AFL Documentation

Release 2.53b

Michal Zalewski

Jul 26, 2019

Contents

CHAPTER 1

What is AFL?

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.

1.1 AFL Quick Start Guide

You should read *Instrumenting programs for AFL* and *Fuzzing with afl-fuzz*. They're pretty short. If you really can't, here's how to hit the ground running:

1) Compile AFL with **`make`**. If build fails, see *Installation instructions* for tips.

2) Find or write a reasonably fast and simple program that takes data from a file or stdin, processes it in a test-worthy way, then exits cleanly. If testing a network service, modify it to run in the foreground and read from stdin. When fuzzing a format that uses checksums, comment out the checksum verification code, too.

   The program *must* crash properly when a fault is encountered. Watch out for custom SIGSEGV or SIGABRT handlers and background processes. For tips on detecting non-crashing flaws, see *Going beyond crashes*.

3) Compile the program / library to be fuzzed using **`afl-gcc`**. A common way to do this would be:

```
CC=/path/to/afl-gcc CXX=/path/to/afl-g++ ./configure --disable-shared
make clean all
```

   If program build fails, ping <afl-users@googlegroups.com>.

4) Get a small but valid input file that makes sense to the program. When fuzzing verbose syntax (SQL, HTTP, etc), create a dictionary as described in dictionaries/README.dictionaries, too.

5) If the program reads from stdin, run **`afl-fuzz`** like so:

```
./afl-fuzz -i testcase_dir -o findings_dir -- \
   /path/to/tested/program [...program's cmdline...]
```

If the program takes input from a file, you can put `@@` in the program's command line; AFL will put an auto-generated file name in there for you.

6) Investigate anything shown in red in the fuzzer UI by promptly consulting "*Understanding the status screen*".

That's it. Sit back, relax, and - time permitting - try to skim through the following:

- *Motivation behind AFL* - A general introduction to AFL,
- *Performance Tips* - Simple tips on how to fuzz more quickly,
- *Understanding the status screen* - An explanation of the tidbits shown in the UI,
- *Tips for parallel fuzzing* - Advice on running AFL on multiple cores.

1.2 Motivation behind AFL

1.2.1 Challenges of guided fuzzing

Fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software; it is responsible for the vast majority of remote code execution and privilege escalation bugs found to date in security-critical software.

Unfortunately, fuzzing is also relatively shallow; blind, random mutations make it very unlikely to reach certain code paths in the tested code, leaving some vulnerabilities firmly outside the reach of this technique.

There have been numerous attempts to solve this problem. One of the early approaches - pioneered by Tavis Ormandy - is corpus distillation. The method relies on coverage signals to select a subset of interesting seeds from a massive, high-quality corpus of candidate files, and then fuzz them by traditional means. The approach works exceptionally well, but requires such a corpus to be readily available. In addition, block coverage measurements provide only a very simplistic understanding of program state, and are less useful for guiding the fuzzing effort in the long haul.

Other, more sophisticated research has focused on techniques such as program flow analysis ("concolic execution"), symbolic execution, or static analysis. All these methods are extremely promising in experimental settings, but tend to suffer from reliability and performance problems in practical uses - and currently do not offer a viable alternative to "dumb" fuzzing techniques.

1.2.2 The AFL approach

American Fuzzy Lop is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow.

Simplifying a bit, the overall algorithm can be summed up as:

1) Load user-supplied initial test cases into the queue,
2) Take next input file from the queue,
3) Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
4) Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
5) If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
6) Go to 2.

The discovered test cases are also periodically culled to eliminate ones that have been obsoleted by newer, higher-coverage finds; and undergo several other instrumentation-driven effort minimization steps.

As a side result of the fuzzing process, the tool creates a small, self-contained corpus of interesting test cases. These are extremely useful for seeding other, labor- or resource-intensive testing regimes - for example, for stress-testing browsers, office applications, graphics suites, or closed-source tools.

The fuzzer is thoroughly tested to deliver out-of-the-box performance far superior to blind fuzzing or coverage-only tools.

1.3 Instrumenting programs for AFL

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/afl/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/afl/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.

Note: ASAN users are advised to review *Using ASAN with AFL* for important caveats.

1.3.1 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.

1.4 Fuzzing with afl-fuzz

To operate correctly, the fuzzer requires one or more starting file that contains a good example of the input data normally expected by the targeted application. There are two basic rules:

- Keep the files small. Under 1 kB is ideal, although not strictly necessary. For a discussion of why size matters, see *Performance Tips*.
- Use multiple test cases only if they are functionally different from each other. There is no point in using fifty different vacation photos to fuzz an image library.

You can find many good examples of starting files in the testcases/ subdirectory that comes with this tool.

PS. If a large corpus of data is available for screening, you may want to use the afl-cmin utility to identify a subset of functionally distinct files that exercise different code paths in the target binary.

1.4.1 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.

Tips for optimizing fuzzing performance are discussed in *Performance Tips*.

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.

