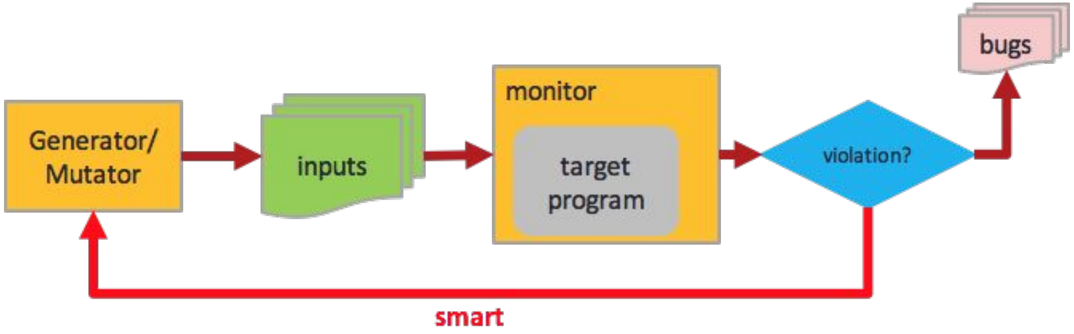
AFL Basics

Chao Zhang

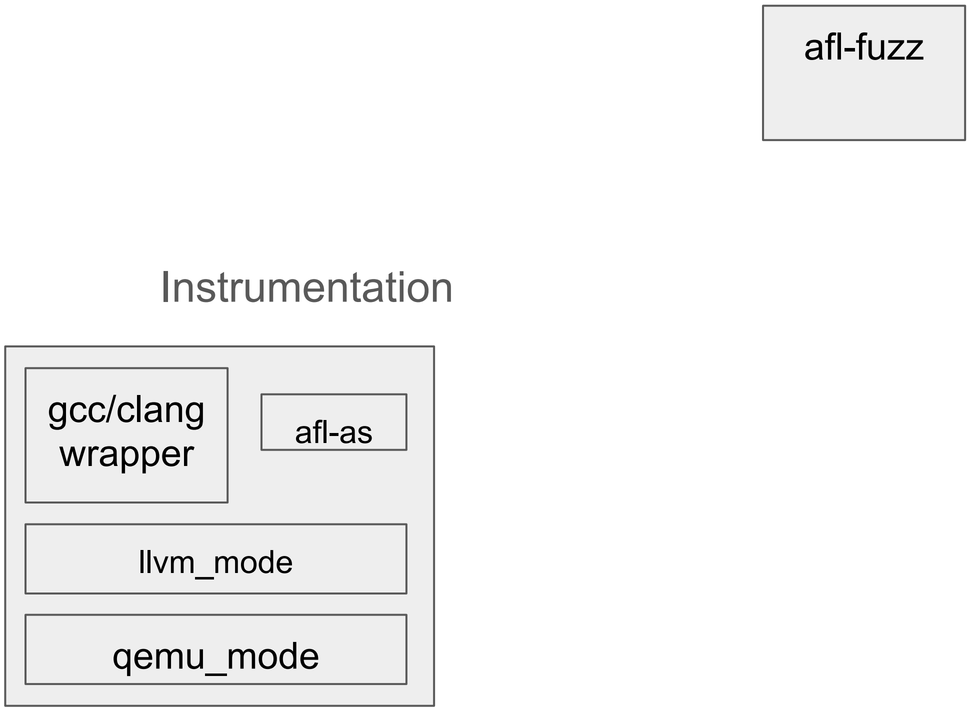
Smart Fuzzer



AFL

- Mutate from SEED inputs, and test them *one by one*.
  - How to mutate?
    - Not our concern, see http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html
  - How to pick seeds one by one?
    - strategy is needed. AFLfast changed the strategy a little bit.
- Keep GOOD mutated inputs, put them into SEED pool.
  - What is GOOD?
    - Hit a new edge, or
    - Hit an edge with different count, i.e., ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
- How to track edge coverage, and detect GOOD?
  - Two bitmaps (arrays of bytes, length = MAP_SIZE, 64k default)
    - trace_bits
    - virgin_bits
  - See following slides

