n"
  "  /* In child process: close fds, resume execution. */\n"
  "\n"
  "  pushl $" STRINGIFY(FORKSRV_FD) "\n"
  "  call  close\n"
  "\n"
  "  pushl $" STRINGIFY((FORKSRV_FD + 1)) "\n"
  "  call  close\n"
  "\n"
  "  addl  $8, %esp\n"
  "\n"
  "  popl %edx\n"
  "  popl %ecx\n"
  "  popl %eax\n"
  "  jmp  __afl_store\n"
  "\n"
  "__afl_die:\n"
  "\n"
  "  xorl %eax, %eax\n"
  "  call _exit\n"
  "\n"
  "__afl_setup_abort:\n"
  "\n"
  "  /* Record setup failure so that we don't keep calling\n"
  "     shmget() / shmat() over and over again. */\n"
  "\n"
  "  incb __afl_setup_failure\n"
  "  popl %ecx\n"
  "  popl %eax\n"
  "  jmp __afl_return\n"
  "\n"
  ".AFL_VARS:\n"
  "\n"
  "  .comm   __afl_area_ptr, 4, 32\n"
  "  .comm   __afl_setup_failure, 1, 32\n"
#ifndef COVERAGE_ONLY
  "  .comm   __afl_prev_loc, 4, 32\n"
```

```
#endif /* !COVERAGE_ONLY */
  "  .comm   __afl_fork_pid, 4, 32\n"
  "  .comm   __afl_temp, 4, 32\n"
  "\n"
  ".AFL_SHM_ENV:\n"
  "  .asciz \"" SHM_ENV_VAR "\"\n"
  "\n"
  "/* --- END --- */\n"
  "\n";

/* The OpenBSD hack is due to lahf and sahf not being recognized by some
   versions of binutils: http://marc.info/?l=openbsd-cvs&m=141636589924400

   The Apple code is a bit different when calling libc functions because
   they are doing relocations differently from everybody else. We also need
   to work around the crash issue with .lcomm and the fact that they don't
   recognize .string. */

#ifdef __APPLE__
#  define CALL_L64(str)     "call _" str "\n"
#else
#  define CALL_L64(str)     "call " str "@PLT\n"
#endif /* ^__APPLE__ */

static const u8* main_payload_64 =

  "\n"
  "/* --- AFL MAIN PAYLOAD (64-BIT) --- */\n"
  "\n"
  ".text\n"
  ".att_syntax\n"
  ".code64\n"
  ".align 8\n"
  "\n"
  "__afl_maybe_log:\n"
  "\n"
#if defined(__OpenBSD__)  || (defined(__FreeBSD__) && (__FreeBSD__ < 9))
  "  .byte 0x9f /* lahf */\n"
#else
  "  lahf\n"
#endif /* ^__OpenBSD__, etc */
  "  seto  %al\n"
  "\n"
  "  /* Check if SHM region is already mapped. */\n"
  "\n"
  "  movq  __afl_area_ptr(%rip), %rdx\n"
  "  testq %rdx, %rdx\n"
  "  je    __afl_setup\n"
  "\n"
  "__afl_store:\n"
  "\n"
  "  /* Calculate and store hit for the code location specified in rcx. */\n"
  "\n"
#ifndef COVERAGE_ONLY
  "  xorq __afl_prev_loc(%rip), %rcx\n"
  "  xorq %rcx, __afl_prev_loc(%rip)\n"
  "  shrq $1, __afl_prev_loc(%rip)\n"
#endif /* ^!COVERAGE_ONLY */
  "\n"
#ifdef SKIP_COUNTS
```

```
  "  orb  $1, (%rdx, %rcx, 1)\n"
#else
  "  incb (%rdx, %rcx, 1)\n"
#endif /* ^SKIP_COUNTS */
  "\n"
  "__afl_return:\n"
  "\n"
  "  addb $127, %al\n"
#if defined(__OpenBSD__)  || (defined(__FreeBSD__) && (__FreeBSD__ < 9))
  "  .byte 0x9e /* sahf */\n"
#else
  "  sahf\n"
#endif /* ^__OpenBSD__, etc */
  "  ret\n"
  "\n"
  ".align 8\n"
  "\n"
  "__afl_setup:\n"
  "\n"
  "  /* Do not retry setup if we had previous failures. */\n"
  "\n"
  "  cmpb $0, __afl_setup_failure(%rip)\n"
  "  jne __afl_return\n"
  "\n"
  "  /* Check out if we have a global pointer on file. */\n"
  "\n"
#ifndef __APPLE__
  "  movq  __afl_global_area_ptr@GOTPCREL(%rip), %rdx\n"
  "  movq  (%rdx), %rdx\n"
#else
  "  movq  __afl_global_area_ptr(%rip), %rdx\n"
#endif /* !^__APPLE__ */
  "  testq %rdx, %rdx\n"
  "  je    __afl_setup_first\n"
  "\n"
  "  movq %rdx, __afl_area_ptr(%rip)\n"
  "  jmp  __afl_store\n"
  "\n"
  "__afl_setup_first:\n"
  "\n"
  "  /* Save everything that is not yet saved and that may be touched by\n"
  "     getenv() and several other libcalls we'll be relying on. */\n"
  "\n"
  "  leaq -352(%rsp), %rsp\n"
  "\n"
  "  movq %rax,   0(%rsp)\n"
  "  movq %rcx,   8(%rsp)\n"
  "  movq %rdi,  16(%rsp)\n"
  "  movq %rsi,  32(%rsp)\n"
  "  movq %r8,   40(%rsp)\n"
  "  movq %r9,   48(%rsp)\n"
  "  movq %r10,  56(%rsp)\n"
  "  movq %r11,  64(%rsp)\n"
  "\n"
  "  movq %xmm0,  96(%rsp)\n"
  "  movq %xmm1,  112(%rsp)\n"
  "  movq %xmm2,  128(%rsp)\n"
  "  movq %xmm3,  144(%rsp)\n"
  "  movq %xmm4,  160(%rsp)\n"
  "  movq %xmm5,  176(%rsp)\n"
```

```
  "  movq %xmm6,  192(%rsp)\n"
  "  movq %xmm7,  208(%rsp)\n"
  "  movq %xmm8,  224(%rsp)\n"
  "  movq %xmm9,  240(%rsp)\n"
  "  movq %xmm10, 256(%rsp)\n"
  "  movq %xmm11, 272(%rsp)\n"
  "  movq %xmm12, 288(%rsp)\n"
  "  movq %xmm13, 304(%rsp)\n"
  "  movq %xmm14, 320(%rsp)\n"
  "  movq %xmm15, 336(%rsp)\n"
  "\n"
  "  /* Map SHM, jumping to __afl_setup_abort if something goes wrong. */\n"
  "\n"
  "  /* The 64-bit ABI requires 16-byte stack alignment. We'll keep the\n"
  "     original stack ptr in the callee-saved r12. */\n"
  "\n"
  "  pushq %r12\n"
  "  movq  %rsp, %r12\n"
  "  subq  $16, %rsp\n"
  "  andq  $0xfffffffffffffff0, %rsp\n"
  "\n"
  "  leaq .AFL_SHM_ENV(%rip), %rdi\n"
  CALL_L64("getenv")
  "\n"
  "  testq %rax, %rax\n"
  "  je    __afl_setup_abort\n"
  "\n"
  "  movq  %rax, %rdi\n"
  CALL_L64("atoi")
  "\n"
  "  xorq %rdx, %rdx   /* shmat flags    */\n"
  "  xorq %rsi, %rsi   /* requested addr */\n"
  "  movq %rax, %rdi   /* SHM ID         */\n"
  CALL_L64("shmat")
  "\n"
  "  cmpq $-1, %rax\n"
  "  je   __afl_setup_abort\n"
  "\n"
  "  /* Store the address of the SHM region. */\n"
  "\n"
  "  movq %rax, %rdx\n"
  "  movq %rax, __afl_area_ptr(%rip)\n"
  "\n"
#ifdef __APPLE__
  "  movq %rax, __afl_global_area_ptr(%rip)\n"
#else
  "  movq __afl_global_area_ptr@GOTPCREL(%rip), %rdx\n"
  "  movq %rax, (%rdx)\n"
#endif /* ^__APPLE__ */
  "  movq %rax, %rdx\n"
  "\n"
  "__afl_forkserver:\n"
  "\n"
  "  /* Enter the fork server mode to avoid the overhead of execve() calls. We\n"
  "     push rdx (area ptr) twice to keep stack alignment neat. */\n"
  "\n"
  "  pushq %rdx\n"
  "  pushq %rdx\n"
  "\n"
  "  /* Phone home and tell the parent that we're OK. (Note that signals with\n"
```

```
"     no SA_RESTART will mess it up). If this fails, assume that the fd is\n"
"     closed because we were execve()d from an instrumented binary, or because\n"
"     the parent doesn't want to use the fork server. */\n"
"\n"
"  movq $4, %rdx                /* length    */\n"
"  leaq __afl_temp(%rip), %rsi /* data      */\n"
"  movq $" STRINGIFY((FORKSRV_FD + 1)) ", %rdi       /* file desc */\n"
CALL_L64("write")
"\n"
"  cmpq $4, %rax\n"
"  jne  __afl_fork_resume\n"
"\n"
"__afl_fork_wait_loop:\n"
"\n"
"  /* Wait for parent by reading from the pipe. Abort if read fails. */\n"
"\n"
"  movq $4, %rdx                /* length    */\n"
"  leaq __afl_temp(%rip), %rsi /* data      */\n"
"  movq $" STRINGIFY(FORKSRV_FD) ", %rdi             /* file desc */\n"
CALL_L64("read")
"  cmpq $4, %rax\n"
"  jne  __afl_die\n"
"\n"
"  /* Once woken up, create a clone of our process. This is an excellent use\n"
"     case for syscall(__NR_clone, 0, CLONE_PARENT), but glibc boneheadedly\n"
"     caches getpid() results and offers no way to update the value, breaking\n"
"     abort(), raise(), and a bunch of other things :-( */\n"
"\n"
CALL_L64("fork")
"  cmpq $0, %rax\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(%rip)\n"
"\n"
"  movq $4, %rdx                   /* length    */\n"
"  leaq __afl_fork_pid(%rip), %rsi /* data      */\n"
"  movq $" STRINGIFY((FORKSRV_FD + 1)) ", %rdi             /* file desc */\n"
CALL_L64("write")
"\n"
"  movq $0, %rdx                   /* no flags  */\n"
"  leaq __afl_temp(%rip), %rsi     /* status    */\n"
"  movq __afl_fork_pid(%rip), %rdi /* PID       */\n"
CALL_L64("waitpid")
"  cmpq $0, %rax\n"
"  jle  __afl_die\n"
"\n"
"  /* Relay wait status to pipe, then loop back. */\n"
"\n"
"  movq $4, %rdx                /* length    */\n"
"  leaq __afl_temp(%rip), %rsi /* data      */\n"
"  movq $" STRINGIFY((FORKSRV_FD + 1)) ", %rdi         /* file desc */\n"
CALL_L64("write")
"\n"
"  jmp  __afl_fork_wait_loop\n"
"\n"
"__afl_fork_resume:\n"
"\n"
```

```
"  /* In child process: close fds, resume execution. */\n"
"\n"
"  movq $" STRINGIFY(FORKSRV_FD) ", %rdi\n"
CALL_L64("close")
"\n"
"  movq $" STRINGIFY((FORKSRV_FD + 1)) ", %rdi\n"
CALL_L64("close")
"\n"
"  popq %rdx\n"
"  popq %rdx\n"
"\n"
"  movq %r12, %rsp\n"
"  popq %r12\n"
"\n"
"  movq  0(%rsp), %rax\n"
"  movq  8(%rsp), %rcx\n"
"  movq 16(%rsp), %rdi\n"
"  movq 32(%rsp), %rsi\n"
"  movq 40(%rsp), %r8\n"
"  movq 48(%rsp), %r9\n"
"  movq 56(%rsp), %r10\n"
"  movq 64(%rsp), %r11\n"
"\n"
"  movq  96(%rsp), %xmm0\n"
"  movq 112(%rsp), %xmm1\n"
"  movq 128(%rsp), %xmm2\n"
"  movq 144(%rsp), %xmm3\n"
"  movq 160(%rsp), %xmm4\n"
"  movq 176(%rsp), %xmm5\n"
"  movq 192(%rsp), %xmm6\n"
"  movq 208(%rsp), %xmm7\n"
"  movq 224(%rsp), %xmm8\n"
"  movq 240(%rsp), %xmm9\n"
"  movq 256(%rsp), %xmm10\n"
"  movq 272(%rsp), %xmm11\n"
"  movq 288(%rsp), %xmm12\n"
"  movq 304(%rsp), %xmm13\n"
"  movq 320(%rsp), %xmm14\n"
"  movq 336(%rsp), %xmm15\n"
"\n"
"  leaq 352(%rsp), %rsp\n"
"\n"
"  jmp  __afl_store\n"
"\n"
"__afl_die:\n"
"\n"
"  xorq %rax, %rax\n"
CALL_L64("_exit")
"\n"
"__afl_setup_abort:\n"
"\n"
"  /* Record setup failure so that we don't keep calling\n"
"     shmget() / shmat() over and over again. */\n"
"\n"
"  incb __afl_setup_failure(%rip)\n"
"\n"
"  movq %r12, %rsp\n"
"  popq %r12\n"
"\n"
"  movq  0(%rsp), %rax\n"
```

```
  "  movq  8(%rsp), %rcx\n"
  "  movq 16(%rsp), %rdi\n"
  "  movq 32(%rsp), %rsi\n"
  "  movq 40(%rsp), %r8\n"
  "  movq 48(%rsp), %r9\n"
  "  movq 56(%rsp), %r10\n"
  "  movq 64(%rsp), %r11\n"
  "\n"
  "  movq  96(%rsp), %xmm0\n"
  "  movq 112(%rsp), %xmm1\n"
  "  movq 128(%rsp), %xmm2\n"
  "  movq 144(%rsp), %xmm3\n"
  "  movq 160(%rsp), %xmm4\n"
  "  movq 176(%rsp), %xmm5\n"
  "  movq 192(%rsp), %xmm6\n"
  "  movq 208(%rsp), %xmm7\n"
  "  movq 224(%rsp), %xmm8\n"
  "  movq 240(%rsp), %xmm9\n"
  "  movq 256(%rsp), %xmm10\n"
  "  movq 272(%rsp), %xmm11\n"
  "  movq 288(%rsp), %xmm12\n"
  "  movq 304(%rsp), %xmm13\n"
  "  movq 320(%rsp), %xmm14\n"
  "  movq 336(%rsp), %xmm15\n"
  "\n"
  "  leaq 352(%rsp), %rsp\n"
  "\n"
  "  jmp __afl_return\n"
  "\n"
  ".AFL_VARS:\n"
  "\n"

#ifdef __APPLE__

  "  .comm   __afl_area_ptr, 8\n"
#ifndef COVERAGE_ONLY
  "  .comm   __afl_prev_loc, 8\n"
#endif /* !COVERAGE_ONLY */
  "  .comm   __afl_fork_pid, 4\n"
  "  .comm   __afl_temp, 4\n"
  "  .comm   __afl_setup_failure, 1\n"

#else

  "  .lcomm   __afl_area_ptr, 8\n"
#ifndef COVERAGE_ONLY
  "  .lcomm   __afl_prev_loc, 8\n"
#endif /* !COVERAGE_ONLY */
  "  .lcomm   __afl_fork_pid, 4\n"
  "  .lcomm   __afl_temp, 4\n"
  "  .lcomm   __afl_setup_failure, 1\n"

#endif /* ^__APPLE__ */

  "  .comm    __afl_global_area_ptr, 8, 8\n"
  "\n"
  ".AFL_SHM_ENV:\n"
  "  .asciz \"" SHM_ENV_VAR "\"\n"
  "\n"
  "/* --- END --- */\n"
```

```
  "\n";

#endif /* !_HAVE_AFL_AS_H */
```

afl-cmin

```
#!/usr/bin/env bash
#
# american fuzzy lop - corpus minimization tool
# ---------------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
#
# Copyright 2014, 2015 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# This tool tries to find the smallest subset of files in the input directory
# that still trigger the full range of instrumentation data points seen in
# the starting corpus. This has two uses:
#
#   - Screening large corpora of input files before using them as a seed for
#     afl-fuzz. The tool will remove functionally redundant files and likely
#     leave you with a much smaller set.
#
#     (In this case, you probably also want to consider running afl-tmin on
#     the individual files later on to reduce their size.)
#
#   - Minimizing the corpus generated organically by afl-fuzz, perhaps when
#     planning to feed it to more resource-intensive tools. The tool achieves
#     this by removing all entries that used to trigger unique behaviors in the
#     past, but have been made obsolete by later finds.
#
# Note that the tool doesn't modify the files themselves. For that, you want
# afl-tmin.
#
# This script must use bash because other shells may have hardcoded limits on
# array sizes.
#

echo "corpus minimization tool for afl-fuzz by <lcamtuf@google.com>"
echo

#########
# SETUP #
#########

# Process command-line options...

MEM_LIMIT=100
TIMEOUT=none

unset IN_DIR OUT_DIR STDIN_FILE EXTRA_PAR MEM_LIMIT_GIVEN \
  AFL_CMIN_CRASHES_ONLY AFL_CMIN_ALLOW_ANY QEMU_MODE
```

```
while getopts "+i:o:f:m:t:eQC" opt; do

  case "$opt" in

    "i")
         IN_DIR="$OPTARG"
         ;;

    "o")
         OUT_DIR="$OPTARG"
         ;;
    "f")
         STDIN_FILE="$OPTARG"
         ;;
    "m")
         MEM_LIMIT="$OPTARG"
         MEM_LIMIT_GIVEN=1
         ;;
    "t")
         TIMEOUT="$OPTARG"
         ;;
    "e")
         EXTRA_PAR="$EXTRA_PAR -e"
         ;;
    "C")
         export AFL_CMIN_CRASHES_ONLY=1
         ;;
    "Q")
         EXTRA_PAR="$EXTRA_PAR -Q"
         test "$MEM_LIMIT_GIVEN" = "" && MEM_LIMIT=250
         QEMU_MODE=1
         ;;
    "?")
         exit 1
         ;;

   esac

done

shift $((OPTIND-1))

TARGET_BIN="$1"

if [ "$TARGET_BIN" = "" -o "$IN_DIR" = "" -o "$OUT_DIR" = "" ]; then

  cat 1>&2 <<_EOF_
Usage: $0 [ options ] -- /path/to/target_app [ ... ]

Required parameters:

  -i dir        - input directory with the starting corpus
  -o dir        - output directory for minimized files

Execution control settings:

  -f file       - location read by the fuzzed program (stdin)
  -m megs       - memory limit for child process ($MEM_LIMIT MB)
  -t msec       - run time limit for child process (none)
```

```
  -Q              - use binary-only instrumentation (QEMU mode)

Minimization settings:

  -C              - keep crashing inputs, reject everything else
  -e              - solve for edge coverage only, ignore hit counts

For additional tips, please consult docs/README.

_EOF_
  exit 1
fi

# Do a sanity check to discourage the use of /tmp, since we can't really
# handle this safely from a shell script.

if [ "$AFL_ALLOW_TMP" = "" ]; then

  echo "$IN_DIR" | grep -qE '^(/var)?/tmp/'
  T1="$?"

  echo "$TARGET_BIN" | grep -qE '^(/var)?/tmp/'
  T2="$?"

  echo "$OUT_DIR" | grep -qE '^(/var)?/tmp/'
  T3="$?"

  echo "$STDIN_FILE" | grep -qE '^(/var)?/tmp/'
  T4="$?"

  echo "$PWD" | grep -qE '^(/var)?/tmp/'
  T5="$?"

  if [ "$T1" = "0" -o "$T2" = "0" -o "$T3" = "0" -o "$T4" = "0" -o "$T5" = "0" ]; then
    echo "[-] Error: do not use this script in /tmp or /var/tmp." 1>&2
    exit 1
  fi

fi

# If @@ is specified, but there's no -f, let's come up with a temporary input
# file name.

TRACE_DIR="$OUT_DIR/.traces"

if [ "$STDIN_FILE" = "" ]; then

  if echo "$*" | grep -qF '@@'; then
    STDIN_FILE="$TRACE_DIR/.cur_input"
  fi

fi

# Check for obvious errors.

if [ ! "$MEM_LIMIT" = "none" ]; then

  if [ "$MEM_LIMIT" -lt "5" ]; then
    echo "[-] Error: dangerously low memory limit." 1>&2
    exit 1
```

```
  fi

fi

if [ ! "$TIMEOUT" = "none" ]; then

  if [ "$TIMEOUT" -lt "10" ]; then
    echo "[-] Error: dangerously low timeout." 1>&2
    exit 1
  fi

fi

if [ ! -f "$TARGET_BIN" -o ! -x "$TARGET_BIN" ]; then

  TNEW="`which "$TARGET_BIN" 2>/dev/null`"

  if [ ! -f "$TNEW" -o ! -x "$TNEW" ]; then
    echo "[-] Error: binary '$TARGET_BIN' not found or not executable." 1>&2
    exit 1
  fi

  TARGET_BIN="$TNEW"

fi

if [ "$AFL_SKIP_BIN_CHECK" = "" -a "$QEMU_MODE" = "" ]; then

  if ! grep -qF "__AFL_SHM_ID" "$TARGET_BIN"; then
    echo "[-] Error: binary '$TARGET_BIN' doesn't appear to be instrumented." 1>&2
    exit 1
  fi

fi

if [ ! -d "$IN_DIR" ]; then
  echo "[-] Error: directory '$IN_DIR' not found." 1>&2
  exit 1
fi

test -d "$IN_DIR/queue" && IN_DIR="$IN_DIR/queue"

find "$OUT_DIR" -name 'id[:_]*' -maxdepth 1 -exec rm -- {} \; 2>/dev/null
rm -rf "$TRACE_DIR" 2>/dev/null

rmdir "$OUT_DIR" 2>/dev/null

if [ -d "$OUT_DIR" ]; then
  echo "[-] Error: directory '$OUT_DIR' exists and is not empty - delete it first." 1>&2
  exit 1
fi

mkdir -m 700 -p "$TRACE_DIR" || exit 1

if [ ! "$STDIN_FILE" = "" ]; then
  rm -f "$STDIN_FILE" || exit 1
  touch "$STDIN_FILE" || exit 1
fi

if [ "$AFL_PATH" = "" ]; then
```

```
  SHOWMAP="${0%/afl-cmin}/afl-showmap"
else
  SHOWMAP="$AFL_PATH/afl-showmap"
fi

if [ ! -x "$SHOWMAP" ]; then
  echo "[-] Error: can't find 'afl-showmap' - please set AFL_PATH." 1>&2
  rm -rf "$TRACE_DIR"
  exit 1
fi

IN_COUNT=$((`ls -- "$IN_DIR" 2>/dev/null | wc -l`))

if [ "$IN_COUNT" = "0" ]; then
  echo "[+] Hmm, no inputs in the target directory. Nothing to be done."
  rm -rf "$TRACE_DIR"
  exit 1
fi

FIRST_FILE=`ls "$IN_DIR" | head -1`

# Make sure that we're not dealing with a directory.

if [ -d "$IN_DIR/$FIRST_FILE" ]; then
  echo "[-] Error: The target directory contains subdirectories - please fix." 1>&2
  rm -rf "$TRACE_DIR"
  exit 1
fi

# Check for the more efficient way to copy files...

if ln "$IN_DIR/$FIRST_FILE" "$TRACE_DIR/.link_test" 2>/dev/null; then
  CP_TOOL=ln
else
  CP_TOOL=cp
fi

# Make sure that we can actually get anything out of afl-showmap before we
# waste too much time.

echo "[*] Testing the target binary..."

if [ "$STDIN_FILE" = "" ]; then

  AFL_CMIN_ALLOW_ANY=1 "$SHOWMAP" -m "$MEM_LIMIT" -t "$TIMEOUT" -o "$TRACE_DIR/.run_test" -Z
$EXTRA_PAR -- "$@" <"$IN_DIR/$FIRST_FILE"

else

  cp "$IN_DIR/$FIRST_FILE" "$STDIN_FILE"
  AFL_CMIN_ALLOW_ANY=1 "$SHOWMAP" -m "$MEM_LIMIT" -t "$TIMEOUT" -o "$TRACE_DIR/.run_test" -Z
$EXTRA_PAR -A "$STDIN_FILE" -- "$@" </dev/null

fi

FIRST_COUNT=$((`grep -c . "$TRACE_DIR/.run_test"`))

if [ "$FIRST_COUNT" -gt "0" ]; then

  echo "[+] OK, $FIRST_COUNT tuples recorded."
```

```
else

  echo "[-] Error: no instrumentation output detected (perhaps crash or timeout)." 1>&2
  test "$AFL_KEEP_TRACES" = "" && rm -rf "$TRACE_DIR"
  exit 1

fi

# Let's roll!

#############################
# STEP 1: COLLECTING TRACES #
#############################

echo "[*] Obtaining traces for input files in '$IN_DIR'..."

(

  CUR=0

  if [ "$STDIN_FILE" = "" ]; then

    while read -r fn; do

      CUR=$((CUR+1))
      printf "\\r    Processing file $CUR/$IN_COUNT... "

      "$SHOWMAP" -m "$MEM_LIMIT" -t "$TIMEOUT" -o "$TRACE_DIR/$fn" -Z $EXTRA_PAR -- "$@"
<"$IN_DIR/$fn"

    done < <(ls "$IN_DIR")

  else

    while read -r fn; do

      CUR=$((CUR+1))
      printf "\\r    Processing file $CUR/$IN_COUNT... "

      cp "$IN_DIR/$fn" "$STDIN_FILE"

      "$SHOWMAP" -m "$MEM_LIMIT" -t "$TIMEOUT" -o "$TRACE_DIR/$fn" -Z $EXTRA_PAR -A
"$STDIN_FILE" -- "$@" </dev/null

    done < <(ls "$IN_DIR")


  fi

)

echo

#########################
# STEP 2: SORTING TUPLES #
#########################

# With this out of the way, we sort all tuples by popularity across all
# datasets. The reasoning here is that we won't be able to avoid the files
```

```
# that trigger unique tuples anyway, so we will want to start with them and
# see what's left.

echo "[*] Sorting trace sets (this may take a while)..."

ls "$IN_DIR" | sed "s#^#$TRACE_DIR/#" | tr '\n' '\0' | xargs -0 -n 1 cat | \
  sort | uniq -c | sort -n >"$TRACE_DIR/.all_uniq"

TUPLE_COUNT=$(( `grep -c . "$TRACE_DIR/.all_uniq"` ))

echo "[+] Found $TUPLE_COUNT unique tuples across $IN_COUNT files."

#####################################
# STEP 3: SELECTING CANDIDATE FILES #
#####################################

# The next step is to find the best candidate for each tuple. The "best"
# part is understood simply as the smallest input that includes a particular
# tuple in its trace. Empirical evidence suggests that this produces smaller
# datasets than more involved algorithms that could be still pulled off in
# a shell script.

echo "[*] Finding best candidates for each tuple..."

CUR=0

while read -r fn; do

  CUR=$((CUR+1))
  printf "\\r    Processing file $CUR/$IN_COUNT... "

  sed "s#\$# $fn#" "$TRACE_DIR/$fn" >>"$TRACE_DIR/.candidate_list"

done < <(ls -rS "$IN_DIR")

echo

##############################
# STEP 4: LOADING CANDIDATES #
##############################

# At this point, we have a file of tuple-file pairs, sorted by file size
# in ascending order (as a consequence of ls -rS). By doing sort keyed
# only by tuple (-k 1,1) and configured to output only the first line for
# every key (-s -u), we end up with the smallest file for each tuple.

echo "[*] Sorting candidate list (be patient)..."

sort -k1,1 -s -u "$TRACE_DIR/.candidate_list" | \
  sed 's/^/BEST_FILE[/;s/ /]="/;s/$/"/' >"$TRACE_DIR/.candidate_script"

if [ ! -s "$TRACE_DIR/.candidate_script" ]; then
  echo "[-] Error: no traces obtained from test cases, check syntax!" 1>&2
  test "$AFL_KEEP_TRACES" = "" && rm -rf "$TRACE_DIR"
  exit 1
fi

# The sed command converted the sorted list to a shell script that populates
# BEST_FILE[tuple]="fname". Let's load that!
```

```
. "$TRACE_DIR/.candidate_script"

##########################
# STEP 5: WRITING OUTPUT #
##########################

# The final trick is to grab the top pick for each tuple, unless said tuple is
# already set due to the inclusion of an earlier candidate; and then put all
# tuples associated with the newly-added file to the "already have" list. The
# loop works from least popular tuples and toward the most common ones.

echo "[*] Processing candidates and writing output files..."

CUR=0

touch "$TRACE_DIR/.already_have"

while read -r cnt tuple; do

  CUR=$((CUR+1))
  printf "\\r    Processing tuple $CUR/$TUPLE_COUNT... "

  # If we already have this tuple, skip it.

  grep -q "^$tuple\$" "$TRACE_DIR/.already_have" && continue

  FN=${BEST_FILE[tuple]}

  $CP_TOOL "$IN_DIR/$FN" "$OUT_DIR/$FN"

  if [ "$((CUR % 5))" = "0" ]; then
    sort -u "$TRACE_DIR/$FN" "$TRACE_DIR/.already_have" >"$TRACE_DIR/.tmp"
    mv -f "$TRACE_DIR/.tmp" "$TRACE_DIR/.already_have"
  else
    cat "$TRACE_DIR/$FN" >>"$TRACE_DIR/.already_have"
  fi

done <"$TRACE_DIR/.all_uniq"

echo

OUT_COUNT=`ls -- "$OUT_DIR" | wc -l`

if [ "$OUT_COUNT" = "1" ]; then
  echo "[!] WARNING: All test cases had the same traces, check syntax!"
fi

echo "[+] Narrowed down to $OUT_COUNT files, saved in '$OUT_DIR'."
echo

test "$AFL_KEEP_TRACES" = "" && rm -rf "$TRACE_DIR"

exit 0
```

afl-fuzz.c

```
/*
  Copyright 2013 Google LLC All rights reserved.
```

```
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - fuzzer code
   --------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   Forkserver design by Jann Horn <jannhorn@googlemail.com>

   This is the real deal: the program takes an instrumented binary and
   attempts a variety of basic fuzzing tricks, paying close attention to
   how they affect the execution path.

*/

#define AFL_MAIN
#include "android-ashmem.h"
#define MESSAGES_TO_STDOUT

#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#define _FILE_OFFSET_BITS 64

#include "config.h"
#include "types.h"
#include "debug.h"
#include "alloc-inl.h"
#include "hash.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <dirent.h>
#include <ctype.h>
#include <fcntl.h>
#include <termios.h>
#include <dlfcn.h>
#include <sched.h>

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/shm.h>
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/file.h>

#if defined(__APPLE__) || defined(__FreeBSD__) || defined (__OpenBSD__)
#  include <sys/sysctl.h>
#endif /* __APPLE__ || __FreeBSD__ || __OpenBSD__ */

/* For systems that have sched_setaffinity; right now just Linux, but one
   can hope... */

#ifdef __linux__
#  define HAVE_AFFINITY 1
#endif /* __linux__ */

/* A toggle to export some variables when building as a library. Not very
   useful for the general public. */

#ifdef AFL_LIB
#  define EXP_ST
#else
#  define EXP_ST static
#endif /* ^AFL_LIB */

/* Lots of globals, but mostly for the status UI and other things where it
   really makes no sense to haul them around as function parameters. */


EXP_ST u8 *in_dir,                    /* Input directory with test cases  */
          *out_file,                  /* File to fuzz, if any             */
          *out_dir,                   /* Working & output directory       */
          *sync_dir,                  /* Synchronization directory        */
          *sync_id,                   /* Fuzzer ID                        */
          *use_banner,                /* Display banner                   */
          *in_bitmap,                 /* Input bitmap                     */
          *doc_path,                  /* Path to documentation dir        */
          *target_path,               /* Path to target binary            */
          *orig_cmdline;              /* Original command line            */

EXP_ST u32 exec_tmout = EXEC_TIMEOUT; /* Configurable exec timeout (ms)   */
static u32 hang_tmout = EXEC_TIMEOUT; /* Timeout used for hang det (ms)   */

EXP_ST u64 mem_limit  = MEM_LIMIT;    /* Memory cap for child (MB)        */

EXP_ST u32 cpu_to_bind = 0;           /* id of free CPU core to bind      */

static u32 stats_update_freq = 1;     /* Stats update frequency (execs)   */

EXP_ST u8  skip_deterministic,        /* Skip deterministic stages?       */
           force_deterministic,       /* Force deterministic stages?      */
           use_splicing,              /* Recombine input files?           */
           dumb_mode,                 /* Run in non-instrumented mode?    */
           score_changed,             /* Scoring for favorites changed?   */
           kill_signal,               /* Signal that killed the child     */
           resuming_fuzz,             /* Resuming an older fuzzing job?   */
           timeout_given,             /* Specific timeout given?          */
           cpu_to_bind_given,         /* Specified cpu_to_bind given?     */
```

```
           not_on_tty,               /* stdout is not a tty             */
           term_too_small,           /* terminal dimensions too small   */
           uses_asan,                /* Target uses ASAN?               */
           no_forkserver,            /* Disable forkserver?             */
           crash_mode,               /* Crash mode! Yeah!               */
           in_place_resume,          /* Attempt in-place resume?        */
           auto_changed,             /* Auto-generated tokens changed?  */
           no_cpu_meter_red,         /* Feng shui on the status screen  */
           no_arith,                 /* Skip most arithmetic ops        */
           shuffle_queue,            /* Shuffle input queue?            */
           bitmap_changed = 1,       /* Time to update bitmap?          */
           qemu_mode,                /* Running in QEMU mode?           */
           skip_requested,           /* Skip request, via SIGUSR1       */
           run_over10m,              /* Run time over 10 minutes?       */
           persistent_mode,          /* Running in persistent mode?     */
           deferred_mode,            /* Deferred forkserver mode?       */
           fast_cal;                 /* Try to calibrate faster?        */

static s32 out_fd,                   /* Persistent fd for out_file      */
           dev_urandom_fd = -1,      /* Persistent fd for /dev/urandom  */
           dev_null_fd = -1,         /* Persistent fd for /dev/null     */
           fsrv_ctl_fd,              /* Fork server control pipe (write) */
           fsrv_st_fd;               /* Fork server status pipe (read)  */

static s32 forksrv_pid,              /* PID of the fork server          */
           child_pid = -1,           /* PID of the fuzzed program       */
           out_dir_fd = -1;          /* FD of the lock file             */

EXP_ST u8* trace_bits;               /* SHM with instrumentation bitmap  */

EXP_ST u8  virgin_bits[MAP_SIZE],    /* Regions yet untouched by fuzzing */
           virgin_tmout[MAP_SIZE],   /* Bits we haven't seen in tmouts   */
           virgin_crash[MAP_SIZE];   /* Bits we haven't seen in crashes  */

static u8  var_bytes[MAP_SIZE];      /* Bytes that appear to be variable */

static s32 shm_id;                   /* ID of the SHM region             */

static volatile u8 stop_soon,        /* Ctrl-C pressed?                  */
                   clear_screen = 1, /* Window resized?                  */
                   child_timed_out;  /* Traced process timed out?        */

EXP_ST u32 queued_paths,             /* Total number of queued testcases */
           queued_variable,          /* Testcases with variable behavior */
           queued_at_start,          /* Total number of initial inputs   */
           queued_discovered,        /* Items discovered during this run */
           queued_imported,          /* Items imported via -S            */
           queued_favored,           /* Paths deemed favorable           */
           queued_with_cov,          /* Paths with new coverage bytes    */
           pending_not_fuzzed,       /* Queued but not done yet          */
           pending_favored,          /* Pending favored paths            */
           cur_skipped_paths,        /* Abandoned inputs in cur cycle    */
           cur_depth,                /* Current path depth               */
           max_depth,                /* Max path depth                   */
           useless_at_start,         /* Number of useless starting paths */
           var_byte_count,           /* Bitmap bytes with var behavior   */
           current_entry,            /* Current queue entry ID           */
           havoc_div = 1;            /* Cycle count divisor for havoc    */

EXP_ST u64 total_crashes,            /* Total number of crashes          */
```

```
           unique_crashes,            /* Crashes with unique signatures   */
           total_tmouts,              /* Total number of timeouts         */
           unique_tmouts,             /* Timeouts with unique signatures  */
           unique_hangs,              /* Hangs with unique signatures     */
           total_execs,               /* Total execve() calls             */
           slowest_exec_ms,           /* Slowest testcase non hang in ms  */
           start_time,                /* Unix start time (ms)             */
           last_path_time,            /* Time for most recent path (ms)   */
           last_crash_time,           /* Time for most recent crash (ms)  */
           last_hang_time,            /* Time for most recent hang (ms)   */
           last_crash_execs,          /* Exec counter at last crash       */
           queue_cycle,               /* Queue round counter              */
           cycles_wo_finds,           /* Cycles without any new paths     */
           trim_execs,                /* Execs done to trim input files   */
           bytes_trim_in,             /* Bytes coming into the trimmer    */
           bytes_trim_out,            /* Bytes coming outa the trimmer    */
           blocks_eff_total,          /* Blocks subject to effector maps  */
           blocks_eff_select;         /* Blocks selected as fuzzable      */

static u32 subseq_tmouts;             /* Number of timeouts in a row      */

static u8 *stage_name = "init",       /* Name of the current fuzz stage   */
          *stage_short,               /* Short stage name                 */
          *syncing_party;             /* Currently syncing with...        */

static s32 stage_cur, stage_max;      /* Stage progression                */
static s32 splicing_with = -1;        /* Splicing with which test case?   */

static u32 master_id, master_max;     /* Master instance job splitting    */

static u32 syncing_case;              /* Syncing with case #...           */

static s32 stage_cur_byte,            /* Byte offset of current stage op  */
           stage_cur_val;             /* Value used for stage op          */

static u8  stage_val_type;            /* Value type (STAGE_VAL_*)         */

static u64 stage_finds[32],           /* Patterns found per fuzz stage    */
           stage_cycles[32];          /* Execs per fuzz stage             */

static u32 rand_cnt;                  /* Random number counter            */

static u64 total_cal_us,              /* Total calibration time (us)      */
           total_cal_cycles;          /* Total calibration cycles         */

static u64 total_bitmap_size,         /* Total bit count for all bitmaps  */
           total_bitmap_entries;      /* Number of bitmaps counted        */

static s32 cpu_core_count;            /* CPU core count                   */

#ifdef HAVE_AFFINITY

static s32 cpu_aff = -1;              /* Selected CPU core                */

#endif /* HAVE_AFFINITY */

static FILE* plot_file;               /* Gnuplot output file              */

struct queue_entry {
```

```
  u8* fname;                          /* File name for the test case      */
  u32 len;                            /* Input length                     */

  u8  cal_failed,                     /* Calibration failed?              */
      trim_done,                      /* Trimmed?                         */
      was_fuzzed,                     /* Had any fuzzing done yet?        */
      passed_det,                     /* Deterministic stages passed?     */
      has_new_cov,                    /* Triggers new coverage?           */
      var_behavior,                   /* Variable behavior?               */
      favored,                        /* Currently favored?               */
      fs_redundant;                   /* Marked as redundant in the fs?   */

  u32 bitmap_size,                    /* Number of bits set in bitmap     */
      exec_cksum;                     /* Checksum of the execution trace  */

  u64 exec_us,                        /* Execution time (us)              */
      handicap,                       /* Number of queue cycles behind    */
      depth;                          /* Path depth                       */

  u8* trace_mini;                     /* Trace bytes, if kept             */
  u32 tc_ref;                         /* Trace bytes ref count            */

  struct queue_entry *next,           /* Next element, if any             */
                     *next_100;       /* 100 elements ahead               */

};

static struct queue_entry *queue,     /* Fuzzing queue (linked list)      */
                          *queue_cur, /* Current offset within the queue  */
                          *queue_top, /* Top of the list                  */
                          *q_prev100; /* Previous 100 marker              */

static struct queue_entry*
  top_rated[MAP_SIZE];                /* Top entries for bitmap bytes     */

struct extra_data {
  u8* data;                           /* Dictionary token data            */
  u32 len;                            /* Dictionary token length          */
  u32 hit_cnt;                        /* Use count in the corpus          */
};

static struct extra_data* extras;     /* Extra tokens to fuzz with        */
static u32 extras_cnt;                /* Total number of tokens read      */

static struct extra_data* a_extras;   /* Automatically selected extras    */
static u32 a_extras_cnt;              /* Total number of tokens available */

static u8* (*post_handler)(u8* buf, u32* len);

/* Interesting values, as per config.h */

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

/* Fuzzing stages */

enum {
  /* 00 */ STAGE_FLIP1,
  /* 01 */ STAGE_FLIP2,
```

```
  /* 02 */ STAGE_FLIP4,
  /* 03 */ STAGE_FLIP8,
  /* 04 */ STAGE_FLIP16,
  /* 05 */ STAGE_FLIP32,
  /* 06 */ STAGE_ARITH8,
  /* 07 */ STAGE_ARITH16,
  /* 08 */ STAGE_ARITH32,
  /* 09 */ STAGE_INTEREST8,
  /* 10 */ STAGE_INTEREST16,
  /* 11 */ STAGE_INTEREST32,
  /* 12 */ STAGE_EXTRAS_UO,
  /* 13 */ STAGE_EXTRAS_UI,
  /* 14 */ STAGE_EXTRAS_AO,
  /* 15 */ STAGE_HAVOC,
  /* 16 */ STAGE_SPLICE
};

/* Stage value types */

enum {
  /* 00 */ STAGE_VAL_NONE,
  /* 01 */ STAGE_VAL_LE,
  /* 02 */ STAGE_VAL_BE
};

/* Execution status fault codes */

enum {
  /* 00 */ FAULT_NONE,
  /* 01 */ FAULT_TMOUT,
  /* 02 */ FAULT_CRASH,
  /* 03 */ FAULT_ERROR,
  /* 04 */ FAULT_NOINST,
  /* 05 */ FAULT_NOBITS
};


/* Get unix time in milliseconds */

static u64 get_cur_time(void) {

  struct timeval tv;
  struct timezone tz;

  gettimeofday(&tv, &tz);

  return (tv.tv_sec * 1000ULL) + (tv.tv_usec / 1000);

}


/* Get unix time in microseconds */

static u64 get_cur_time_us(void) {

  struct timeval tv;
  struct timezone tz;

  gettimeofday(&tv, &tz);
```

```
  return (tv.tv_sec * 1000000ULL) + tv.tv_usec;

}


/* Generate a random number (from 0 to limit - 1). This may
   have slight bias. */

static inline u32 UR(u32 limit) {

  if (unlikely(!rand_cnt--)) {

    u32 seed[2];

    ck_read(dev_urandom_fd, &seed, sizeof(seed), "/dev/urandom");

    srandom(seed[0]);
    rand_cnt = (RESEED_RNG / 2) + (seed[1] % RESEED_RNG);

  }

  return random() % limit;

}


/* Shuffle an array of pointers. Might be slightly biased. */

static void shuffle_ptrs(void** ptrs, u32 cnt) {

  u32 i;

  for (i = 0; i < cnt - 2; i++) {

    u32 j = i + UR(cnt - i);
    void *s = ptrs[i];
    ptrs[i] = ptrs[j];
    ptrs[j] = s;

  }

}


#ifdef HAVE_AFFINITY

/* Build a list of processes bound to specific cores. Returns -1 if nothing
   can be found. Assumes an upper bound of 4k CPUs. */

static void bind_to_free_cpu(void) {

  DIR* d;
  struct dirent* de;
  cpu_set_t c;

  u8 cpu_used[4096] = { 0 };
  u32 i;

  if (cpu_core_count < 2) return;
```

```
if (getenv("AFL_NO_AFFINITY")) {

  WARNF("Not binding to a CPU core (AFL_NO_AFFINITY set).");
  return;

}

d = opendir("/proc");

if (!d) {

  WARNF("Unable to access /proc - can't scan for free CPU cores.");
  return;

}

ACTF("Checking CPU core loadout...");

/* Introduce some jitter, in case multiple AFL tasks are doing the same
   thing at the same time... */

usleep(R(1000) * 250);

/* Scan all /proc/<pid>/status entries, checking for Cpus_allowed_list.
   Flag all processes bound to a specific CPU using cpu_used[]. This will
   fail for some exotic binding setups, but is likely good enough in almost
   all real-world use cases. */

while ((de = readdir(d))) {

  u8* fn;
  FILE* f;
  u8 tmp[MAX_LINE];
  u8 has_vmsize = 0;

  if (!isdigit(de->d_name[0])) continue;

  fn = alloc_printf("/proc/%s/status", de->d_name);

  if (!(f = fopen(fn, "r"))) {
    ck_free(fn);
    continue;
  }

  while (fgets(tmp, MAX_LINE, f)) {

    u32 hval;

    /* Processes without VmSize are probably kernel tasks. */

    if (!strncmp(tmp, "VmSize:\t", 8)) has_vmsize = 1;

    if (!strncmp(tmp, "Cpus_allowed_list:\t", 19) &&
        !strchr(tmp, '-') && !strchr(tmp, ',') &&
        sscanf(tmp + 19, "%u", &hval) == 1 && hval < sizeof(cpu_used) &&
        has_vmsize) {

      cpu_used[hval] = 1;
      break;
```

```
      }

    }

    ck_free(fn);
    fclose(f);

  }

  closedir(d);
  if (cpu_to_bind_given) {

    if (cpu_to_bind >= cpu_core_count)
      FATAL("The CPU core id to bind should be between 0 and %u", cpu_core_count - 1);

    if (cpu_used[cpu_to_bind])
      FATAL("The CPU core #%u to bind is not free!", cpu_to_bind);

    i = cpu_to_bind;

  } else {

    for (i = 0; i < cpu_core_count; i++) if (!cpu_used[i]) break;

  }

  if (i == cpu_core_count) {

    SAYF("\n" cLRD "[-] " cRST
         "Uh-oh, looks like all %u CPU cores on your system are allocated to\n"
         "    other instances of afl-fuzz (or similar CPU-locked tasks). Starting\n"
         "    another fuzzer on this machine is probably a bad plan, but if you are\n"
         "    absolutely sure, you can set AFL_NO_AFFINITY and try again.\n",
         cpu_core_count);

    FATAL("No more free CPU cores");

  }

  OKF("Found a free CPU core, binding to #%u.", i);

  cpu_aff = i;

  CPU_ZERO(&c);
  CPU_SET(i, &c);

  if (sched_setaffinity(0, sizeof(c), &c))
    PFATAL("sched_setaffinity failed");

}

#endif /* HAVE_AFFINITY */

#ifndef IGNORE_FINDS

/* Helper function to compare buffers; returns first and last differing offset. We
   use this to find reasonable locations for splicing two files. */

static void locate_diffs(u8* ptr1, u8* ptr2, u32 len, s32* first, s32* last) {
```

```
  s32 f_loc = -1;
  s32 l_loc = -1;
  u32 pos;

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

    if (*(ptr1++) != *(ptr2++)) {

      if (f_loc == -1) f_loc = pos;
      l_loc = pos;

    }

  }

  *first = f_loc;
  *last = l_loc;

  return;

}

#endif /* !IGNORE_FINDS */


/* Describe integer. Uses 12 cyclic static buffers for return values. The value
   returned should be five characters or less for all the integers we reasonably
   expect to see. */

static u8* DI(u64 val) {

  static u8 tmp[12][16];
  static u8 cur;

  cur = (cur + 1) % 12;

#define CHK_FORMAT(_divisor, _limit_mult, _fmt, _cast) do { \
    if (val < (_divisor) * (_limit_mult)) { \
      sprintf(tmp[cur], _fmt, ((_cast)val) / (_divisor)); \
      return tmp[cur]; \
    } \
  } while (0)

  /* 0-9999 */
  CHK_FORMAT(1, 10000, "%llu", u64);

  /* 10.0k - 99.9k */
  CHK_FORMAT(1000, 99.95, "%0.01fk", double);

  /* 100k - 999k */
  CHK_FORMAT(1000, 1000, "%lluk", u64);

  /* 1.00M - 9.99M */
  CHK_FORMAT(1000 * 1000, 9.995, "%0.02fM", double);

  /* 10.0M - 99.9M */
  CHK_FORMAT(1000 * 1000, 99.95, "%0.01fM", double);

  /* 100M - 999M */
  CHK_FORMAT(1000 * 1000, 1000, "%lluM", u64);
```

```
  /* 1.00G - 9.99G */
  CHK_FORMAT(1000LL * 1000 * 1000, 9.995, "%0.02fG", double);

  /* 10.0G - 99.9G */
  CHK_FORMAT(1000LL * 1000 * 1000, 99.95, "%0.01fG", double);

  /* 100G - 999G */
  CHK_FORMAT(1000LL * 1000 * 1000, 1000, "%lluG", u64);

  /* 1.00T - 9.99G */
  CHK_FORMAT(1000LL * 1000 * 1000 * 1000, 9.995, "%0.02fT", double);

  /* 10.0T - 99.9T */
  CHK_FORMAT(1000LL * 1000 * 1000 * 1000, 99.95, "%0.01fT", double);

  /* 100T+ */
  strcpy(tmp[cur], "infty");
  return tmp[cur];

}


/* Describe float. Similar to the above, except with a single
   static buffer. */

static u8* DF(double val) {

  static u8 tmp[16];

  if (val < 99.995) {
    sprintf(tmp, "%0.02f", val);
    return tmp;
  }

  if (val < 999.95) {
    sprintf(tmp, "%0.01f", val);
    return tmp;
  }

  return DI((u64)val);

}


/* Describe integer as memory size. */

static u8* DMS(u64 val) {

  static u8 tmp[12][16];
  static u8 cur;

  cur = (cur + 1) % 12;

  /* 0-9999 */
  CHK_FORMAT(1, 10000, "%llu B", u64);

  /* 10.0k - 99.9k */
  CHK_FORMAT(1024, 99.95, "%0.01f kB", double);
```

```
  /* 100k - 999k */
  CHK_FORMAT(1024, 1000, "%llu kB", u64);

  /* 1.00M - 9.99M */
  CHK_FORMAT(1024 * 1024, 9.995, "%0.02f MB", double);

  /* 10.0M - 99.9M */
  CHK_FORMAT(1024 * 1024, 99.95, "%0.01f MB", double);

  /* 100M - 999M */
  CHK_FORMAT(1024 * 1024, 1000, "%llu MB", u64);

  /* 1.00G - 9.99G */
  CHK_FORMAT(1024LL * 1024 * 1024, 9.995, "%0.02f GB", double);

  /* 10.0G - 99.9G */
  CHK_FORMAT(1024LL * 1024 * 1024, 99.95, "%0.01f GB", double);

  /* 100G - 999G */
  CHK_FORMAT(1024LL * 1024 * 1024, 1000, "%llu GB", u64);

  /* 1.00T - 9.99G */
  CHK_FORMAT(1024LL * 1024 * 1024 * 1024, 9.995, "%0.02f TB", double);

  /* 10.0T - 99.9T */
  CHK_FORMAT(1024LL * 1024 * 1024 * 1024, 99.95, "%0.01f TB", double);

#undef CHK_FORMAT

  /* 100T+ */
  strcpy(tmp[cur], "infty");
  return tmp[cur];

}


/* Describe time delta. Returns one static buffer, 34 chars of less. */

static u8* DTD(u64 cur_ms, u64 event_ms) {

  static u8 tmp[64];
  u64 delta;
  s32 t_d, t_h, t_m, t_s;

  if (!event_ms) return "none seen yet";

  delta = cur_ms - event_ms;

  t_d = delta / 1000 / 60 / 60 / 24;
  t_h = (delta / 1000 / 60 / 60) % 24;
  t_m = (delta / 1000 / 60) % 60;
  t_s = (delta / 1000) % 60;

  sprintf(tmp, "%s days, %u hrs, %u min, %u sec", DI(t_d), t_h, t_m, t_s);
  return tmp;

}


/* Mark deterministic checks as done for a particular queue entry. We use the
```

```
   .state file to avoid repeating deterministic fuzzing when resuming aborted
   scans. */

static void mark_as_det_done(struct queue_entry* q) {

  u8* fn = strrchr(q->fname, '/');
  s32 fd;

  fn = alloc_printf("%s/queue/.state/deterministic_done/%s", out_dir, fn + 1);

  fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
  if (fd < 0) PFATAL("Unable to create '%s'", fn);
  close(fd);

  ck_free(fn);

  q->passed_det = 1;

}


/* Mark as variable. Create symlinks if possible to make it easier to examine
   the files. */

static void mark_as_variable(struct queue_entry* q) {

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

  ldest = alloc_printf("../../%s", fn);
  fn = alloc_printf("%s/queue/.state/variable_behavior/%s", out_dir, fn);

  if (symlink(ldest, fn)) {

    s32 fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
    if (fd < 0) PFATAL("Unable to create '%s'", fn);
    close(fd);

  }

  ck_free(ldest);
  ck_free(fn);

  q->var_behavior = 1;

}


/* Mark / unmark as redundant (edge-only). This is not used for restoring state,
   but may be useful for post-processing datasets. */

static void mark_as_redundant(struct queue_entry* q, u8 state) {

  u8* fn;
  s32 fd;

  if (state == q->fs_redundant) return;

  q->fs_redundant = state;

  fn = strrchr(q->fname, '/');
```

```
  fn = alloc_printf("%s/queue/.state/redundant_edges/%s", out_dir, fn + 1);

  if (state) {

    fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
    if (fd < 0) PFATAL("Unable to create '%s'", fn);
    close(fd);

  } else {

    if (unlink(fn)) PFATAL("Unable to remove '%s'", fn);

  }

  ck_free(fn);

}


/* Append new test case to the 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();

}


/* Destroy the entire queue. */
```

```
EXP_ST void destroy_queue(void) {

  struct queue_entry *q = queue, *n;

  while (q) {

    n = q->next;
    ck_free(q->fname);
    ck_free(q->trace_mini);
    ck_free(q);
    q = n;

  }

}


/* Write bitmap to file. The bitmap is useful mostly for the secret
   -B option, to focus a separate fuzzing session on a particular
   interesting input without rediscovering all the others. */

EXP_ST void write_bitmap(void) {

  u8* fname;
  s32 fd;

  if (!bitmap_changed) return;
  bitmap_changed = 0;

  fname = alloc_printf("%s/fuzz_bitmap", out_dir);
  fd = open(fname, O_WRONLY | O_CREAT | O_TRUNC, 0600);

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

  ck_write(fd, virgin_bits, MAP_SIZE, fname);

  close(fd);
  ck_free(fname);

}


/* Read bitmap from file. This is for the -B option again. */

EXP_ST void read_bitmap(u8* fname) {

  s32 fd = open(fname, O_RDONLY);

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

  ck_read(fd, virgin_bits, MAP_SIZE, fname);

  close(fd);

}


/* 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.
```

```
   Updates the map, so subsequent calls will always return 0.