1.4.2 Interpreting output

See *Understanding the status screen* for information on how to interpret the displayed stats and monitor the health of the process. Be sure to consult this section 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. This is the synthesized corpus mentioned in section 2. Before using this corpus for any other purposes, you can shrink it to a smaller size using the afl-cmin tool. The tool will find a smaller subset of files offering equivalent edge coverage.
- **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...]
```

If you have gnuplot installed, you can also generate some pretty graphs for any active fuzzing task using afl-plot. For an example of how this looks like, see [http://lcamtuf.coredump.cx/afl/plot/{}](http://lcamtuf.coredump.cx/afl/plot/).

1.4.3 Parallelized fuzzing

Every instance of afl-fuzz takes up roughly one core. This means that on multi-core systems, parallelization is necessary to fully utilize the hardware. For tips on how to fuzz a common target on multiple cores or multiple networked machines, please refer to *Tips for parallel fuzzing*.

The parallel fuzzing mode also offers a simple way for interfacing AFL to other fuzzers, to symbolic or concolic execution engines, and so forth; again, see the last section of *Tips for parallel fuzzing* for tips.

1.4.4 Fuzzer dictionaries

By default, afl-fuzz mutation engine is optimized for compact data formats - say, images, multimedia, compressed data, regular expression syntax, or shell scripts. It is somewhat less suited for languages with particularly verbose and redundant verbiage - notably including HTML, SQL, or JavaScript.

To avoid the hassle of building syntax-aware tools, afl-fuzz provides a way to seed the fuzzing process with an optional dictionary of language keywords, magic headers, or other special tokens associated with the targeted data type – and use that to reconstruct the underlying grammar on the go:

> [http://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html{}](http://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html)

To use this feature, you first need to create a dictionary in one of the two formats discussed in dictionaries/README.dictionaries; and then point the fuzzer to it via the -x option in the command line.

(Several common dictionaries are already provided in that subdirectory, too.)

There is no way to provide more structured descriptions of the underlying syntax, but the fuzzer will likely figure out some of this based on the instrumentation feedback alone. This actually works in practice, say:

> [http://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html{]}(http://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html)

PS. Even when no explicit dictionary is given, afl-fuzz will try to extract existing syntax tokens in the input corpus by watching the instrumentation very closely during deterministic byte flips. This works for some types of parsers and grammars, but isn't nearly as good as the -x mode.

If a dictionary is really hard to come by, another option is to let AFL run for a while, and then use the token capture library that comes as a companion utility with AFL. For that, see libtokencap/README.tokencap.

1.4.5 Crash triage

The coverage-based grouping of crashes usually produces a small data set that can be quickly triaged manually or with a very simple GDB or Valgrind script. Every crash is also traceable to its parent non-crashing test case in the queue, making it easier to diagnose faults.

Having said that, it's important to acknowledge that some fuzzing crashes can be difficult to quickly evaluate for exploitability without a lot of debugging and code analysis work. To assist with this task, afl-fuzz supports a very unique "crash exploration" mode enabled with the -C flag.

In this mode, the fuzzer takes one or more crashing test cases as the input, and uses its feedback-driven fuzzing strategies to very quickly enumerate all code paths that can be reached in the program while keeping it in the crashing state.

Mutations that do not result in a crash are rejected; so are any changes that do not affect the execution path.

The output is a small corpus of files that can be very rapidly examined to see what degree of control the attacker has over the faulting address, or whether it is possible to get past an initial out-of-bounds read - and see what lies beneath.

Oh, one more thing: for test case minimization, give afl-tmin a try. The tool can be operated in a very simple way:

```
$ ./afl-tmin -i test_case -o minimized_result -- /path/to/program [...]
```

The tool works with crashing and non-crashing test cases alike. In the crash mode, it will happily accept instrumented and non-instrumented binaries. In the non-crashing mode, the minimizer relies on standard AFL instrumentation to make the file simpler without altering the execution path.

The minimizer accepts the **`-m`**, **`-t`**, **`-f`** and **`@@`** syntax in a manner compatible with afl-fuzz.

Another recent addition to AFL is the afl-analyze tool. It takes an input file, attempts to sequentially flip bytes, and observes the behavior of the tested program. It then color-codes the input based on which sections appear to be critical, and which are not; while not bulletproof, it can often offer quick insights into complex file formats. More info about its operation can be found near the end of *How AFL works*.

1.4.6 Going beyond crashes

Fuzzing is a wonderful and underutilized technique for discovering non-crashing design and implementation errors, too. Quite a few interesting bugs have been found by modifying the target programs to call abort() when, say:

- Two bignum libraries produce different outputs when given the same fuzzer-generated input,

- An image library produces different outputs when asked to decode the same input image several times in a row,
- A serialization / deserialization library fails to produce stable outputs when iteratively serializing and deserializing fuzzer-supplied data,
- A compression library produces an output inconsistent with the input file when asked to compress and then decompress a particular blob.

Implementing these or similar sanity checks usually takes very little time; if you are the maintainer of a particular package, you can make this code conditional with **`#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION`** (a flag also shared with libfuzzer) or **`#ifdef __AFL_COMPILER`** (this one is just for AFL).

1.5 Installation instructions

This document provides basic installation instructions and discusses known issues for a variety of platforms.

1.5.1 Platforms

Linux on x86

This platform is expected to work well. Compile the program with:

```
$ make
```

You can start using the fuzzer without installation, but it is also possible to install it with:

```
# make install
```

There are no special dependencies to speak of; you will need GNU make and a working compiler (gcc or clang). Some of the optional scripts bundled with the program may depend on bash, gdb, and similar basic tools.

If you are using clang, please review **`llvm_mode/README.llvm`**; the LLVM integration mode can offer substantial performance gains compared to the traditional approach.

You may have to change several settings to get optimal results (most notably, disable crash reporting utilities and switch to a different CPU governor), but afl-fuzz will guide you through that if necessary.

OpenBSD, FreeBSD, NetBSD on x86

Similarly to Linux, these platforms are expected to work well and are regularly tested. Compile everything with GNU make:

```
$ gmake
```

Note that BSD make will *not* work; if you do not have gmake on your system, please install it first. As on Linux, you can use the fuzzer itself without installation, or install it with:

```
# gmake install
```

Keep in mind that if you are using csh as your shell, the syntax of some of the shell commands given in the README and other docs will be different.

The llvm_mode requires a dynamically linked, fully-operational installation of clang. At least on FreeBSD, the clang binaries are static and do not include some of the essential tools, so if you want to make it work, you may need to follow the instructions in llvm_mode/README.llvm.

Beyond that, everything should work as advertised.

The QEMU mode is currently supported only on Linux. I think it's just a QEMU problem, I couldn't get a vanilla copy of user-mode emulation support working correctly on BSD at all.

MacOS X on x86

MacOS X should work, but there are some gotchas due to the idiosyncrasies of the platform. On top of this, I have limited release testing capabilities and depend mostly on user feedback.

To build AFL, install Xcode and follow the general instructions for Linux.

The Xcode 'gcc' tool is just a wrapper for clang, so be sure to use afl-clang to compile any instrumented binaries; afl-gcc will fail unless you have GCC installed from another source (in which case, please specify AFL_CC and AFL_CXX to point to the "real" GCC binaries).

Only 64-bit compilation will work on the platform; porting the 32-bit instrumentation would require a fair amount of work due to the way OS X handles relocations, and today, virtually all MacOS X boxes are 64-bit.

The crash reporting daemon that comes by default with MacOS X will cause problems with fuzzing. You need to turn it off by following the instructions provided here: http://goo.gl/CCcd5u

The fork() semantics on OS X are a bit unusual compared to other unix systems and definitely don't look POSIX-compliant. This means two things:

- Fuzzing will be probably slower than on Linux. In fact, some folks report considerable performance gains by running the jobs inside a Linux VM on MacOS X.
- Some non-portable, platform-specific code may be incompatible with the AFL forkserver. If you run into any problems, set AFL_NO_FORKSRV=1 in the environment before starting afl-fuzz.

User emulation mode of QEMU does not appear to be supported on MacOS X, so black-box instrumentation mode (-Q) will not work.

The llvm_mode requires a fully-operational installation of clang. The one that comes with Xcode is missing some of the essential headers and helper tools. See llvm_mode/README.llvm for advice on how to build the compiler from scratch.

Linux or *BSD on non-x86 systems

Standard build will fail on non-x86 systems, but you should be able to leverage two other options:

- The LLVM mode (see llvm_mode/README.llvm), which does not rely on x86-specific assembly shims. It's fast and robust, but requires a complete installation of clang.
- The QEMU mode (see qemu_mode/README.qemu), which can be also used for fuzzing cross-platform binaries. It's slower and more fragile, but can be used even when you don't have the source for the tested app.

If you're not sure what you need, you need the LLVM mode. To get it, try:

```
$ AFL_NO_X86=1 gmake && gmake -C llvm_mode
```

... and compile your target program with afl-clang-fast or afl-clang-fast++ instead of the traditional afl-gcc or afl-clang wrappers.

Solaris on x86

The fuzzer reportedly works on Solaris, but I have not tested this first-hand, and the user base is fairly small, so I don't have a lot of feedback.

To get the ball rolling, you will need to use GNU make and GCC or clang. I'm being told that the stock version of GCC that comes with the platform does not work properly due to its reliance on a hardcoded location for 'as' (completely ignoring the -B parameter or $PATH).

To fix this, you may want to build stock GCC from the source, like so:

```
$ ./configure --prefix=$HOME/gcc --with-gnu-as --with-gnu-ld \
  --with-gmp-include=/usr/include/gmp --with-mpfr-include=/usr/include/mpfr
$ make
$ sudo make install
```

Do *not* specify **`--with-as=/usr/gnu/bin/as`** - this will produce a GCC binary that ignores the **`-B`** flag and you will be back to square one.

Note that Solaris reportedly comes with crash reporting enabled, which causes problems with crashes being misinterpreted as hangs, similarly to the gotchas for Linux and MacOS X. AFL does not auto-detect crash reporting on this particular platform, but you may need to run the following command:

```
$ coreadm -d global -d global-setid -d process -d proc-setid \
  -d kzone -d log
```

User emulation mode of QEMU is not available on Solaris, so black-box instrumentation mode (**`-Q`**) will not work.

Everything else

You're on your own. On POSIX-compliant systems, you may be able to compile and run the fuzzer; and the LLVM mode may offer a way to instrument non-x86 code.

The fuzzer will not run on Windows. It will also not work under Cygwin. It could be ported to the latter platform fairly easily, but it's a pretty bad idea, because Cygwin is extremely slow. It makes much more sense to use VirtualBox or so to run a hardware-accelerated Linux VM; it will run around 20x faster or so. If you have a *really* compelling use case for Cygwin, let me know.

Although Android on x86 should theoretically work, the stock kernel may have SHM support compiled out, and if so, you may have to address that issue first. It's possible that all you need is this workaround:

> https://github.com/pelya/android-shmem

Joshua J. Drake notes that the Android linker adds a shim that automatically intercepts SIGSEGV and related signals. To fix this issue and be able to see crashes, you need to put this at the beginning of the fuzzed program:

```
signal(SIGILL, SIG_DFL);
signal(SIGABRT, SIG_DFL);
signal(SIGBUS, SIG_DFL);
signal(SIGFPE, SIG_DFL);
signal(SIGSEGV, SIG_DFL);
```

You may need to `#include <signal.h>` first.

1.6 AFL User Guide

1.6.1 Understanding the status screen

This document provides an overview of the status screen - plus tips for troubleshooting any warnings and red text shown in the UI.

0) A note about colors

The status screen and error messages use colors to keep things readable and attract your attention to the most important details. For example, red almost always means "consult this doc" :-)

Unfortunately, the UI will render correctly only if your terminal is using traditional un*x palette (white text on black background) or something close to that.

If you are using inverse video, you may want to change your settings, say:

- For GNOME Terminal, go to Edit > Profile preferences, select the "colors" tab, and from the list of built-in schemes, choose "white on black".
- For the MacOS X Terminal app, open a new window using the "Pro" scheme via the Shell > New Window menu (or make "Pro" your default).

Alternatively, if you really like your current colors, you can edit config.h to comment out USE_COLORS, then do 'make clean all'.

I'm not aware of any other simple way to make this work without causing other side effects - sorry about that.

With that out of the way, let's talk about what's actually on the screen...

1) Process timing

```
+----------------------------------------------------+
|        run time : 0 days, 8 hrs, 32 min, 43 sec    |
|   last new path : 0 days, 0 hrs, 6 min, 40 sec     |
| last uniq crash : none seen yet                    |
|  last uniq hang : 0 days, 1 hrs, 24 min, 32 sec    |
+----------------------------------------------------+
```

This section is fairly self-explanatory: it tells you how long the fuzzer has been running and how much time has elapsed since its most recent finds. This is broken down into "paths" (a shorthand for test cases that trigger new execution patterns), crashes, and hangs.

When it comes to timing: there is no hard rule, but most fuzzing jobs should be expected to run for days or weeks; in fact, for a moderately complex project, the first pass will probably take a day or so. Every now and then, some jobs will be allowed to run for months.

There's one important thing to watch out for: if the tool is not finding new paths within several minutes of starting, you're probably not invoking the target binary correctly and it never gets to parse the input files we're throwing at it; another possible explanations are that the default memory limit (-m) is too restrictive, and the program exits after failing to allocate a buffer very early on; or that the input files are patently invalid and always fail a basic header check.

If there are no new paths showing up for a while, you will eventually see a big red warning in this section, too :-)

2) Overall results