Workflow of AFL



AFL instrumentation

ID is instrumented before each basic block, usually random, in three modes:

- llvm_mode: *afl-llvm-pass.so.cc*,
  - cur_loc = R(MAP_SIZE); where R is random
- gcc: *afl-as.c*,
  - fprintf(outf, use_64bit ? trampoline_fmt_64 : trampoline_fmt_32, R(MAP_SIZE));
- qemu_mode: *afl-qemu-cpu-inl.h*,
  - cur_loc = TB->pc;
  - cur_loc = (cur_loc >> 4) ^ (cur_loc << 8);
  - cur_loc &= MAP_SIZE - 1;

How the instrumentation is done?

- llvm_mode:
  - use afl-clang/afl-clang++ to compile source code, which will invoke code defined in llvm_mode/
- gcc:
  - use afl-gcc/afl-g++ to compile source code, which will generate asm code, and then invoke afl-as
- qemu_mode:
  - use patched qemu, which will instrument ID when BB is translated to TCG blocks.

AFL coverage tracking

(All three instrumentation modes perform the exact same tracking operations.)

Hash for each tuple of BB (i.e., edge).

- cur_block ^ (prev_block >> 1)

The bitmap trace_bit tracks the hit-count of each tuple/edge

- Tuple/edge hash is the index to this bitmap array
- Once an edge is executed, the hit-count increases, trace_bit[hash]++

Note:

- The order of tuples' execution is ignored, not recorded in this bitmap.
- This bitmap is a shared memory between the fuzzer and the fuzzed app
  - The fuzzed app updates this bitmap, then
  - the fuzzer could inspect it to detect whether it is GOOD

We therefore interchangeably use tuple, edge, and hash.

AFL seed filtering

Bitmap virgin_bits tracks whether an edge is hit, and its hit-count range. For each byte (edge/hash)

- If bit ***k*** is set, it means no test case hits this edge for a range of times
  - 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
  - See classify_counts, it will reduce the hit-count to 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7
  - See has_new_bits, **virgin = vir & ~cur;*
- Byte 0xFF means this edge (hash) has not been hit

This bitmap will be saved to file, and could be used later by AFL

- to resume previous fuzzing attempts with same seed inputs).

virgin_bits and trace_bits could be used to detect whether the new sample is GOOD or not

- has_new_bits tells whether it hits a new edge, or hits an edge with different count

AFL detail: execve

- dumb_mode
  - Run the target app in a child process using execve, with shared memory **trace_bits** (bitmap).
    - Parent process could check its exit status, using wait_pid, to recognize crashes.
    - The fuzzed app runs in a different address space, avoid conflicting with the fuzzer.
- forkserver
  - Run the target app in a child process using execve, with shared memory **trace_bits** (bitmap).
  - This child process will stop at the shim code instrumented, behaving like a forkserver
    - It will use fork() to test the fuzzed app from current state, no further execve needed.
    - It will receive commands from the parent process, and sends signal of fuzzed app to parent, using a pipe.

AFL detail: input & output

- app input

AFL usage	out_file	out_fd
-f	argument	0
@@	.cur_input	0
stdin	NULL	open(.cur_input) dup2() will bind it to child process' stdin

- when a new testcase is generated, and to be tested
  - write_to_testcase, then run_target
- app exits/crashes
  - dumb_mode, the fuzzer invokes wait_pid to check the child process' signal
  - forkserver,

AFL detail: forkserver

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

Problem: execve is still heavy-weight, and thus slow. Solution:

- Instrument a shim code to the binary, which
  - allows the heavy part of execve, and stops when hitting **main (**or **_start)** function of fuzzed app
    - How? see next slide
  - waits for the fuzzer's command to test, using the pipe fsrv_ctl_fd
  - once received a "go", it will fork a new process, to fuzz the target app
  - relays the child process' PID, as well as its exit status, to the fuzzer, using pipe fsrv_st_fd
  - and continues to wait for the fuzzer's command
- The fuzzer is the grandparent of the fuzzed app, could not use wait_pid to check
  - Shim will do for it
- The shim code could stop at a later point, rather than **main** **(**or **_start)**
  - llvm_mode/README.llvm