   This function is called after every exec() on a fairly large buffer, so
   it needs to be fast. We do this in 32-bit and 64-bit flavors. */

static inline u8 has_new_bits(u8* virgin_map) {

#ifdef WORD_SIZE_64

  u64* current = (u64*)trace_bits;
  u64* virgin  = (u64*)virgin_map;

  u32  i = (MAP_SIZE >> 3);

#else

  u32* current = (u32*)trace_bits;
  u32* virgin  = (u32*)virgin_map;

  u32  i = (MAP_SIZE >> 2);

#endif /* ^WORD_SIZE_64 */

  u8   ret = 0;

  while (i--) {

    /* Optimize for (*current & *virgin) == 0 - i.e., no bits in current bitmap
       that have not been already cleared from the virgin map - since this will
       almost always be the case. */

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

      if (likely(ret < 2)) {

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

        /* Looks like we have not found any new bytes yet; see if any non-zero
           bytes in current[] are pristine in virgin[]. */

#ifdef WORD_SIZE_64

        if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff) ||
            (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff) ||
            (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff) ||
            (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff)) ret = 2;
        else ret = 1;

#else

        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;

#endif /* ^WORD_SIZE_64 */

      }

      *virgin &= ~*current;
```

```
    }

    current++;
    virgin++;

  }

  if (ret && virgin_map == virgin_bits) bitmap_changed = 1;

  return ret;

}


/* Count the number of bits set in the provided bitmap. Used for the status
   screen several times every second, does not have to be fast. */

static u32 count_bits(u8* mem) {

  u32* ptr = (u32*)mem;
  u32  i   = (MAP_SIZE >> 2);
  u32  ret = 0;

  while (i--) {

    u32 v = *(ptr++);

    /* This gets called on the inverse, virgin bitmap; optimize for sparse
       data. */

    if (v == 0xffffffff) {
      ret += 32;
      continue;
    }

    v -= ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    ret += (((v + (v >> 4)) & 0xF0F0F0F) * 0x01010101) >> 24;

  }

  return ret;

}


#define FF(_b)  (0xff << ((_b) << 3))

/* Count the number of bytes set in the bitmap. Called fairly sporadically,
   mostly to update the status screen or calibrate and examine confirmed
   new paths. */

static u32 count_bytes(u8* mem) {

  u32* ptr = (u32*)mem;
  u32  i   = (MAP_SIZE >> 2);
  u32  ret = 0;

  while (i--) {
```

```
    u32 v = *(ptr++);

    if (!v) continue;
    if (v & FF(0)) ret++;
    if (v & FF(1)) ret++;
    if (v & FF(2)) ret++;
    if (v & FF(3)) ret++;

  }

  return ret;

}


/* Count the number of non-255 bytes set in the bitmap. Used strictly for the
   status screen, several calls per second or so. */

static u32 count_non_255_bytes(u8* mem) {

  u32* ptr = (u32*)mem;
  u32  i   = (MAP_SIZE >> 2);
  u32  ret = 0;

  while (i--) {

    u32 v = *(ptr++);

    /* This is called on the virgin bitmap, so optimize for the most likely
       case. */

    if (v == 0xffffffff) continue;
    if ((v & FF(0)) != FF(0)) ret++;
    if ((v & FF(1)) != FF(1)) ret++;
    if ((v & FF(2)) != FF(2)) ret++;
    if ((v & FF(3)) != FF(3)) ret++;

  }

  return ret;

}


/* Destructively simplify trace by eliminating hit count information
   and replacing it with 0x80 or 0x01 depending on whether the tuple
   is hit or not. Called on every new crash or timeout, should be
   reasonably fast. */

static const u8 simplify_lookup[256] = {

  [0]         = 1,
  [1 ... 255] = 128

};

#ifdef WORD_SIZE_64

static void simplify_trace(u64* mem) {
```

```
  u32 i = MAP_SIZE >> 3;

  while (i--) {

    /* Optimize for sparse bitmaps. */

    if (unlikely(*mem)) {

      u8* mem8 = (u8*)mem;

      mem8[0] = simplify_lookup[mem8[0]];
      mem8[1] = simplify_lookup[mem8[1]];
      mem8[2] = simplify_lookup[mem8[2]];
      mem8[3] = simplify_lookup[mem8[3]];
      mem8[4] = simplify_lookup[mem8[4]];
      mem8[5] = simplify_lookup[mem8[5]];
      mem8[6] = simplify_lookup[mem8[6]];
      mem8[7] = simplify_lookup[mem8[7]];

    } else *mem = 0x0101010101010101ULL;

    mem++;

  }

}

#else

static void simplify_trace(u32* mem) {

  u32 i = MAP_SIZE >> 2;

  while (i--) {

    /* Optimize for sparse bitmaps. */

    if (unlikely(*mem)) {

      u8* mem8 = (u8*)mem;

      mem8[0] = simplify_lookup[mem8[0]];
      mem8[1] = simplify_lookup[mem8[1]];
      mem8[2] = simplify_lookup[mem8[2]];
      mem8[3] = simplify_lookup[mem8[3]];

    } else *mem = 0x01010101;

    mem++;
  }

}

#endif /* ^WORD_SIZE_64 */


/* 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

};

static u16 count_class_lookup16[65536];


EXP_ST void init_count_class16(void) {

  u32 b1, b2;

  for (b1 = 0; b1 < 256; b1++)
    for (b2 = 0; b2 < 256; b2++)
      count_class_lookup16[(b1 << 8) + b2] =
        (count_class_lookup8[b1] << 8) |
        count_class_lookup8[b2];

}


#ifdef WORD_SIZE_64

static inline void classify_counts(u64* mem) {

  u32 i = MAP_SIZE >> 3;

  while (i--) {

    /* Optimize for sparse bitmaps. */

    if (unlikely(*mem)) {

      u16* mem16 = (u16*)mem;

      mem16[0] = count_class_lookup16[mem16[0]];
      mem16[1] = count_class_lookup16[mem16[1]];
      mem16[2] = count_class_lookup16[mem16[2]];
      mem16[3] = count_class_lookup16[mem16[3]];

    }

    mem++;

  }

}

#else
```

```
static inline void classify_counts(u32* mem) {

  u32 i = MAP_SIZE >> 2;

  while (i--) {

    /* Optimize for sparse bitmaps. */

    if (unlikely(*mem)) {

      u16* mem16 = (u16*)mem;

      mem16[0] = count_class_lookup16[mem16[0]];
      mem16[1] = count_class_lookup16[mem16[1]];

    }

    mem++;

  }

}

#endif /* ^WORD_SIZE_64 */


/* Get rid of shared memory (atexit handler). */

static void remove_shm(void) {

  shmctl(shm_id, IPC_RMID, NULL);

}


/* Compact trace bytes into a smaller bitmap. We effectively just drop the
   count information here. This is called only sporadically, for some
   new paths. */

static void minimize_bits(u8* dst, u8* src) {

  u32 i = 0;

  while (i < MAP_SIZE) {

    if (*(src++)) dst[i >> 3] |= 1 << (i & 7);
    i++;

  }

}


/* When we bump into a new path, we call this to see if the path appears
   more "favorable" than any of the existing ones. The purpose of the
   "favorables" is to have a minimal set of paths that trigger all the bits
   seen in the bitmap so far, and focus on fuzzing them at the expense of
   the rest.

   The first step of the process is to maintain a list of top_rated[] entries
```

```
   for every byte in the bitmap. We win that slot if there is no previous
   contender, or if the contender has a more favorable speed x size factor. */

static void update_bitmap_score(struct queue_entry* q) {

  u32 i;
  u64 fav_factor = q->exec_us * q->len;

  /* 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++;

       if (!q->trace_mini) {
         q->trace_mini = ck_alloc(MAP_SIZE >> 3);
         minimize_bits(q->trace_mini, trace_bits);
       }

       score_changed = 1;

     }

}


/* The second part of the mechanism discussed above is a routine that
   goes over top_rated[] entries, and then sequentially grabs winners for
   previously-unseen bytes (temp_v) and marks them as favored, at least
   until the next run. The favored entries are given more air time during
   all fuzzing steps. */

static void cull_queue(void) {

  struct queue_entry* q;
  static u8 temp_v[MAP_SIZE >> 3];
  u32 i;
```

```
  if (dumb_mode || !score_changed) return;

  score_changed = 0;

  memset(temp_v, 255, MAP_SIZE >> 3);

  queued_favored  = 0;
  pending_favored = 0;

  q = queue;

  while (q) {
    q->favored = 0;
    q = q->next;
  }

  /* Let's see if anything in the bitmap isn't captured in temp_v.
     If yes, and if it has a top_rated[] contender, let's use it. */

  for (i = 0; i < MAP_SIZE; i++)
    if (top_rated[i] && (temp_v[i >> 3] & (1 << (i & 7)))) {

      u32 j = MAP_SIZE >> 3;

      /* Remove all bits belonging to the current entry from temp_v. */

      while (j--)
        if (top_rated[i]->trace_mini[j])
          temp_v[j] &= ~top_rated[i]->trace_mini[j];

      top_rated[i]->favored = 1;
      queued_favored++;

      if (!top_rated[i]->was_fuzzed) pending_favored++;

    }

  q = queue;

  while (q) {
    mark_as_redundant(q, !q->favored);
    q = q->next;
  }

}


/* 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 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");

}


/* Load postprocessor, if available. */

static void setup_post(void) {

  void* dh;
  u8* fn = getenv("AFL_POST_LIBRARY");
  u32 tlen = 6;

  if (!fn) return;

  ACTF("Loading postprocessor from '%s'...", fn);

  dh = dlopen(fn, RTLD_NOW);
  if (!dh) FATAL("%s", dlerror());

  post_handler = dlsym(dh, "afl_postprocess");
  if (!post_handler) FATAL("Symbol 'afl_postprocess' not found.");

  /* Do a quick test. It's better to segfault now than later =) */

  post_handler("hello", &tlen);

  OKF("Postprocessor installed successfully.");

}


/* Read all testcases from the input directory, then queue them for testing.
   Called at startup. */

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);

  }

  free(nl); /* not tracked */

  if (!queued_paths) {

    SAYF("\n" cLRD "[-] " cRST
         "Looks like there are no valid test cases in the input directory! The fuzzer\n"
         "    needs one or more test case to start with - ideally, a small file under\n"
         "    1 kB or so. The cases must be stored as regular files directly in the\n"
         "    input directory.\n");

    FATAL("No usable test cases in '%s'", in_dir);

  }

  last_path_time = 0;
  queued_at_start = queued_paths;

}


/* Helper function for load_extras. */

static int compare_extras_len(const void* p1, const void* p2) {
  struct extra_data *e1 = (struct extra_data*)p1,
                    *e2 = (struct extra_data*)p2;

  return e1->len - e2->len;
}

static int compare_extras_use_d(const void* p1, const void* p2) {
  struct extra_data *e1 = (struct extra_data*)p1,
                    *e2 = (struct extra_data*)p2;

  return e2->hit_cnt - e1->hit_cnt;
}


/* Read extras from a file, sort by size. */

static void load_extras_file(u8* fname, u32* min_len, u32* max_len,
                             u32 dict_level) {

  FILE* f;
  u8  buf[MAX_LINE];
  u8  *lptr;
  u32 cur_line = 0;
```

```
f = fopen(fname, "r");

if (!f) PFATAL("Unable to open '%s'", fname);

while ((lptr = fgets(buf, MAX_LINE, f))) {

  u8 *rptr, *wptr;
  u32 klen = 0;

  cur_line++;

  /* Trim on left and right. */

  while (isspace(*lptr)) lptr++;

  rptr = lptr + strlen(lptr) - 1;
  while (rptr >= lptr && isspace(*rptr)) rptr--;
  rptr++;
  *rptr = 0;

  /* Skip empty lines and comments. */

  if (!*lptr || *lptr == '#') continue;

  /* All other lines must end with '"', which we can consume. */

  rptr--;

  if (rptr < lptr || *rptr != '"')
    FATAL("Malformed name=\"value\" pair in line %u.", cur_line);

  *rptr = 0;

  /* Skip alphanumerics and dashes (label). */

  while (isalnum(*lptr) || *lptr == '_') lptr++;

  /* If @number follows, parse that. */

  if (*lptr == '@') {

    lptr++;
    if (atoi(lptr) > dict_level) continue;
    while (isdigit(*lptr)) lptr++;

  }

  /* Skip whitespace and = signs. */

  while (isspace(*lptr) || *lptr == '=') lptr++;

  /* Consume opening '"'. */

  if (*lptr != '"')
    FATAL("Malformed name=\"keyword\" pair in line %u.", cur_line);

  lptr++;

  if (!*lptr) FATAL("Empty keyword in line %u.", cur_line);
```

```
/* Okay, let's allocate memory and copy data between "...", handling
   \xNN escaping, \\, and \". */

extras = ck_realloc_block(extras, (extras_cnt + 1) *
           sizeof(struct extra_data));

wptr = extras[extras_cnt].data = ck_alloc(rptr - lptr);

while (*lptr) {

  char* hexdigits = "0123456789abcdef";

  switch (*lptr) {

    case 1 ... 31:
    case 128 ... 255:
      FATAL("Non-printable characters in line %u.", cur_line);

    case '\\':

      lptr++;

      if (*lptr == '\\' || *lptr == '"') {
        *(wptr++) = *(lptr++);
        klen++;
        break;
      }

      if (*lptr != 'x' || !isxdigit(lptr[1]) || !isxdigit(lptr[2]))
        FATAL("Invalid escaping (not \\xNN) in line %u.", cur_line);

      *(wptr++) =
        ((strchr(hexdigits, tolower(lptr[1])) - hexdigits) << 4) |
        (strchr(hexdigits, tolower(lptr[2])) - hexdigits);

      lptr += 3;
      klen++;

      break;

    default:

      *(wptr++) = *(lptr++);
      klen++;

  }

}

extras[extras_cnt].len = klen;

if (extras[extras_cnt].len > MAX_DICT_FILE)
  FATAL("Keyword too big in line %u (%s, limit is %s)", cur_line,
        DMS(klen), DMS(MAX_DICT_FILE));

if (*min_len > klen) *min_len = klen;
if (*max_len < klen) *max_len = klen;

extras_cnt++;
```

```
  }

  fclose(f);

}


/* Read extras from the extras directory and sort them by size. */

static void load_extras(u8* dir) {

  DIR* d;
  struct dirent* de;
  u32 min_len = MAX_DICT_FILE, max_len = 0, dict_level = 0;
  u8* x;

  /* If the name ends with @, extract level and continue. */

  if ((x = strchr(dir, '@'))) {

    *x = 0;
    dict_level = atoi(x + 1);

  }

  ACTF("Loading extra dictionary from '%s' (level %u)...", dir, dict_level);

  d = opendir(dir);

  if (!d) {

    if (errno == ENOTDIR) {
      load_extras_file(dir, &min_len, &max_len, dict_level);
      goto check_and_sort;
    }

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

  }

  if (x) FATAL("Dictionary levels not supported for directories.");

  while ((de = readdir(d))) {

    struct stat st;
    u8* fn = alloc_printf("%s/%s", dir, de->d_name);
    s32 fd;

    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) {

      ck_free(fn);
      continue;

    }

    if (st.st_size > MAX_DICT_FILE)
```

```
      FATAL("Extra '%s' is too big (%s, limit is %s)", fn,
            DMS(st.st_size), DMS(MAX_DICT_FILE));

    if (min_len > st.st_size) min_len = st.st_size;
    if (max_len < st.st_size) max_len = st.st_size;

    extras = ck_realloc_block(extras, (extras_cnt + 1) *
               sizeof(struct extra_data));

    extras[extras_cnt].data = ck_alloc(st.st_size);
    extras[extras_cnt].len  = st.st_size;

    fd = open(fn, O_RDONLY);

    if (fd < 0) PFATAL("Unable to open '%s'", fn);

    ck_read(fd, extras[extras_cnt].data, st.st_size, fn);

    close(fd);
    ck_free(fn);

    extras_cnt++;

  }

  closedir(d);

check_and_sort:

  if (!extras_cnt) FATAL("No usable files in '%s'", dir);

  qsort(extras, extras_cnt, sizeof(struct extra_data), compare_extras_len);

  OKF("Loaded %u extra tokens, size range %s to %s.", extras_cnt,
      DMS(min_len), DMS(max_len));

  if (max_len > 32)
    WARNF("Some tokens are relatively large (%s) - consider trimming.",
          DMS(max_len));

  if (extras_cnt > MAX_DET_EXTRAS)
    WARNF("More than %u tokens - will use them probabilistically.",
          MAX_DET_EXTRAS);

}




/* Helper function for maybe_add_auto() */

static inline u8 memcmp_nocase(u8* m1, u8* m2, u32 len) {

  while (len--) if (tolower(*(m1++)) ^ tolower(*(m2++))) return 1;
  return 0;

}


/* Maybe add automatic extra. */
```

```
static void maybe_add_auto(u8* mem, u32 len) {

  u32 i;

  /* Allow users to specify that they don't want auto dictionaries. */

  if (!MAX_AUTO_EXTRAS || !USE_AUTO_EXTRAS) return;

  /* Skip runs of identical bytes. */

  for (i = 1; i < len; i++)
    if (mem[0] ^ mem[i]) break;

  if (i == len) return;

  /* Reject builtin interesting values. */

  if (len == 2) {

    i = sizeof(interesting_16) >> 1;

    while (i--)
      if (*((u16*)mem) == interesting_16[i] ||
          *((u16*)mem) == SWAP16(interesting_16[i])) return;

  }

  if (len == 4) {

    i = sizeof(interesting_32) >> 2;

    while (i--)
      if (*((u32*)mem) == interesting_32[i] ||
          *((u32*)mem) == SWAP32(interesting_32[i])) return;

  }

  /* Reject anything that matches existing extras. Do a case-insensitive
     match. We optimize by exploiting the fact that extras[] are sorted
     by size. */

  for (i = 0; i < extras_cnt; i++)
    if (extras[i].len >= len) break;

  for (; i < extras_cnt && extras[i].len == len; i++)
    if (!memcmp_nocase(extras[i].data, mem, len)) return;

  /* Last but not least, check a_extras[] for matches. There are no
     guarantees of a particular sort order. */

  auto_changed = 1;

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

    if (a_extras[i].len == len && !memcmp_nocase(a_extras[i].data, mem, len)) {

      a_extras[i].hit_cnt++;
      goto sort_a_extras;
```

```
    }

  }

  /* At this point, looks like we're dealing with a new entry. So, let's
     append it if we have room. Otherwise, let's randomly evict some other
     entry from the bottom half of the list. */

  if (a_extras_cnt < MAX_AUTO_EXTRAS) {

    a_extras = ck_realloc_block(a_extras, (a_extras_cnt + 1) *
                                sizeof(struct extra_data));

    a_extras[a_extras_cnt].data = ck_memdup(mem, len);
    a_extras[a_extras_cnt].len  = len;
    a_extras_cnt++;

  } else {

    i = MAX_AUTO_EXTRAS / 2 +
        UR((MAX_AUTO_EXTRAS + 1) / 2);

    ck_free(a_extras[i].data);

    a_extras[i].data    = ck_memdup(mem, len);
    a_extras[i].len     = len;
    a_extras[i].hit_cnt = 0;

  }

sort_a_extras:

  /* First, sort all auto extras by use count, descending order. */

  qsort(a_extras, a_extras_cnt, sizeof(struct extra_data),
        compare_extras_use_d);

  /* Then, sort the top USE_AUTO_EXTRAS entries by size. */

  qsort(a_extras, MIN(USE_AUTO_EXTRAS, a_extras_cnt),
        sizeof(struct extra_data), compare_extras_len);

}


/* Save automatically generated extras. */

static void save_auto(void) {

  u32 i;

  if (!auto_changed) return;
  auto_changed = 0;

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

    u8* fn = alloc_printf("%s/queue/.state/auto_extras/auto_%06u", out_dir, i);
    s32 fd;

    fd = open(fn, O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

```
    if (fd < 0) PFATAL("Unable to create '%s'", fn);

    ck_write(fd, a_extras[i].data, a_extras[i].len, fn);

    close(fd);
    ck_free(fn);

  }

}


/* Load automatically generated extras. */

static void load_auto(void) {

  u32 i;

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

    u8  tmp[MAX_AUTO_EXTRA + 1];
    u8* fn = alloc_printf("%s/.state/auto_extras/auto_%06u", in_dir, i);
    s32 fd, len;

    fd = open(fn, O_RDONLY, 0600);

    if (fd < 0) {

      if (errno != ENOENT) PFATAL("Unable to open '%s'", fn);
      ck_free(fn);
      break;

    }

    /* We read one byte more to cheaply detect tokens that are too
       long (and skip them). */

    len = read(fd, tmp, MAX_AUTO_EXTRA + 1);

    if (len < 0) PFATAL("Unable to read from '%s'", fn);

    if (len >= MIN_AUTO_EXTRA && len <= MAX_AUTO_EXTRA)
      maybe_add_auto(tmp, len);

    close(fd);
    ck_free(fn);

  }

  if (i) OKF("Loaded %u auto-discovered dictionary tokens.", i);
  else OKF("No auto-generated dictionary tokens to reuse.");

}


/* Destroy extras. */

static void destroy_extras(void) {
```

```
  u32 i;

  for (i = 0; i < extras_cnt; i++)
    ck_free(extras[i].data);

  ck_free(extras);

  for (i = 0; i < a_extras_cnt; i++)
    ck_free(a_extras[i].data);

  ck_free(a_extras);

}


/* Spin up fork server (instrumented mode only). The idea is explained here:

   http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

   In essence, the instrumentation allows us to skip execve(), and just keep
   cloning a stopped child. So, we just execute once, and then send commands
   through a pipe. The other part of this logic is in afl-as.h. */

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");

}


/* Execute target application, monitoring for timeouts. Return status
   information. The called program will update trace_bits[]. */

static u8 run_target(char** argv, u32 timeout) {

  static struct itimerval it;
  static u32 prev_timed_out = 0;
  static u64 exec_ms = 0;

  int status = 0;
  u32 tb4;

  child_timed_out = 0;

  /* After this memset, trace_bits[] are effectively volatile, so we
     must prevent any earlier operations from venturing into that
     territory. */

  memset(trace_bits, 0, MAP_SIZE);
  MEM_BARRIER();

  /* 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(). There is a bit of code duplication between here and
     init_forkserver(), but c'est la vie. */

  if (dumb_mode == 1 || no_forkserver) {

    child_pid = fork();

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

    if (!child_pid) {

      struct rlimit r;

      if (mem_limit) {

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

#ifdef RLIMIT_AS

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

#else

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

#endif /* ^RLIMIT_AS */

      }

      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);

    }

    /* On Linux, would be faster to use O_CLOEXEC. Maybe TODO. */

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

    /* 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);

    setenv("MSAN_OPTIONS", "exit_code=" STRINGIFY(MSAN_ERROR) ":"
                           "symbolize=0:"
                           "msan_track_origins=0", 0);

    execv(target_path, argv);

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

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

  }

} 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?)");

}

/* Configure timeout, as requested by user, then wait for child to terminate. */

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

setitimer(ITIMER_REAL, &it, NULL);

/* The SIGALRM handler simply kills the child_pid and sets child_timed_out. */

if (dumb_mode == 1 || no_forkserver) {

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

} else {

  s32 res;

  if ((res = read(fsrv_st_fd, &status, 4)) != 4) {

    if (stop_soon) return 0;
    RPFATAL(res, "Unable to communicate with fork server (OOM?)");

  }

}

if (!WIFSTOPPED(status)) child_pid = 0;

getitimer(ITIMER_REAL, &it);
exec_ms = (u64) timeout - (it.it_value.tv_sec * 1000 +
                           it.it_value.tv_usec / 1000);

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

setitimer(ITIMER_REAL, &it, NULL);

total_execs++;

/* Any subsequent operations on trace_bits must not be moved by the
   compiler below this point. Past this location, trace_bits[] behave
   very normally and do not have to be treated as volatile. */

MEM_BARRIER();

tb4 = *(u32*)trace_bits;
```

```
#ifdef WORD_SIZE_64
  classify_counts((u64*)trace_bits);
#else
  classify_counts((u32*)trace_bits);
#endif /* ^WORD_SIZE_64 */

  prev_timed_out = child_timed_out;

  /* Report outcome to caller. */

  if (WIFSIGNALED(status) && !stop_soon) {

    kill_signal = WTERMSIG(status);

    if (child_timed_out && kill_signal == SIGKILL) return FAULT_TMOUT;

    return FAULT_CRASH;

  }

  /* A somewhat nasty hack for MSAN, which doesn't support abort_on_error and
     must use a special exit code. */

  if (uses_asan && WEXITSTATUS(status) == MSAN_ERROR) {
    kill_signal = 0;
    return FAULT_CRASH;
  }

  if ((dumb_mode == 1 || no_forkserver) && tb4 == EXEC_FAIL_SIG)
    return FAULT_ERROR;

  /* It makes sense to account for the slowest units only if the testcase was run
  under the user defined timeout. */
  if (!(timeout > exec_tmout) && (slowest_exec_ms < exec_ms)) {
    slowest_exec_ms = exec_ms;
  }

  return FAULT_NONE;

}


/* Write modified data to file for testing. If out_file is set, the old file
   is unlinked and a new one is created. Otherwise, out_fd is rewound and
   truncated. */

static void write_to_testcase(void* mem, u32 len) {

  s32 fd = out_fd;

  if (out_file) {

    unlink(out_file); /* Ignore errors. */

    fd = open(out_file, O_WRONLY | O_CREAT | O_EXCL, 0600);

    if (fd < 0) PFATAL("Unable to create '%s'", out_file);

  } else lseek(fd, 0, SEEK_SET);
```

```
  ck_write(fd, mem, len, out_file);

  if (!out_file) {

    if (ftruncate(fd, len)) PFATAL("ftruncate() failed");
    lseek(fd, 0, SEEK_SET);

  } else close(fd);

}


/* The same, but with an adjustable gap. Used for trimming. */

static void write_with_gap(void* mem, u32 len, u32 skip_at, u32 skip_len) {

  s32 fd = out_fd;
  u32 tail_len = len - skip_at - skip_len;

  if (out_file) {

    unlink(out_file); /* Ignore errors. */

    fd = open(out_file, O_WRONLY | O_CREAT | O_EXCL, 0600);

    if (fd < 0) PFATAL("Unable to create '%s'", out_file);

  } else lseek(fd, 0, SEEK_SET);

  if (skip_at) ck_write(fd, mem, skip_at, out_file);

  if (tail_len) ck_write(fd, mem + skip_at + skip_len, tail_len, out_file);

  if (!out_file) {

    if (ftruncate(fd, len - skip_len)) PFATAL("ftruncate() failed");
    lseek(fd, 0, SEEK_SET);

  } else close(fd);

}


static void show_stats(void);

/* Calibrate a new test case. This is done when processing the input directory
   to warn about flaky or otherwise problematic test cases early on; and when
   new paths are discovered to detect variable behavior and so on. */

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;

}


/* Examine map coverage. Called once, for first test case. */

static void check_map_coverage(void) {

  u32 i;

  if (count_bytes(trace_bits) < 100) return;

  for (i = (1 << (MAP_SIZE_POW2 - 1)); i < MAP_SIZE; i++)
    if (trace_bits[i]) return;

  WARNF("Recompile binary with newer version of afl to improve coverage!");

}


/* Perform dry run of all test cases to confirm that the app is working as
   expected. This is done only for the initial inputs, and only once. */

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);

    switch (res) {

      case FAULT_NONE:

        if (q == queue) check_map_coverage();

        if (crash_mode) FATAL("Test case '%s' does *NOT* crash", fn);

        break;

      case FAULT_TMOUT:

        if (timeout_given) {

          /* The -t nn+ syntax in the command line sets timeout_given to '2' and
             instructs afl-fuzz to tolerate but skip queue entries that time
             out. */

          if (timeout_given > 1) {
            WARNF("Test case results in a timeout (skipping)");
            q->cal_failed = CAL_CHANCES;
            cal_failures++;
            break;
          }

          SAYF("\n" cLRD "[-] " cRST
               "The program took more than %u ms to process one of the initial test cases.\n"
               "    Usually, the right thing to do is to relax the -t option - or to delete
it\n"
               "    altogether and allow the fuzzer to auto-calibrate. That said, if you
know\n"
               "    what you are doing and want to simply skip the unruly test cases,
append\n"
               "    '+' at the end of the value passed to -t ('-t %u+').\n", exec_tmout,
               exec_tmout);

          FATAL("Test case '%s' results in a timeout", fn);

        } else {

          SAYF("\n" cLRD "[-] " cRST
               "The program took more than %u ms to process one of the initial test cases.\n"
               "    This is bad news; raising the limit with the -t option is possible, but\n"
               "    will probably make the fuzzing process extremely slow.\n\n"
```

```
              "    If this test case is just a fluke, the other option is to just avoid it\n"
              "    altogether, and find one that is less of a CPU hog.\n", exec_tmout);

          FATAL("Test case '%s' results in a timeout", fn);

        }

      case FAULT_CRASH:

        if (crash_mode) break;

        if (skip_crashes) {
          WARNF("Test case results in a crash (skipping)");
          q->cal_failed = CAL_CHANCES;
          cal_failures++;
          break;
        }

        if (mem_limit) {

          SAYF("\n" cLRD "[-] " cRST
               "Oops, the program crashed with one of the test cases provided. There are\n"
               "    several possible explanations:\n\n"

               "    - The test case causes known crashes under normal working conditions.
If\n"
               "      so, please remove it. The fuzzer should be seeded with interesting\n"
               "      inputs - but not ones that cause an outright crash.\n\n"

               "    - The current memory limit (%s) is too low for this program, causing\n"
               "      it to die due to OOM when parsing valid files. To fix this, try\n"
               "      bumping it up with the -m setting in the command line. If in doubt,\n"
               "      try something along the lines of:\n\n"

#ifdef RLIMIT_AS
               "      ( ulimit -Sv $[%llu << 10]; /path/to/binary [...] <testcase )\n\n"
#else
               "      ( ulimit -Sd $[%llu << 10]; /path/to/binary [...] <testcase )\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. Also,\n"
               "      if you are using ASAN, see %s/notes_for_asan.txt.\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"
               "      binaries. To fix this, set AFL_NO_FORKSRV=1 in the environment.\n\n"

#endif /* __APPLE__ */

               "    - Least 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, doc_path);

        } else {
```

```
          SAYF("\n" cLRD "[-] " cRST
               "Oops, the program crashed with one of the test cases provided. There are\n"
               "    several possible explanations:\n\n"

               "    - The test case causes known crashes under normal working conditions.
If\n"
               "      so, please remove it. The fuzzer should be seeded with interesting\n"
               "      inputs - but not ones that cause an outright crash.\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"
               "      binaries. To fix this, set AFL_NO_FORKSRV=1 in the environment.\n\n"

#endif /* __APPLE__ */

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

        }

        FATAL("Test case '%s' results in a crash", fn);

      case FAULT_ERROR:

        FATAL("Unable to execute target application ('%s')", argv[0]);

      case FAULT_NOINST:

        FATAL("No instrumentation detected");

      case FAULT_NOBITS:

        useless_at_start++;

        if (!in_bitmap && !shuffle_queue)
          WARNF("No new instrumentation output, test case may be useless.");

        break;

    }

    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.");

}


/* Helper function: link() if possible, copy otherwise. */

static void link_or_copy(u8* old_path, u8* new_path) {

  s32 i = link(old_path, new_path);
  s32 sfd, dfd;
  u8* tmp;

  if (!i) return;

  sfd = open(old_path, O_RDONLY);
  if (sfd < 0) PFATAL("Unable to open '%s'", old_path);

  dfd = open(new_path, O_WRONLY | O_CREAT | O_EXCL, 0600);
  if (dfd < 0) PFATAL("Unable to create '%s'", new_path);

  tmp = ck_alloc(64 * 1024);

  while ((i = read(sfd, tmp, 64 * 1024)) > 0)
    ck_write(dfd, tmp, i, new_path);

  if (i < 0) PFATAL("read() failed");

  ck_free(tmp);
  close(sfd);
  close(dfd);

}


static void nuke_resume_dir(void);

/* Create hard links for input test cases in the output directory, choosing
   good names and pivoting accordingly. */

static void pivot_inputs(void) {

  struct queue_entry* q = queue;
  u32 id = 0;

  ACTF("Creating hard links for all input files...");

  while (q) {

    u8  *nfn, *rsl = strrchr(q->fname, '/');
    u32 orig_id;

    if (!rsl) rsl = q->fname; else rsl++;

    /* If the original file name conforms to the syntax and the recorded
```

```
       ID matches the one we'd assign, just use the original file name.
       This is valuable for resuming fuzzing runs. */

#ifndef SIMPLE_FILES
#  define CASE_PREFIX "id:"
#else
#  define CASE_PREFIX "id_"
#endif /* ^!SIMPLE_FILES */

    if (!strncmp(rsl, CASE_PREFIX, 3) &&
        sscanf(rsl + 3, "%06u", &orig_id) == 1 && orig_id == id) {

      u8* src_str;
      u32 src_id;

      resuming_fuzz = 1;
      nfn = alloc_printf("%s/queue/%s", out_dir, rsl);

      /* Since we're at it, let's also try to find parent and figure out the
         appropriate depth for this entry. */

      src_str = strchr(rsl + 3, ':');

      if (src_str && sscanf(src_str + 1, "%06u", &src_id) == 1) {

        struct queue_entry* s = queue;
        while (src_id-- && s) s = s->next;
        if (s) q->depth = s->depth + 1;

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

      }

    } else {

      /* No dice - invent a new name, capturing the original one as a
         substring. */

#ifndef SIMPLE_FILES

      u8* use_name = strstr(rsl, ",orig:");

      if (use_name) use_name += 6; else use_name = rsl;
      nfn = alloc_printf("%s/queue/id:%06u,orig:%s", out_dir, id, use_name);

#else

      nfn = alloc_printf("%s/queue/id_%06u", out_dir, id);

#endif /* ^!SIMPLE_FILES */

    }

    /* Pivot to the new queue entry. */

    link_or_copy(q->fname, nfn);
    ck_free(q->fname);
    q->fname = nfn;

    /* Make sure that the passed_det value carries over, too. */
```

```
    if (q->passed_det) mark_as_det_done(q);

    q = q->next;
    id++;

  }

  if (in_place_resume) nuke_resume_dir();

}


#ifndef SIMPLE_FILES

/* Construct a file name for a new test case, capturing the operation
   that led to its discovery. Uses a static buffer. */

static u8* describe_op(u8 hnb) {

  static u8 ret[256];

  if (syncing_party) {

    sprintf(ret, "sync:%s,src:%06u", syncing_party, syncing_case);

  } else {

    sprintf(ret, "src:%06u", current_entry);

    if (splicing_with >= 0)
      sprintf(ret + strlen(ret), "+%06u", splicing_with);

    sprintf(ret + strlen(ret), ",op:%s", stage_short);

    if (stage_cur_byte >= 0) {

      sprintf(ret + strlen(ret), ",pos:%u", stage_cur_byte);

      if (stage_val_type != STAGE_VAL_NONE)
        sprintf(ret + strlen(ret), ",val:%s%+d",
                (stage_val_type == STAGE_VAL_BE) ? "be:" : "",
                stage_cur_val);

    } else sprintf(ret + strlen(ret), ",rep:%u", stage_cur_val);

  }

  if (hnb == 2) strcat(ret, ",+cov");

  return ret;

}

#endif /* !SIMPLE_FILES */


/* Write a message accompanying the crash directory :-) */

static void write_crash_readme(void) {
```

```
  u8* fn = alloc_printf("%s/crashes/README.txt", out_dir);
  s32 fd;
  FILE* f;

  fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
  ck_free(fn);

  /* Do not die on errors here - that would be impolite. */

  if (fd < 0) return;

  f = fdopen(fd, "w");

  if (!f) {
    close(fd);
    return;
  }

  fprintf(f, "Command line used to find this crash:\n\n"

             "%s\n\n"

             "If you can't reproduce a bug outside of afl-fuzz, be sure to set the same\n"
             "memory limit. The limit used for this fuzzing session was %s.\n\n"

             "Need a tool to minimize test cases before investigating the crashes or
sending\n"
             "them to a vendor? Check out the afl-tmin that comes with the fuzzer!\n\n"

             "Found any cool bugs in open-source tools using afl-fuzz? If yes, please drop\n"
             "me a mail at <lcamtuf@coredump.cx> once the issues are fixed - I'd love to\n"
             "add your finds to the gallery at:\n\n"

             "  http://lcamtuf.coredump.cx/afl/\n\n"

             "Thanks :-)\n",

             orig_cmdline, DMS(mem_limit << 20)); /* ignore errors */

  fclose(f);

}


/* Check if the result of an execve() during routine fuzzing is interesting,
   save or queue the input test case for further analysis if so. Returns 1 if
   entry is saved, 0 otherwise. */

static u8 save_if_interesting(char** argv, void* mem, u32 len, u8 fault) {

  u8  *fn = "";
  u8  hnb;
  s32 fd;
  u8  keeping = 0, res;

  if (fault == crash_mode) {

    /* Keep only if there are new bits in the map, add to queue for
       future fuzzing, etc. */
```

```
    if (!(hnb = has_new_bits(virgin_bits))) {
      if (crash_mode) total_crashes++;
      return 0;
    }

#ifndef SIMPLE_FILES

    fn = alloc_printf("%s/queue/id:%06u,%s", out_dir, queued_paths,
                      describe_op(hnb));

#else

    fn = alloc_printf("%s/queue/id_%06u", out_dir, queued_paths);

#endif /* ^!SIMPLE_FILES */

    add_to_queue(fn, len, 0);

    if (hnb == 2) {
      queue_top->has_new_cov = 1;
      queued_with_cov++;
    }

    queue_top->exec_cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

    /* Try to calibrate inline; this also calls update_bitmap_score() when
       successful. */

    res = calibrate_case(argv, queue_top, mem, queue_cycle - 1, 0);

    if (res == FAULT_ERROR)
      FATAL("Unable to execute target application");

    fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
    if (fd < 0) PFATAL("Unable to create '%s'", fn);
    ck_write(fd, mem, len, fn);
    close(fd);

    keeping = 1;

  }

  switch (fault) {

    case FAULT_TMOUT:

      /* Timeouts are not very interesting, but we're still obliged to keep
         a handful of samples. We use the presence of new bits in the
         hang-specific bitmap as a signal of uniqueness. In "dumb" mode, we
         just keep everything. */

      total_tmouts++;

      if (unique_hangs >= KEEP_UNIQUE_HANG) return keeping;

      if (!dumb_mode) {

#ifdef WORD_SIZE_64
        simplify_trace((u64*)trace_bits);
```

```
#else
        simplify_trace((u32*)trace_bits);
#endif /* ^WORD_SIZE_64 */

        if (!has_new_bits(virgin_tmout)) return keeping;

      }

      unique_tmouts++;

      /* Before saving, we make sure that it's a genuine hang by re-running
         the target with a more generous timeout (unless the default timeout
         is already generous). */

      if (exec_tmout < hang_tmout) {

        u8 new_fault;
        write_to_testcase(mem, len);
        new_fault = run_target(argv, hang_tmout);

        /* A corner case that one user reported bumping into: increasing the
           timeout actually uncovers a crash. Make sure we don't discard it if
           so. */

        if (!stop_soon && new_fault == FAULT_CRASH) goto keep_as_crash;

        if (stop_soon || new_fault != FAULT_TMOUT) return keeping;

      }

#ifndef SIMPLE_FILES

      fn = alloc_printf("%s/hangs/id:%06llu,%s", out_dir,
                        unique_hangs, describe_op(0));

#else

      fn = alloc_printf("%s/hangs/id_%06llu", out_dir,
                        unique_hangs);

#endif /* ^!SIMPLE_FILES */

      unique_hangs++;

      last_hang_time = get_cur_time();

      break;

    case FAULT_CRASH:

keep_as_crash:

      /* This is handled in a manner roughly similar to timeouts,
         except for slightly different limits and no need to re-run test
         cases. */

      total_crashes++;

      if (unique_crashes >= KEEP_UNIQUE_CRASH) return keeping;
```

```
      if (!dumb_mode) {

#ifdef WORD_SIZE_64
        simplify_trace((u64*)trace_bits);
#else
        simplify_trace((u32*)trace_bits);
#endif /* ^WORD_SIZE_64 */

        if (!has_new_bits(virgin_crash)) return keeping;

      }

      if (!unique_crashes) write_crash_readme();

#ifndef SIMPLE_FILES

      fn = alloc_printf("%s/crashes/id:%06llu,sig:%02u,%s", out_dir,
                        unique_crashes, kill_signal, describe_op(0));

#else

      fn = alloc_printf("%s/crashes/id_%06llu_%02u", out_dir, unique_crashes,
                        kill_signal);

#endif /* ^!SIMPLE_FILES */

      unique_crashes++;

      last_crash_time = get_cur_time();
      last_crash_execs = total_execs;

      break;

    case FAULT_ERROR: FATAL("Unable to execute target application");

    default: return keeping;

  }

  /* If we're here, we apparently want to save the crash or hang
     test case, too. */

  fd = open(fn, O_WRONLY | O_CREAT | O_EXCL, 0600);
  if (fd < 0) PFATAL("Unable to create '%s'", fn);
  ck_write(fd, mem, len, fn);
  close(fd);

  ck_free(fn);

  return keeping;

}


/* When resuming, try to find the queue position to start from. This makes sense
   only when resuming, and when we can find the original fuzzer_stats. */

static u32 find_start_position(void) {

  static u8 tmp[4096]; /* Ought to be enough for anybody. */
```

```
  u8  *fn, *off;
  s32 fd, i;
  u32 ret;

  if (!resuming_fuzz) return 0;

  if (in_place_resume) fn = alloc_printf("%s/fuzzer_stats", out_dir);
  else fn = alloc_printf("%s/../fuzzer_stats", in_dir);

  fd = open(fn, O_RDONLY);
  ck_free(fn);

  if (fd < 0) return 0;

  i = read(fd, tmp, sizeof(tmp) - 1); (void)i; /* Ignore errors */
  close(fd);

  off = strstr(tmp, "cur_path          : ");
  if (!off) return 0;

  ret = atoi(off + 20);
  if (ret >= queued_paths) ret = 0;
  return ret;

}


/* The same, but for timeouts. The idea is that when resuming sessions without
   -t given, we don't want to keep auto-scaling the timeout over and over
   again to prevent it from growing due to random flukes. */

static void find_timeout(void) {

  static u8 tmp[4096]; /* Ought to be enough for anybody. */

  u8  *fn, *off;
  s32 fd, i;
  u32 ret;

  if (!resuming_fuzz) return;

  if (in_place_resume) fn = alloc_printf("%s/fuzzer_stats", out_dir);
  else fn = alloc_printf("%s/../fuzzer_stats", in_dir);

  fd = open(fn, O_RDONLY);
  ck_free(fn);

  if (fd < 0) return;

  i = read(fd, tmp, sizeof(tmp) - 1); (void)i; /* Ignore errors */
  close(fd);

  off = strstr(tmp, "exec_timeout      : ");
  if (!off) return;

  ret = atoi(off + 20);
  if (ret <= 4) return;

  exec_tmout = ret;
```

```
  timeout_given = 3;

}


/* Update stats file for unattended monitoring. */

static void write_stats_file(double bitmap_cvg, double stability, double eps) {

  static double last_bcvg, last_stab, last_eps;
  static struct rusage usage;

  u8* fn = alloc_printf("%s/fuzzer_stats", out_dir);
  s32 fd;
  FILE* f;

  fd = open(fn, O_WRONLY | O_CREAT | O_TRUNC, 0600);

  if (fd < 0) PFATAL("Unable to create '%s'", fn);

  ck_free(fn);

  f = fdopen(fd, "w");

  if (!f) PFATAL("fdopen() failed");

  /* Keep last values in case we're called from another context
     where exec/sec stats and such are not readily available. */

  if (!bitmap_cvg && !stability && !eps) {
    bitmap_cvg = last_bcvg;
    stability  = last_stab;
    eps        = last_eps;
  } else {
    last_bcvg = bitmap_cvg;
    last_stab = stability;
    last_eps  = eps;
  }

  fprintf(f, "start_time        : %llu\n"
             "last_update       : %llu\n"
             "fuzzer_pid        : %u\n"
             "cycles_done       : %llu\n"
             "execs_done        : %llu\n"
             "execs_per_sec     : %0.02f\n"
             "paths_total       : %u\n"
             "paths_favored     : %u\n"
             "paths_found       : %u\n"
             "paths_imported    : %u\n"
             "max_depth         : %u\n"
             "cur_path          : %u\n" /* Must match find_start_position() */
             "pending_favs      : %u\n"
             "pending_total     : %u\n"
             "variable_paths    : %u\n"
             "stability         : %0.02f%%\n"
             "bitmap_cvg        : %0.02f%%\n"
             "unique_crashes    : %llu\n"
             "unique_hangs      : %llu\n"
             "last_path         : %llu\n"
             "last_crash        : %llu\n"
```

```
              "last_hang         : %llu\n"
              "execs_since_crash : %llu\n"
              "exec_timeout      : %u\n" /* Must match find_timeout() */
              "afl_banner        : %s\n"
              "afl_version       : " VERSION "\n"
              "target_mode       : %s%s%s%s%s%s%s%s\n"
              "command_line      : %s\n"
              "slowest_exec_ms   : %llu\n",
              start_time / 1000, get_cur_time() / 1000, getpid(),
              queue_cycle ? (queue_cycle - 1) : 0, total_execs, eps,
              queued_paths, queued_favored, queued_discovered, queued_imported,
              max_depth, current_entry, pending_favored, pending_not_fuzzed,
              queued_variable, stability, bitmap_cvg, unique_crashes,
              unique_hangs, last_path_time / 1000, last_crash_time / 1000,
              last_hang_time / 1000, total_execs - last_crash_execs,
              exec_tmout, use_banner,
              qemu_mode ? "qemu " : "", dumb_mode ? " dumb " : "",
              no_forkserver ? "no_forksrv " : "", crash_mode ? "crash " : "",
              persistent_mode ? "persistent " : "", deferred_mode ? "deferred " : "",
              (qemu_mode || dumb_mode || no_forkserver || crash_mode ||
               persistent_mode || deferred_mode) ? "" : "default",
              orig_cmdline, slowest_exec_ms);
              /* ignore errors */

  /* Get rss value from the children
     We must have killed the forkserver process and called waitpid
     before calling getrusage */
  if (getrusage(RUSAGE_CHILDREN, &usage)) {
      WARNF("getrusage failed");
  } else if (usage.ru_maxrss == 0) {
    fprintf(f, "peak_rss_mb       : not available while afl is running\n");
  } else {
#ifdef __APPLE__
    fprintf(f, "peak_rss_mb       : %zu\n", usage.ru_maxrss >> 20);
#else
    fprintf(f, "peak_rss_mb       : %zu\n", usage.ru_maxrss >> 10);
#endif /* ^__APPLE__ */
  }

  fclose(f);

}


/* Update the plot file if there is a reason to. */

static void maybe_update_plot_file(double bitmap_cvg, double eps) {

  static u32 prev_qp, prev_pf, prev_pnf, prev_ce, prev_md;
  static u64 prev_qc, prev_uc, prev_uh;

  if (prev_qp == queued_paths && prev_pf == pending_favored &&
      prev_pnf == pending_not_fuzzed && prev_ce == current_entry &&
      prev_qc == queue_cycle && prev_uc == unique_crashes &&
      prev_uh == unique_hangs && prev_md == max_depth) return;

  prev_qp  = queued_paths;
  prev_pf  = pending_favored;
  prev_pnf = pending_not_fuzzed;
  prev_ce  = current_entry;
```

```
  prev_qc  = queue_cycle;
  prev_uc  = unique_crashes;
  prev_uh  = unique_hangs;
  prev_md  = max_depth;

  /* Fields in the file:

     unix_time, cycles_done, cur_path, paths_total, paths_not_fuzzed,
     favored_not_fuzzed, unique_crashes, unique_hangs, max_depth,
     execs_per_sec */

  fprintf(plot_file,
          "%llu, %llu, %u, %u, %u, %u, %0.02f%%, %llu, %llu, %u, %0.02f\n",
          get_cur_time() / 1000, queue_cycle - 1, current_entry, queued_paths,
          pending_not_fuzzed, pending_favored, bitmap_cvg, unique_crashes,
          unique_hangs, max_depth, eps); /* ignore errors */

  fflush(plot_file);

}



/* A helper function for maybe_delete_out_dir(), deleting all prefixed
   files in a directory. */

static u8 delete_files(u8* path, u8* prefix) {

  DIR* d;
  struct dirent* d_ent;

  d = opendir(path);

  if (!d) return 0;

  while ((d_ent = readdir(d))) {

    if (d_ent->d_name[0] != '.' && (!prefix ||
        !strncmp(d_ent->d_name, prefix, strlen(prefix)))) {

      u8* fname = alloc_printf("%s/%s", path, d_ent->d_name);
      if (unlink(fname)) PFATAL("Unable to delete '%s'", fname);
      ck_free(fname);

    }

  }

  closedir(d);

  return !!rmdir(path);

}


/* Get the number of runnable processes, with some simple smoothing. */

static double get_runnable_processes(void) {

  static double res;
```

```
#if defined(__APPLE__) || defined(__FreeBSD__) || defined (__OpenBSD__)

  /* I don't see any portable sysctl or so that would quickly give us the
     number of runnable processes; the 1-minute load average can be a
     semi-decent approximation, though. */

  if (getloadavg(&res, 1) != 1) return 0;

#else

  /* On Linux, /proc/stat is probably the best way; load averages are
     computed in funny ways and sometimes don't reflect extremely short-lived
     processes well. */

  FILE* f = fopen("/proc/stat", "r");
  u8 tmp[1024];
  u32 val = 0;

  if (!f) return 0;

  while (fgets(tmp, sizeof(tmp), f)) {

    if (!strncmp(tmp, "procs_running ", 14) ||
        !strncmp(tmp, "procs_blocked ", 14)) val += atoi(tmp + 14);

  }

  fclose(f);

  if (!res) {

    res = val;

  } else {

    res = res * (1.0 - 1.0 / AVG_SMOOTHING) +
          ((double)val) * (1.0 / AVG_SMOOTHING);

  }

#endif /* ^(__APPLE__ || __FreeBSD__ || __OpenBSD__) */

  return res;

}


/* Delete the temporary directory used for in-place session resume. */

static void nuke_resume_dir(void) {

  u8* fn;

  fn = alloc_printf("%s/_resume/.state/deterministic_done", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/_resume/.state/auto_extras", out_dir);
  if (delete_files(fn, "auto_")) goto dir_cleanup_failed;
```

```
  ck_free(fn);

  fn = alloc_printf("%s/_resume/.state/redundant_edges", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/_resume/.state/variable_behavior", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/_resume/.state", out_dir);
  if (rmdir(fn) && errno != ENOENT) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/_resume", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  return;

dir_cleanup_failed:

  FATAL("_resume directory cleanup failed");

}


/* Delete fuzzer output directory if we recognize it as ours, if the fuzzer
   is not currently running, and if the last run time isn't too great. */

static void maybe_delete_out_dir(void) {

  FILE* f;
  u8 *fn = alloc_printf("%s/fuzzer_stats", out_dir);

  /* See if the output directory is locked. If yes, bail out. If not,
     create a lock that will persist for the lifetime of the process
     (this requires leaving the descriptor open).*/

  out_dir_fd = open(out_dir, O_RDONLY);
  if (out_dir_fd < 0) PFATAL("Unable to open '%s'", out_dir);

#ifndef __sun

  if (flock(out_dir_fd, LOCK_EX | LOCK_NB) && errno == EWOULDBLOCK) {

    SAYF("\n" cLRD "[-] " cRST
         "Looks like the job output directory is being actively used by another\n"
         "    instance of afl-fuzz. You will need to choose a different %s\n"
         "    or stop the other process first.\n",
         sync_id ? "fuzzer ID" : "output location");

    FATAL("Directory '%s' is in use", out_dir);

  }

#endif /* !__sun */

  f = fopen(fn, "r");
```

```
  if (f) {

    u64 start_time, last_update;

    if (fscanf(f, "start_time     : %llu\n"
                  "last_update    : %llu\n", &start_time, &last_update) != 2)
      FATAL("Malformed data in '%s'", fn);

    fclose(f);

    /* Let's see how much work is at stake. */

    if (!in_place_resume && last_update - start_time > OUTPUT_GRACE * 60) {

      SAYF("\n" cLRD "[-] " cRST
           "The job output directory already exists and contains the results of more\n"
           "    than %u minutes worth of fuzzing. To avoid data loss, afl-fuzz will *NOT*\n"
           "    automatically delete this data for you.\n\n"

           "    If you wish to start a new session, remove or rename the directory
manually,\n"
           "    or specify a different output location for this job. To resume the old\n"
           "    session, put '-' as the input directory in the command line ('-i -') and\n"
           "    try again.\n", OUTPUT_GRACE);

       FATAL("At-risk data found in '%s'", out_dir);

    }

  }

  ck_free(fn);

  /* The idea for in-place resume is pretty simple: we temporarily move the old
     queue/ to a new location that gets deleted once import to the new queue/
     is finished. If _resume/ already exists, the current queue/ may be
     incomplete due to an earlier abort, so we want to use the old _resume/
     dir instead, and we let rename() fail silently. */

  if (in_place_resume) {

    u8* orig_q = alloc_printf("%s/queue", out_dir);

    in_dir = alloc_printf("%s/_resume", out_dir);

    rename(orig_q, in_dir); /* Ignore errors */

    OKF("Output directory exists, will attempt session resume.");

    ck_free(orig_q);

  } else {

    OKF("Output directory exists but deemed OK to reuse.");

  }

  ACTF("Deleting old session data...");

  /* Okay, let's get the ball rolling! First, we need to get rid of the entries
```

```
     in <out_dir>/.synced/.../id:*, if any are present. */

  if (!in_place_resume) {

    fn = alloc_printf("%s/.synced", out_dir);
    if (delete_files(fn, NULL)) goto dir_cleanup_failed;
    ck_free(fn);

  }

  /* Next, we need to clean up <out_dir>/queue/.state/ subdirectories: */

  fn = alloc_printf("%s/queue/.state/deterministic_done", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/auto_extras", out_dir);
  if (delete_files(fn, "auto_")) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/redundant_edges", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/queue/.state/variable_behavior", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  /* Then, get rid of the .state subdirectory itself (should be empty by now)
     and everything matching <out_dir>/queue/id:*. */

  fn = alloc_printf("%s/queue/.state", out_dir);
  if (rmdir(fn) && errno != ENOENT) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/queue", out_dir);
  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  /* All right, let's do <out_dir>/crashes/id:* and <out_dir>/hangs/id:*. */

  if (!in_place_resume) {

    fn = alloc_printf("%s/crashes/README.txt", out_dir);
    unlink(fn); /* Ignore errors */
    ck_free(fn);

  }

  fn = alloc_printf("%s/crashes", out_dir);

  /* Make backup of the crashes directory if it's not empty and if we're
     doing in-place resume. */

  if (in_place_resume && rmdir(fn)) {

    time_t cur_t = time(0);
    struct tm* t = localtime(&cur_t);

#ifndef SIMPLE_FILES
```

```
    u8* nfn = alloc_printf("%s.%04u-%02u-%02u-%02u:%02u:%02u", fn,
                           t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                           t->tm_hour, t->tm_min, t->tm_sec);

#else

    u8* nfn = alloc_printf("%s_%04u%02u%02u%02u%02u%02u", fn,
                           t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                           t->tm_hour, t->tm_min, t->tm_sec);

#endif /* ^!SIMPLE_FILES */

    rename(fn, nfn); /* Ignore errors. */
    ck_free(nfn);

  }

  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/hangs", out_dir);

  /* Backup hangs, too. */

  if (in_place_resume && rmdir(fn)) {

    time_t cur_t = time(0);
    struct tm* t = localtime(&cur_t);

#ifndef SIMPLE_FILES

    u8* nfn = alloc_printf("%s.%04u-%02u-%02u-%02u:%02u:%02u", fn,
                           t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                           t->tm_hour, t->tm_min, t->tm_sec);

#else

    u8* nfn = alloc_printf("%s_%04u%02u%02u%02u%02u%02u", fn,
                           t->tm_year + 1900, t->tm_mon + 1, t->tm_mday,
                           t->tm_hour, t->tm_min, t->tm_sec);

#endif /* ^!SIMPLE_FILES */

    rename(fn, nfn); /* Ignore errors. */
    ck_free(nfn);

  }

  if (delete_files(fn, CASE_PREFIX)) goto dir_cleanup_failed;
  ck_free(fn);

  /* And now, for some finishing touches. */

  fn = alloc_printf("%s/.cur_input", out_dir);
  if (unlink(fn) && errno != ENOENT) goto dir_cleanup_failed;
  ck_free(fn);

  fn = alloc_printf("%s/fuzz_bitmap", out_dir);
  if (unlink(fn) && errno != ENOENT) goto dir_cleanup_failed;
```

```
  ck_free(fn);

  if (!in_place_resume) {
    fn  = alloc_printf("%s/fuzzer_stats", out_dir);
    if (unlink(fn) && errno != ENOENT) goto dir_cleanup_failed;
    ck_free(fn);
  }

  fn = alloc_printf("%s/plot_data", out_dir);
  if (unlink(fn) && errno != ENOENT) goto dir_cleanup_failed;
  ck_free(fn);

  OKF("Output dir cleanup successful.");

  /* Wow... is that all? If yes, celebrate! */

  return;

dir_cleanup_failed:

  SAYF("\n" cLRD "[-] " cRST
       "Whoops, the fuzzer tried to reuse your output directory, but bumped into\n"
       "    some files that shouldn't be there or that couldn't be removed - so it\n"
       "    decided to abort! This happened while processing this path:\n\n"

       "    %s\n\n"
       "    Please examine and manually delete the files, or specify a different\n"
       "    output location for the tool.\n", fn);

  FATAL("Output directory cleanup failed");

}


static void check_term_size(void);


/* A spiffy retro stats screen! This is called every stats_update_freq
   execve() calls, plus in several other circumstances. */

static void show_stats(void) {

  static u64 last_stats_ms, last_plot_ms, last_ms, last_execs;
  static double avg_exec;
  double t_byte_ratio, stab_ratio;

  u64 cur_ms;
  u32 t_bytes, t_bits;

  u32 banner_len, banner_pad;
  u8  tmp[256];

  cur_ms = get_cur_time();

  /* If not enough time has passed since last UI update, bail out. */

  if (cur_ms - last_ms < 1000 / UI_TARGET_HZ) return;

  /* Check if we're past the 10 minute mark. */
```

```
if (cur_ms - start_time > 10 * 60 * 1000) run_over10m = 1;

/* Calculate smoothed exec speed stats. */

if (!last_execs) {

  avg_exec = ((double)total_execs) * 1000 / (cur_ms - start_time);

} else {

  double cur_avg = ((double)(total_execs - last_execs)) * 1000 /
                   (cur_ms - last_ms);

  /* If there is a dramatic (5x+) jump in speed, reset the indicator
     more quickly. */

  if (cur_avg * 5 < avg_exec || cur_avg / 5 > avg_exec)
    avg_exec = cur_avg;

  avg_exec = avg_exec * (1.0 - 1.0 / AVG_SMOOTHING) +
             cur_avg * (1.0 / AVG_SMOOTHING);

}

last_ms = cur_ms;
last_execs = total_execs;

/* Tell the callers when to contact us (as measured in execs). */

stats_update_freq = avg_exec / (UI_TARGET_HZ * 10);
if (!stats_update_freq) stats_update_freq = 1;

/* Do some bitmap stats. */

t_bytes = count_non_255_bytes(virgin_bits);
t_byte_ratio = ((double)t_bytes * 100) / MAP_SIZE;

if (t_bytes)
  stab_ratio = 100 - ((double)var_byte_count) * 100 / t_bytes;
else
  stab_ratio = 100;

/* Roughly every minute, update fuzzer stats and save auto tokens. */

if (cur_ms - last_stats_ms > STATS_UPDATE_SEC * 1000) {

  last_stats_ms = cur_ms;
  write_stats_file(t_byte_ratio, stab_ratio, avg_exec);
  save_auto();
  write_bitmap();

}

/* Every now and then, write plot data. */

if (cur_ms - last_plot_ms > PLOT_UPDATE_SEC * 1000) {

  last_plot_ms = cur_ms;
  maybe_update_plot_file(t_byte_ratio, avg_exec);
```

```
  }

  /* Honor AFL_EXIT_WHEN_DONE and AFL_BENCH_UNTIL_CRASH. */

  if (!dumb_mode && cycles_wo_finds > 100 && !pending_not_fuzzed &&
      getenv("AFL_EXIT_WHEN_DONE")) stop_soon = 2;

  if (total_crashes && getenv("AFL_BENCH_UNTIL_CRASH")) stop_soon = 2;

  /* If we're not on TTY, bail out. */

  if (not_on_tty) return;

  /* Compute some mildly useful bitmap stats. */

  t_bits = (MAP_SIZE << 3) - count_bits(virgin_bits);

  /* Now, for the visuals... */

  if (clear_screen) {

    SAYF(TERM_CLEAR CURSOR_HIDE);
    clear_screen = 0;

    check_term_size();

  }

  SAYF(TERM_HOME);

  if (term_too_small) {

    SAYF(cBRI "Your terminal is too small to display the UI.\n"
         "Please resize terminal window to at least 80x25.\n" cRST);

    return;

  }

  /* Let's start by drawing a centered banner. */

  banner_len = (crash_mode ? 24 : 22) + strlen(VERSION) + strlen(use_banner);
  banner_pad = (80 - banner_len) / 2;
  memset(tmp, ' ', banner_pad);

  sprintf(tmp + banner_pad, "%s " cLCY VERSION cLGN
          " (%s)",  crash_mode ? cPIN "peruvian were-rabbit" :
          cYEL "american fuzzy lop", use_banner);

  SAYF("\n%s\n\n", tmp);

  /* "Handy" shortcuts for drawing boxes... */

#define bSTG    bSTART cGRA
#define bH2     bH bH
#define bH5     bH2 bH2 bH
#define bH10    bH5 bH5
#define bH20    bH10 bH10
#define bH30    bH20 bH10
#define SP5     "     "
```

```
#define SP10    SP5 SP5
#define SP20    SP10 SP10

  /* Lord, forgive me this. */

  SAYF(SET_G1 bSTG bLT bH bSTOP cCYA " process timing " bSTG bH30 bH5 bH2 bHB
       bH bSTOP cCYA " overall results " bSTG bH5 bRT "\n");

  if (dumb_mode) {

    strcpy(tmp, cRST);

  } else {

    u64 min_wo_finds = (cur_ms - last_path_time) / 1000 / 60;

    /* First queue cycle: don't stop now! */
    if (queue_cycle == 1 || min_wo_finds < 15) strcpy(tmp, cMGN); else

    /* Subsequent cycles, but we're still making finds. */
    if (cycles_wo_finds < 25 || min_wo_finds < 30) strcpy(tmp, cYEL); else

    /* No finds for a long time and no test cases to try. */
    if (cycles_wo_finds > 100 && !pending_not_fuzzed && min_wo_finds > 120)
      strcpy(tmp, cLGN);

    /* Default: cautiously OK to stop? */
    else strcpy(tmp, cLBL);

  }

  SAYF(bV bSTOP "        run time : " cRST "%-34s " bSTG bV bSTOP
       "  cycles done : %s%-5s  " bSTG bV "\n",
       DTD(cur_ms, start_time), tmp, DI(queue_cycle - 1));

  /* We want to warn people about not seeing new paths after a full cycle,
     except when resuming fuzzing or running in non-instrumented mode. */

  if (!dumb_mode && (last_path_time || resuming_fuzz || queue_cycle == 1 ||
      in_bitmap || crash_mode)) {

    SAYF(bV bSTOP "   last new path : " cRST "%-34s ",
         DTD(cur_ms, last_path_time));

  } else {

    if (dumb_mode)

      SAYF(bV bSTOP "   last new path : " cPIN "n/a" cRST
           " (non-instrumented mode)        ");

     else

      SAYF(bV bSTOP "   last new path : " cRST "none yet " cLRD
           "(odd, check syntax!)      ");

  }

  SAYF(bSTG bV bSTOP "  total paths : " cRST "%-5s  " bSTG bV "\n",
       DI(queued_paths));
```

```
/* Highlight crashes in red if found, denote going over the KEEP_UNIQUE_CRASH
   limit with a '+' appended to the count. */

sprintf(tmp, "%s%s", DI(unique_crashes),
        (unique_crashes >= KEEP_UNIQUE_CRASH) ? "+" : "");

SAYF(bV bSTOP " last uniq crash : " cRST "%-34s " bSTG bV bSTOP
     " uniq crashes : %s%-6s " bSTG bV "\n",
     DTD(cur_ms, last_crash_time), unique_crashes ? cLRD : cRST,
     tmp);

sprintf(tmp, "%s%s", DI(unique_hangs),
       (unique_hangs >= KEEP_UNIQUE_HANG) ? "+" : "");

SAYF(bV bSTOP "  last uniq hang : " cRST "%-34s " bSTG bV bSTOP
     "   uniq hangs : " cRST "%-6s " bSTG bV "\n",
     DTD(cur_ms, last_hang_time), tmp);

SAYF(bVR bH bSTOP cCYA " cycle progress " bSTG bH20 bHB bH bSTOP cCYA
     " map coverage " bSTG bH bHT bH20 bH2 bH bVL "\n");

/* This gets funny because we want to print several variable-length variables
   together, but then cram them into a fixed-width field - so we need to
   put them in a temporary buffer first. */

sprintf(tmp, "%s%s (%0.02f%%)", DI(current_entry),
        queue_cur->favored ? "" : "*",
        ((double)current_entry * 100) / queued_paths);

SAYF(bV bSTOP "  now processing : " cRST "%-17s " bSTG bV bSTOP, tmp);

sprintf(tmp, "%0.02f%% / %0.02f%%", ((double)queue_cur->bitmap_size) *
        100 / MAP_SIZE, t_byte_ratio);

SAYF("    map density : %s%-21s " bSTG bV "\n", t_byte_ratio > 70 ? cLRD :
     ((t_bytes < 200 && !dumb_mode) ? cPIN : cRST), tmp);

sprintf(tmp, "%s (%0.02f%%)", DI(cur_skipped_paths),
        ((double)cur_skipped_paths * 100) / queued_paths);

SAYF(bV bSTOP " paths timed out : " cRST "%-17s " bSTG bV, tmp);

sprintf(tmp, "%0.02f bits/tuple",
        t_bytes ? (((double)t_bits) / t_bytes) : 0);

SAYF(bSTOP " count coverage : " cRST "%-21s " bSTG bV "\n", tmp);

SAYF(bVR bH bSTOP cCYA " stage progress " bSTG bH20 bX bH bSTOP cCYA
     " findings in depth " bSTG bH20 bVL "\n");

sprintf(tmp, "%s (%0.02f%%)", DI(queued_favored),
        ((double)queued_favored) * 100 / queued_paths);

/* Yeah... it's still going on... halp? */

SAYF(bV bSTOP "  now trying : " cRST "%-21s " bSTG bV bSTOP
     " favored paths : " cRST "%-22s " bSTG bV "\n", stage_name, tmp);

if (!stage_max) {
```

```
  sprintf(tmp, "%s/-", DI(stage_cur));

} else {

  sprintf(tmp, "%s/%s (%0.02f%%)", DI(stage_cur), DI(stage_max),
          ((double)stage_cur) * 100 / stage_max);

}

SAYF(bV bSTOP " stage execs : " cRST "%-21s " bSTG bV bSTOP, tmp);

sprintf(tmp, "%s (%0.02f%%)", DI(queued_with_cov),
        ((double)queued_with_cov) * 100 / queued_paths);

SAYF("  new edges on : " cRST "%-22s " bSTG bV "\n", tmp);

sprintf(tmp, "%s (%s%s unique)", DI(total_crashes), DI(unique_crashes),
        (unique_crashes >= KEEP_UNIQUE_CRASH) ? "+" : "");

if (crash_mode) {

  SAYF(bV bSTOP " total execs : " cRST "%-21s " bSTG bV bSTOP
       "   new crashes : %s%-22s " bSTG bV "\n", DI(total_execs),
       unique_crashes ? cLRD : cRST, tmp);

} else {

  SAYF(bV bSTOP " total execs : " cRST "%-21s " bSTG bV bSTOP
       " total crashes : %s%-22s " bSTG bV "\n", DI(total_execs),
       unique_crashes ? cLRD : cRST, tmp);

}

/* Show a warning about slow execution. */

if (avg_exec < 100) {

  sprintf(tmp, "%s/sec (%s)", DF(avg_exec), avg_exec < 20 ?
          "zzzz..." : "slow!");

  SAYF(bV bSTOP "  exec speed : " cLRD "%-21s ", tmp);

} else {

  sprintf(tmp, "%s/sec", DF(avg_exec));
  SAYF(bV bSTOP "  exec speed : " cRST "%-21s ", tmp);

}

sprintf(tmp, "%s (%s%s unique)", DI(total_tmouts), DI(unique_tmouts),
        (unique_hangs >= KEEP_UNIQUE_HANG) ? "+" : "");

SAYF (bSTG bV bSTOP "  total tmouts : " cRST "%-22s " bSTG bV "\n", tmp);

/* Aaaalmost there... hold on! */

SAYF(bVR bH cCYA bSTOP " fuzzing strategy yields " bSTG bH10 bH bHT bH10
     bH5 bHB bH bSTOP cCYA " path geometry " bSTG bH5 bH2 bH bVL "\n");
```

```
if (skip_deterministic) {

  strcpy(tmp, "n/a, n/a, n/a");

} else {

  sprintf(tmp, "%s/%s, %s/%s, %s/%s",
          DI(stage_finds[STAGE_FLIP1]), DI(stage_cycles[STAGE_FLIP1]),
          DI(stage_finds[STAGE_FLIP2]), DI(stage_cycles[STAGE_FLIP2]),
          DI(stage_finds[STAGE_FLIP4]), DI(stage_cycles[STAGE_FLIP4]));

}

SAYF(bV bSTOP "   bit flips : " cRST "%-37s " bSTG bV bSTOP "    levels : "
     cRST "%-10s " bSTG bV "\n", tmp, DI(max_depth));

if (!skip_deterministic)
  sprintf(tmp, "%s/%s, %s/%s, %s/%s",
          DI(stage_finds[STAGE_FLIP8]), DI(stage_cycles[STAGE_FLIP8]),
          DI(stage_finds[STAGE_FLIP16]), DI(stage_cycles[STAGE_FLIP16]),
          DI(stage_finds[STAGE_FLIP32]), DI(stage_cycles[STAGE_FLIP32]));

SAYF(bV bSTOP "  byte flips : " cRST "%-37s " bSTG bV bSTOP "   pending : "
     cRST "%-10s " bSTG bV "\n", tmp, DI(pending_not_fuzzed));

if (!skip_deterministic)
  sprintf(tmp, "%s/%s, %s/%s, %s/%s",
          DI(stage_finds[STAGE_ARITH8]), DI(stage_cycles[STAGE_ARITH8]),
          DI(stage_finds[STAGE_ARITH16]), DI(stage_cycles[STAGE_ARITH16]),
          DI(stage_finds[STAGE_ARITH32]), DI(stage_cycles[STAGE_ARITH32]));

SAYF(bV bSTOP " arithmetics : " cRST "%-37s " bSTG bV bSTOP "  pend fav : "
     cRST "%-10s " bSTG bV "\n", tmp, DI(pending_favored));

if (!skip_deterministic)
  sprintf(tmp, "%s/%s, %s/%s, %s/%s",
          DI(stage_finds[STAGE_INTEREST8]), DI(stage_cycles[STAGE_INTEREST8]),
          DI(stage_finds[STAGE_INTEREST16]), DI(stage_cycles[STAGE_INTEREST16]),
          DI(stage_finds[STAGE_INTEREST32]), DI(stage_cycles[STAGE_INTEREST32]));

SAYF(bV bSTOP "  known ints : " cRST "%-37s " bSTG bV bSTOP " own finds : "
     cRST "%-10s " bSTG bV "\n", tmp, DI(queued_discovered));

if (!skip_deterministic)
  sprintf(tmp, "%s/%s, %s/%s, %s/%s",
          DI(stage_finds[STAGE_EXTRAS_UO]), DI(stage_cycles[STAGE_EXTRAS_UO]),
          DI(stage_finds[STAGE_EXTRAS_UI]), DI(stage_cycles[STAGE_EXTRAS_UI]),
          DI(stage_finds[STAGE_EXTRAS_AO]), DI(stage_cycles[STAGE_EXTRAS_AO]));

SAYF(bV bSTOP "  dictionary : " cRST "%-37s " bSTG bV bSTOP
     "  imported : " cRST "%-10s " bSTG bV "\n", tmp,
     sync_id ? DI(queued_imported) : (u8*)"n/a");

sprintf(tmp, "%s/%s, %s/%s",
        DI(stage_finds[STAGE_HAVOC]), DI(stage_cycles[STAGE_HAVOC]),
        DI(stage_finds[STAGE_SPLICE]), DI(stage_cycles[STAGE_SPLICE]));

SAYF(bV bSTOP "       havoc : " cRST "%-37s " bSTG bV bSTOP, tmp);

if (t_bytes) sprintf(tmp, "%0.02f%%", stab_ratio);
```

```
    else strcpy(tmp, "n/a");

  SAYF(" stability : %s%-10s " bSTG bV "\n", (stab_ratio < 85 && var_byte_count > 40)
       ? cLRD : ((queued_variable && (!persistent_mode || var_byte_count > 20))
       ? cMGN : cRST), tmp);

  if (!bytes_trim_out) {

    sprintf(tmp, "n/a, ");

  } else {

    sprintf(tmp, "%0.02f%%/%s, ",
            ((double)(bytes_trim_in - bytes_trim_out)) * 100 / bytes_trim_in,
            DI(trim_execs));

  }

  if (!blocks_eff_total) {

    u8 tmp2[128];

    sprintf(tmp2, "n/a");
    strcat(tmp, tmp2);

  } else {

    u8 tmp2[128];

    sprintf(tmp2, "%0.02f%%",
            ((double)(blocks_eff_total - blocks_eff_select)) * 100 /
            blocks_eff_total);

    strcat(tmp, tmp2);

  }

  SAYF(bV bSTOP "        trim : " cRST "%-37s " bSTG bVR bH20 bH2 bH2 bRB "\n"
       bLB bH30 bH20 bH2 bH bRB bSTOP cRST RESET_G1, tmp);

  /* Provide some CPU utilization stats. */

  if (cpu_core_count) {

    double cur_runnable = get_runnable_processes();
    u32 cur_utilization = cur_runnable * 100 / cpu_core_count;

    u8* cpu_color = cCYA;

    /* If we could still run one or more processes, use green. */

    if (cpu_core_count > 1 && cur_runnable + 1 <= cpu_core_count)
      cpu_color = cLGN;

    /* If we're clearly oversubscribed, use red. */

    if (!no_cpu_meter_red && cur_utilization >= 150) cpu_color = cLRD;

#ifdef HAVE_AFFINITY
```

```
    if (cpu_aff >= 0) {

      SAYF(SP10 cGRA "[cpu%03u:%s%3u%%" cGRA "]\r" cRST,
           MIN(cpu_aff, 999), cpu_color,
           MIN(cur_utilization, 999));

    } else {

      SAYF(SP10 cGRA "   [cpu:%s%3u%%" cGRA "]\r" cRST,
           cpu_color, MIN(cur_utilization, 999));

   }

#else

    SAYF(SP10 cGRA "   [cpu:%s%3u%%" cGRA "]\r" cRST,
         cpu_color, MIN(cur_utilization, 999));

#endif /* ^HAVE_AFFINITY */

  } else SAYF("\r");

  /* Hallelujah! */

  fflush(0);

}


/* Display quick statistics at the end of processing the input directory,
   plus a bunch of warnings. Some calibration stuff also ended up here,
   along with several hardcoded constants. Maybe clean up eventually. */

static void show_init_stats(void) {

  struct queue_entry* q = queue;
  u32 min_bits = 0, max_bits = 0;
  u64 min_us = 0, max_us = 0;
  u64 avg_us = 0;
  u32 max_len = 0;

  if (total_cal_cycles) avg_us = total_cal_us / total_cal_cycles;

  while (q) {

    if (!min_us || q->exec_us < min_us) min_us = q->exec_us;
    if (q->exec_us > max_us) max_us = q->exec_us;

    if (!min_bits || q->bitmap_size < min_bits) min_bits = q->bitmap_size;
    if (q->bitmap_size > max_bits) max_bits = q->bitmap_size;

    if (q->len > max_len) max_len = q->len;

    q = q->next;

  }

  SAYF("\n");

  if (avg_us > (qemu_mode ? 50000 : 10000))
```

```
  WARNF(cLRD "The target binary is pretty slow! See %s/perf_tips.txt.",
        doc_path);

/* Let's keep things moving with slow binaries. */

if (avg_us > 50000) havoc_div = 10;     /* 0-19 execs/sec   */
else if (avg_us > 20000) havoc_div = 5; /* 20-49 execs/sec  */
else if (avg_us > 10000) havoc_div = 2; /* 50-100 execs/sec */

if (!resuming_fuzz) {

  if (max_len > 50 * 1024)
    WARNF(cLRD "Some test cases are huge (%s) - see %s/perf_tips.txt!",
          DMS(max_len), doc_path);
  else if (max_len > 10 * 1024)
    WARNF("Some test cases are big (%s) - see %s/perf_tips.txt.",
          DMS(max_len), doc_path);

  if (useless_at_start && !in_bitmap)
    WARNF(cLRD "Some test cases look useless. Consider using a smaller set.");

  if (queued_paths > 100)
    WARNF(cLRD "You probably have far too many input files! Consider trimming down.");
  else if (queued_paths > 20)
    WARNF("You have lots of input files; try starting small.");

}

OKF("Here are some useful stats:\n\n"

    cGRA "    Test case count : " cRST "%u favored, %u variable, %u total\n"
    cGRA "       Bitmap range : " cRST "%u to %u bits (average: %0.02f bits)\n"
    cGRA "        Exec timing : " cRST "%s to %s us (average: %s us)\n",
    queued_favored, queued_variable, queued_paths, min_bits, max_bits,
    ((double)total_bitmap_size) / (total_bitmap_entries ? total_bitmap_entries : 1),
    DI(min_us), DI(max_us), DI(avg_us));

if (!timeout_given) {

  /* Figure out the appropriate timeout. The basic idea is: 5x average or
     1x max, rounded up to EXEC_TM_ROUND ms and capped at 1 second.

     If the program is slow, the multiplier is lowered to 2x or 3x, because
     random scheduler jitter is less likely to have any impact, and because
     our patience is wearing thin =) */

  if (avg_us > 50000) exec_tmout = avg_us * 2 / 1000;
  else if (avg_us > 10000) exec_tmout = avg_us * 3 / 1000;
  else exec_tmout = avg_us * 5 / 1000;

  exec_tmout = MAX(exec_tmout, max_us / 1000);
  exec_tmout = (exec_tmout + EXEC_TM_ROUND) / EXEC_TM_ROUND * EXEC_TM_ROUND;

  if (exec_tmout > EXEC_TIMEOUT) exec_tmout = EXEC_TIMEOUT;

  ACTF("No -t option specified, so I'll use exec timeout of %u ms.",
       exec_tmout);

  timeout_given = 1;
```

```
  } else if (timeout_given == 3) {

    ACTF("Applying timeout settings from resumed session (%u ms).", exec_tmout);

  }

  /* In dumb mode, re-running every timing out test case with a generous time
     limit is very expensive, so let's select a more conservative default. */

  if (dumb_mode && !getenv("AFL_HANG_TMOUT"))
    hang_tmout = MIN(EXEC_TIMEOUT, exec_tmout * 2 + 100);

  OKF("All set and ready to roll!");

}


/* Find first power of two greater or equal to val (assuming val under
   2^31). */

static u32 next_p2(u32 val) {

  u32 ret = 1;
  while (val > ret) ret <<= 1;
  return ret;

}


/* Trim all new test cases to save cycles when doing deterministic checks. The
   trimmer uses power-of-two increments somewhere between 1/16 and 1/1024 of
   file size, to keep the stage short and sweet. */

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;

}


/* Write a modified test case, run program, process results. Handle
   error conditions, returning 1 if it's time to bail out. This is
   a helper function for fuzz_one(). */

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;

}


/* Helper to choose random block len for block operations in fuzz_one().
   Doesn't return zero, provided that max_len is > 0. */

static u32 choose_block_len(u32 limit) {

  u32 min_value, max_value;
  u32 rlim = MIN(queue_cycle, 3);

  if (!run_over10m) rlim = 1;

  switch (UR(rlim)) {

    case 0:  min_value = 1;
             max_value = HAVOC_BLK_SMALL;
             break;

    case 1:  min_value = HAVOC_BLK_SMALL;
             max_value = HAVOC_BLK_MEDIUM;
             break;

    default:

             if (UR(10)) {

               min_value = HAVOC_BLK_MEDIUM;
               max_value = HAVOC_BLK_LARGE;

             } else {

               min_value = HAVOC_BLK_LARGE;
               max_value = HAVOC_BLK_XL;

             }
```

```
  }

  if (min_value >= limit) min_value = 1;

  return min_value + UR(MIN(max_value, limit) - min_value + 1);

}


/* Calculate case desirability score to adjust the length of havoc fuzzing.
   A helper function for fuzz_one(). Maybe some of these constants should
   go into config.h. */

static u32 calculate_score(struct queue_entry* q) {

  u32 avg_exec_us = total_cal_us / total_cal_cycles;
  u32 avg_bitmap_size = total_bitmap_size / total_bitmap_entries;
  u32 perf_score = 100;

  /* Adjust score based on execution speed of this path, compared to the
     global average. Multiplier ranges from 0.1x to 3x. Fast inputs are
     less expensive to fuzz, so we're giving them more air time. */

  if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
  else if (q->exec_us * 0.25 > avg_exec_us) perf_score = 25;
  else if (q->exec_us * 0.5 > avg_exec_us) perf_score = 50;
  else if (q->exec_us * 0.75 > avg_exec_us) perf_score = 75;
  else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
  else if (q->exec_us * 3 < avg_exec_us) perf_score = 200;
  else if (q->exec_us * 2 < avg_exec_us) perf_score = 150;

  /* Adjust score based on bitmap size. The working theory is that better
     coverage translates to better targets. Multiplier from 0.25x to 3x. */

  if (q->bitmap_size * 0.3 > avg_bitmap_size) perf_score *= 3;
  else if (q->bitmap_size * 0.5 > avg_bitmap_size) perf_score *= 2;
  else if (q->bitmap_size * 0.75 > avg_bitmap_size) perf_score *= 1.5;
  else if (q->bitmap_size * 3 < avg_bitmap_size) perf_score *= 0.25;
  else if (q->bitmap_size * 2 < avg_bitmap_size) perf_score *= 0.5;
  else if (q->bitmap_size * 1.5 < avg_bitmap_size) perf_score *= 0.75;

  /* Adjust score based on handicap. Handicap is proportional to how late
     in the game we learned about this path. Latecomers are allowed to run
     for a bit longer until they catch up with the rest. */

  if (q->handicap >= 4) {

    perf_score *= 4;
    q->handicap -= 4;

  } else if (q->handicap) {

    perf_score *= 2;
    q->handicap--;

  }

  /* Final adjustment based on input depth, under the assumption that fuzzing
     deeper test cases is more likely to reveal stuff that can't be
     discovered with traditional fuzzers. */
```

```
  switch (q->depth) {

    case 0 ... 3:   break;
    case 4 ... 7:   perf_score *= 2; break;
    case 8 ... 13:  perf_score *= 3; break;
    case 14 ... 25: perf_score *= 4; break;
    default:        perf_score *= 5;

  }

  /* Make sure that we don't go over limit. */

  if (perf_score > HAVOC_MAX_MULT * 100) perf_score = HAVOC_MAX_MULT * 100;

  return perf_score;

}


/* Helper function to see if a particular change (xor_val = old ^ new) could
   be a product of deterministic bit flips with the lengths and stepovers
   attempted by afl-fuzz. This is used to avoid dupes in some of the
   deterministic fuzzing operations that follow bit flips. We also
   return 1 if xor_val is zero, which implies that the old and attempted new
   values are identical and the exec would be a waste of time. */

static u8 could_be_bitflip(u32 xor_val) {

  u32 sh = 0;

  if (!xor_val) return 1;

  /* Shift left until first bit set. */

  while (!(xor_val & 1)) { sh++; xor_val >>= 1; }

  /* 1-, 2-, and 4-bit patterns are OK anywhere. */

  if (xor_val == 1 || xor_val == 3 || xor_val == 15) return 1;

  /* 8-, 16-, and 32-bit patterns are OK only if shift factor is
     divisible by 8, since that's the stepover for these ops. */

  if (sh & 7) return 0;

  if (xor_val == 0xff || xor_val == 0xffff || xor_val == 0xffffffff)
    return 1;

  return 0;

}


/* Helper function to see if a particular value is reachable through
   arithmetic operations. Used for similar purposes. */

static u8 could_be_arith(u32 old_val, u32 new_val, u8 blen) {

  u32 i, ov = 0, nv = 0, diffs = 0;
```

```
if (old_val == new_val) return 1;

/* See if one-byte adjustments to any byte could produce this result. */

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

  u8 a = old_val >> (8 * i),
     b = new_val >> (8 * i);

  if (a != b) { diffs++; ov = a; nv = b; }

}

/* If only one byte differs and the values are within range, return 1. */

if (diffs == 1) {

  if ((u8)(ov - nv) <= ARITH_MAX ||
      (u8)(nv - ov) <= ARITH_MAX) return 1;

}

if (blen == 1) return 0;

/* See if two-byte adjustments to any byte would produce this result. */

diffs = 0;

for (i = 0; i < blen / 2; i++) {

  u16 a = old_val >> (16 * i),
      b = new_val >> (16 * i);

  if (a != b) { diffs++; ov = a; nv = b; }

}

/* If only one word differs and the values are within range, return 1. */

if (diffs == 1) {

  if ((u16)(ov - nv) <= ARITH_MAX ||
      (u16)(nv - ov) <= ARITH_MAX) return 1;

  ov = SWAP16(ov); nv = SWAP16(nv);

  if ((u16)(ov - nv) <= ARITH_MAX ||
      (u16)(nv - ov) <= ARITH_MAX) return 1;

}

/* Finally, let's do the same thing for dwords. */

if (blen == 4) {

  if ((u32)(old_val - new_val) <= ARITH_MAX ||
      (u32)(new_val - old_val) <= ARITH_MAX) return 1;

  new_val = SWAP32(new_val);
```

```
    old_val = SWAP32(old_val);

    if ((u32)(old_val - new_val) <= ARITH_MAX ||
        (u32)(new_val - old_val) <= ARITH_MAX) return 1;

  }

  return 0;

}


/* Last but not least, a similar helper to see if insertion of an
   interesting integer is redundant given the insertions done for
   shorter blen. The last param (check_le) is set if the caller
   already executed LE insertion for current blen and wants to see
   if BE variant passed in new_val is unique. */

static u8 could_be_interest(u32 old_val, u32 new_val, u8 blen, u8 check_le) {

  u32 i, j;

  if (old_val == new_val) return 1;

  /* See if one-byte insertions from interesting_8 over old_val could
     produce new_val. */

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

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

      u32 tval = (old_val & ~(0xff << (i * 8))) |
                 (((u8)interesting_8[j]) << (i * 8));

      if (new_val == tval) return 1;

    }

  }

  /* Bail out unless we're also asked to examine two-byte LE insertions
     as a preparation for BE attempts. */

  if (blen == 2 && !check_le) return 0;

  /* See if two-byte insertions over old_val could give us new_val. */

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

    for (j = 0; j < sizeof(interesting_16) / 2; j++) {

      u32 tval = (old_val & ~(0xffff << (i * 8))) |
                 (((u16)interesting_16[j]) << (i * 8));

      if (new_val == tval) return 1;

      /* Continue here only if blen > 2. */

      if (blen > 2) {
```

```
        tval = (old_val & ~(0xffff << (i * 8))) |
               (SWAP16(interesting_16[j]) << (i * 8));

        if (new_val == tval) return 1;

      }

    }

  }

  if (blen == 4 && check_le) {

    /* See if four-byte insertions could produce the same result
       (LE only). */

    for (j = 0; j < sizeof(interesting_32) / 4; j++)
      if (new_val == (u32)interesting_32[j]) return 1;

  }

  return 0;

}


/* Take the current entry from the queue, fuzz it for a while. This
   function is a tad too long... returns 0 if fuzzed successfully, 1 if
   skipped or bailed out. */

static u8 fuzz_one(char** argv) {

  s32 len, fd, temp_len, i, j;
  u8  *in_buf, *out_buf, *orig_in, *ex_tmp, *eff_map = 0;
  u64 havoc_queued,  orig_hit_cnt, new_hit_cnt;
  u32 splice_cycle = 0, perf_score = 100, orig_perf, prev_cksum, eff_cnt = 1;

  u8  ret_val = 1, doing_det = 0;

  u8  a_collect[MAX_AUTO_EXTRA];
  u32 a_len = 0;

#ifdef IGNORE_FINDS

  /* In IGNORE_FINDS mode, skip any entries that weren't in the
     initial data set. */

  if (queue_cur->depth > 1) return 1;

#else

  if (pending_favored) {

    /* If we have any favored, non-fuzzed new arrivals in the queue,
       possibly skip to them at the expense of already-fuzzed or non-favored
       cases. */

    if ((queue_cur->was_fuzzed || !queue_cur->favored) &&
        UR(100) < SKIP_TO_NEW_PROB) return 1;
```

```
  } else if (!dumb_mode && !queue_cur->favored && queued_paths > 10) {

    /* Otherwise, still possibly skip non-favored cases, albeit less often.
       The odds of skipping stuff are higher for already-fuzzed inputs and
       lower for never-fuzzed entries. */

    if (queue_cycle > 1 && !queue_cur->was_fuzzed) {

      if (UR(100) < SKIP_NFAV_NEW_PROB) return 1;

    } else {

      if (UR(100) < SKIP_NFAV_OLD_PROB) return 1;

    }

  }

#endif /* ^IGNORE_FINDS */

  if (not_on_tty) {
    ACTF("Fuzzing test case #%u (%u total, %llu uniq crashes found)...",
         current_entry, queued_paths, unique_crashes);
    fflush(stdout);
  }

  /* Map the test case into memory. */

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

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

  len = queue_cur->len;

  orig_in = in_buf = mmap(0, len, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);

  if (orig_in == MAP_FAILED) PFATAL("Unable to mmap '%s'", queue_cur->fname);

  close(fd);

  /* We could mmap() out_buf as MAP_PRIVATE, but we end up clobbering every
     single byte anyway, so it wouldn't give us any performance or memory usage
     benefits. */

  out_buf = ck_alloc_nozero(len);

  subseq_tmouts = 0;

  cur_depth = queue_cur->depth;

  /*******************************************
   * CALIBRATION (only if failed earlier on) *
   *******************************************/

  if (queue_cur->cal_failed) {

    u8 res = FAULT_TMOUT;

    if (queue_cur->cal_failed < CAL_CHANCES) {
```

```
    /* Reset exec_cksum to tell calibrate_case to re-execute the testcase
       avoiding the usage of an invalid trace_bits.
       For more info: https://github.com/AFLplusplus/AFLplusplus/pull/425 */

    queue_cur->exec_cksum = 0;

    res = calibrate_case(argv, queue_cur, in_buf, queue_cycle - 1, 0);

    if (res == FAULT_ERROR)
      FATAL("Unable to execute target application");

  }

  if (stop_soon || res != crash_mode) {
    cur_skipped_paths++;
    goto abandon_entry;
  }

}

/************
 * TRIMMING *
 ************/

if (!dumb_mode && !queue_cur->trim_done) {

  u8 res = trim_case(argv, queue_cur, in_buf);

  if (res == FAULT_ERROR)
    FATAL("Unable to execute target application");

  if (stop_soon) {
    cur_skipped_paths++;
    goto abandon_entry;
  }

  /* Don't retry trimming, even if it failed. */

  queue_cur->trim_done = 1;

  if (len != queue_cur->len) len = queue_cur->len;

}

memcpy(out_buf, in_buf, len);

/*********************
 * PERFORMANCE SCORE *
 *********************/

orig_perf = perf_score = calculate_score(queue_cur);

/* 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;

/* Skip deterministic fuzzing if exec path checksum puts this out of scope
```

```
      for this master instance. */

  if (master_max && (queue_cur->exec_cksum % master_max) != master_id - 1)
    goto havoc_stage;

  doing_det = 1;

  /*********************************************
   * SIMPLE BITFLIP (+dictionary construction) *
   *********************************************/

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

  /* Single walking bit. */

  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);

    /* While flipping the least significant bit in every byte, pull of an extra
       trick to detect possible syntax tokens. In essence, the idea is that if
       you have a binary blob like this:

       xxxxxxxxIHDRxxxxxxxx

       ...and changing the leading and trailing bytes causes variable or no
       changes in program flow, but touching any character in the "IHDR" string
       always produces the same, distinctive path, it's highly likely that
       "IHDR" is an atomically-checked magic value of special significance to
       the fuzzed format.

       We do this here, rather than as a separate stage, because it's a nice
       way to keep the operation approximately "free" (i.e., no extra execs).

       Empirically, performing the check when flipping the least significant bit
       is advantageous, compared to doing it at the time of more disruptive
       changes, where the program flow may be affected in more violent ways.

       The caveat is that we won't generate dictionaries in the -d mode or -S
       mode - but that's probably a fair trade-off.
```

```
     This won't work particularly well with paths that exhibit variable
     behavior, but fails gracefully, so we'll carry out the checks anyway.

    */

  if (!dumb_mode && (stage_cur & 7) == 7) {

    u32 cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

    if (stage_cur == stage_max - 1 && cksum == prev_cksum) {

      /* If at end of file and we are still collecting a string, grab the
         final character and force output. */

      if (a_len < MAX_AUTO_EXTRA) a_collect[a_len] = out_buf[stage_cur >> 3];
      a_len++;

      if (a_len >= MIN_AUTO_EXTRA && a_len <= MAX_AUTO_EXTRA)
        maybe_add_auto(a_collect, a_len);

    } else if (cksum != prev_cksum) {

      /* Otherwise, if the checksum has changed, see if we have something
         worthwhile queued up, and collect that if the answer is yes. */

      if (a_len >= MIN_AUTO_EXTRA && a_len <= MAX_AUTO_EXTRA)
        maybe_add_auto(a_collect, a_len);

      a_len = 0;
      prev_cksum = cksum;

    }

    /* Continue collecting string, but only if the bit flip actually made
       any difference - we don't want no-op tokens. */

    if (cksum != queue_cur->exec_cksum) {

      if (a_len < MAX_AUTO_EXTRA) a_collect[a_len] = out_buf[stage_cur >> 3];
      a_len++;

    }

  }

}

new_hit_cnt = queued_paths + unique_crashes;

stage_finds[STAGE_FLIP1]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_FLIP1] += stage_max;

/* Two walking bits. */

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);

  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_FLIP2]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_FLIP2] += stage_max;

  /* Four walking bits. */

  stage_name  = "bitflip 4/1";
  stage_short = "flip4";
  stage_max   = (len << 3) - 3;

  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);
    FLIP_BIT(out_buf, stage_cur + 2);
    FLIP_BIT(out_buf, stage_cur + 3);

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

    FLIP_BIT(out_buf, stage_cur);
    FLIP_BIT(out_buf, stage_cur + 1);
    FLIP_BIT(out_buf, stage_cur + 2);
    FLIP_BIT(out_buf, stage_cur + 3);

  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_FLIP4]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_FLIP4] += stage_max;

  /* Effector map setup. These macros calculate:

     EFF_APOS      - position of a particular file offset in the map.
     EFF_ALEN      - length of a map with a particular number of bytes.
     EFF_SPAN_ALEN - map span for a sequence of bytes.

   */

#define EFF_APOS(_p)          ((_p) >> EFF_MAP_SCALE2)
```

```
#define EFF_REM(_x)           ((_x) & ((1 << EFF_MAP_SCALE2) - 1))
#define EFF_ALEN(_l)          (EFF_APOS(_l) + !!EFF_REM(_l))
#define EFF_SPAN_ALEN(_p, _l) (EFF_APOS((_p) + (_l) - 1) - EFF_APOS(_p) + 1)

  /* Initialize effector map for the next step (see comments below). Always
     flag first and last byte as doing something. */

  eff_map    = ck_alloc(EFF_ALEN(len));
  eff_map[0] = 1;

  if (EFF_APOS(len - 1) != 0) {
    eff_map[EFF_APOS(len - 1)] = 1;
    eff_cnt++;
  }

  /* Walking byte. */

  stage_name  = "bitflip 8/8";
  stage_short = "flip8";
  stage_max   = len;

  orig_hit_cnt = new_hit_cnt;

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

    stage_cur_byte = stage_cur;

    out_buf[stage_cur] ^= 0xFF;

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

    /* We also use this stage to pull off a simple trick: we identify
       bytes that seem to have no effect on the current execution path
       even when fully flipped - and we skip them during more expensive
       deterministic stages, such as arithmetics or known ints. */

    if (!eff_map[EFF_APOS(stage_cur)]) {

      u32 cksum;

      /* If in dumb mode or if the file is very short, just flag everything
         without wasting time on checksums. */

      if (!dumb_mode && len >= EFF_MIN_LEN)
        cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);
      else
        cksum = ~queue_cur->exec_cksum;

      if (cksum != queue_cur->exec_cksum) {
        eff_map[EFF_APOS(stage_cur)] = 1;
        eff_cnt++;
      }

    }

    out_buf[stage_cur] ^= 0xFF;

  }

  /* If the effector map is more than EFF_MAX_PERC dense, just flag the
```

```
    whole thing as worth fuzzing, since we wouldn't be saving much time
    anyway. */

if (eff_cnt != EFF_ALEN(len) &&
    eff_cnt * 100 / EFF_ALEN(len) > EFF_MAX_PERC) {

  memset(eff_map, 1, EFF_ALEN(len));

  blocks_eff_select += EFF_ALEN(len);

} else {

  blocks_eff_select += eff_cnt;

}

blocks_eff_total += EFF_ALEN(len);

new_hit_cnt = queued_paths + unique_crashes;

stage_finds[STAGE_FLIP8]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_FLIP8] += stage_max;

/* Two walking bytes. */

if (len < 2) goto skip_bitflip;

stage_name  = "bitflip 16/8";
stage_short = "flip16";
stage_cur   = 0;
stage_max   = len - 1;

orig_hit_cnt = new_hit_cnt;

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;


}

new_hit_cnt = queued_paths + unique_crashes;

stage_finds[STAGE_FLIP16]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_FLIP16] += stage_max;
```

```
  if (len < 4) goto skip_bitflip;

  /* Four walking bytes. */

  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;

  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_FLIP32]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_FLIP32] += stage_max;

skip_bitflip:

  if (no_arith) goto skip_arith;

  /**********************
   * ARITHMETIC INC/DEC *
   **********************/

  /* 8-bit arithmetics. */

  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;

  /* 16-bit arithmetics, both endians. */

  if (len < 2) goto skip_arith;

  stage_name  = "arith 16/8";
  stage_short = "arith16";
  stage_cur   = 0;
  stage_max   = 4 * (len - 1) * ARITH_MAX;

  orig_hit_cnt = new_hit_cnt;

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