```
+----------------------+
|  cycles done : 0     |
|  total paths : 2095  |
| uniq crashes : 0     |
|   uniq hangs : 19    |
+----------------------+
```

The first field in this section gives you the count of queue passes done so far - that is, the number of times the fuzzer went over all the interesting test cases discovered so far, fuzzed them, and looped back to the very beginning. Every fuzzing session should be allowed to complete at least one cycle; and ideally, should run much longer than that.

As noted earlier, the first pass can take a day or longer, so sit back and relax. If you want to get broader but more shallow coverage right away, try the -d option - it gives you a more familiar experience by skipping the deterministic fuzzing steps. It is, however, inferior to the standard mode in a couple of subtle ways.

To help make the call on when to hit Ctrl-C, the cycle counter is color-coded. It is shown in magenta during the first pass, progresses to yellow if new finds are still being made in subsequent rounds, then blue when that ends - and finally, turns green after the fuzzer hasn't been seeing any action for a longer while.

The remaining fields in this part of the screen should be pretty obvious: there's the number of test cases ("paths") discovered so far, and the number of unique faults. The test cases, crashes, and hangs can be explored in real-time by browsing the output directory, as discussed in *Interpreting output*.

3) Cycle progress

```
+-------------------------------------+
|  now processing : 1296 (61.86%)     |
| paths timed out : 0 (0.00%)         |
+-------------------------------------+
```

This box tells you how far along the fuzzer is with the current queue cycle: it shows the ID of the test case it is currently working on, plus the number of inputs it decided to ditch because they were persistently timing out.

The "*" suffix sometimes shown in the first line means that the currently processed path is not "favored" (a property discussed later on, in section 6).

If you feel that the fuzzer is progressing too slowly, see the note about the -d option in section 2 of this doc.

4) Map coverage

```
+--------------------------------------+
|    map density : 10.15% / 29.07%     |
| count coverage : 4.03 bits/tuple     |
+--------------------------------------+
```

The section provides some trivia about the coverage observed by the instrumentation embedded in the target binary.

The first line in the box tells you how many branch tuples we have already hit, in proportion to how much the bitmap can hold. The number on the left describes the current input; the one on the right is the value for the entire input corpus.

Be wary of extremes:

- Absolute numbers below 200 or so suggest one of three things: that the program is extremely simple; that it is not instrumented properly (e.g., due to being linked against a non-instrumented copy of the target library); or

that it is bailing out prematurely on your input test cases. The fuzzer will try to mark this in pink, just to make you aware.

- Percentages over 70% may very rarely happen with very complex programs that make heavy use of template-generated code.

  Because high bitmap density makes it harder for the fuzzer to reliably discern new program states, I recommend recompiling the binary with AFL_INST_RATIO=10 or so and trying again (see env_variables.txt).

  The fuzzer will flag high percentages in red. Chances are, you will never see that unless you're fuzzing extremely hairy software (say, v8, perl, ffmpeg).

The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every branch, the needle will start to move toward "8.00" (every bit in the 8-bit map hit), but will probably never reach that extreme.

Together, the values can be useful for comparing the coverage of several different fuzzing jobs that rely on the same instrumented binary.

5) Stage progress

```
+-------------------------------------+
|  now trying : interest 32/8         |
| stage execs : 3996/34.4k (11.62%)   |
| total execs : 27.4M                 |
|  exec speed : 891.7/sec             |
+-------------------------------------+
```

This part gives you an in-depth peek at what the fuzzer is actually doing right now. It tells you about the current stage, which can be any of:

- calibration - a pre-fuzzing stage where the execution path is examined to detect anomalies, establish baseline execution speed, and so on. Executed very briefly whenever a new find is being made.
- trim L/S - another pre-fuzzing stage where the test case is trimmed to the shortest form that still produces the same execution path. The length (L) and stepover (S) are chosen in general relationship to file size.
- bitflip L/S - deterministic bit flips. There are L bits toggled at any given time, walking the input file with S-bit increments. The current L/S variants are: 1/1, 2/1, 4/1, 8/8, 16/8, 32/8.
- arith L/8 - deterministic arithmetics. The fuzzer tries to subtract or add small integers to 8-, 16-, and 32-bit values. The stepover is always 8 bits.
- interest L/8 - deterministic value overwrite. The fuzzer has a list of known "interesting" 8-, 16-, and 32-bit values to try. The stepover is 8 bits.
- extras - deterministic injection of dictionary terms. This can be shown as "user" or "auto", depending on whether the fuzzer is using a user-supplied dictionary (-x) or an auto-created one. You will also see "over" or "insert", depending on whether the dictionary words overwrite existing data or are inserted by offsetting the remaining data to accommodate their length.
- havoc - a sort-of-fixed-length cycle with stacked random tweaks. The operations attempted during this stage include bit flips, overwrites with random and "interesting" integers, block deletion, block duplication, plus assorted dictionary-related operations (if a dictionary is supplied in the first place).
- splice - a last-resort strategy that kicks in after the first full queue cycle with no new paths. It is equivalent to 'havoc', except that it first splices together two random inputs from the queue at some arbitrarily selected midpoint.

- sync - a stage used only when -M or -S is set (see *Tips for parallel fuzzing*). No real fuzzing is involved, but the tool scans the output from other fuzzers and imports test cases as necessary. The first time this is done, it may take several minutes or so.

The remaining fields should be fairly self-evident: there's the exec count progress indicator for the current stage, a global exec counter, and a benchmark for the current program execution speed. This may fluctuate from one test case to another, but the benchmark should be ideally over 500 execs/sec most of the time - and if it stays below 100, the job will probably take very long.

The fuzzer will explicitly warn you about slow targets, too. If this happens, see *Performance Tips* for ideas on how to speed things up.

6) Findings in depth

