source code

Alphuzz

1. installation

```
$ git clone https://github.com/artifact11/Alphuzz.git
```

```
Cloning into 'Alphuzz'...
remote: Enumerating objects: 291, done.
remote: Counting objects: 100% (291/291), done.
remote: Compressing objects: 100% (187/187), done.
remote: Total 291 (delta 39), reused 288 (delta 39), pack-reused 0
Receiving objects: 100% (291/291), 24.78 MiB | 3.08 MiB/s, done.
Resolving deltas: 100% (39/39), done.
Checking connectivity... done.
```

```
$ cd Alphuzz
$ make
```

```
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[+] All right, the instrumentation seems to be working!
[+] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[+] All done! Be sure to review README - it's pretty short and useful.
NOTE: If you can read this, your terminal probably uses white background.
This will make the UI hard to read. See docs/status_screen.txt for advice.
```

You can choose to install Alphuzz or not.

```
$ sudo make install
```

2. Instrumenting programs for use with Alphuzz

If you're having a hard time reading such a large section of text, please refer to the examples provided.

When source code is available, instrumentation can be injected by a companion tool that works as a drop-in replacement for gcc or clang in any standard build process for third-party code.

The instrumentation has a fairly modest performance impact; in conjunction with other optimizations implemented by afl-fuzz, most programs can be fuzzed as fast or even faster than possible with traditional tools.

The correct way to recompile the target program may vary depending on the specifics of the build process, but a nearly-universal approach would be:

```
$ CC=/path/to/Alphuzz/afl-gcc ./configure
$ make clean all
```

For C++ programs, you'd would also want to set CXX=/path/to/afl/afl-g++.

The clang wrappers (afl-clang and afl-clang++) can be used in the same way; clang users may also opt to leverage a higher-performance instrumentation mode, as described in llvm_mode/README.llvm.

When testing libraries, you need to find or write a simple program that reads data from stdin or from a file and passes it to the tested library. In such a case, it is essential to link this executable against a static version of the instrumented library, or to make sure that the correct .so file is loaded at runtime (usually by setting LD_LIBRARY_PATH). The simplest option is a static build, usually possible via:

```
$ CC=/path/to/Alphuzz/afl-gcc ./configure --disable-shared
```

Setting `AFL_HARDEN=1` when calling `make` will cause the CC wrapper to automatically enable code hardening options that make it easier to detect simple memory bugs. Libdislocator, a helper library included with AFL (see libdislocator/README.dislocator) can help uncover heap corruption issues, too.

PS. ASAN users are advised to review notes_for_asan.txt file for important caveats.

Example

Here, we use the source code of readelf to demonstrate how to perform instrumentation and how to use Alphuzz to test the program.

First, download the source code and unzip the package.

```
$ wget https://ftp.gnu.org/gnu/binutils/binutils-2.39.tar.gz
$ tar xf binutils-2.39.tar.gz
$ apt install texinfo
```

Then use afl-gcc and afl-g++ to compile the program.

```
$ cd binutils-2.39
$ CC=~/path/to/Alphuzz/afl-gcc CXX=~/.../afl-g++ ./configure
$ make
```

If you have output similar to the following image, you are compiling the program with afl-gcc.

```
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 274 locations (64-bit, non-hardened mode, ratio 100%).
  GEN      eelf32_x86_64.c
```

Now, the instrumentation is complete. We can use Alphuzz to test the program. The fuzzing process itself is carried out by the afl-fuzz utility. Sometimes, afl-fuzz requires some permissions to run.

```
$ echo core >/proc/sys/kernel/core_pattern
$ cd /sys/devices/system/cpu
$ echo performance | tee cpu*/cpufreq/scaling_governor
```