AFL detail: forkserver instrumentation

- llvm_mode
  - case 1: __AFL_INIT(), developers call this function at proper location, rather than at the default _start location.
    - afl-clang-fast defines __AFL_INIT via command line arguments, which
      - defines a signature DEFER_SIG in the app
      - calls __afl_manual_init
  - case 2: __afl_auto_init, afl-clang-fast automatically instrument this initializer function
    - it calls __afl_manual_init, if no DEFER_ENV_VAR is set (avoid conflicting with __AFL_INIT).
      - ther fuzzer will use check_binary to setup DEFER_ENV_VAR if DEFER_SIG exists in app
  - __afl_manual_init, calls
    - __afl_map_shm, and
    - __afl_start_forkserver
      - will check if the parent's forkserver pipe is open. If not, run regular mode, in case the user doesn't want forkserver mode
- gcc (afl-as.h)
  - afl-gcc.c will instrument each BB with trampoline_fmt, which will then invoke
  - __afl_maybe_log, this is the default entry (for both forkserver and fuzzed app)
  - it will call __afl_setup, if the shared memory is not setup, which
    - calls shmat to setup share memory
    - flows into __afl_forkserver, which will check whether the parent's forkserver pipe is open, in case the user doesn't want forkserver
  - it will compute edge hash, and update the shared bitmap
- qemu_mode:
  - it will afl_setup and afl_forkserver, when the afl_entry_point (i.e., _start) is hit.

AFL detail: persistent mode

http://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html

- For applications with a big loop (e.g., Windows message loop), it would be good to test from the loop entry, rather than from the program entry
- forkserver already provides an optimization, but it still needs fork, which is not free
- ideally, we should keep the child process of the fuzzed app, and start testing again from the loop entry

AFL detail: parallel execution

AFL detail: crash exploration

https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html

Given a crash sample, AFL could be driven to find more crash samples (-C), which is

- Related to original crash, but somewhat different

Use this mode, we could filter high-value crashes fast, without debugging it.

AFL detail: more features

https://lcamtuf.blogspot.com/2015/05/lesser-known-features-of-afl-fuzz.html

AFL seed pool

- pool head pointer (won’t change): queue
- pool current seed pointer (picked for mutation): queue_cur
    - Note: each time a seed is picked, mutated and fuzzed, the pool may be extended with GOOD seeds
    - Note: there are several mutation phase: deterministic, …, havoc
    -
- pool last seed pointer (always at top): queue_top
- energy: how many mutations should be generated in the havoc phase

https://github.com/mboehme/aflfast/commit/7819aeccfb74afad1c475ea49b92d27f536e1c51#diff-7055551313949db06ed69a1c16797a15L6394

AFL vs. AFLfast

AFL algorithm on seed picking and mutation:

- each seed is picked in order, following “next” pointer
  - `queue_cur = queue_cur->next;`
- this seed is mutated and fuzzed using fuzz_one
  - this seed may be skipped due to favor or random
  - an energy is assigned to it using `calculate_score`
- once the seed pool end is reached, it goes over to the seed pool head again
- `calculate_score`
  - a ***constant*** energy is assigned to a same seed

AFLfast’s algorithm:

- next seed is picked based on its fuzz_level
  - `chooseNext_queue_cur`
    - smaller fuzz_level is favored
    - less-frequent path (`getPaths`) is favored
- this seed is mutated and fuzzed using fuzz_one
  - this seed may be skipped due to favor or random
  - an energy is assigned to it using `calculate_score`
- once the seed pool end is reached, it goes over to the seed pool head again
- `calculate_score`
  - a ***different*** energy is assigned to a same seed, depending on fuzz_level