```
+-------------------------------------+
| favored paths : 879 (41.96%)        |
|  new edges on : 423 (20.19%)        |
| total crashes : 0 (0 unique)        |
|  total tmouts : 24 (19 unique)      |
+-------------------------------------+
```

This gives you several metrics that are of interest mostly to complete nerds. The section includes the number of paths that the fuzzer likes the most based on a minimization algorithm baked into the code (these will get considerably more air time), and the number of test cases that actually resulted in better edge coverage (versus just pushing the branch hit counters up). There are also additional, more detailed counters for crashes and timeouts.

Note that the timeout counter is somewhat different from the hang counter; this one includes all test cases that exceeded the timeout, even if they did not exceed it by a margin sufficient to be classified as hangs.

7) Fuzzing strategy yields

```
+-----------------------------------------------------+
|   bit flips : 57/289k, 18/289k, 18/288k             |
|  byte flips : 0/36.2k, 4/35.7k, 7/34.6k             |
| arithmetics : 53/2.54M, 0/537k, 0/55.2k             |
|  known ints : 8/322k, 12/1.32M, 10/1.70M            |
|  dictionary : 9/52k, 1/53k, 1/24k                   |
|       havoc : 1903/20.0M, 0/0                       |
|        trim : 20.31%/9201, 17.05%                   |
+-----------------------------------------------------+
```

This is just another nerd-targeted section keeping track of how many paths we have netted, in proportion to the number of execs attempted, for each of the fuzzing strategies discussed earlier on. This serves to convincingly validate assumptions about the usefulness of the various approaches taken by afl-fuzz.

The trim strategy stats in this section are a bit different than the rest. The first number in this line shows the ratio of bytes removed from the input files; the second one corresponds to the number of execs needed to achieve this goal. Finally, the third number shows the proportion of bytes that, although not possible to remove, were deemed to have no effect and were excluded from some of the more expensive deterministic fuzzing steps.

8) Path geometry

```
+--------------------+
|    levels : 5      |
|   pending : 1570   |
|  pend fav : 583    |
| own finds : 0      |
|  imported : 0      |
| stability : 100.00% |
+--------------------+
```

The first field in this section tracks the path depth reached through the guided fuzzing process. In essence: the initial test cases supplied by the user are considered "level 1". The test cases that can be derived from that through traditional fuzzing are considered "level 2"; the ones derived by using these as inputs to subsequent fuzzing rounds are "level 3"; and so forth. The maximum depth is therefore a rough proxy for how much value you're getting out of the instrumentation-guided approach taken by afl-fuzz.

The next field shows you the number of inputs that have not gone through any fuzzing yet. The same stat is also given for "favored" entries that the fuzzer really wants to get to in this queue cycle (the non-favored entries may have to wait a couple of cycles to get their chance).

Next, we have the number of new paths found during this fuzzing section and imported from other fuzzer instances when doing parallelized fuzzing; and the extent to which identical inputs appear to sometimes produce variable behavior in the tested binary.

That last bit is actually fairly interesting: it measures the consistency of observed traces. If a program always behaves the same for the same input data, it will earn a score of 100%. When the value is lower but still shown in purple, the fuzzing process is unlikely to be negatively affected. If it goes into red, you may be in trouble, since AFL will have difficulty discerning between meaningful and "phantom" effects of tweaking the input file.

Now, most targets will just get a 100% score, but when you see lower figures, there are several things to look at:

- The use of uninitialized memory in conjunction with some intrinsic sources of entropy in the tested binary. Harmless to AFL, but could be indicative of a security bug.
- Attempts to manipulate persistent resources, such as left over temporary files or shared memory objects. This is usually harmless, but you may want to double-check to make sure the program isn't bailing out prematurely. Running out of disk space, SHM handles, or other global resources can trigger this, too.
- Hitting some functionality that is actually designed to behave randomly. Generally harmless. For example, when fuzzing sqlite, an input like 'select random();' will trigger a variable execution path.
- Multiple threads executing at once in semi-random order. This is harmless when the 'stability' metric stays over 90% or so, but can become an issue if not. Here's what to try:
  - Use afl-clang-fast from llvm_mode/ - it uses a thread-local tracking model that is less prone to concurrency issues,
  - See if the target can be compiled or run without threads. Common ./configure options include –without-threads, –disable-pthreads, or –disable-openmp.
  - Replace pthreads with GNU Pth (https://www.gnu.org/software/pth/), which allows you to use a deterministic scheduler.
- In persistent mode, minor drops in the "stability" metric can be normal, because not all the code behaves identically when re-entered; but major dips may signify that the code within __AFL_LOOP() is not behaving correctly on subsequent iterations (e.g., due to incomplete clean-up or reinitialization of the state) and that most of the fuzzing effort goes to waste.

The paths where variable behavior is detected are marked with a matching entry in the <out_dir>/queue/.state/variable_behavior/ directory, so you can look them up easily.

9) CPU load