```
u16 orig = *(u16*)(out_buf + i);

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

if (!eff_map[EFF_APOS(i)] && !eff_map[EFF_APOS(i + 1)]) {
  stage_max -= 4 * ARITH_MAX;
  continue;
}

stage_cur_byte = i;

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

  u16 r1 = orig ^ (orig + j),
      r2 = orig ^ (orig - j),
      r3 = orig ^ SWAP16(SWAP16(orig) + j),
      r4 = orig ^ SWAP16(SWAP16(orig) - j);

  /* Try little endian addition and subtraction first. Do it only
     if the operation would affect more than one byte (hence the
     & 0xff overflow checks) and if it couldn't be a product of
     a bitflip. */

  stage_val_type = STAGE_VAL_LE;

  if ((orig & 0xff) + j > 0xff && !could_be_bitflip(r1)) {

    stage_cur_val = j;
    *(u16*)(out_buf + i) = orig + j;

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

  } else stage_max--;

  if ((orig & 0xff) < j && !could_be_bitflip(r2)) {

    stage_cur_val = -j;
    *(u16*)(out_buf + i) = orig - j;

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

  } else stage_max--;

  /* Big endian comes next. Same deal. */

  stage_val_type = STAGE_VAL_BE;


  if ((orig >> 8) + j > 0xff && !could_be_bitflip(r3)) {

    stage_cur_val = j;
    *(u16*)(out_buf + i) = SWAP16(SWAP16(orig) + j);

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

  } else stage_max--;
```

```
    if ((orig >> 8) < j && !could_be_bitflip(r4)) {

      stage_cur_val = -j;
      *(u16*)(out_buf + i) = SWAP16(SWAP16(orig) - j);

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

    } else stage_max--;

    *(u16*)(out_buf + i) = orig;

  }

}

new_hit_cnt = queued_paths + unique_crashes;

stage_finds[STAGE_ARITH16]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_ARITH16] += stage_max;

/* 32-bit arithmetics, both endians. */

if (len < 4) goto skip_arith;

stage_name  = "arith 32/8";
stage_short = "arith32";
stage_cur   = 0;
stage_max   = 4 * (len - 3) * ARITH_MAX;

orig_hit_cnt = new_hit_cnt;

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

  u32 orig = *(u32*)(out_buf + 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 -= 4 * ARITH_MAX;
    continue;
  }

  stage_cur_byte = i;

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

    u32 r1 = orig ^ (orig + j),
        r2 = orig ^ (orig - j),
        r3 = orig ^ SWAP32(SWAP32(orig) + j),
        r4 = orig ^ SWAP32(SWAP32(orig) - j);

    /* Little endian first. Same deal as with 16-bit: we only want to
       try if the operation would have effect on more than two bytes. */

    stage_val_type = STAGE_VAL_LE;

    if ((orig & 0xffff) + j > 0xffff && !could_be_bitflip(r1)) {
```

```
        stage_cur_val = j;
        *(u32*)(out_buf + i) = orig + j;

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

      } else stage_max--;

      if ((orig & 0xffff) < j && !could_be_bitflip(r2)) {

        stage_cur_val = -j;
        *(u32*)(out_buf + i) = orig - j;

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

      } else stage_max--;

      /* Big endian next. */

      stage_val_type = STAGE_VAL_BE;

      if ((SWAP32(orig) & 0xffff) + j > 0xffff && !could_be_bitflip(r3)) {

        stage_cur_val = j;
        *(u32*)(out_buf + i) = SWAP32(SWAP32(orig) + j);

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

      } else stage_max--;

      if ((SWAP32(orig) & 0xffff) < j && !could_be_bitflip(r4)) {

        stage_cur_val = -j;
        *(u32*)(out_buf + i) = SWAP32(SWAP32(orig) - j);

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

      } else stage_max--;

      *(u32*)(out_buf + i) = orig;

    }

  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_ARITH32]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_ARITH32] += stage_max;

skip_arith:

  /**********************
   * INTERESTING VALUES *
   **********************/
```

```
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;

/* Setting 16-bit integers, both endians. */

if (no_arith || len < 2) goto skip_interest;

stage_name  = "interest 16/8";
stage_short = "int16";
stage_cur   = 0;
stage_max   = 2 * (len - 1) * (sizeof(interesting_16) >> 1);
```

```
orig_hit_cnt = new_hit_cnt;

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

  u16 orig = *(u16*)(out_buf + i);

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

  if (!eff_map[EFF_APOS(i)] && !eff_map[EFF_APOS(i + 1)]) {
    stage_max -= sizeof(interesting_16);
    continue;
  }

  stage_cur_byte = i;

  for (j = 0; j < sizeof(interesting_16) / 2; j++) {

    stage_cur_val = interesting_16[j];

    /* Skip if this could be a product of a bitflip, arithmetics,
       or single-byte interesting value insertion. */

    if (!could_be_bitflip(orig ^ (u16)interesting_16[j]) &&
        !could_be_arith(orig, (u16)interesting_16[j], 2) &&
        !could_be_interest(orig, (u16)interesting_16[j], 2, 0)) {

      stage_val_type = STAGE_VAL_LE;

      *(u16*)(out_buf + i) = interesting_16[j];

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

    } else stage_max--;

    if ((u16)interesting_16[j] != SWAP16(interesting_16[j]) &&
        !could_be_bitflip(orig ^ SWAP16(interesting_16[j])) &&
        !could_be_arith(orig, SWAP16(interesting_16[j]), 2) &&
        !could_be_interest(orig, SWAP16(interesting_16[j]), 2, 1)) {

      stage_val_type = STAGE_VAL_BE;

      *(u16*)(out_buf + i) = SWAP16(interesting_16[j]);
      if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
      stage_cur++;

    } else stage_max--;

  }

  *(u16*)(out_buf + i) = orig;

}

new_hit_cnt = queued_paths + unique_crashes;

stage_finds[STAGE_INTEREST16]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_INTEREST16] += stage_max;

if (len < 4) goto skip_interest;
```

```
/* Setting 32-bit integers, both endians. */

stage_name  = "interest 32/8";
stage_short = "int32";
stage_cur   = 0;
stage_max   = 2 * (len - 3) * (sizeof(interesting_32) >> 2);

orig_hit_cnt = new_hit_cnt;

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

  u32 orig = *(u32*)(out_buf + 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 -= sizeof(interesting_32) >> 1;
    continue;
  }

  stage_cur_byte = i;

  for (j = 0; j < sizeof(interesting_32) / 4; j++) {

    stage_cur_val = interesting_32[j];

    /* 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--;

    if ((u32)interesting_32[j] != SWAP32(interesting_32[j]) &&
        !could_be_bitflip(orig ^ SWAP32(interesting_32[j])) &&
        !could_be_arith(orig, SWAP32(interesting_32[j]), 4) &&
        !could_be_interest(orig, SWAP32(interesting_32[j]), 4, 1)) {

      stage_val_type = STAGE_VAL_BE;

      *(u32*)(out_buf + i) = SWAP32(interesting_32[j]);
      if (common_fuzz_stuff(argv, out_buf, len)) goto abandon_entry;
      stage_cur++;

    } else stage_max--;

  }

  *(u32*)(out_buf + i) = orig;
```

```
  }

  new_hit_cnt = queued_paths + unique_crashes;

  stage_finds[STAGE_INTEREST32]  += new_hit_cnt - orig_hit_cnt;
  stage_cycles[STAGE_INTEREST32] += stage_max;

skip_interest:

  /********************
   * DICTIONARY STUFF *
   ********************/

  if (!extras_cnt) goto skip_user_extras;

  /* Overwrite with user-supplied extras. */

  stage_name  = "user extras (over)";
  stage_short = "ext_UO";
  stage_cur   = 0;
  stage_max   = extras_cnt * 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;

    /* Extras are sorted by size, from smallest to largest. This means
       that we don't have to worry about restoring the buffer in
       between writes at a particular offset determined by the outer
       loop. */

    for (j = 0; j < extras_cnt; j++) {

      /* Skip extras probabilistically if extras_cnt > MAX_DET_EXTRAS. Also
         skip them if there's no room to insert the payload, if the token
         is redundant, or if its entire span has no bytes set in the effector
         map. */

      if ((extras_cnt > MAX_DET_EXTRAS && UR(extras_cnt) >= MAX_DET_EXTRAS) ||
          extras[j].len > len - i ||
          !memcmp(extras[j].data, out_buf + i, extras[j].len) ||
          !memchr(eff_map + EFF_APOS(i), 1, EFF_SPAN_ALEN(i, extras[j].len))) {

        stage_max--;
        continue;

      }

      last_len = extras[j].len;
      memcpy(out_buf + i, 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_UO]  += new_hit_cnt - orig_hit_cnt;
stage_cycles[STAGE_EXTRAS_UO] += stage_max;

/* Insertion of user-supplied extras. */

stage_name  = "user extras (insert)";
stage_short = "ext_UI";
stage_cur   = 0;
stage_max   = extras_cnt * (len + 1);

orig_hit_cnt = new_hit_cnt;

ex_tmp = ck_alloc(len + MAX_DICT_FILE);

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

  stage_cur_byte = i;

  for (j = 0; j < extras_cnt; j++) {

    if (len + extras[j].len > MAX_FILE) {
      stage_max--;
      continue;
    }

    /* Insert token */
    memcpy(ex_tmp + i, extras[j].data, extras[j].len);

    /* Copy tail */
    memcpy(ex_tmp + i + extras[j].len, out_buf + i, len - i);

    if (common_fuzz_stuff(argv, ex_tmp, len + extras[j].len)) {
      ck_free(ex_tmp);
      goto abandon_entry;
    }

    stage_cur++;

  }

  /* Copy head */
  ex_tmp[i] = out_buf[i];

}

ck_free(ex_tmp);