Then, use the following command to test the program.

```
$ mkdir in
$ cp /usr/bin/objdump
$ ./Alphuzz/afl-fuzz -i ./in -o ./out -- ./binutils-2.39/binutils/readelf -d @@
```

```
yiru@yiru-virtual-machine:~/artifact$ mkdir in
yiru@yiru-virtual-machine:~/artifact$ cp /usr/bin/objdump ./in
yiru@yiru-virtual-machine:~/artifact$ ./Alphuzz/afl-fuzz -i ./in -o ./out -- ./b
inutils-2.39/binutils/readelf -d @@
```

```
                american fuzzy lop 2.52b (readelf)

┌─ process timing ────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 1 min, 35 sec │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 1 min, 32 sec │  total paths : 59     │
│ last uniq crash : none seen yet     │ uniq crashes : 0      │
│  last uniq hang : none seen yet     │   uniq hangs : 0      │
├─ cycle progress ────────────────────┼─ map coverage ────────┤
│  now processing : 0* (0.00%)        │   map density : 0.79% / 1.28% │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.48 bits/tuple │
├─ stage progress ────────────────────┼─ findings in depth ───┤
│  now trying : bitflip 1/1           │ favored paths : 0 (0.00%)     │
│ stage execs : 119k/2.85M (4.20%)    │  new edges on : 39 (66.10%)   │
│ total execs : 121k                  │ total crashes : 0 (0 unique)  │
│  exec speed : 1282/sec              │  total tmouts : 2 (2 unique)  │
├─ fuzzing strategy yields ──────────┴─ path geometry ──────┤
│   bit flips : 0/0, 0/0, 0/0         │    levels : 2         │
│  byte flips : 0/0, 0/0, 0/0         │   pending : 59        │
│ arithmetics : 0/0, 0/0, 0/0         │  pend fav : 0         │
│  known ints : 0/0, 0/0, 0/0         │ own finds : 58        │
│  dictionary : 0/0, 0/0, 0/0         │  imported : n/a       │
│       havoc : 0/0, 0/0              │ stability : 100.00%   │
│        trim : 0.00%/1379, n/a       ├───────────────────────┘
└─────────────────────────────────────┘             [cpu000: 13%]
```

3. Instrumenting binary-only apps

When source code is *NOT* available, the fuzzer offers experimental support for fast, on-the-fly instrumentation of black-box binaries. This is accomplished with a version of QEMU running in the lesser-known "user space emulation" mode.

QEMU is a project separate from AFL, but you can conveniently build the feature by doing:

```
$ cd qemu_mode
$ ./build_qemu_support.sh
```

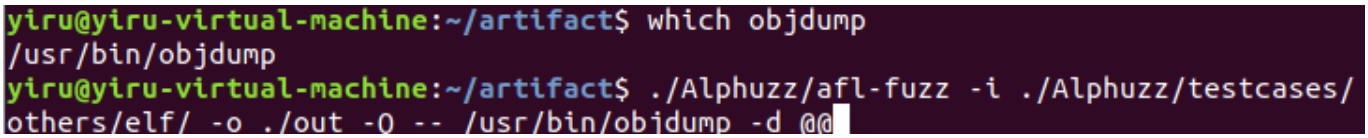
For additional instructions and caveats, see qemu_mode/README.qemu.

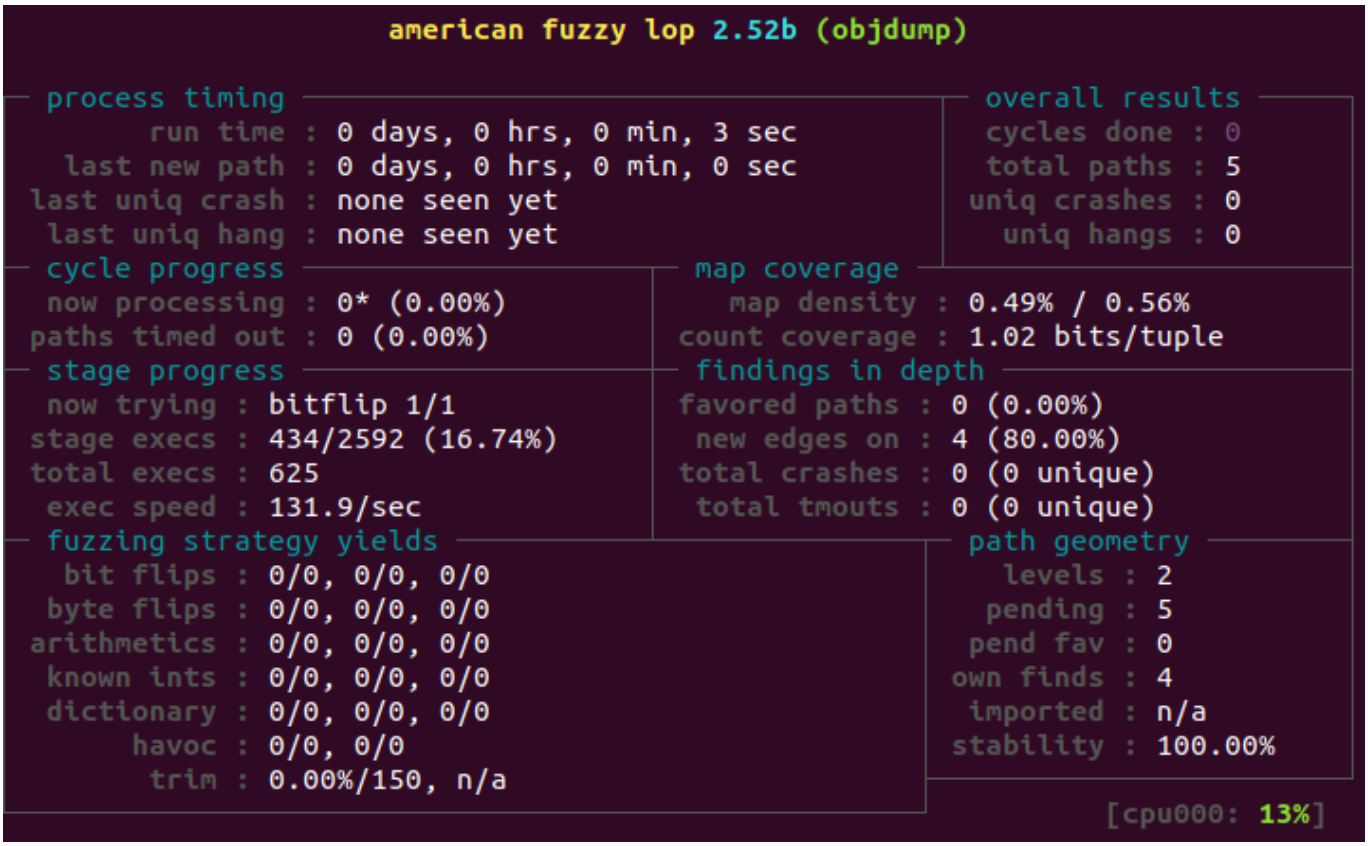
The mode is approximately 2-5x slower than compile-time instrumentation, is less conductive to parallelization, and may have some other quirks.