```
[cpu: 25%]
```

This tiny widget shows the apparent CPU utilization on the local system. It is calculated by taking the number of processes in the "runnable" state, and then comparing it to the number of logical cores on the system.

If the value is shown in green, you are using fewer CPU cores than available on your system and can probably parallelize to improve performance; for tips on how to do that, see *Tips for parallel fuzzing*.

If the value is shown in red, your CPU is *possibly* oversubscribed, and running additional fuzzers may not give you any benefits.

Of course, this benchmark is very simplistic; it tells you how many processes are ready to run, but not how resource-hungry they may be. It also doesn't distinguish between physical cores, logical cores, and virtualized CPUs; the performance characteristics of each of these will differ quite a bit.

If you want a more accurate measurement, you can run the afl-gotcpu utility from the command line.

10) Addendum: status and plot files

For unattended operation, some of the key status screen information can be also found in a machine-readable format in the fuzzer_stats file in the output directory. This includes:

- start_time - unix time indicating the start time of afl-fuzz
- last_update - unix time corresponding to the last update of this file
- fuzzer_pid - PID of the fuzzer process
- cycles_done - queue cycles completed so far
- execs_done - number of execve() calls attempted
- execs_per_sec - current number of execs per second
- paths_total - total number of entries in the queue
- paths_found - number of entries discovered through local fuzzing
- paths_imported - number of entries imported from other instances
- max_depth - number of levels in the generated data set
- cur_path - currently processed entry number
- pending_favs - number of favored entries still waiting to be fuzzed
- pending_total - number of all entries waiting to be fuzzed
- stability - percentage of bitmap bytes that behave consistently
- variable_paths - number of test cases showing variable behavior
- unique_crashes - number of unique crashes recorded
- unique_hangs - number of unique hangs encountered

Most of these map directly to the UI elements discussed earlier on.

On top of that, you can also find an entry called 'plot_data', containing a plottable history for most of these fields. If you have gnuplot installed, you can turn this into a nice progress report with the included 'afl-plot' tool.

1.6.2 Environmental variables

This document discusses the environment variables used by American Fuzzy Lop to expose various exotic functions that may be (rarely) useful for power users or for some types of custom fuzzing setups. See README for the general instruction manual.

1) Settings for afl-gcc, afl-clang, and afl-as

Because they can't directly accept command-line options, the compile-time tools make fairly broad use of environmental variables:

- Setting AFL_HARDEN automatically adds code hardening options when invoking the downstream compiler. This currently includes -D_FORTIFY_SOURCE=2 and -fstack-protector-all. The setting is useful for catching non-crashing memory bugs at the expense of a very slight (sub-5%) performance loss.
- By default, the wrapper appends -O3 to optimize builds. Very rarely, this will cause problems in programs built with -Werror, simply because -O3 enables more thorough code analysis and can spew out additional warnings. To disable optimizations, set AFL_DONT_OPTIMIZE.
- Setting AFL_USE_ASAN automatically enables ASAN, provided that your compiler supports that. Note that fuzzing with ASAN is mildly challenging - see *Using ASAN with AFL*.

  (You can also enable MSAN via AFL_USE_MSAN; ASAN and MSAN come with the same gotchas; the modes are mutually exclusive. UBSAN and other exotic sanitizers are not officially supported yet, but are easy to get to work by hand.)
- Setting AFL_CC, AFL_CXX, and AFL_AS lets you use alternate downstream compilation tools, rather than the default 'clang', 'gcc', or 'as' binaries in your $PATH.
- AFL_PATH can be used to point afl-gcc to an alternate location of afl-as. One possible use of this is experimental/clang_asm_normalize/, which lets you instrument hand-written assembly when compiling clang code by plugging a normalizer into the chain. (There is no equivalent feature for GCC.)
- Setting AFL_INST_RATIO to a percentage between 0 and 100% controls the probability of instrumenting every branch. This is (very rarely) useful when dealing with exceptionally complex programs that saturate the output bitmap. Examples include v8, ffmpeg, and perl.

  (If this ever happens, afl-fuzz will warn you ahead of the time by displaying the "bitmap density" field in fiery red.)

  Setting AFL_INST_RATIO to 0 is a valid choice. This will instrument only the transitions between function entry points, but not individual branches.
- AFL_NO_BUILTIN causes the compiler to generate code suitable for use with libtokencap.so (but perhaps running a bit slower than without the flag).
- TMPDIR is used by afl-as for temporary files; if this variable is not set, the tool defaults to /tmp.
- Setting AFL_KEEP_ASSEMBLY prevents afl-as from deleting instrumented assembly files. Useful for troubleshooting problems or understanding how the tool works. To get them in a predictable place, try something like:

  mkdir assembly_here TMPDIR=$PWD/assembly_here AFL_KEEP_ASSEMBLY=1 make clean all
- Setting AFL_QUIET will prevent afl-cc and afl-as banners from being displayed during compilation, in case you find them distracting.

2) Settings for afl-clang-fast

The native LLVM instrumentation helper accepts a subset of the settings discussed in section #1, with the exception of:

- AFL_AS, since this toolchain does not directly invoke GNU as.
- TMPDIR and AFL_KEEP_ASSEMBLY, since no temporary assembly files are created.

Note that AFL_INST_RATIO will behave a bit differently than for afl-gcc, because functions are *not* instrumented unconditionally - so low values will have a more striking effect. For this tool, 0 is not a valid choice.

3) Settings for afl-fuzz

The main fuzzer binary accepts several options that disable a couple of sanity checks or alter some of the more exotic semantics of the tool:

- Setting AFL_SKIP_CPUFREQ skips the check for CPU scaling policy. This is useful if you can't change the defaults (e.g., no root access to the system) and are OK with some performance loss.
- Setting AFL_NO_FORKSRV disables the forkserver optimization, reverting to fork + execve() call for every tested input. This is useful mostly when working with unruly libraries that create threads or do other crazy things when initializing (before the instrumentation has a chance to run).

  Note that this setting inhibits some of the user-friendly diagnostics normally done when starting up the forkserver and causes a pretty significant performance drop.
- AFL_EXIT_WHEN_DONE causes afl-fuzz to terminate when all existing paths have been fuzzed and there were no new finds for a while. This would be normally indicated by the cycle counter in the UI turning green. May be convenient for some types of automated jobs.
- Setting AFL_NO_AFFINITY disables attempts to bind to a specific CPU core on Linux systems. This slows things down, but lets you run more instances of afl-fuzz than would be prudent (if you really want to).
- AFL_SKIP_CRASHES causes AFL to tolerate crashing files in the input queue. This can help with rare situations where a program crashes only intermittently, but it's not really recommended under normal operating conditions.