new_hit_cnt = queued_paths + unique_crashes;
```

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

skip_user_extras:

  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;

skip_extras:

  /* 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:

  stage_cur_byte = -1;

  /* The havoc stage mutation code is also invoked when splicing files; if the
     splice_cycle variable is set, generate different descriptions and such. */

  if (!splice_cycle) {

    stage_name  = "havoc";
    stage_short = "havoc";
    stage_max   = (doing_det ? HAVOC_CYCLES_INIT : HAVOC_CYCLES) *
                  perf_score / havoc_div / 100;

  } else {

    static u8 tmp[32];

    perf_score = orig_perf;

    sprintf(tmp, "splice %u", splice_cycle);
    stage_name  = tmp;
    stage_short = "splice";
    stage_max   = SPLICE_HAVOC * perf_score / havoc_div / 100;

  }

  if (stage_max < HAVOC_MIN) stage_max = HAVOC_MIN;

  temp_len = len;

  orig_hit_cnt = queued_paths + unique_crashes;

  havoc_queued = queued_paths;

  /* We essentially just do several thousand runs (depending on perf_score)
     where we take the input file and make random stacked tweaks. */

  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;

    }

  }

  new_hit_cnt = queued_paths + unique_crashes;

  if (!splice_cycle) {
    stage_finds[STAGE_HAVOC]  += new_hit_cnt - orig_hit_cnt;
    stage_cycles[STAGE_HAVOC] += stage_max;
  } else {
    stage_finds[STAGE_SPLICE]  += new_hit_cnt - orig_hit_cnt;
    stage_cycles[STAGE_SPLICE] += stage_max;
  }

#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;

  }

#endif /* !IGNORE_FINDS */

  ret_val = 0;

abandon_entry:

  splicing_with = -1;

  /* Update pending_not_fuzzed count if we made it through the calibration
     cycle and have not seen this entry before. */

  if (!stop_soon && !queue_cur->cal_failed && !queue_cur->was_fuzzed) {
    queue_cur->was_fuzzed = 1;
    pending_not_fuzzed--;
    if (queue_cur->favored) pending_favored--;
  }

  munmap(orig_in, queue_cur->len);

  if (in_buf != orig_in) ck_free(in_buf);
  ck_free(out_buf);
  ck_free(eff_map);

  return ret_val;

#undef FLIP_BIT

}


/* Grab interesting test cases from other fuzzers. */

static void sync_fuzzers(char** argv) {

  DIR* sd;
  struct dirent* sd_ent;
  u32 sync_cnt = 0;

  sd = opendir(sync_dir);
  if (!sd) PFATAL("Unable to open '%s'", sync_dir);

  stage_max = stage_cur = 0;
  cur_depth = 0;

  /* Look at the entries created for every other fuzzer in the sync directory. */

  while ((sd_ent = readdir(sd))) {

    static u8 stage_tmp[128];
```

```
DIR* qd;
struct dirent* qd_ent;
u8 *qd_path, *qd_synced_path;
u32 min_accept = 0, next_min_accept;

s32 id_fd;

/* Skip dot files and our own output directory. */

if (sd_ent->d_name[0] == '.' || !strcmp(sync_id, sd_ent->d_name)) continue;

/* Skip anything that doesn't have a queue/ subdirectory. */

qd_path = alloc_printf("%s/%s/queue", sync_dir, sd_ent->d_name);

if (!(qd = opendir(qd_path))) {
  ck_free(qd_path);
  continue;
}

/* Retrieve the ID of the last seen test case. */

qd_synced_path = alloc_printf("%s/.synced/%s", out_dir, sd_ent->d_name);

id_fd = open(qd_synced_path, O_RDWR | O_CREAT, 0600);

if (id_fd < 0) PFATAL("Unable to create '%s'", qd_synced_path);

if (read(id_fd, &min_accept, sizeof(u32)) > 0)
  lseek(id_fd, 0, SEEK_SET);

next_min_accept = min_accept;

/* Show stats */

sprintf(stage_tmp, "sync %u", ++sync_cnt);
stage_name = stage_tmp;
stage_cur  = 0;
stage_max  = 0;

/* For every file queued by this fuzzer, parse ID and see if we have looked at
   it before; exec a test case if not. */

while ((qd_ent = readdir(qd))) {

  u8* path;
  s32 fd;
  struct stat st;

  if (qd_ent->d_name[0] == '.' ||
      sscanf(qd_ent->d_name, CASE_PREFIX "%06u", &syncing_case) != 1 ||
      syncing_case < min_accept) continue;

  /* OK, sounds like a new one. Let's give it a try. */

  if (syncing_case >= next_min_accept)
    next_min_accept = syncing_case + 1;

  path = alloc_printf("%s/%s", qd_path, qd_ent->d_name);
```

```
      /* Allow this to fail in case the other fuzzer is resuming or so... */

      fd = open(path, O_RDONLY);

      if (fd < 0) {
         ck_free(path);
         continue;
      }

      if (fstat(fd, &st)) PFATAL("fstat() failed");

      /* Ignore zero-sized or oversized files. */

      if (st.st_size && st.st_size <= MAX_FILE) {

        u8  fault;
        u8* mem = mmap(0, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

        if (mem == MAP_FAILED) PFATAL("Unable to mmap '%s'", path);

        /* See what happens. We rely on save_if_interesting() to catch major
           errors and save the test case. */

        write_to_testcase(mem, st.st_size);

        fault = run_target(argv, exec_tmout);

        if (stop_soon) return;

        syncing_party = sd_ent->d_name;
        queued_imported += save_if_interesting(argv, mem, st.st_size, fault);
        syncing_party = 0;

        munmap(mem, st.st_size);

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

      }

      ck_free(path);
      close(fd);

    }

    ck_write(id_fd, &next_min_accept, sizeof(u32), qd_synced_path);

    close(id_fd);
    closedir(qd);
    ck_free(qd_path);
    ck_free(qd_synced_path);

  }

  closedir(sd);

}


/* Handle stop signal (Ctrl-C, etc). */
```

```
static void handle_stop_sig(int sig) {

  stop_soon = 1;

  if (child_pid > 0) kill(child_pid, SIGKILL);
  if (forksrv_pid > 0) kill(forksrv_pid, SIGKILL);

}


/* Handle skip request (SIGUSR1). */

static void handle_skipreq(int sig) {

  skip_requested = 1;

}

/* Handle timeout (SIGALRM). */

static void handle_timeout(int sig) {

  if (child_pid > 0) {

    child_timed_out = 1;
    kill(child_pid, SIGKILL);

  } else if (child_pid == -1 && forksrv_pid > 0) {

    child_timed_out = 1;
    kill(forksrv_pid, SIGKILL);

  }

}


/* Do a PATH search and find target binary to see that it exists and
   isn't a shell script - a common and painful mistake. We also check for
   a valid ELF header and for evidence of AFL instrumentation. */

EXP_ST void check_binary(u8* fname) {

  u8* env_path = 0;
  struct stat st;

  s32 fd;
  u8* f_data;
  u32 f_len = 0;

  ACTF("Validating target binary...");

  if (strchr(fname, '/') || !(env_path = getenv("PATH"))) {

    target_path = ck_strdup(fname);
    if (stat(target_path, &st) || !S_ISREG(st.st_mode) ||
        !(st.st_mode & 0111) || (f_len = st.st_size) < 4)
      FATAL("Program '%s' not found or not executable", fname);

  } else {
```

```
  while (env_path) {

    u8 *cur_elem, *delim = strchr(env_path, ':');

    if (delim) {

      cur_elem = ck_alloc(delim - env_path + 1);
      memcpy(cur_elem, env_path, delim - env_path);
      delim++;

    } else cur_elem = ck_strdup(env_path);

    env_path = delim;

    if (cur_elem[0])
      target_path = alloc_printf("%s/%s", cur_elem, fname);
    else
      target_path = ck_strdup(fname);

    ck_free(cur_elem);

    if (!stat(target_path, &st) && S_ISREG(st.st_mode) &&
        (st.st_mode & 0111) && (f_len = st.st_size) >= 4) break;

    ck_free(target_path);
    target_path = 0;

  }

  if (!target_path) FATAL("Program '%s' not found or not executable", fname);

}

if (getenv("AFL_SKIP_BIN_CHECK")) return;

/* Check for blatant user errors. */

if ((!strncmp(target_path, "/tmp/", 5) && !strchr(target_path + 5, '/')) ||
    (!strncmp(target_path, "/var/tmp/", 9) && !strchr(target_path + 9, '/')))
   FATAL("Please don't keep binaries in /tmp or /var/tmp");

fd = open(target_path, O_RDONLY);

if (fd < 0) PFATAL("Unable to open '%s'", target_path);

f_data = mmap(0, f_len, PROT_READ, MAP_PRIVATE, fd, 0);

if (f_data == MAP_FAILED) PFATAL("Unable to mmap file '%s'", target_path);

close(fd);

if (f_data[0] == '#' && f_data[1] == '!') {

  SAYF("\n" cLRD "[-] " cRST
       "Oops, the target binary looks like a shell script. Some build systems will\n"
       "    sometimes generate shell stubs for dynamically linked programs; try static\n"
       "    library mode (./configure --disable-shared) if that's the case.\n\n"

       "    Another possible cause is that you are actually trying to use a shell\n"
```

```
          "    wrapper around the fuzzed component. Invoking shell can slow down the\n"
          "    fuzzing process by a factor of 20x or more; it's best to write the wrapper\n"
          "    in a compiled language instead.\n");

    FATAL("Program '%s' is a shell script", target_path);

  }

#ifndef __APPLE__

  if (f_data[0] != 0x7f || memcmp(f_data + 1, "ELF", 3))
    FATAL("Program '%s' is not an ELF binary", target_path);

#else

  if (f_data[0] != 0xCF || f_data[1] != 0xFA || f_data[2] != 0xED)
    FATAL("Program '%s' is not a 64-bit Mach-O binary", target_path);

#endif /* ^!__APPLE__ */

  if (!qemu_mode && !dumb_mode &&
      !memmem(f_data, f_len, SHM_ENV_VAR, strlen(SHM_ENV_VAR) + 1)) {

    SAYF("\n" cLRD "[-] " cRST
         "Looks like the target binary is not instrumented! The fuzzer depends on\n"
         "    compile-time instrumentation to isolate interesting test cases while\n"
         "    mutating the input data. For more information, and for tips on how to\n"
         "    instrument binaries, please see %s/README.\n\n"

         "    When source code is not available, you may be able to leverage QEMU\n"
         "    mode support. Consult the README for tips on how to enable this.\n"

         "    (It is also possible to use afl-fuzz as a traditional, \"dumb\" fuzzer.\n"
         "    For that, you can use the -n option - but expect much worse results.)\n",
         doc_path);

    FATAL("No instrumentation detected");

  }

  if (qemu_mode &&
      memmem(f_data, f_len, SHM_ENV_VAR, strlen(SHM_ENV_VAR) + 1)) {

    SAYF("\n" cLRD "[-] " cRST
         "This program appears to be instrumented with afl-gcc, but is being run in\n"
         "    QEMU mode (-Q). This is probably not what you want - this setup will be\n"
         "    slow and offer no practical benefits.\n");

    FATAL("Instrumentation found in -Q mode");

  }

  if (memmem(f_data, f_len, "libasan.so", 10) ||
      memmem(f_data, f_len, "__msan_init", 11)) uses_asan = 1;

  /* Detect persistent & deferred init signatures in the binary. */

  if (memmem(f_data, f_len, PERSIST_SIG, strlen(PERSIST_SIG) + 1)) {

    OKF(cPIN "Persistent mode binary detected.");
```

```
    setenv(PERSIST_ENV_VAR, "1", 1);
    persistent_mode = 1;

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

    WARNF("AFL_PERSISTENT is no longer supported and may misbehave!");

  }

  if (memmem(f_data, f_len, DEFER_SIG, strlen(DEFER_SIG) + 1)) {

    OKF(cPIN "Deferred forkserver binary detected.");
    setenv(DEFER_ENV_VAR, "1", 1);
    deferred_mode = 1;

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

    WARNF("AFL_DEFER_FORKSRV is no longer supported and may misbehave!");

  }

  if (munmap(f_data, f_len)) PFATAL("unmap() failed");

}


/* Trim and possibly create a banner for the run. */

static void fix_up_banner(u8* name) {

  if (!use_banner) {

    if (sync_id) {

      use_banner = sync_id;

    } else {

      u8* trim = strrchr(name, '/');
      if (!trim) use_banner = name; else use_banner = trim + 1;

    }

  }

  if (strlen(use_banner) > 40) {

    u8* tmp = ck_alloc(44);
    sprintf(tmp, "%.40s...", use_banner);
    use_banner = tmp;

  }

}


/* Check if we're on TTY. */

static void check_if_tty(void) {
```

```
  struct winsize ws;

  if (getenv("AFL_NO_UI")) {
    OKF("Disabling the UI because AFL_NO_UI is set.");
    not_on_tty = 1;
    return;
  }

  if (ioctl(1, TIOCGWINSZ, &ws)) {

    if (errno == ENOTTY) {
      OKF("Looks like we're not running on a tty, so I'll be a bit less verbose.");
      not_on_tty = 1;
    }

    return;
  }

}


/* Check terminal dimensions after resize. */

static void check_term_size(void) {

  struct winsize ws;

  term_too_small = 0;

  if (ioctl(1, TIOCGWINSZ, &ws)) return;

  if (ws.ws_row == 0 && ws.ws_col == 0) return;
  if (ws.ws_row < 25 || ws.ws_col < 80) term_too_small = 1;

}



/* Display usage hints. */

static void usage(u8* argv0) {

  SAYF("\n%s [ options ] -- /path/to/fuzzed_app [ ... ]\n\n"

       "Required parameters:\n\n"

       "  -i dir        - input directory with test cases\n"
       "  -o dir        - output directory for fuzzer findings\n\n"

       "Execution control settings:\n\n"

       "  -f file       - location read by the fuzzed program (stdin)\n"
       "  -t msec       - timeout for each run (auto-scaled, 50-%u ms)\n"
       "  -m megs       - memory limit for child process (%u MB)\n"
       "  -Q            - use binary-only instrumentation (QEMU mode)\n\n"

       "Fuzzing behavior settings:\n\n"

       "  -d            - quick & dirty mode (skips deterministic steps)\n"
       "  -n            - fuzz without instrumentation (dumb mode)\n"
```

```
        "  -x dir         - optional fuzzer dictionary (see README)\n\n"

        "Other stuff:\n\n"

        "  -T text        - text banner to show on the screen\n"
        "  -M / -S id     - distributed mode (see parallel_fuzzing.txt)\n"
        "  -C             - crash exploration mode (the peruvian rabbit thing)\n"
        "  -V             - show version number and exit\n\n"
        "  -b cpu_id      - bind the fuzzing process to the specified CPU core\n\n"

        "For additional tips, please consult %s/README.\n\n",

        argv0, EXEC_TIMEOUT, MEM_LIMIT, doc_path);

  exit(1);

}


/* Prepare output directories and fds. */

EXP_ST void setup_dirs_fds(void) {

  u8* tmp;
  s32 fd;

  ACTF("Setting up output directories...");

  if (sync_id && mkdir(sync_dir, 0700) && errno != EEXIST)
      PFATAL("Unable to create '%s'", sync_dir);

  if (mkdir(out_dir, 0700)) {

    if (errno != EEXIST) PFATAL("Unable to create '%s'", out_dir);

    maybe_delete_out_dir();

  } else {

    if (in_place_resume)
      FATAL("Resume attempted but old output directory not found");

    out_dir_fd = open(out_dir, O_RDONLY);

#ifndef __sun

    if (out_dir_fd < 0 || flock(out_dir_fd, LOCK_EX | LOCK_NB))
      PFATAL("Unable to flock() output directory.");

#endif /* !__sun */

  }

  /* Queue directory for any starting & discovered paths. */

  tmp = alloc_printf("%s/queue", out_dir);
  if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
  ck_free(tmp);

  /* Top-level directory for queue metadata used for session
```

```
   resume and related tasks. */

tmp = alloc_printf("%s/queue/.state/", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* Directory for flagging queue entries that went through
   deterministic fuzzing in the past. */

tmp = alloc_printf("%s/queue/.state/deterministic_done/", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* Directory with the auto-selected dictionary entries. */

tmp = alloc_printf("%s/queue/.state/auto_extras/", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* The set of paths currently deemed redundant. */

tmp = alloc_printf("%s/queue/.state/redundant_edges/", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* The set of paths showing variable behavior. */

tmp = alloc_printf("%s/queue/.state/variable_behavior/", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* Sync directory for keeping track of cooperating fuzzers. */

if (sync_id) {

  tmp = alloc_printf("%s/.synced/", out_dir);

  if (mkdir(tmp, 0700) && (!in_place_resume || errno != EEXIST))
    PFATAL("Unable to create '%s'", tmp);

  ck_free(tmp);

}

/* All recorded crashes. */

tmp = alloc_printf("%s/crashes", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* All recorded hangs. */

tmp = alloc_printf("%s/hangs", out_dir);
if (mkdir(tmp, 0700)) PFATAL("Unable to create '%s'", tmp);
ck_free(tmp);

/* Generally useful file descriptors. */

dev_null_fd = open("/dev/null", O_RDWR);
if (dev_null_fd < 0) PFATAL("Unable to open /dev/null");
```

```
  dev_urandom_fd = open("/dev/urandom", O_RDONLY);
  if (dev_urandom_fd < 0) PFATAL("Unable to open /dev/urandom");

  /* Gnuplot output file. */

  tmp = alloc_printf("%s/plot_data", out_dir);
  fd = open(tmp, O_WRONLY | O_CREAT | O_EXCL, 0600);
  if (fd < 0) PFATAL("Unable to create '%s'", tmp);
  ck_free(tmp);

  plot_file = fdopen(fd, "w");
  if (!plot_file) PFATAL("fdopen() failed");

  fprintf(plot_file, "# unix_time, cycles_done, cur_path, paths_total, "
                     "pending_total, pending_favs, map_size, unique_crashes, "
                     "unique_hangs, max_depth, execs_per_sec\n");
                     /* ignore errors */

}


/* Setup the output file for fuzzed data, if not using -f. */

EXP_ST void setup_stdio_file(void) {

  u8* fn = alloc_printf("%s/.cur_input", out_dir);

  unlink(fn); /* Ignore errors */

  out_fd = open(fn, O_RDWR | O_CREAT | O_EXCL, 0600);

  if (out_fd < 0) PFATAL("Unable to create '%s'", fn);

  ck_free(fn);

}


/* Make sure that core dumps don't go to a program. */

static void check_crash_handling(void) {

#ifdef __APPLE__

  /* Yuck! There appears to be no simple C API to query for the state of
     loaded daemons on MacOS X, and I'm a bit hesitant to do something
     more sophisticated, such as disabling crash reporting via Mach ports,
     until I get a box to test the code. So, for now, we check for crash
     reporting the awful way. */

  if (system("launchctl list 2>/dev/null | grep -q '\\.ReportCrash$'")) return;

  SAYF("\n" cLRD "[-] " cRST
       "Whoops, your system is configured to forward crash notifications to an\n"
       "    external crash reporting utility. This will cause issues due to the\n"
       "    extended delay between the fuzzed binary malfunctioning and this fact\n"
       "    being relayed to the fuzzer via the standard waitpid() API.\n\n"
       "    To avoid having crashes misinterpreted as timeouts, please run the\n"
       "    following commands:\n\n"
```

```
        "    SL=/System/Library; PL=com.apple.ReportCrash\n"
        "    launchctl unload -w ${SL}/LaunchAgents/${PL}.plist\n"
        "    sudo launchctl unload -w ${SL}/LaunchDaemons/${PL}.Root.plist\n");

  if (!getenv("AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES"))
    FATAL("Crash reporter detected");

#else

  /* This is Linux specific, but I don't think there's anything equivalent on
     *BSD, so we can just let it slide for now. */

  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);

#endif /* ^__APPLE__ */

}


/* Check CPU governor. */

static void check_cpu_governor(void) {

  FILE* f;
  u8 tmp[128];
  u64 min = 0, max = 0;

  if (getenv("AFL_SKIP_CPUFREQ")) return;

  f = fopen("/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor", "r");
  if (!f) return;

  ACTF("Checking CPU scaling governor...");
```

```
  if (!fgets(tmp, 128, f)) PFATAL("fgets() failed");

  fclose(f);

  if (!strncmp(tmp, "perf", 4)) return;

  f = fopen("/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq", "r");

  if (f) {
    if (fscanf(f, "%llu", &min) != 1) min = 0;
    fclose(f);
  }

  f = fopen("/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq", "r");

  if (f) {
    if (fscanf(f, "%llu", &max) != 1) max = 0;
    fclose(f);
  }

  if (min == max) return;

  SAYF("\n" cLRD "[-] " cRST
       "Whoops, your system uses on-demand CPU frequency scaling, adjusted\n"
       "    between %llu and %llu MHz. Unfortunately, the scaling algorithm in the\n"
       "    kernel is imperfect and can miss the short-lived processes spawned by\n"
       "    afl-fuzz. To keep things moving, run these commands as root:\n\n"

       "    cd /sys/devices/system/cpu\n"
       "    echo performance | tee cpu*/cpufreq/scaling_governor\n\n"

       "    You can later go back to the original state by replacing 'performance' with\n"
       "    'ondemand'. If you don't want to change the settings, set AFL_SKIP_CPUFREQ\n"
       "    to make afl-fuzz skip this check - but expect some performance drop.\n",
       min / 1024, max / 1024);

  FATAL("Suboptimal CPU scaling governor");

}


/* Count the number of logical CPU cores. */

static void get_core_count(void) {

  u32 cur_runnable = 0;

#if defined(__APPLE__) || defined(__FreeBSD__) || defined (__OpenBSD__)

  size_t s = sizeof(cpu_core_count);

  /* On *BSD systems, we can just use a sysctl to get the number of CPUs. */

#ifdef __APPLE__

  if (sysctlbyname("hw.logicalcpu", &cpu_core_count, &s, NULL, 0) < 0)
    return;

#else
```

```
  int s_name[2] = { CTL_HW, HW_NCPU };

  if (sysctl(s_name, 2, &cpu_core_count, &s, NULL, 0) < 0) return;

#endif /* ^__APPLE__ */

#else

#ifdef HAVE_AFFINITY

  cpu_core_count = sysconf(_SC_NPROCESSORS_ONLN);

#else

  FILE* f = fopen("/proc/stat", "r");
  u8 tmp[1024];

  if (!f) return;

  while (fgets(tmp, sizeof(tmp), f))
    if (!strncmp(tmp, "cpu", 3) && isdigit(tmp[3])) cpu_core_count++;

  fclose(f);

#endif /* ^HAVE_AFFINITY */

#endif /* ^(__APPLE__ || __FreeBSD__ || __OpenBSD__) */

  if (cpu_core_count > 0) {

    cur_runnable = (u32)get_runnable_processes();

#if defined(__APPLE__) || defined(__FreeBSD__) || defined (__OpenBSD__)

    /* Add ourselves, since the 1-minute average doesn't include that yet. */

    cur_runnable++;

#endif /* __APPLE__ || __FreeBSD__ || __OpenBSD__ */

    OKF("You have %u CPU core%s and %u runnable tasks (utilization: %0.0f%%).",
        cpu_core_count, cpu_core_count > 1 ? "s" : "",
        cur_runnable, cur_runnable * 100.0 / cpu_core_count);

    if (cpu_core_count > 1) {

      if (cur_runnable > cpu_core_count * 1.5) {

        WARNF("System under apparent load, performance may be spotty.");

      } else if (cur_runnable + 1 <= cpu_core_count) {

        OKF("Try parallel jobs - see %s/parallel_fuzzing.txt.", doc_path);

      }

    }

  } else {
```

```
    cpu_core_count = 0;
    WARNF("Unable to figure out the number of CPU cores.");

  }

}


/* Validate and fix up out_dir and sync_dir when using -S. */

static void fix_up_sync(void) {

  u8* x = sync_id;

  if (dumb_mode)
    FATAL("-S / -M and -n are mutually exclusive");

  if (skip_deterministic) {

    if (force_deterministic)
      FATAL("use -S instead of -M -d");
    else
      FATAL("-S already implies -d");

  }

  while (*x) {

    if (!isalnum(*x) && *x != '_' && *x != '-')
      FATAL("Non-alphanumeric fuzzer ID specified via -S or -M");

    x++;

  }

  if (strlen(sync_id) > 32) FATAL("Fuzzer ID too long");

  x = alloc_printf("%s/%s", out_dir, sync_id);

  sync_dir = out_dir;
  out_dir  = x;

  if (!force_deterministic) {
    skip_deterministic = 1;
    use_splicing = 1;
  }

}


/* Handle screen resize (SIGWINCH). */

static void handle_resize(int sig) {
  clear_screen = 1;
}


/* Check ASAN options. */

static void check_asan_opts(void) {
```

```
  u8* x = getenv("ASAN_OPTIONS");

  if (x) {

    if (!strstr(x, "abort_on_error=1"))
      FATAL("Custom ASAN_OPTIONS set without abort_on_error=1 - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom ASAN_OPTIONS set without symbolize=0 - please fix!");

  }

  x = getenv("MSAN_OPTIONS");

  if (x) {

    if (!strstr(x, "exit_code=" STRINGIFY(MSAN_ERROR)))
      FATAL("Custom MSAN_OPTIONS set without exit_code="
            STRINGIFY(MSAN_ERROR) " - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom MSAN_OPTIONS set without symbolize=0 - please fix!");

  }

}


/* Detect @@ in args. */

EXP_ST void detect_file_args(char** argv) {

  u32 i = 0;
  u8* cwd = getcwd(NULL, 0);

  if (!cwd) PFATAL("getcwd() failed");

  while (argv[i]) {

    u8* aa_loc = strstr(argv[i], "@@");

    if (aa_loc) {

      u8 *aa_subst, *n_arg;

      /* If we don't have a file name chosen yet, use a safe default. */

      if (!out_file)
        out_file = alloc_printf("%s/.cur_input", out_dir);

      /* Be sure that we're always using fully-qualified paths. */

      if (out_file[0] == '/') aa_subst = out_file;
      else aa_subst = alloc_printf("%s/%s", cwd, out_file);

      /* Construct a replacement argv value. */

      *aa_loc = 0;
      n_arg = alloc_printf("%s%s%s", argv[i], aa_subst, aa_loc + 2);
      argv[i] = n_arg;
```

```
      *aa_loc = '@';

      if (out_file[0] != '/') ck_free(aa_subst);

    }

    i++;

  }

  free(cwd); /* not tracked */

}


/* Set up signal handlers. More complicated that needs to be, because libc on
   Solaris doesn't resume interrupted reads(), sets SA_RESETHAND when you call
   siginterrupt(), and does other unnecessary things. */

EXP_ST void setup_signal_handlers(void) {

  struct sigaction sa;

  sa.sa_handler   = NULL;
  sa.sa_flags     = SA_RESTART;
  sa.sa_sigaction = NULL;

  sigemptyset(&sa.sa_mask);

  /* Various ways of saying "stop". */

  sa.sa_handler = handle_stop_sig;
  sigaction(SIGHUP, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);

  /* Exec timeout notifications. */

  sa.sa_handler = handle_timeout;
  sigaction(SIGALRM, &sa, NULL);

  /* Window resize */

  sa.sa_handler = handle_resize;
  sigaction(SIGWINCH, &sa, NULL);

  /* SIGUSR1: skip entry */

  sa.sa_handler = handle_skipreq;
  sigaction(SIGUSR1, &sa, NULL);

  /* Things we don't care about. */

  sa.sa_handler = SIG_IGN;
  sigaction(SIGTSTP, &sa, NULL);
  sigaction(SIGPIPE, &sa, NULL);

}
```

```
/* Rewrite argv for QEMU. */

static char** get_qemu_argv(u8* own_loc, char** argv, int argc) {

  char** new_argv = ck_alloc(sizeof(char*) * (argc + 4));
  u8 *tmp, *cp, *rsl, *own_copy;

  /* Workaround for a QEMU stability glitch. */

  setenv("QEMU_LOG", "nochain", 1);

  memcpy(new_argv + 3, argv + 1, sizeof(char*) * argc);

  new_argv[2] = target_path;
  new_argv[1] = "--";

  /* Now we need to actually find the QEMU binary to put in argv[0]. */

  tmp = getenv("AFL_PATH");

  if (tmp) {

    cp = alloc_printf("%s/afl-qemu-trace", tmp);

    if (access(cp, X_OK))
      FATAL("Unable to find '%s'", tmp);

    target_path = new_argv[0] = cp;
    return new_argv;

  }

  own_copy = ck_strdup(own_loc);
  rsl = strrchr(own_copy, '/');

  if (rsl) {

    *rsl = 0;

    cp = alloc_printf("%s/afl-qemu-trace", own_copy);
    ck_free(own_copy);

    if (!access(cp, X_OK)) {

      target_path = new_argv[0] = cp;
      return new_argv;

    }

  } else ck_free(own_copy);

  if (!access(BIN_PATH "/afl-qemu-trace", X_OK)) {

    target_path = new_argv[0] = ck_strdup(BIN_PATH "/afl-qemu-trace");
    return new_argv;

  }

  SAYF("\n" cLRD "[-] " cRST
       "Oops, unable to find the 'afl-qemu-trace' binary. The binary must be built\n"
```

```
        "    separately by following the instructions in qemu_mode/README.qemu. If you\n"
        "    already have the binary installed, you may need to specify AFL_PATH in the\n"
        "    environment.\n\n"

        "    Of course, even without QEMU, afl-fuzz can still work with binaries that are\n"
        "    instrumented at compile time with afl-gcc. It is also possible to use it as a\n"
        "    traditional \"dumb\" fuzzer by specifying '-n' in the command line.\n");

  FATAL("Failed to locate 'afl-qemu-trace'.");

}


/* Make a copy of the current command line. */

static void save_cmdline(u32 argc, char** argv) {

  u32 len = 1, i;
  u8* buf;

  for (i = 0; i < argc; i++)
    len += strlen(argv[i]) + 1;

  buf = orig_cmdline = ck_alloc(len);

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

    u32 l = strlen(argv[i]);

    memcpy(buf, argv[i], l);
    buf += l;

    if (i != argc - 1) *(buf++) = ' ';

  }

  *buf = 0;

}


#ifndef AFL_LIB

/* Main entry point */

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

  s32 opt;
  u64 prev_queued = 0;
  u32 sync_interval_cnt = 0, seek_to;
  u8  *extras_dir = 0;
  u8  mem_limit_given = 0;
  u8  exit_1 = !!getenv("AFL_BENCH_JUST_ONE");
  char** use_argv;

  struct timeval tv;
  struct timezone tz;

  SAYF(cCYA "afl-fuzz " cBRI VERSION cRST " by <lcamtuf@google.com>\n");
```

```
doc_path = access(DOC_PATH, F_OK) ? "docs" : DOC_PATH;

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 'M': { /* master sync ID */

        u8* c;

        if (sync_id) FATAL("Multiple -S or -M options not supported");
        sync_id = ck_strdup(optarg);

        if ((c = strchr(sync_id, ':'))) {

          *c = 0;

          if (sscanf(c + 1, "%u/%u", &master_id, &master_max) != 2 ||
              !master_id || !master_max || master_id > master_max ||
              master_max > 1000000) FATAL("Bogus master ID passed to -M");

        }

        force_deterministic = 1;

      }

      break;

    case 'S':

      if (sync_id) FATAL("Multiple -S or -M options not supported");
      sync_id = ck_strdup(optarg);
      break;

    case 'f': /* target file */

      if (out_file) FATAL("Multiple -f options not supported");
      out_file = optarg;
      break;

    case 'x': /* dictionary */
```

```
      if (extras_dir) FATAL("Multiple -x options not supported");
      extras_dir = optarg;
      break;

    case 't': { /* timeout */

        u8 suffix = 0;

        if (timeout_given) FATAL("Multiple -t options not supported");

        if (sscanf(optarg, "%u%c", &exec_tmout, &suffix) < 1 ||
            optarg[0] == '-') FATAL("Bad syntax used for -t");

        if (exec_tmout < 5) FATAL("Dangerously low value of -t");

        if (suffix == '+') timeout_given = 2; else timeout_given = 1;

        break;

    }

    case 'm': { /* mem limit */

        u8 suffix = 'M';

        if (mem_limit_given) FATAL("Multiple -m options not supported");
        mem_limit_given = 1;

        if (!strcmp(optarg, "none")) {

          mem_limit = 0;
          break;

        }

        if (sscanf(optarg, "%llu%c", &mem_limit, &suffix) < 1 ||
            optarg[0] == '-') FATAL("Bad syntax used for -m");

        switch (suffix) {

          case 'T': mem_limit *= 1024 * 1024; break;
          case 'G': mem_limit *= 1024; break;
          case 'k': mem_limit /= 1024; break;
          case 'M': break;

          default:  FATAL("Unsupported suffix or bad syntax for -m");

        }

        if (mem_limit < 5) FATAL("Dangerously low value of -m");

        if (sizeof(rlim_t) == 4 && mem_limit > 2000)
          FATAL("Value of -m out of range on 32-bit systems");

      }

      break;

    case 'b': { /* bind CPU core */
```

```
    if (cpu_to_bind_given) FATAL("Multiple -b options not supported");
    cpu_to_bind_given = 1;

    if (sscanf(optarg, "%u", &cpu_to_bind) < 1 ||
        optarg[0] == '-') FATAL("Bad syntax used for -b");

    break;

  }

  case 'd': /* skip deterministic */

    if (skip_deterministic) FATAL("Multiple -d options not supported");
    skip_deterministic = 1;
    use_splicing = 1;
    break;

  case 'B': /* load bitmap */

    /* This is a secret undocumented option! It is useful if you find
       an interesting test case during a normal fuzzing process, and want
       to mutate it without rediscovering any of the test cases already
       found during an earlier run.

       To use this mode, you need to point -B to the fuzz_bitmap produced
       by an earlier run for the exact same binary... and that's it.

       I only used this once or twice to get variants of a particular
       file, so I'm not making this an official setting. */

    if (in_bitmap) FATAL("Multiple -B options not supported");

    in_bitmap = optarg;
    read_bitmap(in_bitmap);
    break;

  case 'C': /* crash mode */

    if (crash_mode) FATAL("Multiple -C options not supported");
    crash_mode = FAULT_CRASH;
    break;

  case 'n': /* dumb mode */

    if (dumb_mode) FATAL("Multiple -n options not supported");
    if (getenv("AFL_DUMB_FORKSRV")) dumb_mode = 2; else dumb_mode = 1;

    break;

  case 'T': /* banner */

    if (use_banner) FATAL("Multiple -T options not supported");
    use_banner = optarg;
    break;

  case 'Q': /* QEMU mode */

    if (qemu_mode) FATAL("Multiple -Q options not supported");
    qemu_mode = 1;
```

```
      if (!mem_limit_given) mem_limit = MEM_LIMIT_QEMU;

      break;

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

      /* Version number has been printed already, just quit. */
      exit(0);

    default:

      usage(argv[0]);

  }

if (optind == argc || !in_dir || !out_dir) usage(argv[0]);

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");

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);

}

#endif /* !AFL_LIB */
```

afl-gcc.c

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
```

```
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - wrapper for GCC and clang
   ----------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This program is a drop-in replacement for GCC or clang. The most common way
   of using it is to pass the path to afl-gcc or afl-clang via CC when invoking
   ./configure.

   (Of course, use CXX and point it to afl-g++ / afl-clang++ for C++ code.)

   The wrapper needs to know the path to afl-as (renamed to 'as'). The default
   is /usr/local/lib/afl/. A convenient way to specify alternative directories
   would be to set AFL_PATH.

   If AFL_HARDEN is set, the wrapper will compile the target app with various
   hardening options that may help detect memory management issues more
   reliably. You can also specify AFL_USE_ASAN to enable ASAN.

   If you want to call a non-default compiler as a next step of the chain,
   specify its location via AFL_CC or AFL_CXX.

*/

#define AFL_MAIN

#include "config.h"
#include "types.h"
#include "debug.h"
#include "alloc-inl.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

static u8*  as_path;                /* Path to the AFL 'as' wrapper      */
static u8** cc_params;              /* Parameters passed to the real CC  */
static u32  cc_par_cnt = 1;         /* Param count, including argv0      */
static u8   be_quiet,               /* Quiet mode                        */
            clang_mode;             /* Invoked as afl-clang*?            */


/* Try to find our "fake" GNU assembler in AFL_PATH or at the location derived
   from argv[0]. If that fails, abort. */

static void find_as(u8* argv0) {

  u8 *afl_path = getenv("AFL_PATH");
  u8 *slash, *tmp;

  if (afl_path) {
```

```
    tmp = alloc_printf("%s/as", afl_path);

    if (!access(tmp, X_OK)) {
      as_path = afl_path;
      ck_free(tmp);
      return;
    }

    ck_free(tmp);

  }

  slash = strrchr(argv0, '/');

  if (slash) {

    u8 *dir;

    *slash = 0;
    dir = ck_strdup(argv0);
    *slash = '/';

    tmp = alloc_printf("%s/afl-as", dir);

    if (!access(tmp, X_OK)) {
      as_path = dir;
      ck_free(tmp);
      return;
    }

    ck_free(tmp);
    ck_free(dir);

  }

  if (!access(AFL_PATH "/as", X_OK)) {
    as_path = AFL_PATH;
    return;
  }

  FATAL("Unable to find AFL wrapper binary for 'as'. Please set AFL_PATH");

}


/* Copy argv to cc_params, making the necessary edits. */

static void edit_params(u32 argc, char** argv) {

  u8 fortify_set = 0, asan_set = 0;
  u8 *name;

#if defined(__FreeBSD__) && defined(__x86_64__)
  u8 m32_set = 0;
#endif

  cc_params = ck_alloc((argc + 128) * sizeof(u8*));

  name = strrchr(argv[0], '/');
  if (!name) name = argv[0]; else name++;
```

```
  if (!strncmp(name, "afl-clang", 9)) {

    clang_mode = 1;

    setenv(CLANG_ENV_VAR, "1", 1);

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

  } else {

    /* With GCJ and Eclipse installed, you can actually compile Java! The
       instrumentation will work (amazingly). Alas, unhandled exceptions do
       not call abort(), so afl-fuzz would need to be modified to equate
       non-zero exit codes with crash conditions when working with Java
       binaries. Meh. */

#ifdef __APPLE__

    if (!strcmp(name, "afl-g++")) cc_params[0] = getenv("AFL_CXX");
    else if (!strcmp(name, "afl-gcj")) cc_params[0] = getenv("AFL_GCJ");
    else cc_params[0] = getenv("AFL_CC");

    if (!cc_params[0]) {

      SAYF("\n" cLRD "[-] " cRST
           "On Apple systems, 'gcc' is usually just a wrapper for clang. Please use the\n"
           "    'afl-clang' utility instead of 'afl-gcc'. If you really have GCC installed,\n"
           "    set AFL_CC or AFL_CXX to specify the correct path to that compiler.\n");

      FATAL("AFL_CC or AFL_CXX required on MacOS X");

    }

#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;

}


/* Main entry point */

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

```
  if (isatty(2) && !getenv("AFL_QUIET")) {

    SAYF(cCYA "afl-cc " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  } else be_quiet = 1;

  if (argc < 2) {

    SAYF("\n"
         "This is a helper application for afl-fuzz. It serves as a drop-in replacement\n"
         "for gcc or clang, letting you recompile third-party code with the required\n"
         "runtime instrumentation. A common use pattern would be one of the following:\n\n"

         "  CC=%s/afl-gcc ./configure\n"
         "  CXX=%s/afl-g++ ./configure\n\n"

         "You can specify custom next-stage toolchain via AFL_CC, AFL_CXX, and AFL_AS.\n"
         "Setting AFL_HARDEN enables hardening optimizations in the compiled code.\n\n",
         BIN_PATH, BIN_PATH);

    exit(1);

  }

  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;

}
```

afl-gotcpu.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - free CPU gizmo
   ------------------------------------
```

```
   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This tool provides a fairly accurate measurement of CPU preemption rate.
   It is meant to complement the quick-and-dirty load average widget shown
   in the afl-fuzz UI. See docs/parallel_fuzzing.txt for more info.

   For some work loads, the tool may actually suggest running more instances
   than you have CPU cores. This can happen if the tested program is spending
   a portion of its run time waiting for I/O, rather than being 100%
   CPU-bound.

   The idea for the getrusage()-based approach comes from Jakub Wilk.
*/

#define AFL_MAIN
#include "android-ashmem.h"
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sched.h>

#include <sys/time.h>
#include <sys/times.h>
#include <sys/resource.h>
#include <sys/wait.h>

#include "types.h"
#include "debug.h"

#ifdef __linux__
#  define HAVE_AFFINITY 1
#endif /* __linux__ */


/* Get unix time in microseconds. */

static u64 get_cur_time_us(void) {

  struct timeval tv;
  struct timezone tz;

  gettimeofday(&tv, &tz);

  return (tv.tv_sec * 1000000ULL) + tv.tv_usec;

}


/* Get CPU usage in microseconds. */

static u64 get_cpu_usage_us(void) {

  struct rusage u;

  getrusage(RUSAGE_SELF, &u);
```

```
  return (u.ru_utime.tv_sec * 1000000ULL) + u.ru_utime.tv_usec +
         (u.ru_stime.tv_sec * 1000000ULL) + u.ru_stime.tv_usec;

}


/* Measure preemption rate. */

static u32 measure_preemption(u32 target_ms) {

  static volatile u32 v1, v2;

  u64 st_t, en_t, st_c, en_c, real_delta, slice_delta;
  s32 loop_repeats = 0;

  st_t = get_cur_time_us();
  st_c = get_cpu_usage_us();

repeat_loop:

  v1 = CTEST_BUSY_CYCLES;

  while (v1--) v2++;
  sched_yield();

  en_t = get_cur_time_us();

  if (en_t - st_t < target_ms * 1000) {
    loop_repeats++;
    goto repeat_loop;
  }

  /* Let's see what percentage of this time we actually had a chance to
     run, and how much time was spent in the penalty box. */

  en_c = get_cpu_usage_us();

  real_delta  = (en_t - st_t) / 1000;
  slice_delta = (en_c - st_c) / 1000;

  return real_delta * 100 / slice_delta;

}


/* Do the benchmark thing. */

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

#ifdef HAVE_AFFINITY

  u32 cpu_cnt = sysconf(_SC_NPROCESSORS_ONLN),
      idle_cpus = 0, maybe_cpus = 0, i;

  SAYF(cCYA "afl-gotcpu " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  ACTF("Measuring per-core preemption rate (this will take %0.02f sec)...",
       ((double)CTEST_CORE_TRG_MS) / 1000);

  for (i = 0; i < cpu_cnt; i++) {
```

```
  s32 fr = fork();

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

  if (!fr) {

    cpu_set_t c;
    u32 util_perc;

    CPU_ZERO(&c);
    CPU_SET(i, &c);

    if (sched_setaffinity(0, sizeof(c), &c))
      PFATAL("sched_setaffinity failed for cpu %d", i);

    util_perc = measure_preemption(CTEST_CORE_TRG_MS);

    if (util_perc < 110) {

      SAYF("    Core #%u: " cLGN "AVAILABLE " cRST "(%u%%)\n", i, util_perc);
      exit(0);

    } else if (util_perc < 250) {

      SAYF("    Core #%u: " cYEL "CAUTION " cRST "(%u%%)\n", i, util_perc);
      exit(1);

    }

    SAYF("    Core #%u: " cLRD "OVERBOOKED " cRST "(%u%%)\n" cRST, i,
         util_perc);
    exit(2);

  }

}

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

  int ret;
  if (waitpid(-1, &ret, 0) < 0) PFATAL("waitpid failed");

  if (WEXITSTATUS(ret) == 0) idle_cpus++;
  if (WEXITSTATUS(ret) <= 1) maybe_cpus++;

}

SAYF(cGRA "\n>>> ");

if (idle_cpus) {

  if (maybe_cpus == idle_cpus) {

    SAYF(cLGN "PASS: " cRST "You can run more processes on %u core%s.",
         idle_cpus, idle_cpus > 1 ? "s" : "");

  } else {

    SAYF(cLGN "PASS: " cRST "You can run more processes on %u to %u core%s.",
```

```
          idle_cpus, maybe_cpus, maybe_cpus > 1 ? "s" : "");

    }

    SAYF(cGRA " <<<" cRST "\n\n");
    return 0;

  }

  if (maybe_cpus) {

    SAYF(cYEL "CAUTION: " cRST "You may still have %u core%s available.",
         maybe_cpus, maybe_cpus > 1 ? "s" : "");
    SAYF(cGRA " <<<" cRST "\n\n");
    return 1;

  }

  SAYF(cLRD "FAIL: " cRST "All cores are overbooked.");
  SAYF(cGRA " <<<" cRST "\n\n");
  return 2;

#else

  u32 util_perc;

  SAYF(cCYA "afl-gotcpu " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  /* Run a busy loop for CTEST_TARGET_MS. */

  ACTF("Measuring gross preemption rate (this will take %0.02f sec)...",
       ((double)CTEST_TARGET_MS) / 1000);

  util_perc = measure_preemption(CTEST_TARGET_MS);

  /* Deliver the final verdict. */

  SAYF(cGRA "\n>>> ");

  if (util_perc < 105) {

    SAYF(cLGN "PASS: " cRST "You can probably run additional processes.");

  } else if (util_perc < 130) {

    SAYF(cYEL "CAUTION: " cRST "Your CPU may be somewhat overbooked (%u%%).",
         util_perc);

  } else {

    SAYF(cLRD "FAIL: " cRST "Your CPU is overbooked (%u%%).", util_perc);

  }

  SAYF(cGRA " <<<" cRST "\n\n");

  return (util_perc > 105) + (util_perc > 130);

#endif /* ^HAVE_AFFINITY */
```

```
}
```

afl-plot

```
#!/bin/sh
#
# american fuzzy lop - Advanced Persistent Graphing
# -------------------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
# Based on a design & prototype by Michael Rash.
#
# Copyright 2014, 2015 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#

echo "progress plotting utility for afl-fuzz by <lcamtuf@google.com>"
echo

if [ ! "$#" = "2" ]; then

  cat 1>&2 <<_EOF_
This program generates gnuplot images from afl-fuzz output data. Usage:

$0 afl_state_dir graph_output_dir

The afl_state_dir parameter should point to an existing state directory for any
active or stopped instance of afl-fuzz; while graph_output_dir should point to
an empty directory where this tool can write the resulting plots to.

The program will put index.html and three PNG images in the output directory;
you should be able to view it with any web browser of your choice.

_EOF_

  exit 1

fi

if [ "$AFL_ALLOW_TMP" = "" ]; then

  echo "$1" | grep -qE '^(/var)?/tmp/'
  T1="$?"

  echo "$2" | grep -qE '^(/var)?/tmp/'
  T2="$?"

  if [ "$T1" = "0" -o "$T2" = "0" ]; then

    echo "[-] Error: this script shouldn't be used with shared /tmp directories." 1>&2
    exit 1

  fi
```

```
fi

if [ ! -f "$1/plot_data" ]; then

  echo "[-] Error: input directory is not valid (missing 'plot_data')." 1>&2
  exit 1

fi

BANNER="`cat "$1/fuzzer_stats" | grep '^afl_banner ' | cut -d: -f2- | cut -b2-`"

test "$BANNER" = "" && BANNER="(none)"

GNUPLOT=`which gnuplot 2>/dev/null`

if [ "$GNUPLOT" = "" ]; then

  echo "[-] Error: can't find 'gnuplot' in your \$PATH." 1>&2
  exit 1

fi

mkdir "$2" 2>/dev/null

if [ ! -d "$2" ]; then

  echo "[-] Error: unable to create the output directory - pick another location." 1>&2
  exit 1

fi

rm -f "$2/high_freq.png" "$2/low_freq.png" "$2/exec_speed.png"
mv -f "$2/index.html" "$2/index.html.orig" 2>/dev/null

echo "[*] Generating plots..."

(

cat <<_EOF_
set terminal png truecolor enhanced size 1000,300 butt

set output '$2/high_freq.png'

set xdata time
set timefmt '%s'
set format x "%b %d\n%H:%M"
set tics font 'small'
unset mxtics
unset mytics

set grid xtics linetype 0 linecolor rgb '#e0e0e0'
set grid ytics linetype 0 linecolor rgb '#e0e0e0'
set border linecolor rgb '#50c0f0'
set tics textcolor rgb '#000000'
set key outside

set autoscale xfixmin
set autoscale xfixmax
```

```
plot '$1/plot_data' using 1:4 with filledcurve x1 title 'total paths' linecolor rgb '#000000'
fillstyle transparent solid 0.2 noborder, \\
     '' using 1:3 with filledcurve x1 title 'current path' linecolor rgb '#f0f0f0' fillstyle
transparent solid 0.5 noborder, \\
     '' using 1:5 with lines title 'pending paths' linecolor rgb '#0090ff' linewidth 3, \\
     '' using 1:6 with lines title 'pending favs' linecolor rgb '#c00080' linewidth 3, \\
     '' using 1:2 with lines title 'cycles done' linecolor rgb '#c000f0' linewidth 3

set terminal png truecolor enhanced size 1000,200 butt
set output '$2/low_freq.png'

plot '$1/plot_data' using 1:8 with filledcurve x1 title '' linecolor rgb '#c00080' fillstyle
transparent solid 0.2 noborder, \\
     '' using 1:8 with lines title ' uniq crashes' linecolor rgb '#c00080' linewidth 3, \\
     '' using 1:9 with lines title 'uniq hangs' linecolor rgb '#c000f0' linewidth 3, \\
     '' using 1:10 with lines title 'levels' linecolor rgb '#0090ff' linewidth 3

set terminal png truecolor enhanced size 1000,200 butt
set output '$2/exec_speed.png'

plot '$1/plot_data' using 1:11 with filledcurve x1 title '' linecolor rgb '#0090ff' fillstyle
transparent solid 0.2 noborder, \\
     '$1/plot_data' using 1:11 with lines title '    execs/sec' linecolor rgb '#0090ff'
linewidth 3 smooth bezier;

_EOF_

) | gnuplot

if [ ! -s "$2/exec_speed.png" ]; then

  echo "[-] Error: something went wrong! Perhaps you have an ancient version of gnuplot?" 1>&2
  exit 1

fi

echo "[*] Generating index.html..."

cat >"$2/index.html" <<_EOF_
<table style="font-family: 'Trebuchet MS', 'Tahoma', 'Arial', 'Helvetica'">
<tr><td style="width: 18ex"><b>Banner:</b></td><td>$BANNER</td></tr>
<tr><td><b>Directory:</b></td><td>$1</td></tr>
<tr><td><b>Generated on:</b></td><td>`date`</td></tr>
</table>
<p>
<img src="high_freq.png" width=1000 height=300><p>
<img src="low_freq.png" width=1000 height=200><p>
<img src="exec_speed.png" width=1000 height=200>

_EOF_

# Make it easy to remotely view results when outputting directly to a directory
# served by Apache or other HTTP daemon. Since the plots aren't horribly
# sensitive, this seems like a reasonable trade-off.

chmod 755 "$2"
chmod 644 "$2/high_freq.png" "$2/low_freq.png" "$2/exec_speed.png" "$2/index.html"

echo "[+] All done - enjoy your charts!"
```

```
exit 0
```

afl-showmap.c

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - map display utility
   ----------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   A very simple tool that runs the targeted binary and displays
   the contents of the trace bitmap in a human-readable form. Useful in
   scripts to eliminate redundant inputs and perform other checks.

   Exit code is 2 if the target program crashes; 1 if it times out or
   there is a problem executing it; or 0 if execution is successful.
*/

#define AFL_MAIN
#include "android-ashmem.h"

#include "config.h"
#include "types.h"
#include "debug.h"
#include "alloc-inl.h"
#include "hash.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <dirent.h>
#include <fcntl.h>

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```
#include <sys/resource.h>

static s32 child_pid;                 /* PID of the tested program         */

static u8* trace_bits;                /* SHM with instrumentation bitmap   */

static u8 *out_file,                  /* Trace output file                 */
          *doc_path,                  /* Path to docs                      */
          *target_path,               /* Path to target binary             */
          *at_file;                   /* Substitution string for @@        */

static u32 exec_tmout;                /* Exec timeout (ms)                 */

static u64 mem_limit = MEM_LIMIT;     /* Memory limit (MB)                 */

static s32 shm_id;                    /* ID of the SHM region              */

static u8  quiet_mode,                /* Hide non-essential messages?      */
           edges_only,                /* Ignore hit counts?                */
           cmin_mode,                 /* Generate output in afl-cmin mode? */
           binary_mode,               /* Write output as a binary map      */
           keep_cores;                /* Allow coredumps?                  */

static volatile u8
           stop_soon,                 /* Ctrl-C pressed?                   */
           child_timed_out,           /* Child timed out?                  */
           child_crashed;             /* Child crashed?                    */

/* Classify tuple counts. Instead of mapping to individual bits, as in
   afl-fuzz.c, we map to more user-friendly numbers between 1 and 8. */

static const u8 count_class_human[256] = {

  [0]           = 0,
  [1]           = 1,
  [2]           = 2,
  [3]           = 3,
  [4 ... 7]     = 4,
  [8 ... 15]    = 5,
  [16 ... 31]   = 6,
  [32 ... 127]  = 7,
  [128 ... 255] = 8

};

static const u8 count_class_binary[256] = {

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

};

static void classify_counts(u8* mem, const u8* map) {
```

```
  u32 i = MAP_SIZE;

  if (edges_only) {

    while (i--) {
      if (*mem) *mem = 1;
      mem++;
    }

  } else {

    while (i--) {
      *mem = map[*mem];
      mem++;
    }

  }

}


/* Get rid of shared memory (atexit handler). */

static void remove_shm(void) {

  shmctl(shm_id, IPC_RMID, NULL);

}


/* Configure shared memory. */

static void setup_shm(void) {

  u8* shm_str;

  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);

  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");

}

/* Write results. */

static u32 write_results(void) {

  s32 fd;
```

```
u32 i, ret = 0;

u8  cco = !!getenv("AFL_CMIN_CRASHES_ONLY"),
    caa = !!getenv("AFL_CMIN_ALLOW_ANY");

if (!strncmp(out_file, "/dev/", 5)) {

  fd = open(out_file, O_WRONLY, 0600);
  if (fd < 0) PFATAL("Unable to open '%s'", out_file);

} else if (!strcmp(out_file, "-")) {

  fd = dup(1);
  if (fd < 0) PFATAL("Unable to open stdout");

} else {

  unlink(out_file); /* Ignore errors */
  fd = open(out_file, O_WRONLY | O_CREAT | O_EXCL, 0600);
  if (fd < 0) PFATAL("Unable to create '%s'", out_file);

}


if (binary_mode) {

  for (i = 0; i < MAP_SIZE; i++)
    if (trace_bits[i]) ret++;

  ck_write(fd, trace_bits, MAP_SIZE, out_file);
  close(fd);

} else {

  FILE* f = fdopen(fd, "w");

  if (!f) PFATAL("fdopen() failed");

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

    if (!trace_bits[i]) continue;
    ret++;

    if (cmin_mode) {

      if (child_timed_out) break;
      if (!caa && child_crashed != cco) break;

      fprintf(f, "%u%u\n", trace_bits[i], i);

    } else fprintf(f, "%06u:%u\n", i, trace_bits[i]);

  }

  fclose(f);

}

return ret;
```

```
}


/* Handle timeout signal. */

static void handle_timeout(int sig) {

  child_timed_out = 1;
  if (child_pid > 0) kill(child_pid, SIGKILL);

}


/* Execute target application. */

static void run_target(char** argv) {

  static struct itimerval it;
  int status = 0;

  if (!quiet_mode)
    SAYF("-- Program output begins --\n" cRST);

  MEM_BARRIER();

  child_pid = fork();

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

  if (!child_pid) {

    struct rlimit r;

    if (quiet_mode) {

      s32 fd = open("/dev/null", O_RDWR);

      if (fd < 0 || dup2(fd, 1) < 0 || dup2(fd, 2) < 0) {
        *(u32*)trace_bits = EXEC_FAIL_SIG;
        PFATAL("Descriptor initialization failed");
      }

      close(fd);

    }

    if (mem_limit) {

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

#ifdef RLIMIT_AS

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

#else

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

#endif /* ^RLIMIT_AS */
```

```
  }

  if (!keep_cores) r.rlim_max = r.rlim_cur = 0;
  else r.rlim_max = r.rlim_cur = RLIM_INFINITY;

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

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

  setsid();

  execv(target_path, argv);

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

}

/* Configure timeout, wait for child, cancel timeout. */

if (exec_tmout) {

  child_timed_out = 0;
  it.it_value.tv_sec = (exec_tmout / 1000);
  it.it_value.tv_usec = (exec_tmout % 1000) * 1000;

}

setitimer(ITIMER_REAL, &it, NULL);

if (waitpid(child_pid, &status, 0) <= 0) FATAL("waitpid() failed");

child_pid = 0;
it.it_value.tv_sec = 0;
it.it_value.tv_usec = 0;
setitimer(ITIMER_REAL, &it, NULL);

MEM_BARRIER();

/* Clean up bitmap, analyze exit condition, etc. */

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

classify_counts(trace_bits, binary_mode ?
                count_class_binary : count_class_human);

if (!quiet_mode)
  SAYF(cRST "-- Program output ends --\n");

if (!child_timed_out && !stop_soon && WIFSIGNALED(status))
  child_crashed = 1;

if (!quiet_mode) {

  if (child_timed_out)
    SAYF(cLRD "\n+++ Program timed off +++\n" cRST);
  else if (stop_soon)
    SAYF(cLRD "\n+++ Program aborted by user +++\n" cRST);
  else if (child_crashed)
```

```
      SAYF(cLRD "\n+++ Program killed by signal %u +++\n" cRST, WTERMSIG(status));

  }

}

/* Handle Ctrl-C and the like. */

static void handle_stop_sig(int sig) {

  stop_soon = 1;

  if (child_pid > 0) kill(child_pid, SIGKILL);

}

/* Do basic preparations - persistent fds, filenames, etc. */

static void set_up_environment(void) {

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

  setenv("MSAN_OPTIONS", "exit_code=" STRINGIFY(MSAN_ERROR) ":"
                         "symbolize=0:"
                         "abort_on_error=1:"
                         "allocator_may_return_null=1:"
                         "msan_track_origins=0", 0);

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

}

/* Setup signal handlers, duh. */

static void setup_signal_handlers(void) {

  struct sigaction sa;

  sa.sa_handler   = NULL;
  sa.sa_flags     = SA_RESTART;
  sa.sa_sigaction = NULL;

  sigemptyset(&sa.sa_mask);

  /* Various ways of saying "stop". */

  sa.sa_handler = handle_stop_sig;
  sigaction(SIGHUP, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);
```

```
  /* Exec timeout notifications. */

  sa.sa_handler = handle_timeout;
  sigaction(SIGALRM, &sa, NULL);

}


/* Detect @@ in args. */

static void detect_file_args(char** argv) {

  u32 i = 0;
  u8* cwd = getcwd(NULL, 0);

  if (!cwd) PFATAL("getcwd() failed");

  while (argv[i]) {

    u8* aa_loc = strstr(argv[i], "@@");

    if (aa_loc) {

      u8 *aa_subst, *n_arg;

      if (!at_file) FATAL("@@ syntax is not supported by this tool.");

      /* Be sure that we're always using fully-qualified paths. */

      if (at_file[0] == '/') aa_subst = at_file;
      else aa_subst = alloc_printf("%s/%s", cwd, at_file);

      /* Construct a replacement argv value. */

      *aa_loc = 0;
      n_arg = alloc_printf("%s%s%s", argv[i], aa_subst, aa_loc + 2);
      argv[i] = n_arg;
      *aa_loc = '@';

      if (at_file[0] != '/') ck_free(aa_subst);

    }

    i++;

  }

  free(cwd); /* not tracked */

}


/* Show banner. */

static void show_banner(void) {

  SAYF(cCYA "afl-showmap " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

}
```

```
/* Display usage hints. */

static void usage(u8* argv0) {

  show_banner();

  SAYF("\n%s [ options ] -- /path/to/target_app [ ... ]\n\n"

       "Required parameters:\n\n"

       "  -o file       - file to write the trace data to\n\n"

       "Execution control settings:\n\n"

       "  -t msec       - timeout for each run (none)\n"
       "  -m megs       - memory limit for child process (%u MB)\n"
       "  -Q            - use binary-only instrumentation (QEMU mode)\n\n"

       "Other settings:\n\n"

       "  -q            - sink program's output and don't show messages\n"
       "  -e            - show edge coverage only, ignore hit counts\n"
       "  -c            - allow core dumps\n"
       "  -V            - show version number and exit\n\n"

       "This tool displays raw tuple data captured by AFL instrumentation.\n"
       "For additional help, consult %s/README.\n\n" cRST,

       argv0, MEM_LIMIT, doc_path);

  exit(1);

}


/* Find binary. */

static void find_binary(u8* fname) {

  u8* env_path = 0;
  struct stat st;

  if (strchr(fname, '/') || !(env_path = getenv("PATH"))) {

    target_path = ck_strdup(fname);

    if (stat(target_path, &st) || !S_ISREG(st.st_mode) ||
        !(st.st_mode & 0111) || st.st_size < 4)
      FATAL("Program '%s' not found or not executable", fname);

  } else {

    while (env_path) {

      u8 *cur_elem, *delim = strchr(env_path, ':');

      if (delim) {

        cur_elem = ck_alloc(delim - env_path + 1);
```

```
        memcpy(cur_elem, env_path, delim - env_path);
        delim++;

      } else cur_elem = ck_strdup(env_path);

      env_path = delim;

      if (cur_elem[0])
        target_path = alloc_printf("%s/%s", cur_elem, fname);
      else
        target_path = ck_strdup(fname);

      ck_free(cur_elem);

      if (!stat(target_path, &st) && S_ISREG(st.st_mode) &&
          (st.st_mode & 0111) && st.st_size >= 4) break;

      ck_free(target_path);
      target_path = 0;

    }

    if (!target_path) FATAL("Program '%s' not found or not executable", fname);

  }

}


/* Fix up argv for QEMU. */

static char** get_qemu_argv(u8* own_loc, char** argv, int argc) {

  char** new_argv = ck_alloc(sizeof(char*) * (argc + 4));
  u8 *tmp, *cp, *rsl, *own_copy;

  /* Workaround for a QEMU stability glitch. */

  setenv("QEMU_LOG", "nochain", 1);

  memcpy(new_argv + 3, argv + 1, sizeof(char*) * argc);

  new_argv[2] = target_path;
  new_argv[1] = "--";

  /* Now we need to actually find qemu for argv[0]. */

  tmp = getenv("AFL_PATH");

  if (tmp) {

    cp = alloc_printf("%s/afl-qemu-trace", tmp);

    if (access(cp, X_OK))
      FATAL("Unable to find '%s'", tmp);

    target_path = new_argv[0] = cp;
    return new_argv;

  }
```

```
  own_copy = ck_strdup(own_loc);
  rsl = strrchr(own_copy, '/');

  if (rsl) {

    *rsl = 0;

    cp = alloc_printf("%s/afl-qemu-trace", own_copy);
    ck_free(own_copy);

    if (!access(cp, X_OK)) {

      target_path = new_argv[0] = cp;
      return new_argv;

    }

  } else ck_free(own_copy);

  if (!access(BIN_PATH "/afl-qemu-trace", X_OK)) {

    target_path = new_argv[0] = BIN_PATH "/afl-qemu-trace";
    return new_argv;

  }

  FATAL("Unable to find 'afl-qemu-trace'.");

}


/* Main entry point */

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

  s32 opt;
  u8  mem_limit_given = 0, timeout_given = 0, qemu_mode = 0;
  u32 tcnt;
  char** use_argv;

  doc_path = access(DOC_PATH, F_OK) ? "docs" : DOC_PATH;

  while ((opt = getopt(argc,argv,"+o:m:t:A:eqZQbcV")) > 0)

    switch (opt) {

      case 'o':

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

      case 'm': {

          u8 suffix = 'M';

          if (mem_limit_given) FATAL("Multiple -m options not supported");
          mem_limit_given = 1;
```

```
    if (!strcmp(optarg, "none")) {

      mem_limit = 0;
      break;

    }

    if (sscanf(optarg, "%llu%c", &mem_limit, &suffix) < 1 ||
        optarg[0] == '-') FATAL("Bad syntax used for -m");

    switch (suffix) {

      case 'T': mem_limit *= 1024 * 1024; break;
      case 'G': mem_limit *= 1024; break;
      case 'k': mem_limit /= 1024; break;
      case 'M': break;

      default:  FATAL("Unsupported suffix or bad syntax for -m");

    }

    if (mem_limit < 5) FATAL("Dangerously low value of -m");

    if (sizeof(rlim_t) == 4 && mem_limit > 2000)
      FATAL("Value of -m out of range on 32-bit systems");

  }

  break;

case 't':

  if (timeout_given) FATAL("Multiple -t options not supported");
  timeout_given = 1;

  if (strcmp(optarg, "none")) {
    exec_tmout = atoi(optarg);

    if (exec_tmout < 20 || optarg[0] == '-')
      FATAL("Dangerously low value of -t");

  }

  break;

case 'e':

  if (edges_only) FATAL("Multiple -e options not supported");
  edges_only = 1;
  break;

case 'q':

  if (quiet_mode) FATAL("Multiple -q options not supported");
  quiet_mode = 1;
  break;

case 'Z':

  /* This is an undocumented option to write data in the syntax expected
```

```
          by afl-cmin. Nobody else should have any use for this. */

        cmin_mode  = 1;
        quiet_mode = 1;
        break;

      case 'A':

        /* Another afl-cmin specific feature. */
        at_file = optarg;
        break;

      case 'Q':

        if (qemu_mode) FATAL("Multiple -Q options not supported");
        if (!mem_limit_given) mem_limit = MEM_LIMIT_QEMU;

        qemu_mode = 1;
        break;

      case 'b':

        /* Secret undocumented mode. Writes output in raw binary format
           similar to that dumped by afl-fuzz in <out_dir/queue/fuzz_bitmap. */

        binary_mode = 1;
        break;

      case 'c':

        if (keep_cores) FATAL("Multiple -c options not supported");
        keep_cores = 1;
        break;

      case 'V':

        show_banner();
        exit(0);

      default:

        usage(argv[0]);

    }

  if (optind == argc || !out_file) usage(argv[0]);

  setup_shm();
  setup_signal_handlers();

  set_up_environment();

  find_binary(argv[optind]);

  if (!quiet_mode) {
    show_banner();
    ACTF("Executing '%s'...\n", target_path);
  }

  detect_file_args(argv + optind);
```

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

  run_target(use_argv);

  tcnt = write_results();

  if (!quiet_mode) {

    if (!tcnt) FATAL("No instrumentation detected" cRST);
    OKF("Captured %u tuples in '%s'." cRST, tcnt, out_file);

  }

  exit(child_crashed * 2 + child_timed_out);

}
```

afl-tmin.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - test case minimizer
   ----------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   A simple test case minimizer that takes an input file and tries to remove
   as much data as possible while keeping the binary in a crashing state
   *or* producing consistent instrumentation output (the mode is auto-selected
   based on the initially observed behavior).
*/

#define AFL_MAIN
#include "android-ashmem.h"

#include "config.h"
#include "types.h"
#include "debug.h"
```

```c
#include "alloc-inl.h"
#include "hash.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <signal.h>
#include <dirent.h>
#include <fcntl.h>

#include <sys/wait.h>
#include <sys/time.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/resource.h>

static s32 child_pid;                 /* PID of the tested program         */

static u8 *trace_bits,                /* SHM with instrumentation bitmap   */
          *mask_bitmap;               /* Mask for trace bits (-B)          */

static u8 *in_file,                   /* Minimizer input test case         */
          *out_file,                  /* Minimizer output file             */
          *prog_in,                   /* Targeted program input file       */
          *target_path,               /* Path to target binary             */
          *doc_path;                  /* Path to docs                      */

static u8* in_data;                   /* Input data for trimming           */

static u32 in_len,                    /* Input data length                 */
           orig_cksum,                /* Original checksum                 */
           total_execs,               /* Total number of execs             */
           missed_hangs,              /* Misses due to hangs               */
           missed_crashes,            /* Misses due to crashes             */
           missed_paths,              /* Misses due to exec path diffs     */
           exec_tmout = EXEC_TIMEOUT; /* Exec timeout (ms)                 */

static u64 mem_limit = MEM_LIMIT;     /* Memory limit (MB)                 */

static s32 shm_id,                    /* ID of the SHM region              */
           dev_null_fd = -1;          /* FD to /dev/null                   */

static u8  crash_mode,                /* Crash-centric mode?               */
           exit_crash,                /* Treat non-zero exit as crash?     */
           edges_only,                /* Ignore hit counts?                */
           exact_mode,                /* Require path match for crashes?   */
           use_stdin = 1;             /* Use stdin for program input?      */

static volatile u8
           stop_soon,                 /* Ctrl-C pressed?                   */
           child_timed_out;           /* Child timed out?                  */


/* Classify tuple counts. This is a slow & naive version, but good enough here. */

static const u8 count_class_lookup[256] = {
```