Example

Here, we use objdump to demenstrate how to test binaries in qemu_mode.

```
$ which objdump
$ ./Alphuzz/afl-fuzz -i ./Alphuzz/testcases/others/elf/ -o ./out -Q --
/usr/bin/objdump -d @@
```





4. Fuzzing binaries

The fuzzing process itself is carried out by the afl-fuzz utility. This program requires a read-only directory with initial test cases, a separate place to store its findings, plus a path to the binary to test.

For target binaries that accept input directly from stdin, the usual syntax is:

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program [...params...]
```

For programs that take input from a file, use '@@' to mark the location in the target's command line where the input file name should be placed. The fuzzer will substitute this for you:

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

You can also use the -f option to have the mutated data written to a specific file. This is useful if the program expects a particular file extension or so.

Non-instrumented binaries can be fuzzed in the QEMU mode (add `-Q` in the command line) or in a traditional, blind-fuzzer mode (specify `-n`).

You can use `-t` and `-m` to override the default timeout and memory limit for the executed process; rare examples of targets that may need these settings touched include compilers and video decoders.

Note that afl-fuzz starts by performing an array of deterministic fuzzing steps, which can take several days, but tend to produce neat test cases. If you want quick & dirty results right away - akin to zzuf and other traditional fuzzers - add the `-d` option to the command line.

5. Interpreting output

See the status_screen.txt file for information on how to interpret the displayed stats and monitor the health of the process. Be sure to consult this file especially if any UI elements are highlighted in red.

The fuzzing process will continue until you press Ctrl-C. At minimum, you want to allow the fuzzer to complete one queue cycle, which may take anywhere from a couple of hours to a week or so.

There are three subdirectories created within the output directory and updated in real time:

- queue/
    - test cases for every distinctive execution path, plus all the starting files given by the user.
- crashes/
    - unique test cases that cause the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT). The entries are grouped by the received signal.
- hangs/

- unique test cases that cause the tested program to time out. The default time limit before something is classified as a hang is the larger of 1 second and the value of the -t parameter. The value can be fine-tuned by setting AFL_HANG_TMOUT, but this is rarely necessary.

Crashes and hangs are considered "unique" if the associated execution paths involve any state transitions not seen in previously-recorded faults. If a single bug can be reached in multiple ways, there will be some count inflation early in the process, but this should quickly taper off.

The file names for crashes and hangs are correlated with parent, non-faulting queue entries. This should help with debugging.

When you can't reproduce a crash found by afl-fuzz, the most likely cause is that you are not setting the same memory limit as used by the tool. Try:

```
$ LIMIT_MB=50
$ ( ulimit -Sv $[LIMIT_MB << 10]; /path/to/tested_binary ... )
```

Change LIMIT_MB to match the -m parameter passed to afl-fuzz. On OpenBSD, also change -Sv to -Sd.

Any existing output directory can be also used to resume aborted jobs; try:

```
$ ./afl-fuzz -i- -o existing_output_dir [...etc...]
```