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

};

static void classify_counts(u8* mem) {

  u32 i = MAP_SIZE;

  if (edges_only) {

    while (i--) {
      if (*mem) *mem = 1;
      mem++;
    }

  } else {

    while (i--) {
      *mem = count_class_lookup[*mem];
      mem++;
    }

  }

}


/* Apply mask to classified bitmap (if set). */

static void apply_mask(u32* mem, u32* mask) {

  u32 i = (MAP_SIZE >> 2);

  if (!mask) return;

  while (i--) {

    *mem &= ~*mask;
    mem++;
    mask++;

  }

}


/* See if any bytes are set in the bitmap. */

static inline u8 anything_set(void) {

  u32* ptr = (u32*)trace_bits;
```

```
  u32  i    = (MAP_SIZE >> 2);

  while (i--) if (*(ptr++)) return 1;

  return 0;

}



/* Get rid of shared memory and temp files (atexit handler). */

static void remove_shm(void) {

  if (prog_in) unlink(prog_in); /* Ignore errors */
  shmctl(shm_id, IPC_RMID, NULL);

}


/* Configure shared memory. */

static void setup_shm(void) {

  u8* shm_str;

  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);

  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");

}


/* Read initial file. */

static void read_initial_file(void) {

  struct stat st;
  s32 fd = open(in_file, O_RDONLY);

  if (fd < 0) PFATAL("Unable to open '%s'", in_file);

  if (fstat(fd, &st) || !st.st_size)
    FATAL("Zero-sized input file.");

  if (st.st_size >= TMIN_MAX_FILE)
    FATAL("Input file is too large (%u MB max)", TMIN_MAX_FILE / 1024 / 1024);
```

```
  in_len  = st.st_size;
  in_data = ck_alloc_nozero(in_len);

  ck_read(fd, in_data, in_len, in_file);

  close(fd);

  OKF("Read %u byte%s from '%s'.", in_len, in_len == 1 ? "" : "s", in_file);

}


/* Write output file. */

static s32 write_to_file(u8* path, u8* mem, u32 len) {

  s32 ret;

  unlink(path); /* Ignore errors */

  ret = open(path, O_RDWR | O_CREAT | O_EXCL, 0600);

  if (ret < 0) PFATAL("Unable to create '%s'", path);

  ck_write(ret, mem, len, path);

  lseek(ret, 0, SEEK_SET);

  return ret;

}


/* Handle timeout signal. */

static void handle_timeout(int sig) {

  child_timed_out = 1;
  if (child_pid > 0) kill(child_pid, SIGKILL);

}


/* Execute target application. Returns 0 if the changes are a dud, or
   1 if they should be kept. */

static u8 run_target(char** argv, u8* mem, u32 len, u8 first_run) {

  static struct itimerval it;
  int status = 0;

  s32 prog_in_fd;
  u32 cksum;

  memset(trace_bits, 0, MAP_SIZE);
  MEM_BARRIER();

  prog_in_fd = write_to_file(prog_in, mem, len);

  child_pid = fork();
```

```
  if (child_pid < 0) PFATAL("fork() failed");

  if (!child_pid) {

    struct rlimit r;

    if (dup2(use_stdin ? prog_in_fd : dev_null_fd, 0) < 0 ||
        dup2(dev_null_fd, 1) < 0 ||
        dup2(dev_null_fd, 2) < 0) {

      *(u32*)trace_bits = EXEC_FAIL_SIG;
      PFATAL("dup2() failed");

    }

    close(dev_null_fd);
    close(prog_in_fd);

    setsid();

    if (mem_limit) {

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

#ifdef RLIMIT_AS

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

#else

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

#endif /* ^RLIMIT_AS */

    }

    r.rlim_max = r.rlim_cur = 0;
    setrlimit(RLIMIT_CORE, &r); /* Ignore errors */

    execv(target_path, argv);

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

  }

  close(prog_in_fd);

  /* Configure timeout, wait for child, cancel timeout. */

  child_timed_out = 0;
  it.it_value.tv_sec = (exec_tmout / 1000);
  it.it_value.tv_usec = (exec_tmout % 1000) * 1000;

  setitimer(ITIMER_REAL, &it, NULL);

  if (waitpid(child_pid, &status, 0) <= 0) FATAL("waitpid() failed");

  child_pid = 0;
```

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

setitimer(ITIMER_REAL, &it, NULL);

MEM_BARRIER();

/* Clean up bitmap, analyze exit condition, etc. */

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

classify_counts(trace_bits);
apply_mask((u32*)trace_bits, (u32*)mask_bitmap);
total_execs++;

if (stop_soon) {

  SAYF(cRST cLRD "\n+++ Minimization aborted by user +++\n" cRST);
  close(write_to_file(out_file, in_data, in_len));
  exit(1);

}

/* Always discard inputs that time out. */

if (child_timed_out) {

  missed_hangs++;
  return 0;

}

/* Handle crashing inputs depending on current mode. */

if (WIFSIGNALED(status) ||
    (WIFEXITED(status) && WEXITSTATUS(status) == MSAN_ERROR) ||
    (WIFEXITED(status) && WEXITSTATUS(status) && exit_crash)) {

  if (first_run) crash_mode = 1;

  if (crash_mode) {

    if (!exact_mode) return 1;

  } else {

    missed_crashes++;
    return 0;

  }

} else

/* Handle non-crashing inputs appropriately. */

if (crash_mode) {

  missed_paths++;
  return 0;
```

```
  }

  cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

  if (first_run) orig_cksum = cksum;

  if (orig_cksum == cksum) return 1;

  missed_paths++;
  return 0;

}


/* Find first power of two greater or equal to val. */

static u32 next_p2(u32 val) {

  u32 ret = 1;
  while (val > ret) ret <<= 1;
  return ret;

}


/* Actually minimize! */

static void minimize(char** argv) {

  static u32 alpha_map[256];

  u8* tmp_buf = ck_alloc_nozero(in_len);
  u32 orig_len = in_len, stage_o_len;

  u32 del_len, set_len, del_pos, set_pos, i, alpha_size, cur_pass = 0;
  u32 syms_removed, alpha_del0 = 0, alpha_del1, alpha_del2, alpha_d_total = 0;
  u8  changed_any, prev_del;

  /************************
   * BLOCK NORMALIZATION *
   ************************/

  set_len    = next_p2(in_len / TMIN_SET_STEPS);
  set_pos    = 0;

  if (set_len < TMIN_SET_MIN_SIZE) set_len = TMIN_SET_MIN_SIZE;

  ACTF(cBRI "Stage #0: " cRST "One-time block normalization...");

  while (set_pos < in_len) {

    u8  res;
    u32 use_len = MIN(set_len, in_len - set_pos);

    for (i = 0; i < use_len; i++)
      if (in_data[set_pos + i] != '0') break;

    if (i != use_len) {
```

```
      memcpy(tmp_buf, in_data, in_len);
      memset(tmp_buf + set_pos, '0', use_len);

      res = run_target(argv, tmp_buf, in_len, 0);

      if (res) {

        memset(in_data + set_pos, '0', use_len);
        changed_any = 1;
        alpha_del0 += use_len;

      }

    }

    set_pos += set_len;

  }

  alpha_d_total += alpha_del0;

  OKF("Block normalization complete, %u byte%s replaced.", alpha_del0,
      alpha_del0 == 1 ? "" : "s");

next_pass:

  ACTF(cYEL "--- " cBRI "Pass #%u " cYEL "---", ++cur_pass);
  changed_any = 0;

  /*******************
   * BLOCK DELETION *
   *******************/

  del_len = next_p2(in_len / TRIM_START_STEPS);
  stage_o_len = in_len;

  ACTF(cBRI "Stage #1: " cRST "Removing blocks of data...");

next_del_blksize:

  if (!del_len) del_len = 1;
  del_pos  = 0;
  prev_del = 1;

  SAYF(cGRA "    Block length = %u, remaining size = %u\n" cRST,
       del_len, in_len);

  while (del_pos < in_len) {

    u8  res;
    s32 tail_len;

    tail_len = in_len - del_pos - del_len;
    if (tail_len < 0) tail_len = 0;

    /* If we have processed at least one full block (initially, prev_del == 1),
       and we did so without deleting the previous one, and we aren't at the
       very end of the buffer (tail_len > 0), and the current block is the same
       as the previous one... skip this step as a no-op. */
```

```
  if (!prev_del && tail_len && !memcmp(in_data + del_pos - del_len,
      in_data + del_pos, del_len)) {

    del_pos += del_len;
    continue;

  }

  prev_del = 0;

  /* Head */
  memcpy(tmp_buf, in_data, del_pos);

  /* Tail */
  memcpy(tmp_buf + del_pos, in_data + del_pos + del_len, tail_len);

  res = run_target(argv, tmp_buf, del_pos + tail_len, 0);

  if (res) {

    memcpy(in_data, tmp_buf, del_pos + tail_len);
    prev_del = 1;
    in_len   = del_pos + tail_len;

    changed_any = 1;

  } else del_pos += del_len;

}

if (del_len > 1 && in_len >= 1) {

  del_len /= 2;
  goto next_del_blksize;

}

OKF("Block removal complete, %u bytes deleted.", stage_o_len - in_len);

if (!in_len && changed_any)
  WARNF(cLRD "Down to zero bytes - check the command line and mem limit!" cRST);

if (cur_pass > 1 && !changed_any) goto finalize_all;

/************************
 * ALPHABET MINIMIZATION *
 ************************/

alpha_size   = 0;
alpha_del1   = 0;
syms_removed = 0;

memset(alpha_map, 0, 256 * sizeof(u32));

for (i = 0; i < in_len; i++) {
  if (!alpha_map[in_data[i]]) alpha_size++;
  alpha_map[in_data[i]]++;
}

ACTF(cBRI "Stage #2: " cRST "Minimizing symbols (%u code point%s)...",
```

```
    alpha_size, alpha_size == 1 ? "" : "s");

for (i = 0; i < 256; i++) {

  u32 r;
  u8 res;

  if (i == '0' || !alpha_map[i]) continue;

  memcpy(tmp_buf, in_data, in_len);

  for (r = 0; r < in_len; r++)
    if (tmp_buf[r] == i) tmp_buf[r] = '0';

  res = run_target(argv, tmp_buf, in_len, 0);

  if (res) {

    memcpy(in_data, tmp_buf, in_len);
    syms_removed++;
    alpha_del1 += alpha_map[i];
    changed_any = 1;

  }

}

alpha_d_total += alpha_del1;

OKF("Symbol minimization finished, %u symbol%s (%u byte%s) replaced.",
    syms_removed, syms_removed == 1 ? "" : "s",
    alpha_del1, alpha_del1 == 1 ? "" : "s");

/*************************
 * CHARACTER MINIMIZATION *
 *************************/

alpha_del2 = 0;

ACTF(cBRI "Stage #3: " cRST "Character minimization...");

memcpy(tmp_buf, in_data, in_len);

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

  u8 res, orig = tmp_buf[i];

  if (orig == '0') continue;
  tmp_buf[i] = '0';

  res = run_target(argv, tmp_buf, in_len, 0);

  if (res) {

    in_data[i] = '0';
    alpha_del2++;
    changed_any = 1;

  } else tmp_buf[i] = orig;
```

```
  }

  alpha_d_total += alpha_del2;

  OKF("Character minimization done, %u byte%s replaced.",
      alpha_del2, alpha_del2 == 1 ? "" : "s");

  if (changed_any) goto next_pass;

finalize_all:

  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);

  if (total_execs > 50 && missed_hangs * 10 > total_execs)
    WARNF(cLRD "Frequent timeouts - results may be skewed." cRST);

}



/* Handle Ctrl-C and the like. */

static void handle_stop_sig(int sig) {

  stop_soon = 1;

  if (child_pid > 0) kill(child_pid, SIGKILL);

}


/* Do basic preparations - persistent fds, filenames, etc. */

static void set_up_environment(void) {

  u8* x;

  dev_null_fd = open("/dev/null", O_RDWR);
  if (dev_null_fd < 0) PFATAL("Unable to open /dev/null");

  if (!prog_in) {

    u8* use_dir = ".";

    if (access(use_dir, R_OK | W_OK | X_OK)) {

      use_dir = getenv("TMPDIR");
      if (!use_dir) use_dir = "/tmp";

    }

    prog_in = alloc_printf("%s/.afl-tmin-temp-%u", use_dir, getpid());
```

```
  }

  /* Set sane defaults... */

  x = getenv("ASAN_OPTIONS");

  if (x) {

    if (!strstr(x, "abort_on_error=1"))
      FATAL("Custom ASAN_OPTIONS set without abort_on_error=1 - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom ASAN_OPTIONS set without symbolize=0 - please fix!");

  }

  x = getenv("MSAN_OPTIONS");

  if (x) {

    if (!strstr(x, "exit_code=" STRINGIFY(MSAN_ERROR)))
      FATAL("Custom MSAN_OPTIONS set without exit_code="
            STRINGIFY(MSAN_ERROR) " - please fix!");

    if (!strstr(x, "symbolize=0"))
      FATAL("Custom MSAN_OPTIONS set without symbolize=0 - please fix!");

  }

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

  setenv("MSAN_OPTIONS", "exit_code=" STRINGIFY(MSAN_ERROR) ":"
                         "symbolize=0:"
                         "abort_on_error=1:"
                         "allocator_may_return_null=1:"
                         "msan_track_origins=0", 0);

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

}


/* Setup signal handlers, duh. */

static void setup_signal_handlers(void) {

  struct sigaction sa;

  sa.sa_handler   = NULL;
  sa.sa_flags     = SA_RESTART;
  sa.sa_sigaction = NULL;

  sigemptyset(&sa.sa_mask);
```

```
  /* Various ways of saying "stop". */

  sa.sa_handler = handle_stop_sig;
  sigaction(SIGHUP, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);

  /* Exec timeout notifications. */

  sa.sa_handler = handle_timeout;
  sigaction(SIGALRM, &sa, NULL);

}


/* Detect @@ in args. */

static void detect_file_args(char** argv) {

  u32 i = 0;
  u8* cwd = getcwd(NULL, 0);

  if (!cwd) PFATAL("getcwd() failed");

  while (argv[i]) {

    u8* aa_loc = strstr(argv[i], "@@");

    if (aa_loc) {

      u8 *aa_subst, *n_arg;

      /* Be sure that we're always using fully-qualified paths. */

      if (prog_in[0] == '/') aa_subst = prog_in;
      else aa_subst = alloc_printf("%s/%s", cwd, prog_in);

      /* Construct a replacement argv value. */

      *aa_loc = 0;
      n_arg = alloc_printf("%s%s%s", argv[i], aa_subst, aa_loc + 2);
      argv[i] = n_arg;
      *aa_loc = '@';

      if (prog_in[0] != '/') ck_free(aa_subst);

    }

    i++;

  }

  free(cwd); /* not tracked */

}


/* Display usage hints. */
```

```
static void usage(u8* argv0) {

  SAYF("\n%s [ options ] -- /path/to/target_app [ ... ]\n\n"

       "Required parameters:\n\n"

       "  -i file       - input test case to be shrunk by the tool\n"
       "  -o file       - final output location for the minimized data\n\n"

       "Execution control settings:\n\n"

       "  -f file       - input file read by the tested program (stdin)\n"
       "  -t msec       - timeout for each run (%u ms)\n"
       "  -m megs       - memory limit for child process (%u MB)\n"
       "  -Q            - use binary-only instrumentation (QEMU mode)\n\n"

       "Minimization settings:\n\n"

       "  -e            - solve for edge coverage only, ignore hit counts\n"
       "  -x            - treat non-zero exit codes as crashes\n\n"

       "Other stuff:\n\n"

       "  -V            - show version number and exit\n\n"

       "For additional tips, please consult %s/README.\n\n",

       argv0, EXEC_TIMEOUT, MEM_LIMIT, doc_path);

  exit(1);

}


/* Find binary. */

static void find_binary(u8* fname) {

  u8* env_path = 0;
  struct stat st;

  if (strchr(fname, '/') || !(env_path = getenv("PATH"))) {

    target_path = ck_strdup(fname);

    if (stat(target_path, &st) || !S_ISREG(st.st_mode) ||
        !(st.st_mode & 0111) || st.st_size < 4)
      FATAL("Program '%s' not found or not executable", fname);

  } else {

    while (env_path) {

      u8 *cur_elem, *delim = strchr(env_path, ':');

      if (delim) {

        cur_elem = ck_alloc(delim - env_path + 1);
        memcpy(cur_elem, env_path, delim - env_path);
        delim++;
```

```
      } else cur_elem = ck_strdup(env_path);

      env_path = delim;

      if (cur_elem[0])
        target_path = alloc_printf("%s/%s", cur_elem, fname);
      else
        target_path = ck_strdup(fname);

      ck_free(cur_elem);

      if (!stat(target_path, &st) && S_ISREG(st.st_mode) &&
          (st.st_mode & 0111) && st.st_size >= 4) break;

      ck_free(target_path);
      target_path = 0;

    }

    if (!target_path) FATAL("Program '%s' not found or not executable", fname);

  }

}


/* Fix up argv for QEMU. */

static char** get_qemu_argv(u8* own_loc, char** argv, int argc) {

  char** new_argv = ck_alloc(sizeof(char*) * (argc + 4));
  u8 *tmp, *cp, *rsl, *own_copy;

  /* Workaround for a QEMU stability glitch. */

  setenv("QEMU_LOG", "nochain", 1);

  memcpy(new_argv + 3, argv + 1, sizeof(char*) * argc);

  /* Now we need to actually find qemu for argv[0]. */

  new_argv[2] = target_path;
  new_argv[1] = "--";

  tmp = getenv("AFL_PATH");

  if (tmp) {

    cp = alloc_printf("%s/afl-qemu-trace", tmp);

    if (access(cp, X_OK))
      FATAL("Unable to find '%s'", tmp);

    target_path = new_argv[0] = cp;
    return new_argv;

  }

  own_copy = ck_strdup(own_loc);
```

```
  rsl = strrchr(own_copy, '/');

  if (rsl) {

    *rsl = 0;

    cp = alloc_printf("%s/afl-qemu-trace", own_copy);
    ck_free(own_copy);

    if (!access(cp, X_OK)) {

      target_path = new_argv[0] = cp;
      return new_argv;

    }

  } else ck_free(own_copy);

  if (!access(BIN_PATH "/afl-qemu-trace", X_OK)) {

    target_path = new_argv[0] = BIN_PATH "/afl-qemu-trace";
    return new_argv;

  }

  FATAL("Unable to find 'afl-qemu-trace'.");

}


/* Read mask bitmap from file. This is for the -B option. */

static void read_bitmap(u8* fname) {

  s32 fd = open(fname, O_RDONLY);

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

  ck_read(fd, mask_bitmap, MAP_SIZE, fname);

  close(fd);

}



/* Main entry point */

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

  s32 opt;
  u8  mem_limit_given = 0, timeout_given = 0, qemu_mode = 0;
  char** use_argv;

  doc_path = access(DOC_PATH, F_OK) ? "docs" : DOC_PATH;

  SAYF(cCYA "afl-tmin " cBRI VERSION cRST " by <lcamtuf@google.com>\n");

  while ((opt = getopt(argc,argv,"+i:o:f:m:t:B:xeQV")) > 0)
```

```
  switch (opt) {

    case 'i':

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

    case 'o':

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

    case 'f':

      if (prog_in) FATAL("Multiple -f options not supported");
      use_stdin = 0;
      prog_in   = optarg;
      break;

    case 'e':

      if (edges_only) FATAL("Multiple -e options not supported");
      edges_only = 1;
      break;

    case 'x':

      if (exit_crash) FATAL("Multiple -x options not supported");
      exit_crash = 1;
      break;

    case 'm': {

        u8 suffix = 'M';

        if (mem_limit_given) FATAL("Multiple -m options not supported");
        mem_limit_given = 1;

        if (!strcmp(optarg, "none")) {

          mem_limit = 0;
          break;

        }

        if (sscanf(optarg, "%llu%c", &mem_limit, &suffix) < 1 ||
            optarg[0] == '-') FATAL("Bad syntax used for -m");

        switch (suffix) {

          case 'T': mem_limit *= 1024 * 1024; break;
          case 'G': mem_limit *= 1024; break;
          case 'k': mem_limit /= 1024; break;
          case 'M': break;

          default:  FATAL("Unsupported suffix or bad syntax for -m");

        }
```

```
        if (mem_limit < 5) FATAL("Dangerously low value of -m");

        if (sizeof(rlim_t) == 4 && mem_limit > 2000)
          FATAL("Value of -m out of range on 32-bit systems");

      }

      break;

    case 't':

      if (timeout_given) FATAL("Multiple -t options not supported");
      timeout_given = 1;

      exec_tmout = atoi(optarg);

      if (exec_tmout < 10 || optarg[0] == '-')
        FATAL("Dangerously low value of -t");

      break;

    case 'Q':

      if (qemu_mode) FATAL("Multiple -Q options not supported");
      if (!mem_limit_given) mem_limit = MEM_LIMIT_QEMU;

      qemu_mode = 1;
      break;

    case 'B': /* load bitmap */

      /* This is a secret undocumented option! It is speculated to be useful
         if you have a baseline "boring" input file and another "interesting"
         file you want to minimize.

         You can dump a binary bitmap for the boring file using
         afl-showmap -b, and then load it into afl-tmin via -B. The minimizer
         will then minimize to preserve only the edges that are unique to
         the interesting input file, but ignoring everything from the
         original map.

         The option may be extended and made more official if it proves
         to be useful. */

      if (mask_bitmap) FATAL("Multiple -B options not supported");
      mask_bitmap = ck_alloc(MAP_SIZE);
      read_bitmap(optarg);
      break;

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

      /* Version number has been printed already, just quit. */
      exit(0);

    default:

      usage(argv[0]);

  }
```

```
  if (optind == argc || !in_file || !out_file) usage(argv[0]);

  setup_shm();
  setup_signal_handlers();

  set_up_environment();

  find_binary(argv[optind]);
  detect_file_args(argv + optind);

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

  exact_mode = !!getenv("AFL_TMIN_EXACT");

  SAYF("\n");

  read_initial_file();

  ACTF("Performing dry run (mem limit = %llu MB, timeout = %u ms%s)...",
       mem_limit, exec_tmout, edges_only ? ", edges only" : "");

  run_target(use_argv, in_data, in_len, 1);

  if (child_timed_out)
    FATAL("Target binary times out (adjusting -t may help).");

  if (!crash_mode) {

     OKF("Program terminates normally, minimizing in "
         cCYA "instrumented" cRST " mode.");

     if (!anything_set()) FATAL("No instrumentation detected.");

  } else {

     OKF("Program exits with a signal, minimizing in " cMGN "%scrash" cRST
         " mode.", exact_mode ? "EXACT " : "");

  }

  minimize(use_argv);

  ACTF("Writing output to '%s'...", out_file);

  unlink(prog_in);
  prog_in = NULL;

  close(write_to_file(out_file, in_data, in_len));

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

  exit(0);

}
```

afl-whatsup

```
#!/bin/sh
#
# american fuzzy lop - status check tool
# --------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
#
# Copyright 2015 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# This tool summarizes the status of any locally-running synchronized
# instances of afl-fuzz.
#

echo "status check tool for afl-fuzz by <lcamtuf@google.com>"
echo

if [ "$1" = "-s" ]; then

  SUMMARY_ONLY=1
  DIR="$2"

else

  unset SUMMARY_ONLY
  DIR="$1"

fi

if [ "$DIR" = "" ]; then

  echo "Usage: $0 [ -s ] afl_sync_dir" 1>&2
  echo 1>&2
  echo "The -s option causes the tool to skip all the per-fuzzer trivia and show" 1>&2
  echo "just the summary results. See docs/parallel_fuzzing.txt for additional tips." 1>&2
  echo 1>&2
  exit 1

fi

cd "$DIR" || exit 1

if [ -d queue ]; then

  echo "[-] Error: parameter is an individual output directory, not a sync dir." 1>&2
  exit 1

fi

CUR_TIME=`date +%s`
```

```
TMP=`mktemp -t .afl-whatsup-XXXXXXXX` || TMP=`mktemp -p /data/local/tmp .afl-whatsup-XXXXXXXX`
|| exit 1

ALIVE_CNT=0
DEAD_CNT=0

TOTAL_TIME=0
TOTAL_EXECS=0
TOTAL_EPS=0
TOTAL_CRASHES=0
TOTAL_PFAV=0
TOTAL_PENDING=0

if [ "$SUMMARY_ONLY" = "" ]; then

  echo "Individual fuzzers"
  echo "==================="
  echo

fi

for i in `find . -maxdepth 2 -iname fuzzer_stats | sort`; do

  sed 's/^command_line.*$/_skip:1/;s/[ ]*:[ ]*/="/;s/$/"/' "$i" >"$TMP"
  . "$TMP"

  RUN_UNIX=$((CUR_TIME - start_time))
  RUN_DAYS=$((RUN_UNIX / 60 / 60 / 24))
  RUN_HRS=$(((RUN_UNIX / 60 / 60) % 24))

  if [ "$SUMMARY_ONLY" = "" ]; then

    echo ">>> $afl_banner ($RUN_DAYS days, $RUN_HRS hrs) <<<"
    echo

  fi

  if ! kill -0 "$fuzzer_pid" 2>/dev/null; then

    if [ "$SUMMARY_ONLY" = "" ]; then

      echo "  Instance is dead or running remotely, skipping."
      echo

    fi

    DEAD_CNT=$((DEAD_CNT + 1))
    continue

  fi

  ALIVE_CNT=$((ALIVE_CNT + 1))

  EXEC_SEC=$((execs_done / RUN_UNIX))
  PATH_PERC=$((cur_path * 100 / paths_total))

  TOTAL_TIME=$((TOTAL_TIME + RUN_UNIX))
  TOTAL_EPS=$((TOTAL_EPS + EXEC_SEC))
  TOTAL_EXECS=$((TOTAL_EXECS + execs_done))
  TOTAL_CRASHES=$((TOTAL_CRASHES + unique_crashes))
```

```
  TOTAL_PENDING=$((TOTAL_PENDING + pending_total))
  TOTAL_PFAV=$((TOTAL_PFAV + pending_favs))

  if [ "$SUMMARY_ONLY" = "" ]; then

    echo "  cycle $((cycles_done + 1)), lifetime speed $EXEC_SEC execs/sec, path
$cur_path/$paths_total (${PATH_PERC}%)"

    if [ "$unique_crashes" = "0" ]; then
      echo "  pending $pending_favs/$pending_total, coverage $bitmap_cvg, no crashes yet"
    else
      echo "  pending $pending_favs/$pending_total, coverage $bitmap_cvg, crash count
$unique_crashes (!)"
    fi

    echo

  fi

done

rm -f "$TMP"

TOTAL_DAYS=$((TOTAL_TIME / 60 / 60 / 24))
TOTAL_HRS=$(((TOTAL_TIME / 60 / 60) % 24))

test "$TOTAL_TIME" = "0" && TOTAL_TIME=1

echo "Summary stats"
echo "============="
echo
echo "       Fuzzers alive : $ALIVE_CNT"

if [ ! "$DEAD_CNT" = "0" ]; then
  echo "      Dead or remote : $DEAD_CNT (excluded from stats)"
fi

echo "      Total run time : $TOTAL_DAYS days, $TOTAL_HRS hours"
echo "         Total execs : $((TOTAL_EXECS / 1000 / 1000)) million"
echo "    Cumulative speed : $TOTAL_EPS execs/sec"
echo "       Pending paths : $TOTAL_PFAV faves, $TOTAL_PENDING total"

if [ "$ALIVE_CNT" -gt "1" ]; then
  echo "  Pending per fuzzer : $((TOTAL_PFAV/ALIVE_CNT)) faves, $((TOTAL_PENDING/ALIVE_CNT))
total (on average)"
fi

echo "       Crashes found : $TOTAL_CRASHES locally unique"
echo

exit 0
```

alloc-inl.h

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
```

```
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - error-checking, memory-zeroing alloc routines
   ------------------------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This allocator is not designed to resist malicious attackers (the canaries
   are small and predictable), but provides a robust and portable way to detect
   use-after-free, off-by-one writes, stale pointers, and so on.
*/

#ifndef _HAVE_ALLOC_INL_H
#define _HAVE_ALLOC_INL_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "config.h"
#include "types.h"
#include "debug.h"

/* User-facing macro to sprintf() to a dynamically allocated buffer. */

#define alloc_printf(_str...) ({ \
    u8* _tmp; \
    s32 _len = snprintf(NULL, 0, _str); \
    if (_len < 0) FATAL("Whoa, snprintf() fails?!"); \
    _tmp = ck_alloc(_len + 1); \
    snprintf((char*)_tmp, _len + 1, _str); \
    _tmp; \
  })

/* Macro to enforce allocation limits as a last-resort defense against
   integer overflows. */

#define ALLOC_CHECK_SIZE(_s) do { \
    if ((_s) > MAX_ALLOC) \
      ABORT("Bad alloc request: %u bytes", (_s)); \
  } while (0)

/* Macro to check malloc() failures and the like. */

#define ALLOC_CHECK_RESULT(_r, _s) do { \
    if (!(_r)) \
      ABORT("Out of memory: can't allocate %u bytes", (_s)); \
  } while (0)
```

```
/* Magic tokens used to mark used / freed chunks. */

#define ALLOC_MAGIC_C1  0xFF00FF00 /* Used head (dword)  */
#define ALLOC_MAGIC_F   0xFE00FE00 /* Freed head (dword) */
#define ALLOC_MAGIC_C2  0xF0       /* Used tail (byte)   */

/* Positions of guard tokens in relation to the user-visible pointer. */

#define ALLOC_C1(_ptr)  (((u32*)(_ptr))[-2])
#define ALLOC_S(_ptr)   (((u32*)(_ptr))[-1])
#define ALLOC_C2(_ptr)  (((u8*)(_ptr))[ALLOC_S(_ptr)])

#define ALLOC_OFF_HEAD  8
#define ALLOC_OFF_TOTAL (ALLOC_OFF_HEAD + 1)

/* Allocator increments for ck_realloc_block(). */

#define ALLOC_BLK_INC    256

/* Sanity-checking macros for pointers. */

#define CHECK_PTR(_p) do { \
    if (_p) { \
      if (ALLOC_C1(_p) ^ ALLOC_MAGIC_C1) {\
        if (ALLOC_C1(_p) == ALLOC_MAGIC_F) \
          ABORT("Use after free."); \
        else ABORT("Corrupted head alloc canary."); \
      } \
      if (ALLOC_C2(_p) ^ ALLOC_MAGIC_C2) \
        ABORT("Corrupted tail alloc canary."); \
    } \
  } while (0)

#define CHECK_PTR_EXPR(_p) ({ \
    typeof (_p) _tmp = (_p); \
    CHECK_PTR(_tmp); \
    _tmp; \
  })


/* Allocate a buffer, explicitly not zeroing it. Returns NULL for zero-sized
   requests. */

static inline void* DFL_ck_alloc_nozero(u32 size) {

  void* ret;

  if (!size) return NULL;

  ALLOC_CHECK_SIZE(size);
  ret = malloc(size + ALLOC_OFF_TOTAL);
  ALLOC_CHECK_RESULT(ret, size);

  ret += ALLOC_OFF_HEAD;

  ALLOC_C1(ret) = ALLOC_MAGIC_C1;
  ALLOC_S(ret)  = size;
  ALLOC_C2(ret) = ALLOC_MAGIC_C2;

  return ret;
```

```
}


/* Allocate a buffer, returning zeroed memory. */

static inline void* DFL_ck_alloc(u32 size) {

  void* mem;

  if (!size) return NULL;
  mem = DFL_ck_alloc_nozero(size);

  return memset(mem, 0, size);

}


/* Free memory, checking for double free and corrupted heap. When DEBUG_BUILD
   is set, the old memory will be also clobbered with 0xFF. */

static inline void DFL_ck_free(void* mem) {

  if (!mem) return;

  CHECK_PTR(mem);

#ifdef DEBUG_BUILD

  /* Catch pointer issues sooner. */
  memset(mem, 0xFF, ALLOC_S(mem));

#endif /* DEBUG_BUILD */

  ALLOC_C1(mem) = ALLOC_MAGIC_F;

  free(mem - ALLOC_OFF_HEAD);

}


/* Re-allocate a buffer, checking for issues and zeroing any newly-added tail.
   With DEBUG_BUILD, the buffer is always reallocated to a new addresses and the
   old memory is clobbered with 0xFF. */

static inline void* DFL_ck_realloc(void* orig, u32 size) {

  void* ret;
  u32   old_size = 0;

  if (!size) {

    DFL_ck_free(orig);
    return NULL;

  }

  if (orig) {

    CHECK_PTR(orig);
```

```
#ifndef DEBUG_BUILD
    ALLOC_C1(orig) = ALLOC_MAGIC_F;
#endif /* !DEBUG_BUILD */

    old_size  = ALLOC_S(orig);
    orig     -= ALLOC_OFF_HEAD;

    ALLOC_CHECK_SIZE(old_size);

  }

  ALLOC_CHECK_SIZE(size);

#ifndef DEBUG_BUILD

  ret = realloc(orig, size + ALLOC_OFF_TOTAL);
  ALLOC_CHECK_RESULT(ret, size);

#else

  /* Catch pointer issues sooner: force relocation and make sure that the
     original buffer is wiped. */

  ret = malloc(size + ALLOC_OFF_TOTAL);
  ALLOC_CHECK_RESULT(ret, size);

  if (orig) {

    memcpy(ret + ALLOC_OFF_HEAD, orig + ALLOC_OFF_HEAD, MIN(size, old_size));
    memset(orig + ALLOC_OFF_HEAD, 0xFF, old_size);

    ALLOC_C1(orig + ALLOC_OFF_HEAD) = ALLOC_MAGIC_F;

    free(orig);

  }

#endif /* ^!DEBUG_BUILD */

  ret += ALLOC_OFF_HEAD;

  ALLOC_C1(ret) = ALLOC_MAGIC_C1;
  ALLOC_S(ret)  = size;
  ALLOC_C2(ret) = ALLOC_MAGIC_C2;

  if (size > old_size)
    memset(ret + old_size, 0, size - old_size);

  return ret;

}


/* Re-allocate a buffer with ALLOC_BLK_INC increments (used to speed up
   repeated small reallocs without complicating the user code). */

static inline void* DFL_ck_realloc_block(void* orig, u32 size) {

#ifndef DEBUG_BUILD
```

```
  if (orig) {

    CHECK_PTR(orig);

    if (ALLOC_S(orig) >= size) return orig;

    size += ALLOC_BLK_INC;

  }

#endif /* !DEBUG_BUILD */

  return DFL_ck_realloc(orig, size);

}


/* Create a buffer with a copy of a string. Returns NULL for NULL inputs. */

static inline u8* DFL_ck_strdup(u8* str) {

  void* ret;
  u32   size;

  if (!str) return NULL;

  size = strlen((char*)str) + 1;

  ALLOC_CHECK_SIZE(size);
  ret = malloc(size + ALLOC_OFF_TOTAL);
  ALLOC_CHECK_RESULT(ret, size);

  ret += ALLOC_OFF_HEAD;

  ALLOC_C1(ret) = ALLOC_MAGIC_C1;
  ALLOC_S(ret)  = size;
  ALLOC_C2(ret) = ALLOC_MAGIC_C2;

  return memcpy(ret, str, size);

}


/* Create a buffer with a copy of a memory block. Returns NULL for zero-sized
   or NULL inputs. */

static inline void* DFL_ck_memdup(void* mem, u32 size) {

  void* ret;

  if (!mem || !size) return NULL;

  ALLOC_CHECK_SIZE(size);
  ret = malloc(size + ALLOC_OFF_TOTAL);
  ALLOC_CHECK_RESULT(ret, size);

  ret += ALLOC_OFF_HEAD;

  ALLOC_C1(ret) = ALLOC_MAGIC_C1;
```

```
  ALLOC_S(ret)  = size;
  ALLOC_C2(ret) = ALLOC_MAGIC_C2;

  return memcpy(ret, mem, size);

}


/* Create a buffer with a block of text, appending a NUL terminator at the end.
   Returns NULL for zero-sized or NULL inputs. */

static inline u8* DFL_ck_memdup_str(u8* mem, u32 size) {

  u8* ret;

  if (!mem || !size) return NULL;

  ALLOC_CHECK_SIZE(size);
  ret = malloc(size + ALLOC_OFF_TOTAL + 1);
  ALLOC_CHECK_RESULT(ret, size);

  ret += ALLOC_OFF_HEAD;

  ALLOC_C1(ret) = ALLOC_MAGIC_C1;
  ALLOC_S(ret)  = size;
  ALLOC_C2(ret) = ALLOC_MAGIC_C2;

  memcpy(ret, mem, size);
  ret[size] = 0;

  return ret;

}


#ifndef DEBUG_BUILD

/* In non-debug mode, we just do straightforward aliasing of the above functions
   to user-visible names such as ck_alloc(). */

#define ck_alloc          DFL_ck_alloc
#define ck_alloc_nozero   DFL_ck_alloc_nozero
#define ck_realloc        DFL_ck_realloc
#define ck_realloc_block  DFL_ck_realloc_block
#define ck_strdup         DFL_ck_strdup
#define ck_memdup         DFL_ck_memdup
#define ck_memdup_str     DFL_ck_memdup_str
#define ck_free           DFL_ck_free

#define alloc_report()

#else

/* In debugging mode, we also track allocations to detect memory leaks, and the
   flow goes through one more layer of indirection. */

/* Alloc tracking data structures: */

#define ALLOC_BUCKETS     4096
```

```
struct TRK_obj {
  void *ptr;
  char *file, *func;
  u32  line;
};

#ifdef AFL_MAIN

struct TRK_obj* TRK[ALLOC_BUCKETS];
u32 TRK_cnt[ALLOC_BUCKETS];

#  define alloc_report() TRK_report()

#else

extern struct TRK_obj* TRK[ALLOC_BUCKETS];
extern u32 TRK_cnt[ALLOC_BUCKETS];

#  define alloc_report()

#endif /* ^AFL_MAIN */

/* Bucket-assigning function for a given pointer: */

#define TRKH(_ptr) (((((u32)(_ptr)) >> 16) ^ ((u32)(_ptr))) % ALLOC_BUCKETS)


/* Add a new entry to the list of allocated objects. */

static inline void TRK_alloc_buf(void* ptr, const char* file, const char* func,
                                 u32 line) {

  u32 i, bucket;

  if (!ptr) return;

  bucket = TRKH(ptr);

  /* Find a free slot in the list of entries for that bucket. */

  for (i = 0; i < TRK_cnt[bucket]; i++)

    if (!TRK[bucket][i].ptr) {

      TRK[bucket][i].ptr  = ptr;
      TRK[bucket][i].file = (char*)file;
      TRK[bucket][i].func = (char*)func;
      TRK[bucket][i].line = line;
      return;

    }

  /* No space available - allocate more. */

  TRK[bucket] = DFL_ck_realloc_block(TRK[bucket],
    (TRK_cnt[bucket] + 1) * sizeof(struct TRK_obj));

  TRK[bucket][i].ptr  = ptr;
  TRK[bucket][i].file = (char*)file;
  TRK[bucket][i].func = (char*)func;
```

```
  TRK[bucket][i].line = line;

  TRK_cnt[bucket]++;

}


/* Remove entry from the list of allocated objects. */

static inline void TRK_free_buf(void* ptr, const char* file, const char* func,
                                u32 line) {

  u32 i, bucket;

  if (!ptr) return;

  bucket = TRKH(ptr);

  /* Find the element on the list... */

  for (i = 0; i < TRK_cnt[bucket]; i++)

    if (TRK[bucket][i].ptr == ptr) {

      TRK[bucket][i].ptr = 0;
      return;

    }

  WARNF("ALLOC: Attempt to free non-allocated memory in %s (%s:%u)",
        func, file, line);

}


/* Do a final report on all non-deallocated objects. */

static inline void TRK_report(void) {

  u32 i, bucket;

  fflush(0);

  for (bucket = 0; bucket < ALLOC_BUCKETS; bucket++)
    for (i = 0; i < TRK_cnt[bucket]; i++)
      if (TRK[bucket][i].ptr)
        WARNF("ALLOC: Memory never freed, created in %s (%s:%u)",
              TRK[bucket][i].func, TRK[bucket][i].file, TRK[bucket][i].line);

}


/* Simple wrappers for non-debugging functions: */

static inline void* TRK_ck_alloc(u32 size, const char* file, const char* func,
                                 u32 line) {

  void* ret = DFL_ck_alloc(size);
  TRK_alloc_buf(ret, file, func, line);
  return ret;
```

```
}


static inline void* TRK_ck_realloc(void* orig, u32 size, const char* file,
                                   const char* func, u32 line) {

  void* ret = DFL_ck_realloc(orig, size);
  TRK_free_buf(orig, file, func, line);
  TRK_alloc_buf(ret, file, func, line);
  return ret;

}


static inline void* TRK_ck_realloc_block(void* orig, u32 size, const char* file,
                                         const char* func, u32 line) {

  void* ret = DFL_ck_realloc_block(orig, size);
  TRK_free_buf(orig, file, func, line);
  TRK_alloc_buf(ret, file, func, line);
  return ret;

}


static inline void* TRK_ck_strdup(u8* str, const char* file, const char* func,
                                  u32 line) {

  void* ret = DFL_ck_strdup(str);
  TRK_alloc_buf(ret, file, func, line);
  return ret;

}


static inline void* TRK_ck_memdup(void* mem, u32 size, const char* file,
                                  const char* func, u32 line) {

  void* ret = DFL_ck_memdup(mem, size);
  TRK_alloc_buf(ret, file, func, line);
  return ret;

}


static inline void* TRK_ck_memdup_str(void* mem, u32 size, const char* file,
                                      const char* func, u32 line) {

  void* ret = DFL_ck_memdup_str(mem, size);
  TRK_alloc_buf(ret, file, func, line);
  return ret;

}


static inline void TRK_ck_free(void* ptr, const char* file,
                               const char* func, u32 line) {

  TRK_free_buf(ptr, file, func, line);
```

```
  DFL_ck_free(ptr);

}

/* Aliasing user-facing names to tracking functions: */

#define ck_alloc(_p1) \
  TRK_ck_alloc(_p1, __FILE__, __FUNCTION__, __LINE__)

#define ck_alloc_nozero(_p1) \
  TRK_ck_alloc(_p1, __FILE__, __FUNCTION__, __LINE__)

#define ck_realloc(_p1, _p2) \
  TRK_ck_realloc(_p1, _p2, __FILE__, __FUNCTION__, __LINE__)

#define ck_realloc_block(_p1, _p2) \
  TRK_ck_realloc_block(_p1, _p2, __FILE__, __FUNCTION__, __LINE__)

#define ck_strdup(_p1) \
  TRK_ck_strdup(_p1, __FILE__, __FUNCTION__, __LINE__)

#define ck_memdup(_p1, _p2) \
  TRK_ck_memdup(_p1, _p2, __FILE__, __FUNCTION__, __LINE__)

#define ck_memdup_str(_p1, _p2) \
  TRK_ck_memdup_str(_p1, _p2, __FILE__, __FUNCTION__, __LINE__)

#define ck_free(_p1) \
  TRK_ck_free(_p1, __FILE__, __FUNCTION__, __LINE__)

#endif /* ^!DEBUG_BUILD */

#endif /* ! _HAVE_ALLOC_INL_H */
```

android-ashmem.h

```
#ifdef __ANDROID__
#ifndef _ANDROID_ASHMEM_H
#define _ANDROID_ASHMEM_H

#include <fcntl.h>
#include <linux/ashmem.h>
#include <linux/shm.h>
#include <sys/ioctl.h>
#include <sys/mman.h>

#if __ANDROID_API__ >= 26
#define shmat bionic_shmat
#define shmctl bionic_shmctl
#define shmdt bionic_shmdt
#define shmget bionic_shmget
#endif
#include <sys/shm.h>
#undef shmat
#undef shmctl
#undef shmdt
#undef shmget
#include <stdio.h>
```

```
#define ASHMEM_DEVICE "/dev/ashmem"

static inline int shmctl(int __shmid, int __cmd, struct shmid_ds *__buf) {
  int ret = 0;
  if (__cmd == IPC_RMID) {
    int length = ioctl(__shmid, ASHMEM_GET_SIZE, NULL);
    struct ashmem_pin pin = {0, length};
    ret = ioctl(__shmid, ASHMEM_UNPIN, &pin);
    close(__shmid);
  }

  return ret;
}

static inline int shmget(key_t __key, size_t __size, int __shmflg) {
  (void) __shmflg;
  int fd, ret;
  char ourkey[11];

  fd = open(ASHMEM_DEVICE, O_RDWR);
  if (fd < 0)
    return fd;

  sprintf(ourkey, "%d", __key);
  ret = ioctl(fd, ASHMEM_SET_NAME, ourkey);
  if (ret < 0)
    goto error;

  ret = ioctl(fd, ASHMEM_SET_SIZE, __size);
  if (ret < 0)
    goto error;

  return fd;

error:
  close(fd);
  return ret;
}

static inline void *shmat(int __shmid, const void *__shmaddr, int __shmflg) {
  (void) __shmflg;
  int size;
  void *ptr;

  size = ioctl(__shmid, ASHMEM_GET_SIZE, NULL);
  if (size < 0) {
    return NULL;
  }

  ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, __shmid, 0);
  if (ptr == MAP_FAILED) {
    return NULL;
  }

  return ptr;
}

#endif /* !_ANDROID_ASHMEM_H */
#endif /* !__ANDROID__ */
```

Android.bp

```
cc_defaults {
  name: "afl-defaults",

  cflags: [
    "-funroll-loops",
    "-Wno-pointer-sign",
    "-Wno-pointer-arith",
    "-Wno-sign-compare",
    "-Wno-unused-parameter",
    "-Wno-unused-function",
    "-Wno-format",
    "-Wno-user-defined-warnings",
    "-DUSE_TRACE_PC=1",
    "-DBIN_PATH=\"out/host/linux-x86/bin\"",
    "-DDOC_PATH=\"out/host/linux-x86/shared/doc/afl\"",
    "-D__USE_GNU",
  ],
}

cc_binary {
  name: "afl-fuzz",
  static_executable: true,
  host_supported: true,

  defaults: [
    "afl-defaults",
  ],

  srcs: [
    "afl-fuzz.c",
  ],
}

cc_binary {
  name: "afl-showmap",
  static_executable: true,
  host_supported: true,

  defaults: [
    "afl-defaults",
  ],

  srcs: [
    "afl-showmap.c",
  ],
}

cc_binary {
  name: "afl-tmin",
  static_executable: true,
  host_supported: true,

  defaults: [
    "afl-defaults",
  ],
```

```
  srcs: [
    "afl-tmin.c",
  ],
}

cc_binary {
  name: "afl-analyze",
  static_executable: true,
  host_supported: true,

  defaults: [
    "afl-defaults",
  ],

  srcs: [
    "afl-analyze.c",
  ],
}

cc_binary {
  name: "afl-gotcpu",
  static_executable: true,
  host_supported: true,

  defaults: [
    "afl-defaults",
  ],

  srcs: [
    "afl-gotcpu.c",
  ],
}

cc_binary_host {
  name: "afl-clang-fast",
  static_executable: true,

  defaults: [
    "afl-defaults",
  ],

  cflags: [
    "-D__ANDROID__",
    "-DAFL_PATH=\"out/host/linux-x86/lib64\"",
  ],

  srcs: [
    "llvm_mode/afl-clang-fast.c",
  ],
}

cc_binary_host {
  name: "afl-clang-fast++",
  static_executable: true,

  defaults: [
    "afl-defaults",
  ],
```

```
  cflags: [
    "-D__ANDROID__",
    "-DAFL_PATH=\"out/host/linux-x86/lib64\"",
  ],

  srcs: [
    "llvm_mode/afl-clang-fast.c",
  ],
}

cc_library_static {
  name: "afl-llvm-rt",
  compile_multilib: "both",
  vendor_available: true,
  host_supported: true,
  recovery_available: true,

  defaults: [
    "afl-defaults",
  ],

  srcs: [
    "llvm_mode/afl-llvm-rt.o.c",
  ],
}
```

config.h

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - vaguely configurable bits
   ----------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>
*/

#ifndef _HAVE_CONFIG_H
#define _HAVE_CONFIG_H

#include "types.h"

/* Version string: */
```

```
#define VERSION             "2.57b"

/*******************************************************
 *                                                     *
 *  Settings that may be of interest to power users:   *
 *                                                     *
 *******************************************************/

/* Comment out to disable terminal colors (note that this makes afl-analyze
   a lot less nice): */

#define USE_COLOR

/* Comment out to disable fancy ANSI boxes and use poor man's 7-bit UI: */

#define FANCY_BOXES

/* Default timeout for fuzzed code (milliseconds). This is the upper bound,
   also used for detecting hangs; the actual value is auto-scaled: */

#define EXEC_TIMEOUT        1000

/* Timeout rounding factor when auto-scaling (milliseconds): */

#define EXEC_TM_ROUND       20

/* 64bit arch MACRO */
#if (defined (__x86_64__) || defined (__arm64__) || defined (__aarch64__))
#define WORD_SIZE_64 1
#endif

/* Default memory limit for child process (MB): */

#ifndef WORD_SIZE_64
#  define MEM_LIMIT         25
#else
#  define MEM_LIMIT         50
#endif /* ^!WORD_SIZE_64 */

/* Default memory limit when running in QEMU mode (MB): */

#define MEM_LIMIT_QEMU      200

/* Number of calibration cycles per every new test case (and for test
   cases that show variable behavior): */

#define CAL_CYCLES          8
#define CAL_CYCLES_LONG     40

/* Number of subsequent timeouts before abandoning an input file: */

#define TMOUT_LIMIT         250

/* Maximum number of unique hangs or crashes to record: */

#define KEEP_UNIQUE_HANG    500
#define KEEP_UNIQUE_CRASH   5000

/* Baseline number of random tweaks during a single 'havoc' stage: */
```

```
#define HAVOC_CYCLES        256
#define HAVOC_CYCLES_INIT   1024

/* Maximum multiplier for the above (should be a power of two, beware
   of 32-bit int overflows): */

#define HAVOC_MAX_MULT      16

/* Absolute minimum number of havoc cycles (after all adjustments): */

#define HAVOC_MIN           16

/* Maximum stacking for havoc-stage tweaks. The actual value is calculated
   like this:

   n = random between 1 and HAVOC_STACK_POW2
   stacking = 2^n

   In other words, the default (n = 7) produces 2, 4, 8, 16, 32, 64, or
   128 stacked tweaks: */

#define HAVOC_STACK_POW2    7

/* Caps on block sizes for cloning and deletion operations. Each of these
   ranges has a 33% probability of getting picked, except for the first
   two cycles where smaller blocks are favored: */

#define HAVOC_BLK_SMALL     32
#define HAVOC_BLK_MEDIUM    128
#define HAVOC_BLK_LARGE     1500

/* Extra-large blocks, selected very rarely (<5% of the time): */

#define HAVOC_BLK_XL        32768

/* Probabilities of skipping non-favored entries in the queue, expressed as
   percentages: */

#define SKIP_TO_NEW_PROB    99 /* ...when there are new, pending favorites */
#define SKIP_NFAV_OLD_PROB  95 /* ...no new favs, cur entry already fuzzed */
#define SKIP_NFAV_NEW_PROB  75 /* ...no new favs, cur entry not fuzzed yet */

/* Splicing cycle count: */

#define SPLICE_CYCLES       15

/* Nominal per-splice havoc cycle length: */

#define SPLICE_HAVOC        32

/* Maximum offset for integer addition / subtraction stages: */

#define ARITH_MAX           35

/* Limits for the test case trimmer. The absolute minimum chunk size; and
   the starting and ending divisors for chopping up the input file: */

#define TRIM_MIN_BYTES      4
#define TRIM_START_STEPS    16
#define TRIM_END_STEPS      1024
```

```
/* Maximum size of input file, in bytes (keep under 100MB): */

#define MAX_FILE            (1 * 1024 * 1024)

/* The same, for the test case minimizer: */

#define TMIN_MAX_FILE       (10 * 1024 * 1024)

/* Block normalization steps for afl-tmin: */

#define TMIN_SET_MIN_SIZE   4
#define TMIN_SET_STEPS      128

/* Maximum dictionary token size (-x), in bytes: */

#define MAX_DICT_FILE       128

/* Length limits for auto-detected dictionary tokens: */

#define MIN_AUTO_EXTRA      3
#define MAX_AUTO_EXTRA      32

/* Maximum number of user-specified dictionary tokens to use in deterministic
   steps; past this point, the "extras/user" step will be still carried out,
   but with proportionally lower odds: */

#define MAX_DET_EXTRAS      200

/* Maximum number of auto-extracted dictionary tokens to actually use in fuzzing
   (first value), and to keep in memory as candidates. The latter should be much
   higher than the former. */

#define USE_AUTO_EXTRAS     50
#define MAX_AUTO_EXTRAS     (USE_AUTO_EXTRAS * 10)

/* Scaling factor for the effector map used to skip some of the more
   expensive deterministic steps. The actual divisor is set to
   2^EFF_MAP_SCALE2 bytes: */

#define EFF_MAP_SCALE2      3

/* Minimum input file length at which the effector logic kicks in: */

#define EFF_MIN_LEN         128

/* Maximum effector density past which everything is just fuzzed
   unconditionally (%): */

#define EFF_MAX_PERC        90

/* UI refresh frequency (Hz): */

#define UI_TARGET_HZ        5

/* Fuzzer stats file and plot update intervals (sec): */

#define STATS_UPDATE_SEC    60
#define PLOT_UPDATE_SEC     5
```

```
/* Smoothing divisor for CPU load and exec speed stats (1 - no smoothing). */

#define AVG_SMOOTHING       16

/* Sync interval (every n havoc cycles): */

#define SYNC_INTERVAL       5

/* Output directory reuse grace period (minutes): */

#define OUTPUT_GRACE        25

/* Uncomment to use simple file names (id_NNNNNN): */

// #define SIMPLE_FILES

/* List of interesting values to use in fuzzing. */

#define INTERESTING_8 \
  -128,          /* Overflow signed 8-bit when decremented  */ \
  -1,            /*                                         */ \
   0,            /*                                         */ \
   1,            /*                                         */ \
   16,           /* One-off with common buffer size         */ \
   32,           /* One-off with common buffer size         */ \
   64,           /* One-off with common buffer size         */ \
   100,          /* One-off with common buffer size         */ \
   127           /* Overflow signed 8-bit when incremented  */

#define INTERESTING_16 \
  -32768,        /* Overflow signed 16-bit when decremented */ \
  -129,          /* Overflow signed 8-bit                   */ \
   128,          /* Overflow signed 8-bit                   */ \
   255,          /* Overflow unsig 8-bit when incremented   */ \
   256,          /* Overflow unsig 8-bit                    */ \
   512,          /* One-off with common buffer size         */ \
   1000,         /* One-off with common buffer size         */ \
   1024,         /* One-off with common buffer size         */ \
   4096,         /* One-off with common buffer size         */ \
   32767         /* Overflow signed 16-bit when incremented */

#define INTERESTING_32 \
  -2147483648LL, /* Overflow signed 32-bit when decremented */ \
  -100663046,    /* Large negative number (endian-agnostic) */ \
  -32769,        /* Overflow signed 16-bit                  */ \
   32768,        /* Overflow signed 16-bit                  */ \
   65535,        /* Overflow unsig 16-bit when incremented  */ \
   65536,        /* Overflow unsig 16 bit                   */ \
   100663045,    /* Large positive number (endian-agnostic) */ \
   2147483647    /* Overflow signed 32-bit when incremented */

/*************************************************************
 *                                                           *
 *  Really exotic stuff you probably don't want to touch:    *
 *                                                           *
 *************************************************************/

/* Call count interval between reseeding the libc PRNG from /dev/urandom: */

#define RESEED_RNG          10000
```

```
/* Maximum line length passed from GCC to 'as' and used for parsing
   configuration files: */

#define MAX_LINE            8192

/* Environment variable used to pass SHM ID to the called program. */

#define SHM_ENV_VAR         "__AFL_SHM_ID"

/* Other less interesting, internal-only variables. */

#define CLANG_ENV_VAR       "__AFL_CLANG_MODE"
#define AS_LOOP_ENV_VAR     "__AFL_AS_LOOPCHECK"
#define PERSIST_ENV_VAR     "__AFL_PERSISTENT"
#define DEFER_ENV_VAR       "__AFL_DEFER_FORKSRV"

/* In-code signatures for deferred and persistent mode. */

#define PERSIST_SIG         "##SIG_AFL_PERSISTENT##"
#define DEFER_SIG           "##SIG_AFL_DEFER_FORKSRV##"

/* Distinctive bitmap signature used to indicate failed execution: */

#define EXEC_FAIL_SIG       0xfee1dead

/* Distinctive exit code used to indicate MSAN trip condition: */

#define MSAN_ERROR          86

/* Designated file descriptors for forkserver commands (the application will
   use FORKSRV_FD and FORKSRV_FD + 1): */

#define FORKSRV_FD          198

/* Fork server init timeout multiplier: we'll wait the user-selected
   timeout plus this much for the fork server to spin up. */

#define FORK_WAIT_MULT      10

/* Calibration timeout adjustments, to be a bit more generous when resuming
   fuzzing sessions or trying to calibrate already-added internal finds.
   The first value is a percentage, the other is in milliseconds: */

#define CAL_TMOUT_PERC      125
#define CAL_TMOUT_ADD       50

/* Number of chances to calibrate a case before giving up: */

#define CAL_CHANCES         3

/* Map size for the traced binary (2^MAP_SIZE_POW2). Must be greater than
   2; you probably want to keep it under 18 or so for performance reasons
   (adjusting AFL_INST_RATIO when compiling is probably a better way to solve
   problems with complex programs). You need to recompile the target binary
   after changing this - otherwise, SEGVs may ensue. */

#define MAP_SIZE_POW2       16
#define MAP_SIZE            (1 << MAP_SIZE_POW2)
```

```
/* Maximum allocator request size (keep well under INT_MAX): */

#define MAX_ALLOC           0x40000000

/* A made-up hashing seed: */

#define HASH_CONST          0xa5b35705

/* Constants for afl-gotcpu to control busy loop timing: */

#define  CTEST_TARGET_MS    5000
#define  CTEST_CORE_TRG_MS  1000
#define  CTEST_BUSY_CYCLES  (10 * 1000 * 1000)

/* Uncomment this to use inferior block-coverage-based instrumentation. Note
   that you need to recompile the target binary for this to have any effect: */

// #define COVERAGE_ONLY

/* Uncomment this to ignore hit counts and output just one bit per tuple.
   As with the previous setting, you will need to recompile the target
   binary: */

// #define SKIP_COUNTS

/* Uncomment this to use instrumentation data to record newly discovered paths,
   but do not use them as seeds for fuzzing. This is useful for conveniently
   measuring coverage that could be attained by a "dumb" fuzzing algorithm: */

// #define IGNORE_FINDS

#endif /* ! _HAVE_CONFIG_H */
```

debug.h

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - debug / error handling macros
   --------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>
*/
```

```
#ifndef _HAVE_DEBUG_H
#define _HAVE_DEBUG_H

#include <errno.h>

#include "types.h"
#include "config.h"

/*******************
 * Terminal colors *
 *******************/

#ifdef USE_COLOR

#  define cBLK "\x1b[0;30m"
#  define cRED "\x1b[0;31m"
#  define cGRN "\x1b[0;32m"
#  define cBRN "\x1b[0;33m"
#  define cBLU "\x1b[0;34m"
#  define cMGN "\x1b[0;35m"
#  define cCYA "\x1b[0;36m"
#  define cLGR "\x1b[0;37m"
#  define cGRA "\x1b[1;90m"
#  define cLRD "\x1b[1;91m"
#  define cLGN "\x1b[1;92m"
#  define cYEL "\x1b[1;93m"
#  define cLBL "\x1b[1;94m"
#  define cPIN "\x1b[1;95m"
#  define cLCY "\x1b[1;96m"
#  define cBRI "\x1b[1;97m"
#  define cRST "\x1b[0m"

#  define bgBLK "\x1b[40m"
#  define bgRED "\x1b[41m"
#  define bgGRN "\x1b[42m"
#  define bgBRN "\x1b[43m"
#  define bgBLU "\x1b[44m"
#  define bgMGN "\x1b[45m"
#  define bgCYA "\x1b[46m"
#  define bgLGR "\x1b[47m"
#  define bgGRA "\x1b[100m"
#  define bgLRD "\x1b[101m"
#  define bgLGN "\x1b[102m"
#  define bgYEL "\x1b[103m"
#  define bgLBL "\x1b[104m"
#  define bgPIN "\x1b[105m"
#  define bgLCY "\x1b[106m"
#  define bgBRI "\x1b[107m"

#else

#  define cBLK ""
#  define cRED ""
#  define cGRN ""
#  define cBRN ""
#  define cBLU ""
#  define cMGN ""
#  define cCYA ""
#  define cLGR ""
#  define cGRA ""
```

```
#  define cLRD ""
#  define cLGN ""
#  define cYEL ""
#  define cLBL ""
#  define cPIN ""
#  define cLCY ""
#  define cBRI ""
#  define cRST ""

#  define bgBLK ""
#  define bgRED ""
#  define bgGRN ""
#  define bgBRN ""
#  define bgBLU ""
#  define bgMGN ""
#  define bgCYA ""
#  define bgLGR ""
#  define bgGRA ""
#  define bgLRD ""
#  define bgLGN ""
#  define bgYEL ""
#  define bgLBL ""
#  define bgPIN ""
#  define bgLCY ""
#  define bgBRI ""

#endif /* ^USE_COLOR */

/*************************
 * Box drawing sequences *
 *************************/

#ifdef FANCY_BOXES

#  define SET_G1   "\x1b)0"       /* Set G1 for box drawing    */
#  define RESET_G1 "\x1b)B"       /* Reset G1 to ASCII         */
#  define bSTART   "\x0e"         /* Enter G1 drawing mode     */
#  define bSTOP    "\x0f"         /* Leave G1 drawing mode     */
#  define bH       "q"            /* Horizontal line           */
#  define bV       "x"            /* Vertical line             */
#  define bLT      "l"            /* Left top corner           */
#  define bRT      "k"            /* Right top corner          */
#  define bLB      "m"            /* Left bottom corner        */
#  define bRB      "j"            /* Right bottom corner       */
#  define bX       "n"            /* Cross                     */
#  define bVR      "t"            /* Vertical, branch right    */
#  define bVL      "u"            /* Vertical, branch left     */
#  define bHT      "v"            /* Horizontal, branch top    */
#  define bHB      "w"            /* Horizontal, branch bottom */

#else

#  define SET_G1   ""
#  define RESET_G1 ""
#  define bSTART   ""
#  define bSTOP    ""
#  define bH       "-"
#  define bV       "|"
#  define bLT      "+"
#  define bRT      "+"
```

```
#  define bLB      "+"
#  define bRB      "+"
#  define bX       "+"
#  define bVR      "+"
#  define bVL      "+"
#  define bHT      "+"
#  define bHB      "+"

#endif /* ^FANCY_BOXES */

/***********************
 * Misc terminal codes *
 ***********************/

#define TERM_HOME     "\x1b[H"
#define TERM_CLEAR    TERM_HOME "\x1b[2J"
#define cEOL          "\x1b[0K"
#define CURSOR_HIDE   "\x1b[?25l"
#define CURSOR_SHOW   "\x1b[?25h"

/************************
 * Debug & error macros *
 ************************/

/* Just print stuff to the appropriate stream. */

#ifdef MESSAGES_TO_STDOUT
#  define SAYF(x...)    printf(x)
#else
#  define SAYF(x...)    fprintf(stderr, x)
#endif /* ^MESSAGES_TO_STDOUT */

/* Show a prefixed warning. */

#define WARNF(x...) do { \
    SAYF(cYEL "[!] " cBRI "WARNING: " cRST x); \
    SAYF(cRST "\n"); \
  } while (0)

/* Show a prefixed "doing something" message. */

#define ACTF(x...) do { \
    SAYF(cLBL "[*] " cRST x); \
    SAYF(cRST "\n"); \
  } while (0)

/* Show a prefixed "success" message. */

#define OKF(x...) do { \
    SAYF(cLGN "[+] " cRST x); \
    SAYF(cRST "\n"); \
  } while (0)

/* Show a prefixed fatal error message (not used in afl). */

#define BADF(x...) do { \
    SAYF(cLRD "\n[-] " cRST x); \
    SAYF(cRST "\n"); \
  } while (0)
```

```
/* Die with a verbose non-OS fatal error message. */

#define FATAL(x...) do { \
    SAYF(bSTOP RESET_G1 CURSOR_SHOW cRST cLRD "\n[-] PROGRAM ABORT : " \
         cBRI x); \
    SAYF(cLRD "\n         Location : " cRST "%s(), %s:%u\n\n", \
         __FUNCTION__, __FILE__, __LINE__); \
    exit(1); \
  } while (0)

/* Die by calling abort() to provide a core dump. */

#define ABORT(x...) do { \
    SAYF(bSTOP RESET_G1 CURSOR_SHOW cRST cLRD "\n[-] PROGRAM ABORT : " \
         cBRI x); \
    SAYF(cLRD "\n    Stop location : " cRST "%s(), %s:%u\n\n", \
         __FUNCTION__, __FILE__, __LINE__); \
    abort(); \
  } while (0)

/* Die while also including the output of perror(). */

#define PFATAL(x...) do { \
    fflush(stdout); \
    SAYF(bSTOP RESET_G1 CURSOR_SHOW cRST cLRD "\n[-]  SYSTEM ERROR : " \
         cBRI x); \
    SAYF(cLRD "\n    Stop location : " cRST "%s(), %s:%u\n", \
         __FUNCTION__, __FILE__, __LINE__); \
    SAYF(cLRD "       OS message : " cRST "%s\n", strerror(errno)); \
    exit(1); \
  } while (0)

/* Die with FAULT() or PFAULT() depending on the value of res (used to
   interpret different failure modes for read(), write(), etc). */

#define RPFATAL(res, x...) do { \
    if (res < 0) PFATAL(x); else FATAL(x); \
  } while (0)

/* Error-checking versions of read() and write() that call RPFATAL() as
   appropriate. */

#define ck_write(fd, buf, len, fn) do { \
    u32 _len = (len); \
    s32 _res = write(fd, buf, _len); \
    if (_res != _len) RPFATAL(_res, "Short write to %s", fn); \
  } while (0)

#define ck_read(fd, buf, len, fn) do { \
    u32 _len = (len); \
    s32 _res = read(fd, buf, _len); \
    if (_res != _len) RPFATAL(_res, "Short read from %s", fn); \
  } while (0)

#endif /* ! _HAVE_DEBUG_H */
```

> dictionaries

gif.dict

```
#
# AFL dictionary for GIF images
# -----------------------------
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

header_87a="87a"
header_89a="89a"
header_gif="GIF"

marker_2c=","
marker_3b=";"

section_2101="!\x01\x12"
section_21f9="!\xf9\x04"
section_21fe="!\xfe"
section_21ff="!\xff\x11"
```

html_tags.dict

```
#
# AFL dictionary for HTML parsers (tags only)
# -------------------------------------------
#
# A basic collection of HTML tags likely to matter to HTML parsers. Does *not*
# include any attributes or attribute values.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

tag_a="<a>"
tag_abbr="<abbr>"
tag_acronym="<acronym>"
tag_address="<address>"
tag_annotation_xml="<annotation-xml>"
tag_applet="<applet>"
tag_area="<area>"
tag_article="<article>"
tag_aside="<aside>"
tag_audio="<audio>"
tag_b="<b>"
tag_base="<base>"
tag_basefont="<basefont>"
tag_bdi="<bdi>"
tag_bdo="<bdo>"
tag_bgsound="<bgsound>"
tag_big="<big>"
tag_blink="<blink>"
tag_blockquote="<blockquote>"
tag_body="<body>"
tag_br="<br>"
tag_button="<button>"
tag_canvas="<canvas>"
tag_caption="<caption>"
```

```
tag_center="<center>"
tag_cite="<cite>"
tag_code="<code>"
tag_col="<col>"
tag_colgroup="<colgroup>"
tag_data="<data>"
tag_datalist="<datalist>"
tag_dd="<dd>"
tag_del="<del>"
tag_desc="<desc>"
tag_details="<details>"
tag_dfn="<dfn>"
tag_dir="<dir>"
tag_div="<div>"
tag_dl="<dl>"
tag_dt="<dt>"
tag_em="<em>"
tag_embed="<embed>"
tag_fieldset="<fieldset>"
tag_figcaption="<figcaption>"
tag_figure="<figure>"
tag_font="<font>"
tag_footer="<footer>"
tag_foreignobject="<foreignobject>"
tag_form="<form>"
tag_frame="<frame>"
tag_frameset="<frameset>"
tag_h1="<h1>"
tag_h2="<h2>"
tag_h3="<h3>"
tag_h4="<h4>"
tag_h5="<h5>"
tag_h6="<h6>"
tag_head="<head>"
tag_header="<header>"
tag_hgroup="<hgroup>"
tag_hr="<hr>"
tag_html="<html>"
tag_i="<i>"
tag_iframe="<iframe>"
tag_image="<image>"
tag_img="<img>"
tag_input="<input>"
tag_ins="<ins>"
tag_isindex="<isindex>"
tag_kbd="<kbd>"
tag_keygen="<keygen>"
tag_label="<label>"
tag_legend="<legend>"
tag_li="<li>"
tag_link="<link>"
tag_listing="<listing>"
tag_main="<main>"
tag_malignmark="<malignmark>"
tag_map="<map>"
tag_mark="<mark>"
tag_marquee="<marquee>"
tag_math="<math>"
tag_menu="<menu>"
tag_menuitem="<menuitem>"
```

```
tag_meta="<meta>"
tag_meter="<meter>"
tag_mglyph="<mglyph>"
tag_mi="<mi>"
tag_mn="<mn>"
tag_mo="<mo>"
tag_ms="<ms>"
tag_mtext="<mtext>"
tag_multicol="<multicol>"
tag_nav="<nav>"
tag_nextid="<nextid>"
tag_nobr="<nobr>"
tag_noembed="<noembed>"
tag_noframes="<noframes>"
tag_noscript="<noscript>"
tag_object="<object>"
tag_ol="<ol>"
tag_optgroup="<optgroup>"
tag_option="<option>"
tag_output="<output>"
tag_p="<p>"
tag_param="<param>"
tag_plaintext="<plaintext>"
tag_pre="<pre>"
tag_progress="<progress>"
tag_q="<q>"
tag_rb="<rb>"
tag_rp="<rp>"
tag_rt="<rt>"
tag_rtc="<rtc>"
tag_ruby="<ruby>"
tag_s="<s>"
tag_samp="<samp>"
tag_script="<script>"
tag_section="<section>"
tag_select="<select>"
tag_small="<small>"
tag_source="<source>"
tag_spacer="<spacer>"
tag_span="<span>"
tag_strike="<strike>"
tag_strong="<strong>"
tag_style="<style>"
tag_sub="<sub>"
tag_summary="<summary>"
tag_sup="<sup>"
tag_svg="<svg>"
tag_table="<table>"
tag_tbody="<tbody>"
tag_td="<td>"
tag_template="<template>"
tag_textarea="<textarea>"
tag_tfoot="<tfoot>"
tag_th="<th>"
tag_thead="<thead>"
tag_time="<time>"
tag_title="<title>"
tag_tr="<tr>"
tag_track="<track>"
tag_tt="<tt>"
```

```
tag_u="<u>"
tag_ul="<ul>"
tag_var="<var>"
tag_video="<video>"
tag_wbr="<wbr>"
tag_xmp="<xmp>"
```

jpeg.dict

```
#
# AFL dictionary for JPEG images
# ------------------------------
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

header_jfif="JFIF\x00"
header_jfxx="JFXX\x00"

section_ffc0="\xff\xc0"
section_ffc2="\xff\xc2"
section_ffc4="\xff\xc4"
section_ffd0="\xff\xd0"
section_ffd8="\xff\xd8"
section_ffd9="\xff\xd9"
section_ffda="\xff\xda"
section_ffdb="\xff\xdb"
section_ffdd="\xff\xdd"
section_ffe0="\xff\xe0"
section_ffe1="\xff\xe1"
section_fffe="\xff\xfe"
```

js.dict

```
#
# AFL dictionary for JavaScript
# -----------------------------
#
# Contains basic reserved keywords and syntax building blocks.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

keyword_arguments="arguments"
keyword_break="break"
keyword_case="case"
keyword_catch="catch"
keyword_const="const"
keyword_continue="continue"
keyword_debugger="debugger"
keyword_decodeURI="decodeURI"
keyword_default="default"
keyword_delete="delete"
keyword_do="do"
keyword_else="else"
keyword_escape="escape"
```

```
keyword_eval="eval"
keyword_export="export"
keyword_finally="finally"
keyword_for="for (a=0;a<2;a++)"
keyword_function="function"
keyword_if="if"
keyword_in="in"
keyword_instanceof="instanceof"
keyword_isNaN="isNaN"
keyword_let="let"
keyword_new="new"
keyword_parseInt="parseInt"
keyword_return="return"
keyword_switch="switch"
keyword_this="this"
keyword_throw="throw"
keyword_try="try"
keyword_typeof="typeof"
keyword_var="var"
keyword_void="void"
keyword_while="while"
keyword_with="with"

misc_1=" 1"
misc_a="a"
misc_array=" [1]"
misc_assign=" a=1"
misc_code_block=" {1}"
misc_colon_num=" 1:"
misc_colon_string=" 'a':"
misc_comma=" ,"
misc_comment_block=" /* */"
misc_comment_line=" //"
misc_cond=" 1?2:3"
misc_dec=" --"
misc_div=" /"
misc_equals=" ="
misc_fn=" a()"
misc_identical=" ==="
misc_inc=" ++"
misc_minus=" -"
misc_modulo=" %"
misc_parentheses=" ()"
misc_parentheses_1=" (1)"
misc_parentheses_1x4=" (1,1,1,1)"
misc_parentheses_a=" (a)"
misc_period="."
misc_plus=" +"
misc_plus_assign=" +="
misc_regex=" /a/g"
misc_rol=" <<<"
misc_semicolon=" ;"
misc_serialized_object=" {'a': 1}"
misc_string=" 'a'"
misc_unicode=" '\\u0001'"

object_Array=" Array"
object_Boolean=" Boolean"
object_Date=" Date"
object_Function=" Function"
```

```
object_Infinity=" Infinity"
object_Int8Array=" Int8Array"
object_Math=" Math"
object_NaN=" NaN"
object_Number=" Number"
object_Object=" Object"
object_RegExp=" RegExp"
object_String=" String"
object_Symbol=" Symbol"
object_false=" false"
object_null=" null"
object_true=" true"

prop_charAt=".charAt"
prop_concat=".concat"
prop_constructor=".constructor"
prop_destructor=".destructor"
prop_length=".length"
prop_match=".match"
prop_proto=".__proto__"
prop_prototype=".prototype"
prop_slice=".slice"
prop_toCode=".toCode"
prop_toString=".toString"
prop_valueOf=".valueOf"
```

json.dict

```
#
# AFL dictionary for JSON
# -----------------------
#
# Just the very basics.
#
# Inspired by a dictionary by Jakub Wilk <jwilk@jwilk.net>
#

"0"
",0"
":0"
"0:"
"-1.2e+3"

"true"
"false"
"null"

"\"\""
",\"\""
":\"\""
"\"\":"

"{}"
",{}"
":{}"
"{\"\":0}"
"{{}}"
```

```
"[]"
",[]"
":[]"
"[0]"
"[[]]"

"''"
"\\"
"\\b"
"\\f"
"\\n"
"\\r"
"\\t"
"\\u0000"
"\\x00"
"\\0"
"\\uD800\\uDC00"
"\\uDBFF\\uDFFF"

"\"\":0"
"//"
"/**/"
```

pdf.dict

```
#
# AFL dictionary for PDF
# ----------------------
#
# This is a pretty big PDF dictionary constructed by Ben by manually reviewing
# the spec and combining that with the data pulled out of a corpus of sample
# PDFs.
#
# Contributed by Ben Nagy <ben@iagu.net>
#

"#"
"%"
"%%"
"%%EOF"
"%FDF-1.7"
"%PDF-1.7"
"("
"(/xdp:xdp)"
"(\\001)"
"(config)"
"(datasets)"
"(template)"
"(xdp:xdp)"
")"
"-1"
"-1.0"
".."
"/"
"/#23clipboard"
"/.notdef"
"/1"
"/1.0"
```

"/1.3"
"/3D"
"/3DA"
"/3DAnimationStyle"
"/3DB"
"/3DD"
"/3DI"
"/3DLightingScheme"
"/3DRenderMode"
"/3DV"
"/3DView"
"/90pv-RKSJ-H"
"/A"
"/A0"
"/A85"
"/AA"
"/AAIC"
"/AAPL"
"/ABCDEF+ACaslonPro-Regular"
"/ABCDEF+AJensonPro-LtIt"
"/ABCDEF+AdobeCorpID-MinionRg"
"/ABCDEF+Arial,Bold"
"/ABCDEF+BankGothicMdBT"
"/ABCDEF+Bauhaus-Heavy"
"/ABCDEF+BluesClues"
"/ABCDEF+BodegaSans"
"/ABCDEF+BodoniMTCondensed"
"/ABCDEF+BookAntiqua"
"/ABCDEF+CMBX10"
"/ABCDEF+CaflischScriptPro-Regular"
"/ABCDEF+CityBlueprint"
"/ABCDEF+CourierNewPSMT"
"/ABCDEF+FixedsysExcelsior2.00"
"/ABCDEF+MSTT31854bd45bo188067S00"
"/ABCDEF+MinionPro-BoldCnIt"
"/ABCDEF+MyriadMM-It_400_300_"
"/ABCDEF+Wingdings"
"/ABCDEF+ZapfDingbats"
"/AC"
"/ADBE"
"/ADB_DEVICE_DEFAULT_STYLE"
"/ADB_DefaultStyle"
"/ADB_NO_TRAP_STYLE"
"/AE"
"/AESV2"
"/AGaramond"
"/AH"
"/AI8DstIndex"
"/AI8SrcIndex"
"/AIMetaData"
"/AIPDFPrivateData1"
"/AIS"
"/AL"
"/AN"
"/AP"
"/AS"
"/ASCII85Decode"
"/ASCIIHexDecode"
"/ASomewhatLongerName"
"/AU"

"/Aacute"
"/Acc.#20Prod.#202501#20#2F2#20#20"
"/Accounts#20payable"
"/AccurateScreens"
"/Acircumflex"
"/AcroForm"
"/Action"
"/Actual"
"/Add"
"/Adieresis"
"/Adobe"
"/Adobe#20PDF#20Library"
"/Adobe.PPKLite"
"/AdobeCorpID-Acrobat"
"/AdobeCorpID-MinionRg"
"/AdobePhotoshop"
"/Agrave"
"/All"
"/AllKO"
"/AllOn"
"/Alt"
"/Alternate"
"/AlternatePresentations"
"/Alternates"
"/Amex"
"/And"
"/Angle"
"/Annot"
"/Annots"
"/AntiAlias"
"/AnyOn"
"/Apag_PDFX_Checkup"
"/App"
"/Architecture-Normal"
"/Arial"
"/Aring"
"/Art"
"/ArtBox"
"/Article"
"/Artifact"
"/Artwork"
"/Ascent"
"/Aspect"
"/Assistant"
"/Atilde"
"/AuthEvent"
"/Author"
"/Avenir-Heavy"
"/Avenir-MediumOblique"
"/AvgWidth"
"/BBox"
"/BC"
"/BCL"
"/BDC"
"/BDL"
"/BE"
"/BFSOL"
"/BG"
"/BG2"
"/BM"

"/BMC"
"/BS"
"/BW"
"/Bank"
"/BaseEncoding"
"/BaseFont"
"/BaseState"
"/BaseVersion"
"/Birch"
"/BitsPerComponent"
"/BitsPerCoordinate"
"/BitsPerFlag"
"/BitsPerSample"
"/Bl"
"/BlCDel"
"/BlMiNu"
"/Black"
"/BlackIs1"
"/BlackOP"
"/BlackPoint"
"/BleedBox"
"/Blend"
"/Block"
"/Blue"
"/BluesClues"
"/Bookshelf"
"/Border"
"/Bounds"
"/BoxColorInfo"
"/Btn"
"/BulmerMT-BoldDisplay"
"/ByteRange"
"/C"
"/C0"
"/C0_0"
"/C1"
"/C2w"
"/C3"
"/CALS_AIS"
"/CALS_BM"
"/CALS_HT"
"/CALS_SMASK"
"/CALS_ca"
"/CAM"
"/CB"
"/CC"
"/CCH"
"/CCITTFaxDecode"
"/CD"
"/CDL"
"/CEN"
"/CF"
"/CFM"
"/CI"
"/CIDFontType0"
"/CIDFontType0C"
"/CIDFontType2"
"/CIDInit"
"/CIDSet"
"/CIDSystemInfo"

"/CIDToGIDMap"
"/CMV_LabBar"
"/CMV_LabControl"
"/CMYK"
"/CMYK#20#2880,#208,#2034,#200#29"
"/CMap"
"/CMapName"
"/CMapType"
"/CMapVersion"
"/CO"
"/CP"
"/CS"
"/CS0"
"/CT"
"/CV"
"/CalGray"
"/CalRGB"
"/CapHeight"
"/Caption"
"/Caslon540BT-Roman"
"/CaslonBT-Bold"
"/CaslonBT-BoldItalic"
"/Catalog"
"/Category"
"/Ccedilla"
"/CenturySchoolbookBT-Roman"
"/Ch"
"/Chair"
"/Chap"
"/Chaparral-Display"
"/CharProcs"
"/CharSet"
"/CheckSum"
"/Circle"
"/ClarendonBT-Black"
"/ClassMap"
"/Clearface-Black"
"/Clip"
"/ClippedText"
"/Cn"
"/Collection"
"/CollectionItem"
"/CollectionSchema"
"/CollectionSubitem"
"/Color"
"/ColorBurn"
"/ColorDodge"
"/ColorMatch"
"/ColorSpace"
"/ColorTransform"
"/ColorType"
"/Colorants"
"/Colors"
"/Columns"
"/ComicSansMS,Bold"
"/Comment"
"/Comments"
"/Company"
"/Compatibility"
"/Compatible"

"/Components"
"/CompressArt"
"/Condensed"
"/Configs"
"/Consultant"
"/ContainerVersion"
"/Contents"
"/Coords"
"/Copy"
"/Copy#20center"
"/Cor"
"/Corner#20surface"
"/CosineDot"
"/Count"
"/Cour"
"/Courier"
"/Create"
"/CreationDate"
"/Creator"
"/CreatorInfo"
"/CreatorVersion"
"/CropBox"
"/CropFixed"
"/CropRect"
"/Crypt"
"/CryptFilter"
"/CryptFilterDecodeParms"
"/Cs12"
"/Cs3"
"/Cyan"
"/D"
"/DA"
"/DCTDecode"
"/DIC#202525p*"
"/DIS"
"/DL"
"/DOS"
"/DP"
"/DR"
"/DS"
"/DSz"
"/DV"
"/DW"
"/DamagedRowsBeforeError"
"/Darken"
"/Data"
"/Date"
"/Decode"
"/DecodeParms"
"/DefEmbeddedFile"
"/Default"
"/DefaultCryptFilter"
"/DefaultForPrinting"
"/DefaultRGB"
"/Delete"
"/Delta"
"/DescendantFonts"
"/Descent"
"/Description"
"/Design"

"/Dest"
"/DestOutputProfile"
"/DestOutputProfileRef"
"/Dests"
"/DeviceCMYK"
"/DeviceGray"
"/DeviceN"
"/DeviceRGB"
"/Difference"
"/Differences"
"/DigestLocation"
"/DigestMethod"
"/DigestValue"
"/Dimmed"
"/Direction"
"/DisplayDocTitle"
"/Dissolve"
"/Div"
"/Dm"
"/DocMDP"
"/DocOpen"
"/Document"
"/Documents"
"/Domain"
"/Door"
"/DotGain"
"/Draw"
"/Dt"
"/Dur"
"/Dynamic#20connector"
"/E"
"/EF"
"/EFF"
"/EMC"
"/Eacute"
"/EarlyChange"
"/Ecircumflex"
"/Edieresis"
"/Editable"
"/Egrave"
"/EmbedFonts"
"/EmbedICCProfile"
"/Embedded"
"/EmbeddedFile"
"/EmbeddedFiles"
"/Encode"
"/EncodedByteAlign"
"/Encoding"
"/Encrypt"
"/EncryptMetadata"
"/EndIndent"
"/EndOfBlock"
"/EndOfLine"
"/Euro"
"/Euro.037"
"/Event"
"/ExData"
"/Exchange-Pro"
"/Exclude"
"/Exclusion"

"/Executive"
"/Export"
"/ExportCrispy"
"/ExportState"
"/ExtGState"
"/Extend"
"/Extends"
"/ExtensionLevel"
"/Extensions"
"/F1"
"/F1.0"
"/F12"
"/F13"
"/F3"
"/F5"
"/F6"
"/F7"
"/F8"
"/FB"
"/FD"
"/FDecodeParms"
"/FFilter"
"/FICL"
"/FM"
"/FOV"
"/FRM"
"/FS"
"/FT"
"/Facilities"
"/Fade"
"/False"
"/Feature"
"/FedEx#20Orange"
"/FedEx#20Purple"
"/Field"
"/Fields"
"/Figure"
"/File"
"/Files"
"/Filespec"
"/FillIn"
"/Filter"
"/First"
"/FirstChar"
"/FirstPage"
"/Fit"
"/FitB"
"/FitBH"
"/FitBV"
"/FitH"
"/FitR"
"/FitV"
"/FitWindow"
"/FixedPrint"
"/Flags"
"/FlateDecode"
"/Fm0"
"/Fm4"
"/Fo"
"/Focoltone#201047"

"/Font"
"/FontBBox"
"/FontDescriptor"
"/FontFamily"
"/FontFile"
"/FontFile2"
"/FontMatrix"
"/FontName"
"/FontStretch"
"/FontWeight"
"/Form"
"/FormEx"
"/FormType"
"/FreeText"
"/FreeTextCallout"
"/Frequency"
"/FullSave"
"/FullScreen"
"/Function"
"/FunctionType"
"/Functions"
"/Futura-Bold"
"/Futura-CondensedExtraBold"
"/G"
"/GO2"
"/GLGR"
"/GS0"
"/GS1"
"/GS2"
"/GTS"
"/GTS_PDFA1"
"/GTS_PDFX"
"/GTS_PDFXConformance"
"/GTS_PDFXVersion"
"/GWG#20Green"
"/Gamma"
"/Garamond"
"/Georgia,Bold"
"/GoTo"
"/GoTo3DView"
"/GoToE"
"/GoToR"
"/Gold"
"/Goudy"
"/Gray"
"/Green"
"/GreymantleMVB"
"/GrotesqueMT"
"/Group"
"/H"
"/HDAG_Tools"
"/HKana"
"/HT"
"/HT2"
"/Halftone"
"/HalftoneName"
"/HalftoneType"
"/HardLight"
"/HeBo"
"/Head1"

"/Headlamp"
"/Height"
"/HeiseiMin"
"/Helv"
"/Helvetica"
"/Helvetica-Bold"
"/Helvetica-BoldOblique"
"/Helvetica-Condensed"
"/HelveticaNeue-Black"
"/Hide"
"/HonMincho-M"
"/Horizontal"
"/Hue"
"/I"
"/IO"
"/IC"
"/ICCBased"
"/ICCVersion"
"/ID"
"/IDS"
"/IDTree"
"/IEC"
"/IF"
"/IN"
"/ISO32000Registry"
"/ISO_PDFE1"
"/ISO_PDFEVersion"
"/IT"
"/ITO"
"/ITP"
"/IV"
"/IX"
"/Icircumflex"
"/Icon"
"/Identity"
"/Identity-H"
"/IgnEP"
"/Illustrator"
"/Illustrator8.0"
"/Im0"
"/Im1"
"/Im2"
"/Im3"
"/Im4"
"/Image"
"/Image1"
"/ImageB"
"/ImageC"
"/ImageI"
"/ImageMask"
"/ImageResources"
"/ImageType"
"/Import"
"/ImportData"
"/ImpressBT-Regular"
"/Index"
"/Indexed"
"/Info"
"/Information#20services"
"/Ink"

"/InkList"
"/InsertPages"
"/Insignia"
"/IntegerItem"
"/Intent"
"/Interpolate"
"/ItalicAngle"
"/ItcKabel-Ultra"
"/Item1"
"/Item2"
"/JBIG2Decode"
"/JBIG2Globals"
"/JPXDecode"
"/JS"
"/JT"
"/JTC"
"/JTF"
"/JTFile"
"/JTM"
"/JavaScript"
"/JobTicketContents"
"/Justify"
"/Keywords"
"/Kids"
"/L"
"/L1"
"/L1a"
"/L1b"
"/L2R"
"/L5O188"
"/LBody"
"/LI"
"/LL"
"/LLE"
"/LLO"
"/LS"
"/LSP"
"/LZW"
"/LZWDecode"
"/Lab"
"/Lang"
"/Last"
"/LastChar"
"/LastItem"
"/LastModified"
"/Lateral#20file"
"/Launch"
"/Layout"
"/Lbl"
"/Leading"
"/Legal"
"/Length"
"/Length1"
"/Length2"
"/Length3"
"/LetterspaceFlags"
"/Lighten"
"/Limits"
"/Line"
"/LineDimension"

"/LineHeight"
"/Linear"
"/Linearized"
"/Link"
"/Locked"
"/LogoGreen"
"/LrTb"
"/Lslash"
"/Luminosity"
"/M"
"/MB"
"/MC"
"/MC0"
"/MCD"
"/MCID"
"/MCR"
"/MD5"
"/MH"
"/MIT"
"/MK"
"/MMType1"
"/MP"
"/MR"
"/MS"
"/MUX#20#2F#20DEMUX"
"/Mac"
"/MacRomanEncoding"
"/Magenta"
"/Manager"
"/MarkInfo"
"/Marked"
"/MarkedPDF"
"/Marker#20board"
"/Markup3D"
"/Mask"
"/Mastercard"
"/Matrix"
"/Max"
"/MaxLen"
"/MaxWidth"
"/Me"
"/Measure"
"/MediaBox"
"/MetaData"
"/Min"
"/MinionMM"
"/MissingWidth"
"/MixedContainer"
"/MixingHints"
"/ModDate"
"/Mode"
"/Modify"
"/Movie"
"/Msg"
"/MurrayHillBT-Bold"
"/MxGeom"
"/MxLaNu"
"/MxPts"
"/MyriadPro-Black"
"/NA"

"/NChannel"
"/ND"
"/NL"
"/NM"
"/NR"
"/Name"
"/Name1"
"/Named"
"/Names"
"/NeedsRendering"
"/NewCenturySchlbk-Italic"
"/NewWindow"
"/Next"
"/NextPage"
"/No"
"/NonEFontNoWarn"
"/NonStruct"
"/None"
"/Normal"
"/Not"
"/NotDefSpecial"
"/NumBlock"
"/Nums"
"/OB"
"/OBJR"
"/OC"
"/OC2"
"/OC3"
"/OC4"
"/OCG"
"/OCGs"
"/OCL"
"/OCMD"
"/OCProperties"
"/OE"
"/OFF"
"/OLN"
"/ON"
"/OOL"
"/OPBG"
"/OPBS"
"/OPI"
"/OPM"
"/OS"
"/OT"
"/Oacute"
"/Obj"
"/ObjStm"
"/Ocircumflex"
"/Odieresis"
"/Ograve"
"/Omega"
"/OneColumn"
"/Online"
"/Open"
"/OpenAction"
"/Operation"
"/Opt"
"/OptionSet"
"/Options"

"/Or"
"/Orange"
"/Order"
"/Ordering"
"/OriginalLayerName"
"/Oslash"
"/Otilde"
"/Outlines"
"/OutputCondition"
"/OutputConditionIdentifier"
"/OutputIntent"
"/OutputIntents"
"/Overlay"
"/P0"
"/P1"
"/P2"
"/P2,#2300ff007900000000,PANTONE#20151#20C"
"/PANTONE"
"/PANTONE#20158-5#20CVS"
"/PANTONE#20221#20CVU"
"/PANTONE#203405#20C"
"/PANTONE#20399#20CVC"
"/PANTONE#20Blue#20072#20C"
"/PANTONE#20Orange#20021#20C"
"/PANTONE#20Orange#20021#20CVC"
"/PANTONE#20Yellow#20C"
"/PC"
"/PDFDocEncoding"
"/PIX"
"/PO"
"/PS"
"/PUBLISHER"
"/PZ"
"/Pa0"
"/Page"
"/PageElement"
"/PageLabels"
"/PageLayout"
"/PageMode"
"/PageRange"
"/Pages"
"/PaintType"
"/Palatino,Bold"
"/Pale#20Brown.c"
"/Panose"
"/Paper#20tray"
"/Para"
"/Params"
"/Parent"
"/ParentTree"
"/ParentTreeNextKey"
"/Part"
"/Pattern"
"/PatternType"
"/PcZ"
"/Perceptual"
"/Perms"
"/Pg"
"/Pgf"
"/PieceInfo"

"/PitStop"
"/Placement"
"/Play"
"/Polygon"
"/PolygonCloud"
"/Popup"
"/Position"
"/PowerUpPDF"
"/PrOut"
"/PrRGBGra"
"/PrRGBIma"
"/Predictor"
"/PresSteps"
"/PreserveRB"
"/Prev"
"/PrevPage"
"/Preview"
"/Print"
"/PrintRecord"
"/PrintScaling"
"/PrintState"
"/PrintStyle"
"/Printed"
"/PrintingOrder"
"/Private"
"/ProcSet"
"/Process"
"/ProcessBlue"
"/ProcessGreen"
"/ProcessRed"
"/Producer"
"/ProfileCS"
"/ProfileName"
"/Prop_Build"
"/Properties"
"/Proportional"
"/PubSec"
"/Q"
"/QuadPoints"
"/R1"
"/RBGroups"
"/RC"
"/RD"
"/REC"
"/REx"
"/RF"
"/RGB"
"/RI"
"/RICMYKGra"
"/RICMYKIma"
"/RICalGra"
"/RICalIma"
"/RIDefault"
"/RIDevNGra"
"/RIDevNIma"
"/RIRGBGra"
"/RIRGBIma"
"/RL"
"/RM"
"/RV"

"/Range"
"/Rect"
"/Red"
"/Redact"
"/Ref"
"/Reference"
"/Registry"
"/RegistryName"
"/RelativeColorimetric"
"/Rendition"
"/Renditions"
"/Requirements"
"/ResetForm"
"/Resolution"
"/Resources"
"/ReversedChars"
"/RoleMap"
"/Root"
"/Rotate"
"/Round"
"/RoundTrip"
"/RoundtripVersion"
"/Router"
"/Rows"
"/RunLengthDecode"
"/Ryumin"
"/SA"
"/SBDraft"
"/SC"
"/SE"
"/SFSSL"
"/SFTWS"
"/SI"
"/SL"
"/SLA"
"/SM"
"/SMask"
"/SMaskInData"
"/SP"
"/SPS"
"/STL"
"/SU"
"/SW"
"/Saturation"
"/SaveAs"
"/SaveContents"
"/SaveResource"
"/SavedBy"
"/Scaron"
"/Schema"
"/Screen"
"/Sect"
"/SemiCondensed"
"/SemiExpanded"
"/Separation"
"/SeparationInfo"
"/SetOCGState"
"/SettingsFileName"
"/Sh0"
"/Sh1"

"/Shading"
"/ShadingType"
"/Shape"
"/Sig"
"/SigFlags"
"/SigRef"
"/Signature"
"/Signed"
"/SinglePage"
"/Size"
"/SlideShow"
"/SoftLight"
"/Solid"
"/Solidities"
"/SomeName"
"/Sort"
"/Sound"
"/Space"
"/SpaceAfter"
"/SpaceBefore"
"/Span"
"/SpawnTemplate"
"/SpdrArt"
"/SpiderInfo"
"/Split"
"/Spot"
"/Spot1"
"/Spot2"
"/SpotFunction"
"/SpotMap"
"/St"
"/Stamp"
"/StandardImageFileData"
"/Star"
"/Start"
"/StartIndent"
"/StartResource"
"/State"
"/StdCF"
"/StemH"
"/StemV"
"/Stm"
"/StmF"
"/Stop"
"/Story"
"/StrF"
"/StrikeOut"
"/StringItem"
"/StructElem"
"/StructParent"
"/StructParents"
"/StructTreeRoot"
"/Style"
"/SubFilter"
"/SubType"
"/Subdictionary"
"/Subform"
"/Subj"
"/Subject"
"/SubmitForm"

"/SubmitStandalone"
"/SubsetFontsBelow"
"/SubsetFontsRatio"
"/Supplement"
"/Swiss721BT-Black"
"/Switch"
"/T"
"/T1"
"/T1_0"
"/TB"
"/TC"
"/TCS"
"/TF"
"/TID"
"/TK"
"/TM"
"/TO"
"/TOC"
"/TOCI"
"/TOYO#200004pc"
"/TP"
"/TR"
"/TR2"
"/TRUMATCH#206-e"
"/TS"
"/TSV"
"/TT"
"/TT0"
"/TTRefMan"
"/TU"
"/TV"
"/TW"
"/TWS"
"/TWY"
"/Tabs"
"/TagSuspect"
"/TargetCS"
"/Technical"
"/Template"
"/TemplateInstantiated"
"/Templates"
"/Text"
"/TextAlign"
"/TextBox"
"/TextIndent"
"/The"
"/This"
"/Thorn"
"/Thread"
"/Threads"
"/Thumb"
"/Thumbnail"
"/Thumbs"
"/Ti"
"/TiBI"
"/TilingType"
"/Times-BoldItalic"
"/Times-Roman"
"/Title"
"/ToUnicode"

"/Toggle"
"/Trans"
"/TransferFunction"
"/TransformMethod"
"/TransformParams"
"/Transparency"
"/TrapInfo"
"/TrapMagicNumber"
"/TrapRegions"
"/TrapSet"
"/Trapped"
"/Trapping"
"/TrappingDetails"
"/TrappingParameters"
"/TrimBox"
"/True"
"/TrueType"
"/TrustedMode"
"/TwoColumnLeft"
"/Tx"
"/Type"
"/Type0"
"/U3D"
"/UA"
"/UCR"
"/UCR2"
"/UIDOffset"
"/UR"
"/UR3"
"/URI"
"/URL"
"/URLs"
"/Uacute"
"/Ucircumflex"
"/Udieresis"
"/Ugrave"
"/Univers-BoldExt"
"/Unix"
"/Unknown"
"/Usage"
"/UseAttachments"
"/UseNone"
"/UseOC"
"/UseOutlines"
"/UseThumbs"
"/UsedCMYK"
"/UserProperties"
"/UserUnit"
"/V2"
"/VA"
"/VE"
"/VP"
"/Verdana,Bold"
"/Version"
"/Vertical"
"/VeryLastItem"
"/View"
"/ViewerPreferences"
"/Visa"
"/Visible"

"/Volume"
"/W2"
"/WAI"
"/WAN"
"/WMode"
"/WP"
"/WarnockPro-BoldIt"
"/Watermark"
"/WebCapture"
"/Which"
"/WhiteBG"
"/WhitePoint"
"/Widget"
"/Width"
"/Widths"
"/Win"
"/WinAnsiEncoding"
"/Window"
"/Windows"
"/Work#20surface"
"/Workbook"
"/Worksheet"
"/WritingMode"
"/X"
"/X1"
"/XFA"
"/XHeight"
"/XML"
"/XN"
"/XObject"
"/XRef"
"/XRefStm"
"/XStep"
"/XUID"
"/XYZ"
"/Y"
"/YStep"
"/Yacute"
"/Ydieresis"
"/Yellow"
"/Z"
"/Z7KNXbN"
"/ZaDb"
"/ZapfDingbats"
"/Zcaron"
"/Zoom"
"/_No_paragraph_style_"
"/a1"
"/acute"
"/adbe.pkcs7.detached"
"/ampersand"
"/apple"
"/approxequal"
"/asciicircum"
"/asciitilde"
"/asterisk"
"/at"
"/audio#2Fmpeg"
"/b"
"/backslash"

```
"/bar"
"/blank"
"/braceleft"
"/braceright"
"/bracketleft"
"/bracketright"
"/breve"
"/brokenbar"
"/bullet"
"/c108"
"/cCompKind"
"/cCompQuality"
"/cCompression"
"/cRes"
"/cResolution"
"/ca"
"/caron"
"/cedilla"
"/cent"
"/circumflex"
"/colon"
"/comma"
"/copyright"
"/currency"
"/dagger"
"/daggerdbl"
"/degree"
"/deviceNumber"
"/dieresis"
"/divide"
"/dollar"
"/dotaccent"
"/dotlessi"
"/dotlessj"
"/eight"
"/ellipsis"
"/emdash"
"/endash"
"/equal"
"/eth"
"/exclam"
"/exclamdown"
"/f"
"/ff"
"/ffi"
"/ffl"
"/fi"
"/five"
"/fl"
"/florin"
"/four"
"/fraction"
"/gCompKind"
"/gCompQuality"
"/gCompression"
"/gRes"
"/gResolution"
"/germandbls"
"/go1"
"/grave"
```

```
"/greater"
"/greaterequal"
"/guillemotleft"
"/guillemotright"
"/guilsinglleft"
"/guilsinglright"
"/hungarumlaut"
"/hyphen"
"/iacute"
"/idieresis"
"/igrave"
"/infinity"
"/integral"
"/j"
"/k"
"/less"
"/lessequal"
"/logicalnot"
"/lozenge"
"/lt#20blue"
"/mCompKind"
"/mCompression"
"/mRes"
"/mResolution"
"/macron"
"/minus"
"/mu"
"/multiply"
"/n"
"/n0"
"/nine"
"/notequal"
"/ntilde"
"/numbersign"
"/o"
"/ogonek"
"/one"
"/onehalf"
"/onequarter"
"/onesuperior"
"/op"
"/ordfeminine"
"/ordmasculine"
"/p"
"/pageH"
"/pageV"
"/paragraph"
"/parenleft"
"/parenright"
"/partialdiff"
"/pdf"
"/pdfx"
"/percent"
"/period"
"/periodcentered"
"/perthousand"
"/pi"
"/plus"
"/plusminus"
"/pms#208400"
```

"/printX"
"/product"
"/question"
"/questiondown"
"/quotedbl"
"/quotedblbase"
"/quotedblleft"
"/quotedblright"
"/quoteleft"
"/quoteright"
"/quotesinglbase"
"/quotesingle"
"/r"
"/radical"
"/registered"
"/ring"
"/s"
"/s1"
"/sd1"
"/sd2"
"/section"
"/semicolon"
"/seven"
"/six"
"/slash"
"/sterling"
"/summation"
"/thinspace"
"/three"
"/threequarters"
"/threesuperior"
"/tilde"
"/trademark"
"/two"
"/twosuperior"
"/u"
"/underscore"
"/v"
"/w"
"/y1"
"/yen"
"/yes"
"/zero"
"0 R"
"1"
"1.0"
"<"
"<<"
">"
">>"
"Adobe.PPKLite"
"Adobe.PubSec"
"B*"
"BDC"
"BI"
"BMC"
"BT"
"BX"
"CS"
"DP"

```
"Do"
"EI"
"EMC"
"ET"
"EX"
"Entrust.PPKEF"
"ID"
"MP"
"R"
"T*"
"TJ"
"TL"
"Tc"
"Td"
"Tf"
"Tj"
"Tm"
"Tr"
"Ts"
"Tw"
"W*"
"["
"[0.0 0.0 0.0 0.0 0.0 0.0]"
"[1 1 1]"
"[1.0 -1.0 1.0 -1.0]"
"[1.0 -1.0]"
"\\"
"]"
"abs"
"adbe.pkcs7.s3"
"adbe.pkcs7.s4"
"adbe.pkcs7.s5"
"add"
"and"
"atan"
"begin"
"beginarrangedfont"
"beginbfchar"
"begincidrange"
"begincmap"
"begincodespacerange"
"beginnotdefchar"
"beginnotdefrange"
"beginusematrix"
"bitshift"
"ceiling"
"cm"
"copy"
"cos"
"cvi"
"cvr"
"d0"
"d1"
"div"
"dup"
"end"
"endarrangedfont"
"endbfchar"
"endcidrange"
"endcmap"
```

```
"endcodespacerange"
"endnotdefchar"
"endnotdefrange"
"endobj"
"endstream"
"endusematrix"
"eq"
"exch"
"exp"
"f*"
"false"
"findresource"
"floor"
"ge"
"gs"
"gt"
"idiv"
"if"
"ifelse"
"index"
"le"
"ln"
"log"
"lt"
"mod"
"mul"
"ne"
"neg"
"not"
"null"
"obj"
"or"
"page"
"pop"
"re"
"rg"
"ri"
"roll"
"round"
"sin"
"sqrt"
"startxref"
"stream"
"sub"
"trailer"
"true"
"truncate"
"usecmap"
"usefont"
"xor"
"xref"
"{"
"}"
```

perl.dict

```
#
# AFL dictionary for fuzzing Perl
# -------------------------------
#
# Created by @RandomDhiraj
#

"<:crlf"
"fwrite()"
"fread()"
":raw:utf8"
":raw:eol(LF)"
"Perl_invert()"
":raw:eol(CRLF)"
"Perl_PerlIO_eof()"
```

png.dict

```
#
# AFL dictionary for PNG images
# -----------------------------
#
# Just the basic, standard-originating sections; does not include vendor
# extensions.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

header_png="\x89PNG\x0d\x0a\x1a\x0a"

section_IDAT="IDAT"
section_IEND="IEND"
section_IHDR="IHDR"
section_PLTE="PLTE"
section_bKGD="bKGD"
section_cHRM="cHRM"
section_fRAc="fRAc"
section_gAMA="gAMA"
section_gIFg="gIFg"
section_gIFt="gIFt"
section_gIFx="gIFx"
section_hIST="hIST"
section_iCCP="iCCP"
section_iTXt="iTXt"
section_oFFs="oFFs"
section_pCAL="pCAL"
section_pHYs="pHYs"
section_sBIT="sBIT"
section_sCAL="sCAL"
section_sPLT="sPLT"
section_sRGB="sRGB"
section_sTER="sTER"
section_tEXt="tEXt"
section_tIME="tIME"
```

```
section_tRNS="tRNS"
section_zTXt="zTXt"
```

regexp.dict

```
#
# AFL dictionary for JS regex
# --------------------------
#
# Contains various regular expressions.
#
# Created by Yang Guo <yangguo@chromium.org>
#
"?"
"abc"
"()"
"[]"
"abc|def"
"abc|def|ghi"
"^xxx$"
"ab\\b\\d\\bcd"
"\\w|\\d"
"a*?"
"abc+"
"abc+?"
"xyz?"
"xyz??"
"xyz{0,1}"
"xyz{0,1}?"
"xyz{93}"
"xyz{1,32}"
"xyz{1,32}?"
"xyz{1,}"
"xyz{1,}?"
"a\\fb\\nc\\rd\\te\\vf"
"a\\nb\\bc"
"(?:foo)"
"(?: foo )"
"foo|(bar|baz)|quux"
"foo(?=bar)baz"
"foo(?!bar)baz"
"foo(?<=bar)baz"
"foo(?<!bar)baz"
"()"
"(?=)"
"[]"
"[x]"
"[xyz]"
"[a-zA-Z0-9]"
"[-123]"
"[^123]"
"]"
"}"
"[a-b-c]"
"[x\\dz]"
"[\\d-z]"
"[\\d-\\d]"
"[z-\\d]"
```

```
"\\cj\\cJ\\ci\\cI\\ck\\cK"
"\\c!"
"\\c_"
"\\c~"
"[\\c!]"
"[\\c_]"
"[\\c~]"
"[\\ca]"
"[\\cz]"
"[\\cA]"
"[\\cZ]"
"[\\c1]"
"\\[\\]\\{\\}\\(\\)\\%\\^\\#\\ "
"[\\[\\]\\{\\}\\(\\)\\%\\^\\#\\ ]"
"\\8"
"\\9"
"\\11"
"\\11a"
"\\011"
"\\118"
"\\111"
"\\1111"
"(x)(x)(x)\\1"
"(x)(x)(x)\\2"
"(x)(x)(x)\\3"
"(x)(x)(x)\\4"
"(x)(x)(x)\\1*"
"(x)(x)(x)\\3*"
"(x)(x)(x)\\4*"
"(x)(x)(x)(x)(x)(x)(x)(x)(x)(x)\\10"
"(x)(x)(x)(x)(x)(x)(x)(x)(x)(x)\\11"
"(a)\\1"
"(a\\1)"
"(\\1a)"
"(\\2)(\\1)"
"(?=a){0,10}a"
"(?=a){1,10}a"
"(?=a){9,10}a"
"(?!a)?a"
"\\1(a)"
"(?!(a))\\1"
"(?!\\1(a\\1)\\1)\\1"
"\\1\\2(a(?:\\1(b\\1\\2))\\2)\\1"
"[\\0]"
"[\\11]"
"[\\11a]"
"[\\011]"
"[\\00011]"
"[\\118]"
"[\\111]"
"[\\1111]"
"\\x60"
"\\x3z"
"\\c"
"\\u0034"
"\\u003z"
"foo[z]*"
"\\u{12345}"
"\\u{12345}\\u{23456}"
"\\u{12345}{3}"
```

```
"\\u{12345}*"
"\\ud808\\udf45*"
"[\\ud808\\udf45-\\ud809\\udccc]"
"a"
"a|b"
"a\\n"
"a$"
"a\\b!"
"a\\Bb"
"a*?"
"a?"
"a??"
"a{0,1}?"
"a{1,2}?"
"a+?"
"(a)"
"(a)\\1"
"(\\1a)"
"\\1(a)"
"a\\s"
"a\\S"
"a\\D"
"a\\w"
"a\\W"
"a."
"a\\q"
"a[a]"
"a[^a]"
"a[a-z]"
"a(?:b)"
"a(?=b)"
"a(?!b)"
"\\x60"
"\\u0060"
"\\cA"
"\\q"
"\\1112"
"(a)\\1"
"(?!a)?a\\1"
"(?:(?=a))a\\1"
"a{}"
"a{,}"
"a{"
"a{z}"
"a{12z}"
"a{12,"
"a{12,3b"
"{}"
"{,}"
"{"
"{z}"
"{1z}"
"{12,"
"{12,3b"
"a"
"abc"
"a[bc]d"
"a|bc"
"ab|c"
"a||bc"
```

```
"(?:ab)"
"(?:ab|cde)"
"(?:ab)|cde"
"(ab)"
"(ab|cde)"
"(ab)\\1"
"(ab|cde)\\1"
"(?:ab)?"
"(?:ab)+"
"a?"
"a+"
"a??"
"a*?"
"a+?"
"(?:a?)?"
"(?:a+)?"
"(?:a?)+"
"(?:a*)+"
"(?:a+)+"
"(?:a?)*"
"(?:a*)*"
"(?:a+)*"
"a{0}"
"(?:a+){0,0}"
"a*b"
"a+b"
"a*b|c"
"a+b|c"
"(?:a{5,1000000}){3,1000000}"
"(?:ab){4,7}"
"a\\bc"
"a\\sc"
"a\\Sc"
"a(?=b)c"
"a(?=bbb|bb)c"
"a(?!bbb|bb)c"
"\xe2\x81\xa3"
"[\xe2\x81\xa3]"
"\xed\xb0\x80"
"\xed\xa0\x80"
"(\xed\xb0\x80)\x01"
"((\xed\xa0\x80))\x02"
"\xf0\x9f\x92\xa9"
"\x01"
"\x0f"
"[-\xf0\x9f\x92\xa9]+"
"[\xf0\x9f\x92\xa9-\xf4\x8f\xbf\xbf]"
"(?<=)"
"(?<=a)"
"(?<!)"
"(?<!a)"
"(?<a>)"
"(?<a>.)"
"(?<a>.)\\k<a>"
"\\p{Script=Greek}"
"\\P{sc=Greek}"
"\\p{Script_Extensions=Greek}"
"\\P{scx=Greek}"
"\\p{General_Category=Decimal_Number}"
"\\P{gc=Decimal_Number}"
```

```
"\\p{gc=Nd}"
"\\P{Decimal_Number}"
"\\p{Nd}"
"\\P{Any}"
"\\p{Changes_When_NFKC_Casefolded}"
"(?:a?)??"
"a?)"xyz{93}"
"{93}"
"a{12za?)?"
"[\x8f]"
"[\xf0\x9f\x92\xa9-\xf4\x8f\xbf\x92\xa9-\xf4\x8f\xbf\xbf]"
"[\x92\xa9-\xf4\x8f\xbf\xbf]"
"\\1\\2(b\\1\\2))\\2)\\1"
"\\1\\2(a(?:\\1\\2))\\2)\\1"
"?:\\1"
"\\1(b\\1\\2))\\2)\\1"
"\\1\\2(a(?:\\1(b\\1\\2))\\2)\\1"
"foo(?=bar)bar)baz"
"fo(?o(?o(?o(?=bar)baz"
"foo(?=bar)baz"
"foo(?=bar)bar)az"
```

sql.dict

```
#
# AFL dictionary for SQL
# ----------------------
#
# Modeled based on SQLite documentation, contains some number of SQLite
# extensions. Other dialects of SQL may benefit from customized dictionaries.
#
# If you append @1 to the file name when loading this dictionary, afl-fuzz
# will also additionally load a selection of pragma keywords that are very
# specific to SQLite (and are probably less interesting from the security
# standpoint, because they are usually not allowed in non-privileged
# contexts).
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

function_abs=" abs(1)"
function_avg=" avg(1)"
function_changes=" changes()"
function_char=" char(1)"
function_coalesce=" coalesce(1,1)"
function_count=" count(1)"
function_date=" date(1,1,1)"
function_datetime=" datetime(1,1,1)"
function_decimal=" decimal(1,1)"
function_glob=" glob(1,1)"
function_group_concat=" group_concat(1,1)"
function_hex=" hex(1)"
function_ifnull=" ifnull(1,1)"
function_instr=" instr(1,1)"
function_julianday=" julianday(1,1,1)"
function_last_insert_rowid=" last_insert_rowid()"
function_length=" length(1)"
function_like=" like(1,1)"
```

```
function_likelihood=" likelihood(1,1)"
function_likely=" likely(1)"
function_load_extension=" load_extension(1,1)"
function_lower=" lower(1)"
function_ltrim=" ltrim(1,1)"
function_max=" max(1,1)"
function_min=" min(1,1)"
function_nullif=" nullif(1,1)"
function_printf=" printf(1,1)"
function_quote=" quote(1)"
function_random=" random()"
function_randomblob=" randomblob(1)"
function_replace=" replace(1,1,1)"
function_round=" round(1,1)"
function_rtrim=" rtrim(1,1)"
function_soundex=" soundex(1)"
function_sqlite_compileoption_get=" sqlite_compileoption_get(1)"
function_sqlite_compileoption_used=" sqlite_compileoption_used(1)"
function_sqlite_source_id=" sqlite_source_id()"
function_sqlite_version=" sqlite_version()"
function_strftime=" strftime(1,1,1,1)"
function_substr=" substr(1,1,1)"
function_sum=" sum(1)"
function_time=" time(1,1,1)"
function_total=" total(1)"
function_total_changes=" total_changes()"
function_trim=" trim(1,1)"
function_typeof=" typeof(1)"
function_unicode=" unicode(1)"
function_unlikely=" unlikely(1)"
function_upper=" upper(1)"
function_varchar=" varchar(1)"
function_zeroblob=" zeroblob(1)"

keyword_ABORT="ABORT"
keyword_ACTION="ACTION"
keyword_ADD="ADD"
keyword_AFTER="AFTER"
keyword_ALL="ALL"
keyword_ALTER="ALTER"
keyword_ANALYZE="ANALYZE"
keyword_AND="AND"
keyword_AS="AS"
keyword_ASC="ASC"
keyword_ATTACH="ATTACH"
keyword_AUTOINCREMENT="AUTOINCREMENT"
keyword_BEFORE="BEFORE"
keyword_BEGIN="BEGIN"
keyword_BETWEEN="BETWEEN"
keyword_BY="BY"
keyword_CASCADE="CASCADE"
keyword_CASE="CASE"
keyword_CAST="CAST"
keyword_CHECK="CHECK"
keyword_COLLATE="COLLATE"
keyword_COLUMN="COLUMN"
keyword_COMMIT="COMMIT"
keyword_CONFLICT="CONFLICT"
keyword_CONSTRAINT="CONSTRAINT"
keyword_CREATE="CREATE"
```

```
keyword_CROSS="CROSS"
keyword_CURRENT_DATE="CURRENT_DATE"
keyword_CURRENT_TIME="CURRENT_TIME"
keyword_CURRENT_TIMESTAMP="CURRENT_TIMESTAMP"
keyword_DATABASE="DATABASE"
keyword_DEFAULT="DEFAULT"
keyword_DEFERRABLE="DEFERRABLE"
keyword_DEFERRED="DEFERRED"
keyword_DELETE="DELETE"
keyword_DESC="DESC"
keyword_DETACH="DETACH"
keyword_DISTINCT="DISTINCT"
keyword_DROP="DROP"
keyword_EACH="EACH"
keyword_ELSE="ELSE"
keyword_END="END"
keyword_ESCAPE="ESCAPE"
keyword_EXCEPT="EXCEPT"
keyword_EXCLUSIVE="EXCLUSIVE"
keyword_EXISTS="EXISTS"
keyword_EXPLAIN="EXPLAIN"
keyword_FAIL="FAIL"
keyword_FOR="FOR"
keyword_FOREIGN="FOREIGN"
keyword_FROM="FROM"
keyword_FULL="FULL"
keyword_GLOB="GLOB"
keyword_GROUP="GROUP"
keyword_HAVING="HAVING"
keyword_IF="IF"
keyword_IGNORE="IGNORE"
keyword_IMMEDIATE="IMMEDIATE"
keyword_IN="IN"
keyword_INDEX="INDEX"
keyword_INDEXED="INDEXED"
keyword_INITIALLY="INITIALLY"
keyword_INNER="INNER"
keyword_INSERT="INSERT"
keyword_INSTEAD="INSTEAD"
keyword_INTERSECT="INTERSECT"
keyword_INTO="INTO"
keyword_IS="IS"
keyword_ISNULL="ISNULL"
keyword_JOIN="JOIN"
keyword_KEY="KEY"
keyword_LEFT="LEFT"
keyword_LIKE="LIKE"
keyword_LIMIT="LIMIT"
keyword_MATCH="MATCH"
keyword_NATURAL="NATURAL"
keyword_NO="NO"
keyword_NOT="NOT"
keyword_NOTNULL="NOTNULL"
keyword_NULL="NULL"
keyword_OF="OF"
keyword_OFFSET="OFFSET"
keyword_ON="ON"
keyword_OR="OR"
keyword_ORDER="ORDER"
keyword_OUTER="OUTER"
```

```
keyword_PLAN="PLAN"
keyword_PRAGMA="PRAGMA"
keyword_PRIMARY="PRIMARY"
keyword_QUERY="QUERY"
keyword_RAISE="RAISE"
keyword_RECURSIVE="RECURSIVE"
keyword_REFERENCES="REFERENCES"
keyword_REGEXP="REGEXP"
keyword_REINDEX="REINDEX"
keyword_RELEASE="RELEASE"
keyword_RENAME="RENAME"
keyword_REPLACE="REPLACE"
keyword_RESTRICT="RESTRICT"
keyword_RIGHT="RIGHT"
keyword_ROLLBACK="ROLLBACK"
keyword_ROW="ROW"
keyword_SAVEPOINT="SAVEPOINT"
keyword_SELECT="SELECT"
keyword_SET="SET"
keyword_TABLE="TABLE"
keyword_TEMP="TEMP"
keyword_TEMPORARY="TEMPORARY"
keyword_THEN="THEN"
keyword_TO="TO"
keyword_TRANSACTION="TRANSACTION"
keyword_TRIGGER="TRIGGER"
keyword_UNION="UNION"
keyword_UNIQUE="UNIQUE"
keyword_UPDATE="UPDATE"
keyword_USING="USING"
keyword_VACUUM="VACUUM"
keyword_VALUES="VALUES"
keyword_VIEW="VIEW"
keyword_VIRTUAL="VIRTUAL"
keyword_WHEN="WHEN"
keyword_WHERE="WHERE"
keyword_WITH="WITH"
keyword_WITHOUT="WITHOUT"

operator_concat=" || "
operator_ebove_eq=" >="

snippet_1eq1=" 1=1"
snippet_at=" @1"
snippet_backticks=" `a`"
snippet_blob=" blob"
snippet_brackets=" [a]"
snippet_colon=" :1"
snippet_comment=" /* */"
snippet_date="2001-01-01"
snippet_dollar=" $1"
snippet_dotref=" a.b"
snippet_fmtY="%Y"
snippet_int=" int"
snippet_neg1=" -1"
snippet_pair=" a,b"
snippet_parentheses=" (1)"
snippet_plus2days="+2 days"
snippet_qmark=" ?1"
snippet_semicolon=" ;"
```

```
snippet_star=" *"
snippet_string_pair=" \"a\",\"b\""

string_dbl_q=" \"a\""
string_escaped_q=" 'a''b'"
string_single_q=" 'a'"

pragma_application_id@1=" application_id"
pragma_auto_vacuum@1=" auto_vacuum"
pragma_automatic_index@1=" automatic_index"
pragma_busy_timeout@1=" busy_timeout"
pragma_cache_size@1=" cache_size"
pragma_cache_spill@1=" cache_spill"
pragma_case_sensitive_like@1=" case_sensitive_like"
pragma_checkpoint_fullfsync@1=" checkpoint_fullfsync"
pragma_collation_list@1=" collation_list"
pragma_compile_options@1=" compile_options"
pragma_count_changes@1=" count_changes"
pragma_data_store_directory@1=" data_store_directory"
pragma_database_list@1=" database_list"
pragma_default_cache_size@1=" default_cache_size"
pragma_defer_foreign_keys@1=" defer_foreign_keys"
pragma_empty_result_callbacks@1=" empty_result_callbacks"
pragma_encoding@1=" encoding"
pragma_foreign_key_check@1=" foreign_key_check"
pragma_foreign_key_list@1=" foreign_key_list"
pragma_foreign_keys@1=" foreign_keys"
pragma_freelist_count@1=" freelist_count"
pragma_full_column_names@1=" full_column_names"
pragma_fullfsync@1=" fullfsync"
pragma_ignore_check_constraints@1=" ignore_check_constraints"
pragma_incremental_vacuum@1=" incremental_vacuum"
pragma_index_info@1=" index_info"
pragma_index_list@1=" index_list"
pragma_integrity_check@1=" integrity_check"
pragma_journal_mode@1=" journal_mode"
pragma_journal_size_limit@1=" journal_size_limit"
pragma_legacy_file_format@1=" legacy_file_format"
pragma_locking_mode@1=" locking_mode"
pragma_max_page_count@1=" max_page_count"
pragma_mmap_size@1=" mmap_size"
pragma_page_count@1=" page_count"
pragma_page_size@1=" page_size"
pragma_parser_trace@1=" parser_trace"
pragma_query_only@1=" query_only"
pragma_quick_check@1=" quick_check"
pragma_read_uncommitted@1=" read_uncommitted"
pragma_recursive_triggers@1=" recursive_triggers"
pragma_reverse_unordered_selects@1=" reverse_unordered_selects"
pragma_schema_version@1=" schema_version"
pragma_secure_delete@1=" secure_delete"
pragma_short_column_names@1=" short_column_names"
pragma_shrink_memory@1=" shrink_memory"
pragma_soft_heap_limit@1=" soft_heap_limit"
pragma_stats@1=" stats"
pragma_synchronous@1=" synchronous"
pragma_table_info@1=" table_info"
pragma_temp_store@1=" temp_store"
pragma_temp_store_directory@1=" temp_store_directory"
pragma_threads@1=" threads"
```

```
pragma_user_version@1=" user_version"
pragma_vdbe_addoptrace@1=" vdbe_addoptrace"
pragma_vdbe_debug@1=" vdbe_debug"
pragma_vdbe_listing@1=" vdbe_listing"
pragma_vdbe_trace@1=" vdbe_trace"
pragma_wal_autocheckpoint@1=" wal_autocheckpoint"
pragma_wal_checkpoint@1=" wal_checkpoint"
pragma_writable_schema@1=" writable_schema"
```

tiff.dict

```
#
# AFL dictionary for TIFF images
# ------------------------------
#
# Just the basic, standard-originating sections; does not include vendor
# extensions.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

header_ii="II*\x00"
header_mm="MM\x00*"

section_100="\x00\x01"
section_101="\x01\x01"
section_102="\x02\x01"
section_103="\x03\x01"
section_106="\x06\x01"
section_107="\x07\x01"
section_10D="\x0d\x01"
section_10E="\x0e\x01"
section_10F="\x0f\x01"
section_110="\x10\x01"
section_111="\x11\x01"
section_112="\x12\x01"
section_115="\x15\x01"
section_116="\x16\x01"
section_117="\x17\x01"
section_11A="\x1a\x01"
section_11B="\x1b\x01"
section_11C="\x1c\x01"
section_11D="\x1d\x01"
section_11E="\x1e\x01"
section_11F="\x1f\x01"
section_122="\"\x01"
section_123="#\x01"
section_124="$\x01"
section_125="%\x01"
section_128="(\x01"
section_129=")\x01"
section_12D="-\x01"
section_131="1\x01"
section_132="2\x01"
section_13B=";\x01"
section_13C="<\x01"
section_13D="=\x01"
section_13E=">\x01"
```

```
section_13F="?\x01"
section_140="@\x01"
section_FE="\xfe\x00"
section_FF="\xff\x00"
```

webp.dict

```
#
# AFL dictionary for WebP images
# ------------------------------
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

header_RIFF="RIFF"
header_WEBP="WEBP"

section_ALPH="ALPH"
section_ANIM="ANIM"
section_ANMF="ANMF"
section_EXIF="EXIF"
section_FRGM="FRGM"
section_ICCP="ICCP"
section_VP8="VP8 "
section_VP8L="VP8L"
section_VP8X="VP8X"
section_XMP="XMP "
```

xml.dict

```
#
# AFL dictionary for XML
# ----------------------
#
# Several basic syntax elements and attributes, modeled on libxml2.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

attr_encoding=" encoding=\"1\""
attr_generic=" a=\"1\""
attr_href=" href=\"1\""
attr_standalone=" standalone=\"no\""
attr_version=" version=\"1\""
attr_xml_base=" xml:base=\"1\""
attr_xml_id=" xml:id=\"1\""
attr_xml_lang=" xml:lang=\"1\""
attr_xml_space=" xml:space=\"1\""
attr_xmlns=" xmlns=\"1\""

entity_builtin="&lt;"
entity_decimal="&#1;"
entity_external="&a;"
entity_hex="&#x1;"

string_any="ANY"
```

```
string_brackets="[]"
string_cdata="CDATA"
string_col_fallback=":fallback"
string_col_generic=":a"
string_col_include=":include"
string_dashes="--"
string_empty="EMPTY"
string_empty_dblquotes="\"\""
string_empty_quotes="''"
string_entities="ENTITIES"
string_entity="ENTITY"
string_fixed="#FIXED"
string_id="ID"
string_idref="IDREF"
string_idrefs="IDREFS"
string_implied="#IMPLIED"
string_nmtoken="NMTOKEN"
string_nmtokens="NMTOKENS"
string_notation="NOTATION"
string_parentheses="()"
string_pcdata="#PCDATA"
string_percent="%a"
string_public="PUBLIC"
string_required="#REQUIRED"
string_schema=":schema"
string_system="SYSTEM"
string_ucs4="UCS-4"
string_utf16="UTF-16"
string_utf8="UTF-8"
string_xmlns="xmlns:"

tag_attlist="<!ATTLIST"
tag_cdata="<![CDATA["
tag_close="</a>"
tag_doctype="<!DOCTYPE"
tag_element="<!ELEMENT"
tag_entity="<!ENTITY"
tag_ignore="<![IGNORE["
tag_include="<![INCLUDE["
tag_notation="<!NOTATION"
tag_open="<a>"
tag_open_close="<a />"
tag_open_exclamation="<!"
tag_open_q="<?"
tag_sq2_close="]]>"
tag_xml_q="<?xml?>"
```

> experimental

> argv_fuzzing

argv-fuzz-inl.h

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
```

```
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/


/*
   american fuzzy lop - sample argv fuzzing wrapper
   ------------------------------------------------

   Written by Michal Zalewski <lcamtuf@google.com>

   This file shows a simple way to fuzz command-line parameters with stock
   afl-fuzz. To use, add:

   #include "/path/to/argv-fuzz-inl.h"

   ...to the file containing main(), ideally placing it after all the
   standard includes. Next, put AFL_INIT_ARGV(); near the very beginning of
   main().

   This will cause the program to read NUL-delimited input from stdin and
   put it in argv[]. Two subsequent NULs terminate the array.

   If you would like to always preserve argv[0], use this instead:
   AFL_INIT_SET0("prog_name");
*/

#ifndef _HAVE_ARGV_FUZZ_INL
#define _HAVE_ARGV_FUZZ_INL

#include <unistd.h>
#include <ctype.h>

#define AFL_INIT_ARGV() do { argv = afl_init_argv(&argc); } while (0)

#define AFL_INIT_SET0(_p) do { \
    argv = afl_init_argv(&argc); \
    argv[0] = (_p); \
    if (!argc) argc = 1; \
  } while (0)

#define MAX_CMDLINE_LEN 100000
#define MAX_CMDLINE_PAR 1000

static char** afl_init_argv(int* argc) {

  static char  in_buf[MAX_CMDLINE_LEN];
  static char* ret[MAX_CMDLINE_PAR];

  char* ptr = in_buf;
  int   rc  = 1; /* start after argv[0] */
```

```
  if (read(0, in_buf, MAX_CMDLINE_LEN - 2) < 0);

  while (*ptr) {

    ret[rc] = ptr;

    /* insert '\0' at the end of ret[rc] on first space-sym */
    while (*ptr && !isspace(*ptr)) ptr++;
    *ptr = '\0';
    ptr++;

    /* skip more space-syms */
    while (*ptr && isspace(*ptr)) ptr++;

    rc++;
  }

  *argc = rc;

  return ret;

}

#undef MAX_CMDLINE_LEN
#undef MAX_CMDLINE_PAR

#endif /* !_HAVE_ARGV_FUZZ_INL */
```

test.c

```
#include <stdio.h>
#include "argv-fuzz-inl.h"

int main(int argc, char *argv[]) {
    AFL_INIT_SET0("a.out");

    int i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d] = '%s'\n", i, argv[i]);
    return 0;
}
```

> asan_cgroups

limit_memory.sh

```
#!/usr/bin/env bash
#
# Copyright 2015 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
```

```
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ------------------------------------------------
# american fuzzy lop - limit memory using cgroups
# ------------------------------------------------
#
# Written by Samir Khakimov <samir.hakim@nyu.edu> and
#            David A. Wheeler <dwheeler@ida.org>
#
# Edits to bring the script in line with afl-cmin and other companion scripts
# by Michal Zalewski <lcamtuf@google.com>. All bugs are my fault.
#
# This tool allows the amount of actual memory allocated to a program
# to be limited on Linux systems using cgroups, instead of the traditional
# setrlimit() API. This helps avoid the address space problems discussed in
# docs/notes_for_asan.txt.
#
# Important: the limit covers *both* afl-fuzz and the fuzzed binary. In some
# hopefully rare circumstances, afl-fuzz could be killed before the fuzzed
# task.
#

echo "cgroup tool for afl-fuzz by <samir.hakim@nyu.edu> and <dwheeler@ida.org>"
echo

unset NEW_USER
MEM_LIMIT="50"

while getopts "+u:m:" opt; do

  case "$opt" in

    "u")
         NEW_USER="$OPTARG"
         ;;

    "m")
         MEM_LIMIT="$[OPTARG]"
         ;;

    "?")
         exit 1
         ;;

   esac

done

if [ "$MEM_LIMIT" -lt "5" ]; then
  echo "[-] Error: malformed or dangerously low value of -m." 1>&2
  exit 1
fi

shift $((OPTIND-1))

TARGET_BIN="$1"
```

```
if [ "$TARGET_BIN" = "" -o "$NEW_USER" = "" ]; then

  cat 1>&2 <<_EOF_
Usage: $0 [ options ] -- /path/to/afl-fuzz [ ...afl options... ]

Required parameters:

  -u user   - run the fuzzer as a specific user after setting up limits

Optional parameters:

  -m megs   - set memory limit to a specified value ($MEM_LIMIT MB)

This tool configures cgroups-based memory limits for a fuzzing job to simplify
the task of fuzzing ASAN or MSAN binaries. You would normally want to use it in
conjunction with '-m none' passed to the afl-fuzz binary itself, say:

  $0 -u joe ./afl-fuzz -i input -o output -m none /path/to/target

_EOF_

  exit 1

fi

# Basic sanity checks

if [ ! "`uname -s`" = "Linux" ]; then
 echo "[-] Error: this tool does not support non-Linux systems." 1>&2
 exit 1
fi

if [ ! "`id -u`" = "0" ]; then
 echo "[-] Error: you need to run this script as root (sorry!)." 1>&2
 exit 1
fi

if ! type cgcreate 2>/dev/null 1>&2; then

  echo "[-] Error: you need to install cgroup tools first." 1>&2

  if type apt-get 2>/dev/null 1>&2; then
    echo "    (Perhaps 'apt-get install cgroup-bin' will work.)" 1>&2
  elif type yum 2>/dev/null 1>&2; then
    echo "    (Perhaps 'yum install libcgroup-tools' will work.)" 1>&2
  fi

  exit 1

fi

if ! id -u "$NEW_USER" 2>/dev/null 1>&2; then
  echo "[-] Error: user '$NEW_USER' does not seem to exist." 1>&2
  exit 1
fi

# Create a new cgroup path if necessary... We used PID-keyed groups to keep
# parallel afl-fuzz tasks separate from each other.
```

```
CID="afl-$NEW_USER-$$"

CPATH="/sys/fs/cgroup/memory/$CID"

if [ ! -d "$CPATH" ]; then

  cgcreate -a "$NEW_USER" -g memory:"$CID" || exit 1

fi

# Set the appropriate limit...

if [ -f "$CPATH/memory.memsw.limit_in_bytes" ]; then

  echo "${MEM_LIMIT}M" > "$CPATH/memory.limit_in_bytes" 2>/dev/null
  echo "${MEM_LIMIT}M" > "$CPATH/memory.memsw.limit_in_bytes" || exit 1
  echo "${MEM_LIMIT}M" > "$CPATH/memory.limit_in_bytes" || exit 1

elif grep -qE 'partition|file' /proc/swaps; then

  echo "[-] Error: your system requires swap to be disabled first (swapoff -a)." 1>&2
  exit 1

else

  echo "${MEM_LIMIT}M" > "$CPATH/memory.limit_in_bytes" || exit 1

fi

# All right. At this point, we can just run the command.

cgexec -g "memory:$CID" su -c "$*" "$NEW_USER"

cgdelete -g "memory:$CID"
```

> bash_shellshock

shellshock-fuzz.diff

```
This patch shows a very simple way to find post-Shellshock bugs in bash, as
discussed here:

  http://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html

In essence, it shows a way to fuzz environmental variables. Instructions:

1) Download bash 4.3, apply this patch, compile with:

   CC=/path/to/afl-gcc ./configure
   make clean all

   Note that the harness puts the fuzzed output in $TEST_VARIABLE. With
   Florian's Shellshock patch (bash43-028), this is no longer passed down
   to the parser.

2) Create and cd to an empty directory, put the compiled bash binary in
   there, and run these commands:
```

```
   mkdir in_dir
   echo -n '() { a() { a; }; : >b; }' >in_dir/script.txt

3) Run the fuzzer with:

   /path/to/afl-fuzz -d -i in_dir -o out_dir ./bash -c :

   The -d parameter is advisable only if the tested shell is fairly slow
   or if you are in a hurry; will cover more ground faster, but
   less systematically.

4) Watch for crashes in out_dir/crashes/. Also watch for any new files
   created in cwd if you're interested in non-crash RCEs (files will be
   created whenever the shell executes "foo>bar" or something like
   that). You can correlate their creation date with new entries in
   out_dir/queue/.

   You can also modify the bash binary to directly check for more subtle
   fault conditions, or use the synthesized entries in out_dir/queue/
   as a seed for other, possibly slower or more involved testing regimes.

   Expect several hours to get decent coverage.

--- bash-4.3/shell.c.orig   2014-01-14 14:04:32.000000000 +0100
+++ bash-4.3/shell.c    2015-04-30 05:56:46.000000000 +0200
@@ -371,6 +371,14 @@
   env = environ;
 #endif /* __OPENNT */

+  {
+
+    static char val[1024 * 16];
+    read(0, val, sizeof(val) - 1);
+    setenv("TEST_VARIABLE", val, 1);
+
+  }
+
   USE_VAR(argc);
   USE_VAR(argv);
   USE_VAR(env);
```

> canvas_harness

canvas_harness.html

```
<html>
<!--

  american fuzzy lop - <canvas> harness
  -------------------------------------

  Written and maintained by Michal Zalewski <lcamtuf@google.com>

  Copyright 2013, 2014 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:
```

```
    http://www.apache.org/licenses/LICENSE-2.0

  A simple harness for going through afl-generated test cases, rendering them in
  the browser environment, and discovering the use of uninitialized memory and
  similar bugs. This code led to the discovery of a fair number of library and
  browser security bugs!

  The url_list[] array is a placeholder; for this to work properly, it needs to
  be initialized with web-reachable paths to individual test cases. This can
  be done manually or with a simple script.

-->

<body onload="set_images()">

<div id="status"></div>

<div id="image_div"></div>

<canvas height=64 width=64 id=cvs></canvas>

<h2>Results</h2>

<ul id="output"></ul>

<script>

var c = document.getElementById('cvs');
var ctx = c.getContext('2d');

var url_list = [
  "images/id:000000,[...].jpg",
  "images/id:000001,[...].jpg",
  /* ... */
  null
];

var USE_IMAGES = 50;
var cur_image = 0;

if (location.hash) cur_image = parseInt(location.hash.substr(1));

var loaded = 0;
var image_obj = [];

var msie_cleanup;

function check_results() {

  var uniques = [];

  clearTimeout(msie_cleanup);

  ctx.clearRect(0, 0, 64, 64);

  uniques.push(image_obj[0].imgdata);

  for (var i = 1; i < USE_IMAGES; i++) {
```

```
    if (!image_obj[i].imgdata) continue;

    if (image_obj[0].imgdata != image_obj[i].imgdata) {

      for (var j = 1; j < uniques.length; j++)
        if (uniques[j] == image_obj[i].imgdata) break;

      if (j == uniques.length) uniques.push(image_obj[i].imgdata);


    }

  }

  if (uniques.length > 1) {

    var str = '<li> Image ' + url_list[cur_image] + ' has ' + uniques.length + ' variants: ';

    for (var i = 0; i < uniques.length; i++)
      str += '<img src="' + uniques[i] + '">';

    document.getElementById('output').innerHTML += str;

  }

  cur_image++;
  set_images();
}


function count_image() {

  if (!this.complete || this.counted) return;

  this.counted = true;

  loaded++;

  ctx.clearRect(0, 0, 64, 64);

  try {
    ctx.drawImage(this, 0, 0, 64, 64);
  } catch (e) { }

  this.imgdata = c.toDataURL();

  if (loaded == USE_IMAGES) check_results();
}


function set_images() {

  loaded = 0;

  document.getElementById('status').innerHTML = 'Now processing ' + cur_image + '...';
  location.hash = '#' + cur_image;

  if (url_list[cur_image] == null) {
    alert('Done!');
    return;
```

```
  }

  restart_images();

  msie_cleanup = setTimeout(check_results, 5000);

  for (var i = 0; i < USE_IMAGES; i++)
    image_obj[i].src = url_list[cur_image] + '?' + Math.random();

}


function restart_images() {

  for (var i = 0; i < USE_IMAGES; i++)
    if (image_obj[i]) image_obj[i].counted = true;

  document.getElementById('image_div').innerHTML = '';
  image_obj = [];

  for (var i = 0; i < USE_IMAGES; i++) {

    image_obj[i] = new Image();
    image_obj[i].height = 64;
    image_obj[i].width = 64;
    image_obj[i].onerror = count_image;
    image_obj[i].onload = count_image;

    document.getElementById('image_div').appendChild(image_obj[i]);

  }

}

</script>

<iframe src='http://www.cnn.com/'></iframe>
```

> clang_asm_normalize

as

```
#!/bin/sh
#
# american fuzzy lop - clang assembly normalizer
# ----------------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
# The idea for this wrapper comes from Ryan Govostes.
#
# Copyright 2013, 2014 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
```

```
# This 'as' wrapper should allow you to instrument unruly, hand-written
# assembly with afl-as.
#
# Usage:
#
# export AFL_REAL_PATH=/path/to/directory/with/afl-as/
# AFL_PATH=/path/to/this/directory/ make clean all

if [ "$#" -lt "2" ]; then
  echo "[-] Error: this utility can't be called directly." 1>&2
  exit 1
fi

if [ "$AFL_REAL_PATH" = "" ]; then
  echo "[-] Error: AFL_REAL_PATH not set!" 1>&2
  exit 1
fi

if [ ! -x "$AFL_REAL_PATH/afl-as" ]; then
  echo "[-] Error: AFL_REAL_PATH does not contain the 'afl-as' binary." 1>&2
  exit 1
fi

unset __AFL_AS_CMDLINE __AFL_FNAME

while [ ! "$#" = "0" ]; do

  if [ "$#" = "1" ]; then
    __AFL_FNAME="$1"
  else
    __AFL_AS_CMDLINE="${__AFL_AS_CMDLINE} $1"
  fi

  shift

done

test "$TMPDIR" = "" && TMPDIR=/tmp

TMPFILE=`mktemp $TMPDIR/.afl-XXXXXXXXXX.s`

test "$TMPFILE" = "" && exit 1

clang -cc1as -filetype asm -output-asm-variant 0 "${__AFL_FNAME}" >"$TMPFILE"

ERR="$?"

if [ ! "$ERR" = "0" ]; then
  rm -f "$TMPFILE"
  exit $ERR
fi

"$AFL_REAL_PATH/afl-as" ${__AFL_AS_CMDLINE} "$TMPFILE"

ERR="$?"

rm -f "$TMPFILE"

exit "$ERR"
```

> crash_triage

triage_crashes.sh

```
#!/bin/sh
#
# Copyright 2013 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# american fuzzy lop - crash triage utility
# -----------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
#
# Note that this assumes that the targeted application reads from stdin
# and requires no other cmdline parameters. Modify as needed if this is
# not the case.
#
# Note that on OpenBSD, you may need to install a newer version of gdb
# (e.g., from ports). You can set GDB=/some/path to point to it if
# necessary.
#

echo "crash triage utility for afl-fuzz by <lcamtuf@google.com>"
echo

ulimit -v 100000 2>/dev/null
ulimit -d 100000 2>/dev/null

if [ "$#" -lt "2" ]; then
  echo "Usage: $0 /path/to/afl_output_dir /path/to/tested_binary [...target params...]" 1>&2
  echo 1>&2
  exit 1
fi

DIR="$1"
BIN="$2"
shift
shift

if [ "$AFL_ALLOW_TMP" = "" ]; then

  echo "$DIR" | grep -qE '^(/var)?/tmp/'
  T1="$?"

  echo "$BIN" | grep -qE '^(/var)?/tmp/'
  T2="$?"
```

```
  if [ "$T1" = "0" -o "$T2" = "0" ]; then
    echo "[-] Error: do not use shared /tmp or /var/tmp directories with this script." 1>&2
    exit 1
  fi

fi

if
 [ "$GDB" = "" ]; then
  GDB=gdb
fi

if [ ! -f "$BIN" -o ! -x "$BIN" ]; then
  echo "[-] Error: binary '$2' not found or is not executable." 1>&2
  exit 1
fi

if [ ! -d "$DIR/queue" ]; then
  echo "[-] Error: directory '$1' not found or not created by afl-fuzz." 1>&2
  exit 1
fi

CCOUNT=$((`ls -- "$DIR/crashes" 2>/dev/null | wc -l`))

if [ "$CCOUNT" = "0" ]; then
  echo "No crashes recorded in the target directory - nothing to be done."
  exit 0
fi

echo

for crash in $DIR/crashes/id:*; do

  id=`basename -- "$crash" | cut -d, -f1 | cut -d: -f2`
  sig=`basename -- "$crash" | cut -d, -f2 | cut -d: -f2`

  # Grab the args, converting @@ to $crash

  use_args=""
  use_stdio=1

  for a in $@; do

    if [ "$a" = "@@" ] ; then
      use_args="$use_args $crash"
      unset use_stdio
    else
      use_args="$use_args $a"
    fi

  done

  # Strip the trailing space
  use_args="${use_args# }"

  echo "+++ ID $id, SIGNAL $sig +++"
  echo

  if [ "$use_stdio" = "1" ]; then
```

```
    $GDB --batch -q --ex "r $use_args <$crash" --ex 'back' --ex 'disass $pc, $pc+16' --ex
'info reg' --ex 'quit' "$BIN" 0</dev/null
  else
    $GDB --batch -q --ex "r $use_args" --ex 'back' --ex 'disass $pc, $pc+16' --ex 'info reg' -
-ex 'quit' "$BIN" 0</dev/null
  fi
  echo

done
```

> distributed_fuzzing

sync_script.sh

```
#!/bin/sh
#
# Copyright 2014 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ----------------------------------------
# american fuzzy lop - fuzzer synchronization tool
# -----------------------------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
#
# To make this script work:
#
# - Edit FUZZ_HOSTS, FUZZ_DOMAIN, FUZZ_USER, and SYNC_DIR to reflect your
#   environment.
#
# - Make sure that the system you are running this on can log into FUZZ_HOSTS
#   without a password (authorized_keys or otherwise).
#
# - Make sure that every fuzzer is running with -o pointing to SYNC_DIR and -S
#   that consists of its local host name, followed by an underscore, and then
#   by some host-local fuzzer ID.
#

# Hosts to synchronize the data across.
FUZZ_HOSTS='host1 host2 host3 host4'

# Domain for all hosts
FUZZ_DOMAIN='example.com'

# Remote user for SSH
FUZZ_USER=bob

# Directory to synchronize
```

```
SYNC_DIR='/home/bob/sync_dir'

# Interval (seconds) between sync attempts
SYNC_INTERVAL=$((30 * 60))

if [ "$AFL_ALLOW_TMP" = "" ]; then

  if [ "$PWD" = "/tmp" -o "$PWD" = "/var/tmp" ]; then
    echo "[-] Error: do not use shared /tmp or /var/tmp directories with this script." 1>&2
    exit 1
  fi

fi

rm -rf .sync_tmp 2>/dev/null
mkdir .sync_tmp || exit 1

while :; do

  # Pull data in...

  for host in $FUZZ_HOSTS; do

    echo "[*] Retrieving data from ${host}.${FUZZ_DOMAIN}..."

    ssh -o 'passwordauthentication no' ${FUZZ_USER}@${host}.$FUZZ_DOMAIN \
      "cd '$SYNC_DIR' && tar -czf - ${host}_*/[qf]*" >".sync_tmp/${host}.tgz"

  done

  # Distribute data. For large fleets, see tips in the docs/ directory.

  for dst_host in $FUZZ_HOSTS; do

    echo "[*] Distributing data to ${dst_host}.${FUZZ_DOMAIN}..."

    for src_host in $FUZZ_HOSTS; do

      test "$src_host" = "$dst_host" && continue

      echo "    Sending fuzzer data from ${src_host}.${FUZZ_DOMAIN}..."

      ssh -o 'passwordauthentication no' ${FUZZ_USER}@$dst_host \
        "cd '$SYNC_DIR' && tar -xkzf -" <".sync_tmp/${src_host}.tgz"

    done

  done

  echo "[+] Done. Sleeping for $SYNC_INTERVAL seconds (Ctrl-C to quit)."

  sleep $SYNC_INTERVAL

done
```

> libpng_no_checksum

libpng-nocrc.patch

```
--- pngrutil.c.orig 2014-06-12 03:35:16.000000000 +0200
+++ pngrutil.c  2014-07-01 05:08:31.000000000 +0200
@@ -268,7 +268,11 @@
    if (need_crc != 0)
    {
       crc = png_get_uint_32(crc_bytes);
-      return ((int)(crc != png_ptr->crc));
+
+      if (crc != png_ptr->crc)
+        fprintf(stderr, "NOTE: CRC in the file is 0x%08x, change to 0x%08x\n", crc, png_ptr-
>crc);
+
+      return ((int)(1 != 1));
    }

    else
```

> persistent_demo

persistent_demo.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - persistent mode example
   --------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This file demonstrates the high-performance "persistent mode" that may be
   suitable for fuzzing certain fast and well-behaved libraries, provided that
   they are stateless or that their internal state can be easily reset
   across runs.

   To make this work, the library and this shim need to be compiled in LLVM
   mode using afl-clang-fast (other compiler wrappers will *not* work).
*/

#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>


/* Main entry point. */

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

  char buf[100]; /* Example-only buffer, you'd replace it with other global or
                    local variables appropriate for your use case. */

  /* The number passed to __AFL_LOOP() controls the maximum number of
     iterations before the loop exits and the program is allowed to
     terminate normally. This limits the impact of accidental memory leaks
     and similar hiccups. */

  while (__AFL_LOOP(1000)) {

    /*** PLACEHOLDER CODE ***/

    /* STEP 1: Fully re-initialize all critical variables. In our example, this
               involves zeroing buf[], our input buffer. */

    memset(buf, 0, 100);

    /* STEP 2: Read input data. When reading from stdin, no special preparation
               is required. When reading from a named file, you need to close
               the old descriptor and reopen the file first!

               Beware of reading from buffered FILE* objects such as stdin. Use
               raw file descriptors or call fopen() / fdopen() in every pass. */

    read(0, buf, 100);

    /* STEP 3: This is where we'd call the tested library on the read data.
               We just have some trivial inline code that faults on 'foo!'. */

    if (buf[0] == 'f') {
      printf("one\n");
      if (buf[1] == 'o') {
        printf("two\n");
        if (buf[2] == 'o') {
          printf("three\n");
          if (buf[3] == '!') {
            printf("four\n");
            abort();
          }
        }
      }
    }

    /*** END PLACEHOLDER CODE ***/

  }

  /* Once the loop is exited, terminate normally - AFL will restart the process
     when this happens, with a clean slate when it comes to allocated memory,
     leftover file descriptors, etc. */
```

```
  return 0;

}
```

> post_library

post_library.so.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - postprocessor library example
   --------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   Postprocessor libraries can be passed to afl-fuzz to perform final cleanup
   of any mutated test cases - for example, to fix up checksums in PNG files.

   Please heed the following warnings:

   1) In almost all cases, it is more productive to comment out checksum logic
      in the targeted binary (as shown in ../libpng_no_checksum/). One possible
      exception is the process of fuzzing binary-only software in QEMU mode.

   2) The use of postprocessors for anything other than checksums is questionable
      and may cause more harm than good. AFL is normally pretty good about
      dealing with length fields, magic values, etc.

   3) Postprocessors that do anything non-trivial must be extremely robust to
      gracefully handle malformed data and other error conditions - otherwise,
      they will crash and take afl-fuzz down with them. Be wary of reading past
      *len and of integer overflows when calculating file offsets.

   In other words, THIS IS PROBABLY NOT WHAT YOU WANT - unless you really,
   honestly know what you're doing =)

   With that out of the way: the postprocessor library is passed to afl-fuzz
   via AFL_POST_LIBRARY. The library must be compiled with:

     gcc -shared -Wall -O3 post_library.so.c -o post_library.so

   AFL will call the afl_postprocess() function for every mutated output buffer.
```

```
   From there, you have three choices:

   1) If you don't want to modify the test case, simply return the original
      buffer pointer ('in_buf').

   2) If you want to skip this test case altogether and have AFL generate a
      new one, return NULL. Use this sparingly - it's faster than running
      the target program with patently useless inputs, but still wastes CPU
      time.

   3) If you want to modify the test case, allocate an appropriately-sized
      buffer, move the data into that buffer, make the necessary changes, and
      then return the new pointer. You can update *len if necessary, too.

      Note that the buffer will *not* be freed for you. To avoid memory leaks,
      you need to free it or reuse it on subsequent calls (as shown below).

      *** DO NOT MODIFY THE ORIGINAL 'in_buf' BUFFER. ***

    Aight. The example below shows a simple postprocessor that tries to make
    sure that all input files start with "GIF89a".

    PS. If you don't like C, you can try out the unix-based wrapper from
    Ben Nagy instead: https://github.com/bnagy/aflfix

 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Header that must be present at the beginning of every test case: */

#define HEADER "GIF89a"

/* The actual postprocessor routine called by afl-fuzz: */

const unsigned char* afl_postprocess(const unsigned char* in_buf,
                                     unsigned int* len) {

  static unsigned char* saved_buf;
  unsigned char* new_buf;

  /* Skip execution altogether for buffers shorter than 6 bytes (just to
     show how it's done). We can trust *len to be sane. */

  if (*len < strlen(HEADER)) return NULL;

  /* Do nothing for buffers that already start with the expected header. */

  if (!memcmp(in_buf, HEADER, strlen(HEADER))) return in_buf;

  /* Allocate memory for new buffer, reusing previous allocation if
     possible. */

  new_buf = realloc(saved_buf, *len);

  /* If we're out of memory, the most graceful thing to do is to return the
     original buffer and give up on modifying it. Let AFL handle OOM on its
     own later on. */
```

```
  if (!new_buf) return in_buf;
  saved_buf = new_buf;

  /* Copy the original data to the new location. */

  memcpy(new_buf, in_buf, *len);

  /* Insert the new header. */

  memcpy(new_buf, HEADER, strlen(HEADER));

  /* Return modified buffer. No need to update *len in this particular case,
     as we're not changing it. */

  return new_buf;

}
```

post_library_png.so.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - postprocessor for PNG
   ------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   See post_library.so.c for a general discussion of how to implement
   postprocessors. This specific postprocessor attempts to fix up PNG
   checksums, providing a slightly more complicated example than found
   in post_library.so.c.

   Compile with:

     gcc -shared -Wall -O3 post_library_png.so.c -o post_library_png.so -lz
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <zlib.h>
```

```
#include <arpa/inet.h>

/* A macro to round an integer up to 4 kB. */

#define UP4K(_i) ((((_i) >> 12) + 1) << 12)

const unsigned char* afl_postprocess(const unsigned char* in_buf,
                                     unsigned int* len) {

  static unsigned char* saved_buf;
  static unsigned int   saved_len;

  unsigned char* new_buf = (unsigned char*)in_buf;
  unsigned int pos = 8;

  /* Don't do anything if there's not enough room for the PNG header
     (8 bytes). */

  if (*len < 8) return in_buf;

  /* Minimum size of a zero-length PNG chunk is 12 bytes; if we
     don't have that, we can bail out. */

  while (pos + 12 <= *len) {

    unsigned int chunk_len, real_cksum, file_cksum;

    /* Chunk length is the first big-endian dword in the chunk. */

    chunk_len = ntohl(*(uint32_t*)(in_buf + pos));

    /* Bail out if chunk size is too big or goes past EOF. */

    if (chunk_len > 1024 * 1024 || pos + 12 + chunk_len > *len) break;

    /* Chunk checksum is calculated for chunk ID (dword) and the actual
       payload. */

    real_cksum = htonl(crc32(0, in_buf + pos + 4, chunk_len + 4));

    /* The in-file checksum is the last dword past the chunk data. */

    file_cksum = *(uint32_t*)(in_buf + pos + 8 + chunk_len);

    /* If the checksums do not match, we need to fix the file. */

    if (real_cksum != file_cksum) {

      /* First modification? Make a copy of the input buffer. Round size
         up to 4 kB to minimize the number of reallocs needed. */

      if (new_buf == in_buf) {

        if (*len <= saved_len) {

          new_buf = saved_buf;

        } else {
```

```
          new_buf = realloc(saved_buf, UP4K(*len));
          if (!new_buf) return in_buf;
          saved_buf = new_buf;
          saved_len = UP4K(*len);
          memcpy(new_buf, in_buf, *len);

        }

      }

      *(uint32_t*)(new_buf + pos + 8 + chunk_len) = real_cksum;

    }

    /* Skip the entire chunk and move to the next one. */

    pos += 12 + chunk_len;

  }

  return new_buf;

}
```

hash.h

```
/*
  Copyright 2016 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - hashing function
   -------------------------------------

   The hash32() function is a variant of MurmurHash3, a good
   non-cryptosafe hashing function developed by Austin Appleby.

   For simplicity, this variant does *NOT* accept buffer lengths
   that are not divisible by 8 bytes. The 32-bit version is otherwise
   similar to the original; the 64-bit one is a custom hack with
   mostly-unproven properties.

   Austin's original code is public domain.

   Other code written and maintained by Michal Zalewski <lcamtuf@google.com>
*/
```

```
#ifndef _HAVE_HASH_H
#define _HAVE_HASH_H

#include "types.h"

#ifdef __x86_64__

#define ROL64(_x, _r)  ((((u64)(_x)) << (_r)) | (((u64)(_x)) >> (64 - (_r))))

static inline u32 hash32(const void* key, u32 len, u32 seed) {

  const u64* data = (u64*)key;
  u64 h1 = seed ^ len;

  len >>= 3;

  while (len--) {

    u64 k1 = *data++;

    k1 *= 0x87c37b91114253d5ULL;
    k1  = ROL64(k1, 31);
    k1 *= 0x4cf5ad432745937fULL;

    h1 ^= k1;
    h1  = ROL64(h1, 27);
    h1  = h1 * 5 + 0x52dce729;

  }

  h1 ^= h1 >> 33;
  h1 *= 0xff51afd7ed558ccdULL;
  h1 ^= h1 >> 33;
  h1 *= 0xc4ceb9fe1a85ec53ULL;
  h1 ^= h1 >> 33;

  return h1;

}

#else

#define ROL32(_x, _r)  ((((u32)(_x)) << (_r)) | (((u32)(_x)) >> (32 - (_r))))

static inline u32 hash32(const void* key, u32 len, u32 seed) {

  const u32* data  = (u32*)key;
  u32 h1 = seed ^ len;

  len >>= 2;

  while (len--) {

    u32 k1 = *data++;

    k1 *= 0xcc9e2d51;
    k1  = ROL32(k1, 15);
    k1 *= 0x1b873593;
```

```
    h1 ^= k1;
    h1  = ROL32(h1, 13);
    h1  = h1 * 5 + 0xe6546b64;

  }

  h1 ^= h1 >> 16;
  h1 *= 0x85ebca6b;
  h1 ^= h1 >> 13;
  h1 *= 0xc2b2ae35;
  h1 ^= h1 >> 16;

  return h1;

}

#endif /* ^__x86_64__ */

#endif /* !_HAVE_HASH_H */
```

> libdislocator

libdislocator.so.c

```
/*
  Copyright 2016 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*

   american fuzzy lop - dislocator, an abusive allocator
   -----------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This is a companion library that can be used as a drop-in replacement
   for the libc allocator in the fuzzed binaries. See README.dislocator for
   more info.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <sys/mman.h>
```

```
#include "../config.h"
#include "../types.h"

#ifndef PAGE_SIZE
#  define PAGE_SIZE 4096
#endif /* !PAGE_SIZE */

#ifndef MAP_ANONYMOUS
#  define MAP_ANONYMOUS MAP_ANON
#endif /* !MAP_ANONYMOUS */

/* Error / message handling: */

#define DEBUGF(_x...) do { \
    if (alloc_verbose) { \
      if (++call_depth == 1) { \
        fprintf(stderr, "[AFL] " _x); \
        fprintf(stderr, "\n"); \
      } \
      call_depth--; \
    } \
  } while (0)

#define FATAL(_x...) do { \
    if (++call_depth == 1) { \
      fprintf(stderr, "*** [AFL] " _x); \
      fprintf(stderr, " ***\n"); \
      abort(); \
    } \
    call_depth--; \
  } while (0)

/* Macro to count the number of pages needed to store a buffer: */

#define PG_COUNT(_l) (((_l) + (PAGE_SIZE - 1)) / PAGE_SIZE)

/* Canary & clobber bytes: */

#define ALLOC_CANARY  0xAACCAACC
#define ALLOC_CLOBBER 0xCC

#define PTR_C(_p) (((u32*)(_p))[-1])
#define PTR_L(_p) (((u32*)(_p))[-2])

/* Configurable stuff (use AFL_LD_* to set): */

static u32 max_mem = MAX_ALLOC;         /* Max heap usage to permit         */
static u8  alloc_verbose,               /* Additional debug messages        */
           hard_fail,                   /* abort() when max_mem exceeded?   */
           no_calloc_over;              /* abort() on calloc() overflows?   */

static __thread size_t total_mem;       /* Currently allocated mem          */

static __thread u32 call_depth;         /* To avoid recursion via fprintf() */


/* This is the main alloc function. It allocates one page more than necessary,
   sets that tailing page to PROT_NONE, and then increments the return address
   so that it is right-aligned to that boundary. Since it always uses mmap(),
```

```
   the returned memory will be zeroed. */

static void* __dislocator_alloc(size_t len) {

  void* ret;


  if (total_mem + len > max_mem || total_mem + len < total_mem) {

    if (hard_fail)
      FATAL("total allocs exceed %u MB", max_mem / 1024 / 1024);

    DEBUGF("total allocs exceed %u MB, returning NULL",
           max_mem / 1024 / 1024);

    return NULL;

  }

  /* We will also store buffer length and a canary below the actual buffer, so
     let's add 8 bytes for that. */

  ret = mmap(NULL, (1 + PG_COUNT(len + 8)) * PAGE_SIZE, PROT_READ | PROT_WRITE,
             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

  if (ret == (void*)-1) {

    if (hard_fail) FATAL("mmap() failed on alloc (OOM?)");

    DEBUGF("mmap() failed on alloc (OOM?)");

    return NULL;

  }

  /* Set PROT_NONE on the last page. */

  if (mprotect(ret + PG_COUNT(len + 8) * PAGE_SIZE, PAGE_SIZE, PROT_NONE))
    FATAL("mprotect() failed when allocating memory");

  /* Offset the return pointer so that it's right-aligned to the page
     boundary. */

  ret += PAGE_SIZE * PG_COUNT(len + 8) - len - 8;

  /* Store allocation metadata. */

  ret += 8;

  PTR_L(ret) = len;
  PTR_C(ret) = ALLOC_CANARY;

  total_mem += len;

  return ret;

}


/* The "user-facing" wrapper for calloc(). This just checks for overflows and
```

```
   displays debug messages if requested. */

void* calloc(size_t elem_len, size_t elem_cnt) {

  void* ret;

  size_t len = elem_len * elem_cnt;

  /* Perform some sanity checks to detect obvious issues... */

  if (elem_cnt && len / elem_cnt != elem_len) {

    if (no_calloc_over) {
      DEBUGF("calloc(%zu, %zu) would overflow, returning NULL", elem_len, elem_cnt);
      return NULL;
    }

    FATAL("calloc(%zu, %zu) would overflow", elem_len, elem_cnt);

  }

  ret = __dislocator_alloc(len);

  DEBUGF("calloc(%zu, %zu) = %p [%zu total]", elem_len, elem_cnt, ret,
         total_mem);

  return ret;

}


/* The wrapper for malloc(). Roughly the same, also clobbers the returned
   memory (unlike calloc(), malloc() is not guaranteed to return zeroed
   memory). */

void* malloc(size_t len) {

  void* ret;

  ret = __dislocator_alloc(len);

  DEBUGF("malloc(%zu) = %p [%zu total]", len, ret, total_mem);

  if (ret && len) memset(ret, ALLOC_CLOBBER, len);

  return ret;

}


/* The wrapper for free(). This simply marks the entire region as PROT_NONE.
   If the region is already freed, the code will segfault during the attempt to
   read the canary. Not very graceful, but works, right? */

void free(void* ptr) {

  u32 len;

  DEBUGF("free(%p)", ptr);
```

```
  if (!ptr) return;

  if (PTR_C(ptr) != ALLOC_CANARY) FATAL("bad allocator canary on free()");

  len = PTR_L(ptr);

  total_mem -= len;

  /* Protect everything. Note that the extra page at the end is already
     set as PROT_NONE, so we don't need to touch that. */

  ptr -= PAGE_SIZE * PG_COUNT(len + 8) - len - 8;

  if (mprotect(ptr - 8, PG_COUNT(len + 8) * PAGE_SIZE, PROT_NONE))
    FATAL("mprotect() failed when freeing memory");

  /* Keep the mapping; this is wasteful, but prevents ptr reuse. */

}


/* Realloc is pretty straightforward, too. We forcibly reallocate the buffer,
   move data, and then free (aka mprotect()) the original one. */

void* realloc(void* ptr, size_t len) {

  void* ret;

  ret = malloc(len);

  if (ret && ptr) {

    if (PTR_C(ptr) != ALLOC_CANARY) FATAL("bad allocator canary on realloc()");

    memcpy(ret, ptr, MIN(len, PTR_L(ptr)));
    free(ptr);

  }

  DEBUGF("realloc(%p, %zu) = %p [%zu total]", ptr, len, ret, total_mem);

  return ret;

}


__attribute__((constructor)) void __dislocator_init(void) {

  u8* tmp = getenv("AFL_LD_LIMIT_MB");

  if (tmp) {

    max_mem = atoi(tmp) * 1024 * 1024;
    if (!max_mem) FATAL("Bad value for AFL_LD_LIMIT_MB");

  }

  alloc_verbose = !!getenv("AFL_LD_VERBOSE");
  hard_fail = !!getenv("AFL_LD_HARD_FAIL");
  no_calloc_over = !!getenv("AFL_LD_NO_CALLOC_OVER");
```

```
}
```

Makefile

```
#
# american fuzzy lop - libdislocator
# ----------------------------------
#
# Written by Michal Zalewski <lcamtuf@google.com>
#
# Copyright 2016 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#

PREFIX      ?= /usr/local
HELPER_PATH  = $(PREFIX)/lib/afl

VERSION     = $(shell grep '^\#define VERSION ' ../config.h | cut -d '"' -f2)

CFLAGS      ?= -O3 -funroll-loops
CFLAGS      += -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign

all: libdislocator.so

libdislocator.so: libdislocator.so.c ../config.h
    $(CC) $(CFLAGS) -shared -fPIC $< -o $@ $(LDFLAGS)

.NOTPARALLEL: clean

clean:
    rm -f *.o *.so *~ a.out core core.[1-9][0-9]*
    rm -f libdislocator.so

install: all
    install -m 755 libdislocator.so $${DESTDIR}$(HELPER_PATH)
    install -m 644 README.dislocator $${DESTDIR}$(HELPER_PATH)
```

> libtokencap

libtokencap.so.c

```
/*
  Copyright 2016 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:
```

```
   http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*

   american fuzzy lop - extract tokens passed to strcmp / memcmp
   -------------------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>

   This Linux-only companion library allows you to instrument strcmp(),
   memcmp(), and related functions to automatically extract tokens.
   See README.tokencap for more info.
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "../types.h"
#include "../config.h"

#ifndef __linux__
#  error "Sorry, this library is Linux-specific for now!"
#endif /* !__linux__ */


/* Mapping data and such */

#define MAX_MAPPINGS 1024

static struct mapping {
  void *st, *en;
} __tokencap_ro[MAX_MAPPINGS];

static u32   __tokencap_ro_cnt;
static u8    __tokencap_ro_loaded;
static FILE* __tokencap_out_file;


/* Identify read-only regions in memory. Only parameters that fall into these
   ranges are worth dumping when passed to strcmp() and so on. Read-write
   regions are far more likely to contain user input instead. */

static void __tokencap_load_mappings(void) {

  u8 buf[MAX_LINE];
  FILE* f = fopen("/proc/self/maps", "r");

  __tokencap_ro_loaded = 1;

  if (!f) return;

  while (fgets(buf, MAX_LINE, f)) {
```

```
    u8 rf, wf;
    void* st, *en;

    if (sscanf(buf, "%p-%p %c%c", &st, &en, &rf, &wf) != 4) continue;
    if (wf == 'w' || rf != 'r') continue;

    __tokencap_ro[__tokencap_ro_cnt].st = (void*)st;
    __tokencap_ro[__tokencap_ro_cnt].en = (void*)en;

    if (++__tokencap_ro_cnt == MAX_MAPPINGS) break;

  }

  fclose(f);

}


/* Check an address against the list of read-only mappings. */

static u8 __tokencap_is_ro(const void* ptr) {

  u32 i;

  if (!__tokencap_ro_loaded) __tokencap_load_mappings();

  for (i = 0; i < __tokencap_ro_cnt; i++)
    if (ptr >= __tokencap_ro[i].st && ptr <= __tokencap_ro[i].en) return 1;

  return 0;

}


/* Dump an interesting token to output file, quoting and escaping it
   properly. */

static void __tokencap_dump(const u8* ptr, size_t len, u8 is_text) {

  u8 buf[MAX_AUTO_EXTRA * 4 + 1];
  u32 i;
  u32 pos = 0;

  if (len < MIN_AUTO_EXTRA || len > MAX_AUTO_EXTRA || !__tokencap_out_file)
    return;

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

    if (is_text && !ptr[i]) break;

    switch (ptr[i]) {

      case 0 ... 31:
      case 127 ... 255:
      case '\"':
      case '\\':

        sprintf(buf + pos, "\\x%02x", ptr[i]);
        pos += 4;
```

```
        break;

      default:

        buf[pos++] = ptr[i];

    }

  }

  buf[pos] = 0;

  fprintf(__tokencap_out_file, "\"%s\"\n", buf);

}


/* Replacements for strcmp(), memcmp(), and so on. Note that these will be used
   only if the target is compiled with -fno-builtins and linked dynamically. */

#undef strcmp

int strcmp(const char* str1, const char* str2) {

  if (__tokencap_is_ro(str1)) __tokencap_dump(str1, strlen(str1), 1);
  if (__tokencap_is_ro(str2)) __tokencap_dump(str2, strlen(str2), 1);

  while (1) {

    unsigned char c1 = *str1, c2 = *str2;

    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    if (!c1) return 0;
    str1++; str2++;

  }

}


#undef strncmp

int strncmp(const char* str1, const char* str2, size_t len) {

  if (__tokencap_is_ro(str1)) __tokencap_dump(str1, len, 1);
  if (__tokencap_is_ro(str2)) __tokencap_dump(str2, len, 1);

  while (len--) {

    unsigned char c1 = *str1, c2 = *str2;

    if (!c1) return 0;
    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    str1++; str2++;

  }

  return 0;

}
```

```
#undef strcasecmp

int strcasecmp(const char* str1, const char* str2) {

  if (__tokencap_is_ro(str1)) __tokencap_dump(str1, strlen(str1), 1);
  if (__tokencap_is_ro(str2)) __tokencap_dump(str2, strlen(str2), 1);

  while (1) {

    unsigned char c1 = tolower(*str1), c2 = tolower(*str2);

    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    if (!c1) return 0;
    str1++; str2++;

  }

}


#undef strncasecmp

int strncasecmp(const char* str1, const char* str2, size_t len) {

  if (__tokencap_is_ro(str1)) __tokencap_dump(str1, len, 1);
  if (__tokencap_is_ro(str2)) __tokencap_dump(str2, len, 1);

  while (len--) {

    unsigned char c1 = tolower(*str1), c2 = tolower(*str2);

    if (!c1) return 0;
    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    str1++; str2++;

  }

  return 0;

}


#undef memcmp

int memcmp(const void* mem1, const void* mem2, size_t len) {

  if (__tokencap_is_ro(mem1)) __tokencap_dump(mem1, len, 0);
  if (__tokencap_is_ro(mem2)) __tokencap_dump(mem2, len, 0);

  while (len--) {

    unsigned char c1 = *(const char*)mem1, c2 = *(const char*)mem2;
    if (c1 != c2) return (c1 > c2) ? 1 : -1;
    mem1++; mem2++;

  }

  return 0;
```

```
}


#undef strstr

char* strstr(const char* haystack, const char* needle) {

  if (__tokencap_is_ro(haystack))
    __tokencap_dump(haystack, strlen(haystack), 1);

  if (__tokencap_is_ro(needle))
    __tokencap_dump(needle, strlen(needle), 1);

  do {
    const char* n = needle;
    const char* h = haystack;

    while(*n && *h && *n == *h) n++, h++;

    if(!*n) return (char*)haystack;

  } while (*(haystack++));

  return 0;

}


#undef strcasestr

char* strcasestr(const char* haystack, const char* needle) {

  if (__tokencap_is_ro(haystack))
    __tokencap_dump(haystack, strlen(haystack), 1);

  if (__tokencap_is_ro(needle))
    __tokencap_dump(needle, strlen(needle), 1);

  do {

    const char* n = needle;
    const char* h = haystack;

    while(*n && *h && tolower(*n) == tolower(*h)) n++, h++;

    if(!*n) return (char*)haystack;

  } while(*(haystack++));

  return 0;

}


/* Init code to open the output file (or default to stderr). */

__attribute__((constructor)) void __tokencap_init(void) {

  u8* fn = getenv("AFL_TOKEN_FILE");
```

```
  if (fn) __tokencap_out_file = fopen(fn, "a");
  if (!__tokencap_out_file) __tokencap_out_file = stderr;

}
```

Makefile

```
#
# american fuzzy lop - libtokencap
# --------------------------------
#
# Written by Michal Zalewski <lcamtuf@google.com>
#
# Copyright 2016 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#

PREFIX      ?= /usr/local
HELPER_PATH  = $(PREFIX)/lib/afl

VERSION     = $(shell grep '^\#define VERSION ' ../config.h | cut -d '"' -f2)

CFLAGS      ?= -O3 -funroll-loops
CFLAGS      += -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign

all: libtokencap.so

libtokencap.so: libtokencap.so.c ../config.h
    $(CC) $(CFLAGS) -shared -fPIC $< -o $@ $(LDFLAGS)

.NOTPARALLEL: clean

clean:
    rm -f *.o *.so *~ a.out core core.[1-9][0-9]*
    rm -f libtokencap.so

install: all
    install -m 755 libtokencap.so $${DESTDIR}$(HELPER_PATH)
    install -m 644 README.tokencap $${DESTDIR}$(HELPER_PATH)
```

> llvm_mode

afl-clang-fast.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
```

```
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - LLVM-mode wrapper for clang
   ------------------------------------------------

   Written by Laszlo Szekeres <lszekeres@google.com> and
              Michal Zalewski <lcamtuf@google.com>

   LLVM integration design comes from Laszlo Szekeres.

   This program is a drop-in replacement for clang, similar in most respects
   to ../afl-gcc. It tries to figure out compilation mode, adds a bunch
   of flags, and then calls the real compiler.
*/

#define AFL_MAIN

#include "../config.h"
#include "../types.h"
#include "../debug.h"
#include "../alloc-inl.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

static u8*  obj_path;               /* Path to runtime libraries         */
static u8** cc_params;              /* Parameters passed to the real CC  */
static u32  cc_par_cnt = 1;         /* Param count, including argv0      */


/* Try to find the runtime libraries. If that fails, abort. */

static void find_obj(u8* argv0) {

  u8 *afl_path = getenv("AFL_PATH");
  u8 *slash, *tmp;

  if (afl_path) {

    tmp = alloc_printf("%s/afl-llvm-rt.o", afl_path);

    if (!access(tmp, R_OK)) {
      obj_path = afl_path;
      ck_free(tmp);
      return;
    }
```

```
    ck_free(tmp);

  }

  slash = strrchr(argv0, '/');

  if (slash) {

    u8 *dir;

    *slash = 0;
    dir = ck_strdup(argv0);
    *slash = '/';

    tmp = alloc_printf("%s/afl-llvm-rt.o", dir);

    if (!access(tmp, R_OK)) {
      obj_path = dir;
      ck_free(tmp);
      return;
    }

    ck_free(tmp);
    ck_free(dir);

  }

  if (!access(AFL_PATH "/afl-llvm-rt.o", R_OK)) {
    obj_path = AFL_PATH;
    return;
  }

  FATAL("Unable to find 'afl-llvm-rt.o' or 'afl-llvm-pass.so'. Please set AFL_PATH");

}


/* Copy argv to cc_params, making the necessary edits. */

static void edit_params(u32 argc, char** argv) {

  u8 fortify_set = 0, asan_set = 0, x_set = 0, bit_mode = 0;
  u8 *name;

  cc_params = ck_alloc((argc + 128) * sizeof(u8*));

  name = strrchr(argv[0], '/');
  if (!name) name = argv[0]; else name++;

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

  /* There are two ways to compile afl-clang-fast. In the traditional mode, we
     use afl-llvm-pass.so to inject instrumentation. In the experimental
     'trace-pc-guard' mode, we use native LLVM instrumentation callbacks
```

```
     instead. The latter is a very recent addition - see:

     http://clang.llvm.org/docs/SanitizerCoverage.html#tracing-pcs-with-guards */

#ifdef USE_TRACE_PC
  cc_params[cc_par_cnt++] = "-fsanitize-coverage=trace-pc-guard";
#ifndef __ANDROID__
  cc_params[cc_par_cnt++] = "-mllvm";
  cc_params[cc_par_cnt++] = "-sanitizer-coverage-block-threshold=0";
#endif
#else
  cc_params[cc_par_cnt++] = "-Xclang";
  cc_params[cc_par_cnt++] = "-load";
  cc_params[cc_par_cnt++] = "-Xclang";
  cc_params[cc_par_cnt++] = alloc_printf("%s/afl-llvm-pass.so", obj_path);
#endif /* ^USE_TRACE_PC */

  cc_params[cc_par_cnt++] = "-Qunused-arguments";

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

    if (!strcmp(cur, "-m32")) bit_mode = 32;
    if (!strcmp(cur, "armv7a-linux-androideabi")) bit_mode = 32;
    if (!strcmp(cur, "-m64")) bit_mode = 64;

    if (!strcmp(cur, "-x")) x_set = 1;

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

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

    if (!strcmp(cur, "-Wl,-z,defs") ||
        !strcmp(cur, "-Wl,--no-undefined")) continue;

    cc_params[cc_par_cnt++] = cur;

  }

  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) {

    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";

    }

  }

#ifdef USE_TRACE_PC

  if (getenv("AFL_INST_RATIO"))
    FATAL("AFL_INST_RATIO not available at compile time with 'trace-pc'.");

#endif /* USE_TRACE_PC */

  if (!getenv("AFL_DONT_OPTIMIZE")) {

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

  }

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

  /* When the user tries to use persistent or deferred forkserver modes by
     appending a single line to the program, we want to reliably inject a
     signature into the binary (to be picked up by afl-fuzz) and we want
     to call a function from the runtime .o file. This is unnecessarily
     painful for three reasons:

     1) We need to convince the compiler not to optimize out the signature.
        This is done with __attribute__((used)).

     2) We need to convince the linker, when called with -Wl,--gc-sections,
        not to do the same. This is done by forcing an assignment to a
        'volatile' pointer.

     3) We need to declare __afl_persistent_loop() in the global namespace,
```

```
        but doing this within a method in a class is hard - :: and extern "C"
        are forbidden and __attribute__((alias(...))) doesn't work. Hence the
        __asm__ aliasing trick.

   */

  cc_params[cc_par_cnt++] = "-D__AFL_LOOP(_A)="
    "({ static volatile char *_B __attribute__((used)); "
    " _B = (char*)\"" PERSIST_SIG "\"; "
#ifdef __APPLE__
    "__attribute__((visibility(\"default\"))) "
    "int _L(unsigned int) __asm__(\"___afl_persistent_loop\"); "
#else
    "__attribute__((visibility(\"default\"))) "
    "int _L(unsigned int) __asm__(\"__afl_persistent_loop\"); "
#endif /* ^__APPLE__ */
    "_L(_A); })";

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

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

#ifndef __ANDROID__
  switch (bit_mode) {

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

    case 32:
      cc_params[cc_par_cnt++] = alloc_printf("%s/afl-llvm-rt-32.o", obj_path);

      if (access(cc_params[cc_par_cnt - 1], R_OK))
        FATAL("-m32 is not supported by your compiler");

      break;

    case 64:
      cc_params[cc_par_cnt++] = alloc_printf("%s/afl-llvm-rt-64.o", obj_path);

      if (access(cc_params[cc_par_cnt - 1], R_OK))
        FATAL("-m64 is not supported by your compiler");

      break;

  }
#endif
```

```
  cc_params[cc_par_cnt] = NULL;

}


/* Main entry point */

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

  if (isatty(2) && !getenv("AFL_QUIET")) {

#ifdef USE_TRACE_PC
    SAYF(cCYA "afl-clang-fast [tpcg] " cBRI VERSION  cRST " by <lszekeres@google.com>\n");
#else
    SAYF(cCYA "afl-clang-fast " cBRI VERSION  cRST " by <lszekeres@google.com>\n");
#endif /* ^USE_TRACE_PC */

  }

  if (argc < 2) {

    SAYF("\n"
         "This is a helper application for afl-fuzz. It serves as a drop-in replacement\n"
         "for clang, letting you recompile third-party code with the required runtime\n"
         "instrumentation. A common use pattern would be one of the following:\n\n"

         "  CC=%s/afl-clang-fast ./configure\n"
         "  CXX=%s/afl-clang-fast++ ./configure\n\n"

         "In contrast to the traditional afl-clang tool, this version is implemented as\n"
         "an LLVM pass and tends to offer improved performance with slow programs.\n\n"

         "You can specify custom next-stage toolchain via AFL_CC and AFL_CXX. Setting\n"
         "AFL_HARDEN enables hardening optimizations in the compiled code.\n\n",
         BIN_PATH, BIN_PATH);

    exit(1);

  }


#ifndef __ANDROID__
  find_obj(argv[0]);
#endif

  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;

}
```

afl-llvm-pass.so.cc

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - LLVM-mode instrumentation pass
   ---------------------------------------------------

   Written by Laszlo Szekeres <lszekeres@google.com> and
              Michal Zalewski <lcamtuf@google.com>

   LLVM integration design comes from Laszlo Szekeres. C bits copied-and-pasted
   from afl-as.c are Michal's fault.

   This library is plugged into LLVM when invoking clang through afl-clang-fast.
   It tells the compiler to add code roughly equivalent to the bits discussed
   in ../afl-as.h.
*/

#define AFL_LLVM_PASS

#include "../config.h"
#include "../debug.h"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "llvm/ADT/Statistic.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/Debug.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;

namespace {

  class AFLCoverage : public ModulePass {

    public:

      static char ID;
      AFLCoverage() : ModulePass(ID) { }
```

```
    bool runOnModule(Module &M) override;

    // StringRef getPassName() const override {
    //  return "American Fuzzy Lop Instrumentation";
    // }

  };

}


char AFLCoverage::ID = 0;


bool AFLCoverage::runOnModule(Module &M) {

  LLVMContext &C = M.getContext();

  IntegerType *Int8Ty  = IntegerType::getInt8Ty(C);
  IntegerType *Int32Ty = IntegerType::getInt32Ty(C);

  /* Show a banner */

  char be_quiet = 0;

  if (isatty(2) && !getenv("AFL_QUIET")) {

    SAYF(cCYA "afl-llvm-pass " cBRI VERSION cRST " by <lszekeres@google.com>\n");

  } else be_quiet = 1;

  /* Decide instrumentation ratio */

  char* inst_ratio_str = getenv("AFL_INST_RATIO");
  unsigned int inst_ratio = 100;

  if (inst_ratio_str) {

    if (sscanf(inst_ratio_str, "%u", &inst_ratio) != 1 || !inst_ratio ||
        inst_ratio > 100)
      FATAL("Bad value of AFL_INST_RATIO (must be between 1 and 100)");

  }

  /* Get globals for the SHM region and the previous location. Note that
     __afl_prev_loc is thread-local. */

  GlobalVariable *AFLMapPtr =
      new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
                         GlobalValue::ExternalLinkage, 0, "__afl_area_ptr");

  GlobalVariable *AFLPrevLoc = new GlobalVariable(
      M, Int32Ty, false, GlobalValue::ExternalLinkage, 0, "__afl_prev_loc",
      0, GlobalVariable::GeneralDynamicTLSModel, 0, false);

  /* Instrument all the things! */

  int inst_blocks = 0;
```

```
  for (auto &F : M)
    for (auto &BB : F) {

      BasicBlock::iterator IP = BB.getFirstInsertionPt();
      IRBuilder<> IRB(&(*IP));

      if (AFL_R(100) >= inst_ratio) continue;

      /* Make up cur_loc */

      unsigned int cur_loc = AFL_R(MAP_SIZE);

      ConstantInt *CurLoc = ConstantInt::get(Int32Ty, cur_loc);

      /* Load prev_loc */

      LoadInst *PrevLoc = IRB.CreateLoad(AFLPrevLoc);
      PrevLoc->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
      Value *PrevLocCasted = IRB.CreateZExt(PrevLoc, IRB.getInt32Ty());

      /* Load SHM pointer */

      LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
      Value *MapPtrIdx =
          IRB.CreateGEP(MapPtr, IRB.CreateXor(PrevLocCasted, CurLoc));

      /* Update bitmap */

      LoadInst *Counter = IRB.CreateLoad(MapPtrIdx);
      Counter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));
      Value *Incr = IRB.CreateAdd(Counter, ConstantInt::get(Int8Ty, 1));
      IRB.CreateStore(Incr, MapPtrIdx)
          ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));

      /* Set prev_loc to cur_loc >> 1 */

      StoreInst *Store =
          IRB.CreateStore(ConstantInt::get(Int32Ty, cur_loc >> 1), AFLPrevLoc);
      Store->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C, None));

      inst_blocks++;

    }

  /* Say something nice. */

  if (!be_quiet) {

    if (!inst_blocks) WARNF("No instrumentation targets found.");
    else OKF("Instrumented %u locations (%s mode, ratio %u%%).",
             inst_blocks, getenv("AFL_HARDEN") ? "hardened" :
             ((getenv("AFL_USE_ASAN") || getenv("AFL_USE_MSAN")) ?
              "ASAN/MSAN" : "non-hardened"), inst_ratio);

  }

  return true;

}
```

```
static void registerAFLPass(const PassManagerBuilder &,
                            legacy::PassManagerBase &PM) {

  PM.add(new AFLCoverage());

}


static RegisterStandardPasses RegisterAFLPass(
    PassManagerBuilder::EP_ModuleOptimizerEarly, registerAFLPass);

static RegisterStandardPasses RegisterAFLPass0(
    PassManagerBuilder::EP_EnabledOnOptLevel0, registerAFLPass);
```

afl-llvm-rt.o.c

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - LLVM instrumentation bootstrap
   ---------------------------------------------------

   Written by Laszlo Szekeres <lszekeres@google.com> and
              Michal Zalewski <lcamtuf@google.com>

   LLVM integration design comes from Laszlo Szekeres.

   This code is the rewrite of afl-as.h's main_payload.
*/

#include "../android-ashmem.h"
#include "../config.h"
#include "../types.h"

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>

#include <sys/mman.h>
```

```
#include <sys/shm.h>
#include <sys/wait.h>
#include <sys/types.h>

/* This is a somewhat ugly hack for the experimental 'trace-pc-guard' mode.
   Basically, we need to make sure that the forkserver is initialized after
   the LLVM-generated runtime initialization pass, not before. */

#ifdef USE_TRACE_PC
#  define CONST_PRIO 5
#else
#  define CONST_PRIO 0
#endif /* ^USE_TRACE_PC */


/* Globals needed by the injected instrumentation. The __afl_area_initial region
   is used for instrumentation output before __afl_map_shm() has a chance to run.
   It will end up as .comm, so it shouldn't be too wasteful. */

u8  __afl_area_initial[MAP_SIZE];
u8* __afl_area_ptr = __afl_area_initial;

__thread u32 __afl_prev_loc;


/* Running in persistent mode? */

static u8 is_persistent;


/* SHM setup. */

static void __afl_map_shm(void) {

  u8 *id_str = getenv(SHM_ENV_VAR);

  /* If we're running under AFL, attach to the appropriate region, replacing the
     early-stage __afl_area_initial region that is needed to allow some really
     hacky .init code to work correctly in projects such as OpenSSL. */

  if (id_str) {

    u32 shm_id = atoi(id_str);

    __afl_area_ptr = shmat(shm_id, NULL, 0);

    /* Whooooops. */

    if (__afl_area_ptr == (void *)-1) _exit(1);

    /* Write something into the bitmap so that even with low AFL_INST_RATIO,
       our parent doesn't give up on us. */

    __afl_area_ptr[0] = 1;

  }

}
```

```
/* Fork server logic. */

static void __afl_start_forkserver(void) {

  static u8 tmp[4];
  s32 child_pid;

  u8  child_stopped = 0;

  /* Phone home and tell the parent that we're OK. If parent isn't there,
     assume we're not running in forkserver mode and just execute program. */

  if (write(FORKSRV_FD + 1, tmp, 4) != 4) return;

  while (1) {

    u32 was_killed;
    int status;

    /* Wait for parent by reading from the pipe. Abort if read fails. */

    if (read(FORKSRV_FD, &was_killed, 4) != 4) _exit(1);

    /* If we stopped the child in persistent mode, but there was a race
       condition and afl-fuzz already issued SIGKILL, write off the old
       process. */

    if (child_stopped && was_killed) {
      child_stopped = 0;
      if (waitpid(child_pid, &status, 0) < 0) _exit(1);
    }

    if (!child_stopped) {

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

      child_pid = fork();
      if (child_pid < 0) _exit(1);

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

      if (!child_pid) {

        close(FORKSRV_FD);
        close(FORKSRV_FD + 1);
        return;

      }

    } else {

      /* Special handling for persistent mode: if the child is alive but
         currently stopped, simply restart it with SIGCONT. */

      kill(child_pid, SIGCONT);
      child_stopped = 0;

    }

    /* In parent process: write PID to pipe, then wait for child. */
```

```
    if (write(FORKSRV_FD + 1, &child_pid, 4) != 4) _exit(1);

    if (waitpid(child_pid, &status, is_persistent ? WUNTRACED : 0) < 0)
      _exit(1);

    /* In persistent mode, the child stops itself with SIGSTOP to indicate
       a successful run. In this case, we want to wake it up without forking
       again. */

    if (WIFSTOPPED(status)) child_stopped = 1;

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

    if (write(FORKSRV_FD + 1, &status, 4) != 4) _exit(1);

  }

}


/* A simplified persistent mode handler, used as explained in README.llvm. */

int __afl_persistent_loop(unsigned int max_cnt) {

  static u8  first_pass = 1;
  static u32 cycle_cnt;

  if (first_pass) {

    /* Make sure that every iteration of __AFL_LOOP() starts with a clean slate.
       On subsequent calls, the parent will take care of that, but on the first
       iteration, it's our job to erase any trace of whatever happened
       before the loop. */

    if (is_persistent) {

      memset(__afl_area_ptr, 0, MAP_SIZE);
      __afl_area_ptr[0] = 1;
      __afl_prev_loc = 0;
    }

    cycle_cnt  = max_cnt;
    first_pass = 0;
    return 1;

  }

  if (is_persistent) {

    if (--cycle_cnt) {

      raise(SIGSTOP);

      __afl_area_ptr[0] = 1;
      __afl_prev_loc = 0;

      return 1;

    } else {
```

```
      /* When exiting __AFL_LOOP(), make sure that the subsequent code that
         follows the loop is not traced. We do that by pivoting back to the
         dummy output region. */

      __afl_area_ptr = __afl_area_initial;

    }

  }

  return 0;

}


/* This one can be called from user code when deferred forkserver mode
    is enabled. */

void __afl_manual_init(void) {

  static u8 init_done;

  if (!init_done) {

    __afl_map_shm();
    __afl_start_forkserver();
    init_done = 1;

  }

}


/* Proper initialization routine. */

__attribute__((constructor(CONST_PRIO))) void __afl_auto_init(void) {

  is_persistent = !!getenv(PERSIST_ENV_VAR);

  if (getenv(DEFER_ENV_VAR)) return;

  __afl_manual_init();

}


/* The following stuff deals with supporting -fsanitize-coverage=trace-pc-guard.
   It remains non-operational in the traditional, plugin-backed LLVM mode.
   For more info about 'trace-pc-guard', see README.llvm.

   The first function (__sanitizer_cov_trace_pc_guard) is called back on every
   edge (as opposed to every basic block). */

void __sanitizer_cov_trace_pc_guard(uint32_t* guard) {
  __afl_area_ptr[*guard]++;
}


/* Init callback. Populates instrumentation IDs. Note that we're using
```

```
   ID of 0 as a special value to indicate non-instrumented bits. That may
   still touch the bitmap, but in a fairly harmless way. */

void __sanitizer_cov_trace_pc_guard_init(uint32_t* start, uint32_t* stop) {

  u32 inst_ratio = 100;
  u8* x;

  if (start == stop || *start) return;

  x = getenv("AFL_INST_RATIO");
  if (x) inst_ratio = atoi(x);

  if (!inst_ratio || inst_ratio > 100) {
    fprintf(stderr, "[-] ERROR: Invalid AFL_INST_RATIO (must be 1-100).\n");
    abort();
  }

  /* Make sure that the first element in the range is always set - we use that
     to avoid duplicate calls (which can happen as an artifact of the underlying
     implementation in LLVM). */

  *(start++) = R(MAP_SIZE - 1) + 1;

  while (start < stop) {

    if (R(100) < inst_ratio) *start = R(MAP_SIZE - 1) + 1;
    else *start = 0;

    start++;

  }

}
```

Makefile

```
#
# american fuzzy lop - LLVM instrumentation
# -----------------------------------------
#
# Written by Laszlo Szekeres <lszekeres@google.com> and
#            Michal Zalewski <lcamtuf@google.com>
#
# LLVM integration design comes from Laszlo Szekeres.
#
# Copyright 2015, 2016 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#   http://www.apache.org/licenses/LICENSE-2.0
#

PREFIX      ?= /usr/local
HELPER_PATH  = $(PREFIX)/lib/afl
BIN_PATH     = $(PREFIX)/bin
```

```
VERSION     = $(shell grep '^\#define VERSION ' ../config.h | cut -d '"' -f2)

LLVM_CONFIG ?= llvm-config

CFLAGS      ?= -O3 -funroll-loops
CFLAGS      += -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign \
               -DAFL_PATH=\"$(HELPER_PATH)\" -DBIN_PATH=\"$(BIN_PATH)\" \
               -DVERSION=\"$(VERSION)\"
ifdef AFL_TRACE_PC
  CFLAGS    += -DUSE_TRACE_PC=1
endif

CXXFLAGS    ?= -O3 -funroll-loops
CXXFLAGS    += -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign \
               -DVERSION=\"$(VERSION)\" -Wno-variadic-macros

# Mark nodelete to work around unload bug in upstream LLVM 5.0+
CLANG_CFL    = `$(LLVM_CONFIG) --cxxflags` -Wl,-znodelete -fno-rtti -fpic $(CXXFLAGS)
CLANG_LFL    = `$(LLVM_CONFIG) --ldflags` $(LDFLAGS)

# User teor2345 reports that this is required to make things work on MacOS X.

ifeq "$(shell uname)" "Darwin"
  CLANG_LFL += -Wl,-flat_namespace -Wl,-undefined,suppress
endif

# We were using llvm-config --bindir to get the location of clang, but
# this seems to be busted on some distros, so using the one in $PATH is
# probably better.

ifeq "$(origin CC)" "default"
  CC         = clang
  CXX        = clang++
endif

ifndef AFL_TRACE_PC
  PROGS      = ../afl-clang-fast ../afl-llvm-pass.so ../afl-llvm-rt.o ../afl-llvm-rt-32.o
../afl-llvm-rt-64.o
else
  PROGS      = ../afl-clang-fast ../afl-llvm-rt.o ../afl-llvm-rt-32.o ../afl-llvm-rt-64.o
endif

all: test_deps $(PROGS) test_build all_done

test_deps:
ifndef AFL_TRACE_PC
    @echo "[*] Checking for working 'llvm-config'..."
    @which $(LLVM_CONFIG) >/dev/null 2>&1 || ( echo "[-] Oops, can't find 'llvm-config'.
Install clang or set \$$LLVM_CONFIG or \$$PATH beforehand."; echo "    (Sometimes, the binary
will be named llvm-config-3.5 or something like that.)"; exit 1 )
else
    @echo "[!] Note: using -fsanitize=trace-pc mode (this will fail with older LLVM)."
endif
    @echo "[*] Checking for working '$(CC)'..."
    @which $(CC) >/dev/null 2>&1 || ( echo "[-] Oops, can't find '$(CC)'. Make sure that it's
in your \$$PATH (or set \$$CC and \$$CXX)."; exit 1 )
    @echo "[*] Checking for '../afl-showmap'..."
    @test -f ../afl-showmap || ( echo "[-] Oops, can't find '../afl-showmap'. Be sure to
compile AFL first."; exit 1 )
```

```
    @echo "[+] All set and ready to build."

../afl-clang-fast: afl-clang-fast.c | test_deps
    $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)
    ln -sf afl-clang-fast ../afl-clang-fast++

../afl-llvm-pass.so: afl-llvm-pass.so.cc | test_deps
    $(CXX) $(CLANG_CFL) -shared $< -o $@ $(CLANG_LFL)

../afl-llvm-rt.o: afl-llvm-rt.o.c | test_deps
    $(CC) $(CFLAGS) -fPIC -c $< -o $@

../afl-llvm-rt-32.o: afl-llvm-rt.o.c | test_deps
    @printf "[*] Building 32-bit variant of the runtime (-m32)... "
    @$(CC) $(CFLAGS) -m32 -fPIC -c $< -o $@ 2>/dev/null; if [ "$$?" = "0" ]; then echo
"success!"; else echo "failed (that's fine)"; fi

../afl-llvm-rt-64.o: afl-llvm-rt.o.c | test_deps
    @printf "[*] Building 64-bit variant of the runtime (-m64)... "
    @$(CC) $(CFLAGS) -m64 -fPIC -c $< -o $@ 2>/dev/null; if [ "$$?" = "0" ]; then echo
"success!"; else echo "failed (that's fine)"; fi

test_build: $(PROGS)
    @echo "[*] Testing the CC wrapper and instrumentation output..."
    unset AFL_USE_ASAN AFL_USE_MSAN AFL_INST_RATIO; AFL_QUIET=1 AFL_PATH=. AFL_CC=$(CC)
../afl-clang-fast $(CFLAGS) ../test-instr.c -o test-instr $(LDFLAGS)
# Use /dev/null to avoid problems with optimization messing up expected
# branches. See https://github.com/google/AFL/issues/30.
    ../afl-showmap -m none -q -o .test-instr0 ./test-instr < /dev/null
    echo 1 | ../afl-showmap -m none -q -o .test-instr1 ./test-instr
    @rm -f test-instr
    @cmp -s .test-instr0 .test-instr1; DR="$$?"; rm -f .test-instr0 .test-instr1; if [ "$$DR"
= "0" ]; then echo; echo "Oops, the instrumentation does not seem to be behaving correctly!";
echo; echo "Please ping <lcamtuf@google.com> to troubleshoot the issue."; echo; exit 1; fi
    @echo "[+] All right, the instrumentation seems to be working!"

all_done: test_build
    @echo "[+] All done! You can now use '../afl-clang-fast' to compile programs."

.NOTPARALLEL: clean

clean:
    rm -f *.o *.so *~ a.out core core.[1-9][0-9]* test-instr .test-instr0 .test-instr1
    rm -f $(PROGS) ../afl-clang-fast++
```

Makefile

```
#
# american fuzzy lop - makefile
# -----------------------------
#
# Written and maintained by Michal Zalewski <lcamtuf@google.com>
#
# Copyright 2013, 2014, 2015, 2016, 2017 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
```

```
#
#   http://www.apache.org/licenses/LICENSE-2.0
#

PROGNAME    = afl
VERSION     = $(shell grep '^\#define VERSION ' config.h | cut -d '"' -f2)

PREFIX     ?= /usr/local
BIN_PATH    = $(PREFIX)/bin
HELPER_PATH = $(PREFIX)/lib/afl
DOC_PATH    = $(PREFIX)/share/doc/afl
MISC_PATH   = $(PREFIX)/share/afl

# PROGS intentionally omit afl-as, which gets installed elsewhere.

PROGS       = afl-gcc afl-fuzz afl-showmap afl-tmin afl-gotcpu afl-analyze
SH_PROGS    = afl-plot afl-cmin afl-whatsup

CFLAGS     ?= -O3 -funroll-loops
CFLAGS     += -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign \
          -DAFL_PATH=\"$(HELPER_PATH)\" -DDOC_PATH=\"$(DOC_PATH)\" \
          -DBIN_PATH=\"$(BIN_PATH)\"

ifneq "$(filter Linux GNU%,$(shell uname))" ""
  LDFLAGS  += -ldl
endif

ifeq "$(findstring clang, $(shell $(CC) --version 2>/dev/null))" ""
  TEST_CC   = afl-gcc
else
  TEST_CC   = afl-clang
endif

COMM_HDR    = alloc-inl.h config.h debug.h types.h

all: test_x86 $(PROGS) afl-as test_build all_done

ifndef AFL_NO_X86

test_x86:
    @echo "[*] Checking for the ability to compile x86 code..."
    @echo 'main() { __asm__("xorb %al, %al"); }' | $(CC) -w -x c - -o .test || ( echo; echo "Oops, looks like your compiler can't generate x86 code."; echo; echo "Don't panic! You can use the LLVM or QEMU mode, but see docs/INSTALL first."; echo "(To ignore this error, set AFL_NO_X86=1 and try again.)"; echo; exit 1 )
    @rm -f .test
    @echo "[+] Everything seems to be working, ready to compile."

else

test_x86:
    @echo "[!] Note: skipping x86 compilation checks (AFL_NO_X86 set)."

endif

afl-gcc: afl-gcc.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)
    set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $$i; done

afl-as: afl-as.c afl-as.h $(COMM_HDR) | test_x86
```

```
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)
    ln -sf afl-as as

afl-fuzz: afl-fuzz.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)

afl-showmap: afl-showmap.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)

afl-tmin: afl-tmin.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)

afl-analyze: afl-analyze.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)

afl-gotcpu: afl-gotcpu.c $(COMM_HDR) | test_x86
    $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS)

ifndef AFL_NO_X86

test_build: afl-gcc afl-as afl-showmap
    @echo "[*] Testing the CC wrapper and instrumentation output..."
    unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./$(TEST_CC)
$(CFLAGS) test-instr.c -o test-instr $(LDFLAGS)
    ./afl-showmap -m none -q -o .test-instr0 ./test-instr < /dev/null
    echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
    @rm -f test-instr
    @cmp -s .test-instr0 .test-instr1; DR="$$?"; rm -f .test-instr0 .test-instr1; if [ "$$DR"
= "0" ]; then echo; echo "Oops, the instrumentation does not seem to be behaving correctly!";
echo; echo "Please ping <lcamtuf@google.com> to troubleshoot the issue."; echo; exit 1; fi
    @echo "[+] All right, the instrumentation seems to be working!"

else

test_build: afl-gcc afl-as afl-showmap
    @echo "[!] Note: skipping build tests (you may need to use LLVM or QEMU mode)."

endif

all_done: test_build
    @if [ ! "`which clang 2>/dev/null`" = "" ]; then echo "[+] LLVM users: see
llvm_mode/README.llvm for a faster alternative to afl-gcc."; fi
    @echo "[+] All done! Be sure to review README - it's pretty short and useful."
    @if [ "`uname`" = "Darwin" ]; then printf "\nWARNING: Fuzzing on MacOS X is slow because
of the unusually high overhead of\nfork() on this OS. Consider using Linux or *BSD. You can
also use VirtualBox\n(virtualbox.org) to put AFL inside a Linux or *BSD VM.\n\n"; fi
    @! tty <&1 >/dev/null || printf "\033[0;30mNOTE: If you can read this, your terminal
probably uses white background.\nThis will make the UI hard to read. See
docs/status_screen.txt for advice.\033[0m\n" 2>/dev/null

.NOTPARALLEL: clean

clean:
    rm -f $(PROGS) afl-as as afl-g++ afl-clang afl-clang++ *.o *~ a.out core core.[1-9][0-9]*
*.stackdump test .test test-instr .test-instr0 .test-instr1 qemu_mode/qemu-2.10.0.tar.bz2 afl-
qemu-trace
    rm -rf out_dir qemu_mode/qemu-2.10.0
    $(MAKE) -C llvm_mode clean
    $(MAKE) -C libdislocator clean
    $(MAKE) -C libtokencap clean
```

```
install: all
    mkdir -p -m 755 $${DESTDIR}$(BIN_PATH) $${DESTDIR}$(HELPER_PATH) $${DESTDIR}$(DOC_PATH)
$${DESTDIR}$(MISC_PATH)
    rm -f $${DESTDIR}$(BIN_PATH)/afl-plot.sh
    install -m 755 $(PROGS) $(SH_PROGS) $${DESTDIR}$(BIN_PATH)
    rm -f $${DESTDIR}$(BIN_PATH)/afl-as
    if [ -f afl-qemu-trace ]; then install -m 755 afl-qemu-trace $${DESTDIR}$(BIN_PATH); fi
ifndef AFL_TRACE_PC
    if [ -f afl-clang-fast -a -f afl-llvm-pass.so -a -f afl-llvm-rt.o ]; then set -e; install
-m 755 afl-clang-fast $${DESTDIR}$(BIN_PATH); ln -sf afl-clang-fast
$${DESTDIR}$(BIN_PATH)/afl-clang-fast++; install -m 755 afl-llvm-pass.so afl-llvm-rt.o
$${DESTDIR}$(HELPER_PATH); fi
else
    if [ -f afl-clang-fast -a -f afl-llvm-rt.o ]; then set -e; install -m 755 afl-clang-fast
$${DESTDIR}$(BIN_PATH); ln -sf afl-clang-fast $${DESTDIR}$(BIN_PATH)/afl-clang-fast++; install
-m 755 afl-llvm-rt.o $${DESTDIR}$(HELPER_PATH); fi
endif
    if [ -f afl-llvm-rt-32.o ]; then set -e; install -m 755 afl-llvm-rt-32.o
$${DESTDIR}$(HELPER_PATH); fi
    if [ -f afl-llvm-rt-64.o ]; then set -e; install -m 755 afl-llvm-rt-64.o
$${DESTDIR}$(HELPER_PATH); fi
    set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc
$${DESTDIR}$(BIN_PATH)/$$i; done
    install -m 755 afl-as $${DESTDIR}$(HELPER_PATH)
    ln -sf afl-as $${DESTDIR}$(HELPER_PATH)/as
    install -m 644 README.md docs/ChangeLog docs/*.txt $${DESTDIR}$(DOC_PATH)
    cp -r testcases/ $${DESTDIR}$(MISC_PATH)
    cp -r dictionaries/ $${DESTDIR}$(MISC_PATH)

publish: clean
    test "`basename $$PWD`" = "AFL" || exit 1
    test -f ~/www/afl/releases/$(PROGNAME)-$(VERSION).tgz; if [ "$$?" = "0" ]; then echo; echo
"Change program version in config.h, mmkay?"; echo; exit 1; fi
    cd ..; rm -rf $(PROGNAME)-$(VERSION); cp -pr $(PROGNAME) $(PROGNAME)-$(VERSION); \
      tar -cvz -f ~/www/afl/releases/$(PROGNAME)-$(VERSION).tgz $(PROGNAME)-$(VERSION)
    chmod 644 ~/www/afl/releases/$(PROGNAME)-$(VERSION).tgz
    ( cd ~/www/afl/releases/; ln -s -f $(PROGNAME)-$(VERSION).tgz $(PROGNAME)-latest.tgz )
    cat docs/README >~/www/afl/README.txt
    cat docs/status_screen.txt >~/www/afl/status_screen.txt
    cat docs/historical_notes.txt >~/www/afl/historical_notes.txt
    cat docs/technical_details.txt >~/www/afl/technical_details.txt
    cat docs/ChangeLog >~/www/afl/ChangeLog.txt
    cat docs/QuickStartGuide.txt >~/www/afl/QuickStartGuide.txt
    echo -n "$(VERSION)" >~/www/afl/version.txt
```

> qemu_mode

build_qemu_support.sh

```
#!/bin/sh
#
# Copyright 2015 Google LLC All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
```

```
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ----------------------------------------
# american fuzzy lop - QEMU build script
# ---------------------------------------
#
# Written by Andrew Griffiths <agriffiths@google.com> and
#            Michal Zalewski <lcamtuf@google.com>
#
# This script downloads, patches, and builds a version of QEMU with
# minor tweaks to allow non-instrumented binaries to be run under
# afl-fuzz.
#
# The modifications reside in patches/*. The standalone QEMU binary
# will be written to ../afl-qemu-trace.
#


VERSION="2.10.0"
QEMU_URL="http://download.qemu-project.org/qemu-${VERSION}.tar.xz"
QEMU_SHA384="68216c935487bc8c0596ac309e1e3ee75c2c4ce898aab796faa321db5740609ced365fedda025678d
072d09ac892810 5"

echo "================================================="
echo "AFL binary-only instrumentation QEMU build script"
echo "================================================="
echo

echo "[*] Performing basic sanity checks..."

if [ ! "`uname -s`" = "Linux" ]; then

  echo "[-] Error: QEMU instrumentation is supported only on Linux."
  exit 1

fi

if [ ! -f "patches/afl-qemu-cpu-inl.h" -o ! -f "../config.h" ]; then

  echo "[-] Error: key files not found - wrong working directory?"
  exit 1

fi

if [ ! -f "../afl-showmap" ]; then

  echo "[-] Error: ../afl-showmap not found - compile AFL first!"
  exit 1

fi


for i in libtool wget python automake autoconf sha384sum bison iconv; do
```

```
  T=`which "$i" 2>/dev/null`

  if [ "$T" = "" ]; then

    echo "[-] Error: '$i' not found, please install first."
    exit 1

  fi

done

if [ ! -d "/usr/include/glib-2.0/" -a ! -d "/usr/local/include/glib-2.0/" ]; then

  echo "[-] Error: devel version of 'glib2' not found, please install first."
  exit 1

fi

if echo "$CC" | grep -qF /afl-; then

  echo "[-] Error: do not use afl-gcc or afl-clang to compile this tool."
  exit 1

fi

echo "[+] All checks passed!"

ARCHIVE="`basename -- "$QEMU_URL"`"

CKSUM=`sha384sum -- "$ARCHIVE" 2>/dev/null | cut -d' ' -f1`

if [ ! "$CKSUM" = "$QEMU_SHA384" ]; then

  echo "[*] Downloading QEMU ${VERSION} from the web..."
  rm -f "$ARCHIVE"
  wget -O "$ARCHIVE" -- "$QEMU_URL" || exit 1

  CKSUM=`sha384sum -- "$ARCHIVE" 2>/dev/null | cut -d' ' -f1`

fi

if [ "$CKSUM" = "$QEMU_SHA384" ]; then

  echo "[+] Cryptographic signature on $ARCHIVE checks out."

else

  echo "[-] Error: signature mismatch on $ARCHIVE (perhaps download error?)."
  exit 1

fi

echo "[*] Uncompressing archive (this will take a while)..."

rm -rf "qemu-${VERSION}" || exit 1
tar xf "$ARCHIVE" || exit 1

echo "[+] Unpacking successful."

echo "[*] Configuring QEMU for $CPU_TARGET..."
```

```
ORIG_CPU_TARGET="$CPU_TARGET"

test "$CPU_TARGET" = "" && CPU_TARGET="`uname -m`"
test "$CPU_TARGET" = "i686" && CPU_TARGET="i386"

cd qemu-$VERSION || exit 1

echo "[*] Applying patches..."

patch -p1 <../patches/elfload.diff || exit 1
patch -p1 <../patches/cpu-exec.diff || exit 1
patch -p1 <../patches/syscall.diff || exit 1
patch -p1 <../patches/configure.diff || exit 1
patch -p1 <../patches/memfd.diff || exit 1

echo "[+] Patching done."

# --enable-pie seems to give a couple of exec's a second performance
# improvement, much to my surprise. Not sure how universal this is..

CFLAGS="-O3 -ggdb" ./configure --disable-system \
  --enable-linux-user --disable-gtk --disable-sdl --disable-vnc \
  --target-list="${CPU_TARGET}-linux-user" --enable-pie --enable-kvm || exit 1

echo "[+] Configuration complete."

echo "[*] Attempting to build QEMU (fingers crossed!)..."

make || exit 1

echo "[+] Build process successful!"

echo "[*] Copying binary..."

cp -f "${CPU_TARGET}-linux-user/qemu-${CPU_TARGET}" "../../afl-qemu-trace" || exit 1

cd ..
ls -l ../afl-qemu-trace || exit 1

echo "[+] Successfully created '../afl-qemu-trace'."

if [ "$ORIG_CPU_TARGET" = "" ]; then

  echo "[*] Testing the build..."

  cd ..

  make >/dev/null || exit 1

  gcc test-instr.c -o test-instr || exit 1

  unset AFL_INST_RATIO

  # We shouldn't need the /dev/null hack because program isn't compiled with any
  # optimizations.
  echo 0 | ./afl-showmap -m none -Q -q -o .test-instr0 ./test-instr || exit 1
  echo 1 | ./afl-showmap -m none -Q -q -o .test-instr1 ./test-instr || exit 1

  rm -f test-instr
```

```
  cmp -s .test-instr0 .test-instr1
  DR="$?"

  rm -f .test-instr0 .test-instr1

  if [ "$DR" = "0" ]; then

    echo "[-] Error: afl-qemu-trace instrumentation doesn't seem to work!"
    exit 1

  fi

  echo "[+] Instrumentation tests passed. "
  echo "[+] All set, you can now use the -Q mode in afl-fuzz!"

else

  echo "[!] Note: can't test instrumentation when CPU_TARGET set."
  echo "[+] All set, you can now (hopefully) use the -Q mode in afl-fuzz!"

fi

exit 0
```

> patches

afl-qemu-cpu-inl.h

```
/*
  Copyright 2015 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/

/*
   american fuzzy lop - high-performance binary-only instrumentation
   -----------------------------------------------------------------

   Written by Andrew Griffiths <agriffiths@google.com> and
              Michal Zalewski <lcamtuf@google.com>

   Idea & design very much by Andrew Griffiths.

   This code is a shim patched into the separately-distributed source
   code of QEMU 2.10.0. It leverages the built-in QEMU tracing functionality
   to implement AFL-style instrumentation and to take care of the remaining
   parts of the AFL fork server logic.
```

```
   The resulting QEMU binary is essentially a standalone instrumentation
   tool; for an example of how to leverage it for other purposes, you can
   have a look at afl-showmap.c.
*/

#include <sys/shm.h>
#include "../../config.h"

/*************************
 * VARIOUS AUXILIARY STUFF *
 *************************/

/* A snippet patched into tb_find_slow to inform the parent process that
   we have hit a new block that hasn't been translated yet, and to tell
   it to translate within its own context, too (this avoids translation
   overhead in the next forked-off copy). */

#define AFL_QEMU_CPU_SNIPPET1 do { \
    afl_request_tsl(pc, cs_base, flags); \
  } while (0)

/* 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)

/* We use one additional file descriptor to relay "needs translation"
   messages between the child and the fork server. */

#define TSL_FD (FORKSRV_FD - 1)

/* This is equivalent to afl-as.h: */

static unsigned char *afl_area_ptr;

/* Exported variables populated by the code patched into elfload.c: */

abi_ulong afl_entry_point, /* ELF entry point (_start) */
          afl_start_code,  /* .text start pointer      */
          afl_end_code;    /* .text end pointer        */

/* Set in the child process in forkserver mode: */

static unsigned char afl_fork_child;
unsigned int afl_forksrv_pid;

/* Instrumentation ratio: */

static unsigned int afl_inst_rms = MAP_SIZE;

/* Function declarations. */
```

```
static void afl_setup(void);
static void afl_forkserver(CPUState*);
static inline void afl_maybe_log(abi_ulong);

static void afl_wait_tsl(CPUState*, int);
static void afl_request_tsl(target_ulong, target_ulong, uint64_t);

/* Data structure passed around by the translate handlers: */

struct afl_tsl {
  target_ulong pc;
  target_ulong cs_base;
  uint64_t flags;
};

/* Some forward decls: */

TranslationBlock *tb_htable_lookup(CPUState*, target_ulong, target_ulong, uint32_t);
static inline TranslationBlock *tb_find(CPUState*, TranslationBlock*, int);

/*************************
 * ACTUAL IMPLEMENTATION *
 *************************/

/* Set up SHM region and initialize other stuff. */

static void afl_setup(void) {

  char *id_str = getenv(SHM_ENV_VAR),
       *inst_r = getenv("AFL_INST_RATIO");

  int shm_id;

  if (inst_r) {

    unsigned int r;

    r = atoi(inst_r);

    if (r > 100) r = 100;
    if (!r) r = 1;

    afl_inst_rms = MAP_SIZE * r / 100;

  }

  if (id_str) {

    shm_id = atoi(id_str);
    afl_area_ptr = shmat(shm_id, NULL, 0);

    if (afl_area_ptr == (void*)-1) exit(1);

    /* With AFL_INST_RATIO set to a low value, we want to touch the bitmap
       so that the parent doesn't give up on us. */

    if (inst_r) afl_area_ptr[0] = 1;


  }
```

```
  if (getenv("AFL_INST_LIBS")) {

    afl_start_code = 0;
    afl_end_code   = (abi_ulong)-1;

  }

  /* pthread_atfork() seems somewhat broken in util/rcu.c, and I'm
     not entirely sure what is the cause. This disables that
     behaviour, and seems to work alright? */

  rcu_disable_atfork();

}


/* Fork server logic, invoked once we hit _start. */

static void afl_forkserver(CPUState *cpu) {

  static unsigned char tmp[4];

  if (!afl_area_ptr) return;

  /* Tell the parent that we're alive. If the parent doesn't want
     to talk, assume that we're not running in forkserver mode. */

  if (write(FORKSRV_FD + 1, tmp, 4) != 4) return;

  afl_forksrv_pid = getpid();

  /* All right, let's await orders... */

  while (1) {

    pid_t child_pid;
    int status, t_fd[2];

    /* Whoops, parent dead? */

    if (read(FORKSRV_FD, tmp, 4) != 4) exit(2);

    /* Establish a channel with child to grab translation commands. We'll
       read from t_fd[0], child will write to TSL_FD. */

    if (pipe(t_fd) || dup2(t_fd[1], TSL_FD) < 0) exit(3);
    close(t_fd[1]);

    child_pid = fork();
    if (child_pid < 0) exit(4);

    if (!child_pid) {

      /* Child process. Close descriptors and run free. */

      afl_fork_child = 1;
      close(FORKSRV_FD);
      close(FORKSRV_FD + 1);
      close(t_fd[0]);
```

```
      return;

    }

    /* Parent. */

    close(TSL_FD);

    if (write(FORKSRV_FD + 1, &child_pid, 4) != 4) exit(5);

    /* Collect translation requests until child dies and closes the pipe. */

    afl_wait_tsl(cpu, t_fd[0]);

    /* Get and relay exit status to parent. */

    if (waitpid(child_pid, &status, 0) < 0) exit(6);
    if (write(FORKSRV_FD + 1, &status, 4) != 4) exit(7);

  }

}


/* The equivalent of the tuple logging routine from afl-as.h. */

static inline void afl_maybe_log(abi_ulong cur_loc) {

  static __thread abi_ulong prev_loc;

  /* Optimize for cur_loc > afl_end_code, which is the most likely case on
     Linux systems. */

  if (cur_loc > afl_end_code || cur_loc < afl_start_code || !afl_area_ptr)
    return;

  /* Looks like QEMU always maps to fixed locations, so ASAN is not a
     concern. Phew. But instruction addresses may be aligned. Let's mangle
     the value to get something quasi-uniform. */

  cur_loc  = (cur_loc >> 4) ^ (cur_loc << 8);
  cur_loc &= MAP_SIZE - 1;

  /* Implement probabilistic instrumentation by looking at scrambled block
     address. This keeps the instrumented locations stable across runs. */

  if (cur_loc >= afl_inst_rms) return;

  afl_area_ptr[cur_loc ^ prev_loc]++;
  prev_loc = cur_loc >> 1;

}


/* This code is invoked whenever QEMU decides that it doesn't have a
   translation of a particular block and needs to compute it. When this happens,
   we tell the parent to mirror the operation, so that the next fork() has a
   cached copy. */

static void afl_request_tsl(target_ulong pc, target_ulong cb, uint64_t flags) {
```

```
  struct afl_tsl t;

  if (!afl_fork_child) return;

  t.pc      = pc;
  t.cs_base = cb;
  t.flags   = flags;

  if (write(TSL_FD, &t, sizeof(struct afl_tsl)) != sizeof(struct afl_tsl))
    return;

}

/* This is the other side of the same channel. Since timeouts are handled by
   afl-fuzz simply killing the child, we can just wait until the pipe breaks. */

static void afl_wait_tsl(CPUState *cpu, int fd) {

  struct afl_tsl t;
  TranslationBlock *tb;

  while (1) {

    /* Broken pipe means it's time to return to the fork server routine. */

    if (read(fd, &t, sizeof(struct afl_tsl)) != sizeof(struct afl_tsl))
      break;

    tb = tb_htable_lookup(cpu, t.pc, t.cs_base, t.flags);

    if(!tb) {
      mmap_lock();
      tb_lock();
      tb_gen_code(cpu, t.pc, t.cs_base, t.flags, 0);
      mmap_unlock();
      tb_unlock();
    }

  }

  close(fd);

}
```

configure.diff

```
--- qemu-2.10.0-clean/configure 2019-08-01 23:04:12.511396481 +0200
+++ qemu-2.10.0/configure   2019-08-01 23:04:32.936429232 +0200
@@ -3855,7 +3855,7 @@
 # check if memfd is supported
 memfd=no
 cat > $TMPC << EOF
-#include <sys/memfd.h>
+#include <sys/mman.h>

 int main(void)
 {
```

cpu-exec.diff

```
--- qemu-2.10.0-rc3-clean/accel/tcg/cpu-exec.c  2017-08-15 11:39:41.000000000 -0700
+++ qemu-2.10.0-rc3/accel/tcg/cpu-exec.c    2017-08-22 14:34:55.868730680 -0700
@@ -36,6 +36,8 @@
 #include "sysemu/cpus.h"
 #include "sysemu/replay.h"

+#include "../patches/afl-qemu-cpu-inl.h"
+
 /* -icount align implementation. */

 typedef struct SyncClocks {
@@ -144,6 +146,8 @@
     int tb_exit;
     uint8_t *tb_ptr = itb->tc_ptr;

+    AFL_QEMU_CPU_SNIPPET2;
+
     qemu_log_mask_and_addr(CPU_LOG_EXEC, itb->pc,
                            "Trace %p [%d: " TARGET_FMT_lx "] %s\n",
                            itb->tc_ptr, cpu->cpu_index, itb->pc,
@@ -365,6 +369,7 @@
             if (!tb) {
                 /* if no translated code available, then translate it now */
                 tb = tb_gen_code(cpu, pc, cs_base, flags, 0);
+                AFL_QEMU_CPU_SNIPPET1;
             }

             mmap_unlock();
```

elfload.diff

```
--- qemu-2.10.0-rc3-clean/linux-user/elfload.c  2017-08-15 11:39:41.000000000 -0700
+++ qemu-2.10.0-rc3/linux-user/elfload.c    2017-08-22 14:33:57.397127516 -0700
@@ -20,6 +20,8 @@

 #define ELF_OSABI   ELFOSABI_SYSV

+extern abi_ulong afl_entry_point, afl_start_code, afl_end_code;
+
 /* from personality.h */

 /*
```

```
@@ -2085,6 +2087,8 @@
     info->brk = 0;
     info->elf_flags = ehdr->e_flags;

+    if (!afl_entry_point) afl_entry_point = info->entry;
+
     for (i = 0; i < ehdr->e_phnum; i++) {
         struct elf_phdr *eppnt = phdr + i;
         if (eppnt->p_type == PT_LOAD) {
@@ -2118,9 +2122,11 @@
             if (elf_prot & PROT_EXEC) {
                 if (vaddr < info->start_code) {
                     info->start_code = vaddr;
+                    if (!afl_start_code) afl_start_code = vaddr;
                 }
                 if (vaddr_ef > info->end_code) {
                     info->end_code = vaddr_ef;
+                    if (!afl_end_code) afl_end_code = vaddr_ef;
                 }
             }
             if (elf_prot & PROT_WRITE) {
```

memfd.diff

```
--- qemu-2.10.0-clean/util/memfd.c  2019-08-01 23:04:12.562396563 +0200
+++ qemu-2.10.0/util/memfd.c    2019-08-01 23:06:47.882646792 +0200
@@ -31,9 +31,7 @@

 #include "qemu/memfd.h"

-#ifdef CONFIG_MEMFD
-#include <sys/memfd.h>
-#elif defined CONFIG_LINUX
+#if defined CONFIG_LINUX && !defined CONFIG_MEMFD
 #include <sys/syscall.h>
 #include <asm/unistd.h>
```

syscall.diff

```
--- qemu-2.10.0-rc3-clean/linux-user/syscall.c  2017-08-15 11:39:41.000000000 -0700
+++ qemu-2.10.0-rc3/linux-user/syscall.c    2017-08-22 14:34:03.193088186 -0700
@@ -116,6 +116,8 @@

 #include "qemu.h"

+extern unsigned int afl_forksrv_pid;
+
 #ifndef CLONE_IO
 #define CLONE_IO                0x80000000      /* Clone io context */
 #endif
@@ -11688,8 +11690,21 @@
         break;

     case TARGET_NR_tgkill:
-        ret = get_errno(safe_tgkill((int)arg1, (int)arg2,
+        if ((int)arg1 == getpid())
+            ret = get_errno(safe_tgkill((int)arg1, (int)arg2,
+                            target_to_host_signal(arg3)));
+        else {
+            ret = -EPERM;
+            if ((int)arg1 == afl_forksrv_pid && afl_forksrv_pid &&
+                (int)arg2 == SIGKILL) {
+                ret = get_errno(safe_tgkill((int)arg1, (int)arg2,
                             target_to_host_signal(arg3)));
+            }
+        }
+        break;
+
+    case TARGET_NR_tkill:
+        ret = get_errno(safe_tkill((int)arg1, target_to_host_signal(arg2)));
         break;
```

```
-                         target_to_host_signal(arg3)));
+
+        {
+          int pid  = (int)arg1,
+              tgid = (int)arg2,
+              sig  = (int)arg3;
+
+          /* Not entirely sure if the below is correct for all architectures. */
+
+          if(afl_forksrv_pid && afl_forksrv_pid == pid && sig == SIGABRT)
+              pid = tgid = getpid();
+
+          ret = get_errno(safe_tgkill(pid, tgid, target_to_host_signal(sig)));
+
+        }
+
          break;

 #ifdef TARGET_NR_set_robust_list
```

test-instr.c

```c
/*
  Copyright 2014 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/


/*
   american fuzzy lop - a trivial program to test the build
   --------------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

  char buf[8];

  if (read(0, buf, 8) < 1) {
    printf("Hum?\n");
    exit(1);
  }
```

```
  if (buf[0] == '0')
    printf("Looks like a zero to me!\n");
  else
    printf("A non-zero value? How quaint!\n");

  exit(0);

}
```

test-libfuzzer-target.c

```
/*
  Copyright 2019 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/


/*
   american fuzzy lop - a trivial program to test libFuzzer target fuzzing.
   ------------------------------------------------------------------------

   Initially written and maintained by Michal Zalewski.
*/

#include <stddef.h>
#include <stdint.h>
#include <stdio.h>

// TODO(metzman): Create a test/ directory to store this and other similar
// files.
int LLVMFuzzerTestOneInput(uint8_t* buf, size_t size) {
  if (size < 2)
    return 0;

  if (buf[0] == '0')
    printf("Looks like a zero to me!\n");
  else
    printf("A non-zero value? How quaint!\n");

  return 0;
}
```

types.h

```
/*
  Copyright 2013 Google LLC All rights reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
*/


/*
   american fuzzy lop - type definitions and minor macros
   ------------------------------------------------------

   Written and maintained by Michal Zalewski <lcamtuf@google.com>
*/

#ifndef _HAVE_TYPES_H
#define _HAVE_TYPES_H

#include <stdint.h>
#include <stdlib.h>

typedef uint8_t  u8;
typedef uint16_t u16;
typedef uint32_t u32;

/*

   Ugh. There is an unintended compiler / glibc #include glitch caused by
   combining the u64 type an %llu in format strings, necessitating a workaround.

   In essence, the compiler is always looking for 'unsigned long long' for %llu.
   On 32-bit systems, the u64 type (aliased to uint64_t) is expanded to
   'unsigned long long' in <bits/types.h>, so everything checks out.

   But on 64-bit systems, it is #ifdef'ed in the same file as 'unsigned long'.
   Now, it only happens in circumstances where the type happens to have the
   expected bit width, *but* the compiler does not know that... and complains
   about 'unsigned long' being unsafe to pass to %llu.

 */

#ifdef __x86_64__
typedef unsigned long long u64;
#else
typedef uint64_t u64;
#endif /* ^__x86_64__ */

typedef int8_t   s8;
typedef int16_t  s16;
typedef int32_t  s32;
typedef int64_t  s64;
```

```
#ifndef MIN
#  define MIN(_a,_b) ((_a) > (_b) ? (_b) : (_a))
#  define MAX(_a,_b) ((_a) > (_b) ? (_a) : (_b))
#endif /* !MIN */

#define SWAP16(_x) ({ \
    u16 _ret = (_x); \
    (u16)((_ret << 8) | (_ret >> 8)); \
  })

#define SWAP32(_x) ({ \
    u32 _ret = (_x); \
    (u32)((_ret << 24) | (_ret >> 24) | \
          ((_ret << 8) & 0x00FF0000) | \
          ((_ret >> 8) & 0x0000FF00)); \
  })

#ifdef AFL_LLVM_PASS
#  define AFL_R(x) (random() % (x))
#else
#  define R(x) (random() % (x))
#endif /* ^AFL_LLVM_PASS */

#define STRINGIFY_INTERNAL(x) #x
#define STRINGIFY(x) STRINGIFY_INTERNAL(x)

#define MEM_BARRIER() \
  __asm__ volatile("" ::: "memory")

#define likely(_x)   __builtin_expect(!!(_x), 1)
#define unlikely(_x)  __builtin_expect(!!(_x), 0)

#endif /* ! _HAVE_TYPES_H */
```