- Setting AFL_HANG_TMOUT allows you to specify a different timeout for deciding if a particular test case is a "hang". The default is 1 second or the value of the -t parameter, whichever is larger. Dialing the value down can be useful if you are very concerned about slow inputs, or if you don't want AFL to spend too much time classifying that stuff and just rapidly put all timeouts in that bin.
- AFL_NO_ARITH causes AFL to skip most of the deterministic arithmetics. This can be useful to speed up the fuzzing of text-based file formats.
- AFL_SHUFFLE_QUEUE randomly reorders the input queue on startup. Requested by some users for unorthodox parallelized fuzzing setups, but not advisable otherwise.
- When developing custom instrumentation on top of afl-fuzz, you can use AFL_SKIP_BIN_CHECK to inhibit the checks for non-instrumented binaries and shell scripts; and AFL_DUMB_FORKSRV in conjunction with the -n setting to instruct afl-fuzz to still follow the fork server protocol without expecting any instrumentation data in return.
- When running in the -M or -S mode, setting AFL_IMPORT_FIRST causes the fuzzer to import test cases from other instances before doing anything else. This makes the "own finds" counter in the UI more accurate. Beyond counter aesthetics, not much else should change.
- Setting AFL_POST_LIBRARY allows you to configure a postprocessor for mutated files - say, to fix up checksums. See experimental/post_library/ for more.

- AFL_FAST_CAL keeps the calibration stage about 2.5x faster (albeit less precise), which can help when starting a session against a slow target.
- The CPU widget shown at the bottom of the screen is fairly simplistic and may complain of high load prematurely, especially on systems with low core counts. To avoid the alarming red color, you can set AFL_NO_CPU_RED.
- In QEMU mode (-Q), AFL_PATH will be searched for afl-qemu-trace.
- Setting AFL_PRELOAD causes AFL to set LD_PRELOAD for the target binary without disrupting the afl-fuzz process itself. This is useful, among other things, for bootstrapping libdislocator.so.
- Setting AFL_NO_UI inhibits the UI altogether, and just periodically prints some basic stats. This behavior is also automatically triggered when the output from afl-fuzz is redirected to a file or to a pipe.
- If you are Jakub, you may need AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES. Others need not apply.
- Benchmarking only: AFL_BENCH_JUST_ONE causes the fuzzer to exit after processing the first queue entry; and AFL_BENCH_UNTIL_CRASH causes it to exit soon after the first crash is found.

4) Settings for afl-qemu-trace

The QEMU wrapper used to instrument binary-only code supports several settings:

- It is possible to set AFL_INST_RATIO to skip the instrumentation on some of the basic blocks, which can be useful when dealing with very complex binaries.
- Setting AFL_INST_LIBS causes the translator to also instrument the code inside any dynamically linked libraries (notably including glibc).
- The underlying QEMU binary will recognize any standard "user space emulation" variables (e.g., QEMU_STACK_SIZE), but there should be no reason to touch them.

5) Settings for afl-cmin

The corpus minimization script offers very little customization:

- Setting AFL_PATH offers a way to specify the location of afl-showmap and afl-qemu-trace (the latter only in -Q mode).
- AFL_KEEP_TRACES makes the tool keep traces and other metadata used for minimization and normally deleted at exit. The files can be found in the **`<out_dir>/.traces/*`**.
- AFL_ALLOW_TMP permits this and some other scripts to run in /tmp. This is a modest security risk on multi-user systems with rogue users, but should be safe on dedicated fuzzing boxes.

6) Settings for afl-tmin

Virtually nothing to play with. Well, in QEMU mode (-Q), AFL_PATH will be searched for afl-qemu-trace. In addition to this, TMPDIR may be used if a temporary file can't be created in the current working directory.

You can specify AFL_TMIN_EXACT if you want afl-tmin to require execution paths to match when minimizing crashes. This will make minimization less useful, but may prevent the tool from "jumping" from one crashing condition to another in very buggy software. You probably want to combine it with the -e flag.

7) Settings for afl-analyze

You can set AFL_ANALYZE_HEX to get file offsets printed as hexadecimal instead of decimal.

8) Settings for libdislocator.so

The library honors three environmental variables:

- AFL_LD_LIMIT_MB caps the size of the maximum heap usage permitted by the library, in megabytes. The default value is 1 GB. Once this is exceeded, allocations will return NULL.
- AFL_LD_HARD_FAIL alters the behavior by calling abort() on excessive allocations, thus causing what AFL would perceive as a crash. Useful for programs that are supposed to maintain a specific memory footprint.
- AFL_LD_VERBOSE causes the library to output some diagnostic messages that may be useful for pinpointing the cause of any observed issues.
- AFL_LD_NO_CALLOC_OVER inhibits abort() on calloc() overflows. Most of the common allocators check for that internally and return NULL, so it's a security risk only in more exotic setups.

9) Settings for libtokencap.so

This library accepts AFL_TOKEN_FILE to indicate the location to which the discovered tokens should be written.

10) Third-party variables set by afl-fuzz & other tools

Several variables are not directly interpreted by afl-fuzz, but are set to optimal values if not already present in the environment:

- By default, LD_BIND_NOW is set to speed up fuzzing by forcing the linker to do all the work before the fork server kicks in. You can override this by setting LD_BIND_LAZY beforehand, but it is almost certainly pointless.
- By default, ASAN_OPTIONS are set to:

  abort_on_error=1 detect_leaks=0 symbolize=0 allocator_may_return_null=1

  If you want to set your own options, be sure to include abort_on_error=1 - otherwise, the fuzzer will not be able to detect crashes in the tested app. Similarly, include symbolize=0, since without it, AFL may have difficulty telling crashes and hangs apart.
- In the same vein, by default, MSAN_OPTIONS are set to:

  exit_code=86 (required for legacy reasons) abort_on_error=1 symbolize=0 msan_track_origins=0 allocator_may_return_null=1

  Be sure to include the first one when customizing anything, since some MSAN versions don't call abort() on error, and we need a way to detect faults.

1.6.3 Tips for parallel fuzzing

This document talks about synchronizing afl-fuzz jobs on a single machine or across a fleet of systems. See README for the general instruction manual.

1) Introduction

Every copy of afl-fuzz will take up one CPU core. This means that on an n-core system, you can almost always run around n concurrent fuzzing jobs with virtually no performance hit (you can use the afl-gotcpu tool to make sure).

In fact, if you rely on just a single job on a multi-core system, you will be underutilizing the hardware. So, parallelization is usually the right way to go.

When targeting multiple unrelated binaries or using the tool in "dumb" (-n) mode, it is perfectly fine to just start up several fully separate instances of afl-fuzz. The picture gets more complicated when you want to have multiple fuzzers hammering a common target: if a hard-to-hit but interesting test case is synthesized by one fuzzer, the remaining instances will not be able to use that input to guide their work.

To help with this problem, afl-fuzz offers a simple way to synchronize test cases on the fly.

2) Single-system parallelization

If you wish to parallelize a single job across multiple cores on a local system, simply create a new, empty output directory ("sync dir") that will be shared by all the instances of afl-fuzz; and then come up with a naming scheme for every instance - say, "fuzzer01", "fuzzer02", etc.

Run the first one ("master", -M) like this:

```
$ ./afl-fuzz -i testcase_dir -o sync_dir -M fuzzer01 [...other stuff...]
```

... and then, start up secondary (-S) instances like this:

```
$ ./afl-fuzz -i testcase_dir -o sync_dir -S fuzzer02 [...other stuff...]
$ ./afl-fuzz -i testcase_dir -o sync_dir -S fuzzer03 [...other stuff...]
```

Each fuzzer will keep its state in a separate subdirectory, like so:

```
/path/to/sync_dir/fuzzer01/
```

Each instance will also periodically rescan the top-level sync directory for any test cases found by other fuzzers - and will incorporate them into its own fuzzing when they are deemed interesting enough.

The difference between the -M and -S modes is that the master instance will still perform deterministic checks; while the secondary instances will proceed straight to random tweaks. If you don't want to do deterministic fuzzing at all, it's OK to run all instances with -S. With very slow or complex targets, or when running heavily parallelized jobs, this is usually a good plan.

Note that running multiple -M instances is wasteful, although there is an experimental support for parallelizing the deterministic checks. To leverage that, you need to create -M instances like so:

```
$ ./afl-fuzz -i testcase_dir -o sync_dir -M masterA:1/3 [...]
$ ./afl-fuzz -i testcase_dir -o sync_dir -M masterB:2/3 [...]
$ ./afl-fuzz -i testcase_dir -o sync_dir -M masterC:3/3 [...]
```

... where the first value after ':' is the sequential ID of a particular master instance (starting at 1), and the second value is the total number of fuzzers to distribute the deterministic fuzzing across. Note that if you boot up fewer fuzzers than indicated by the second number passed to -M, you may end up with poor coverage.

You can also monitor the progress of your jobs from the command line with the provided afl-whatsup tool. When the instances are no longer finding new paths, it's probably time to stop.

WARNING: Exercise caution when explicitly specifying the -f option. Each fuzzer must use a separate temporary file; otherwise, things will go south. One safe example may be:

```
$ ./afl-fuzz [...] -S fuzzer10 -f file10.txt ./fuzzed/binary @@
$ ./afl-fuzz [...] -S fuzzer11 -f file11.txt ./fuzzed/binary @@
$ ./afl-fuzz [...] -S fuzzer12 -f file12.txt ./fuzzed/binary @@
```

This is not a concern if you use @@ without -f and let afl-fuzz come up with the file name.

3) Multi-system parallelization

The basic operating principle for multi-system parallelization is similar to the mechanism explained in section 2. The key difference is that you need to write a simple script that performs two actions:

- Uses SSH with authorized_keys to connect to every machine and retrieve a tar archive of the /path/to/sync_dir/<fuzzer_id>/queue/ directories for every <fuzzer_id> local to the machine. It's best to use a naming scheme that includes host name in the fuzzer ID, so that you can do something like:

```
for s in {1..10}; do
  ssh user@host${s} "tar -czf - sync/host${s}_fuzzid*/[qf]*" >host${s}.tgz
done
```

- Distributes and unpacks these files on all the remaining machines, e.g.:

```
for s in {1..10}; do
  for d in {1..10}; do
    test "$s" = "$d" && continue
    ssh user@host${d} 'tar -kxzf -' <host${s}.tgz
  done
done
```

There is an example of such a script in experimental/distributed_fuzzing/; you can also find a more featured, experimental tool developed by Martijn Bogaard at:

https://github.com/MartijnB/disfuzz-afl

Another client-server implementation from Richo Healey is:

https://github.com/richo/roving

Note that these third-party tools are unsafe to run on systems exposed to the Internet or to untrusted users.

When developing custom test case sync code, there are several optimizations to keep in mind:

- The synchronization does not have to happen very often; running the task every 30 minutes or so may be perfectly fine.
- There is no need to synchronize crashes/ or hangs/; you only need to copy over queue/* (and ideally, also fuzzer_stats).
- It is not necessary (and not advisable!) to overwrite existing files; the -k option in tar is a good way to avoid that.
- There is no need to fetch directories for fuzzers that are not running locally on a particular machine, and were simply copied over onto that system during earlier runs.
- For large fleets, you will want to consolidate tarballs for each host, as this will let you use n SSH connections for sync, rather than n*(n-1).

  You may also want to implement staged synchronization. For example, you could have 10 groups of systems, with group 1 pushing test cases only to group 2; group 2 pushing them only to group 3; and so on, with group eventually 10 feeding back to group 1.

  This arrangement would allow test interesting cases to propagate across the fleet without having to copy every fuzzer queue to every single host.
- You do not want a "master" instance of afl-fuzz on every system; you should run them all with -S, and just designate a single process somewhere within the fleet to run with -M.

It is *not* advisable to skip the synchronization script and run the fuzzers directly on a network filesystem; unexpected latency and unkillable processes in I/O wait state can mess things up.

4) Remote monitoring and data collection

You can use screen, nohup, tmux, or something equivalent to run remote instances of afl-fuzz. If you redirect the program's output to a file, it will automatically switch from a fancy UI to more limited status reports. There is also basic machine-readable information always written to the fuzzer_stats file in the output directory. Locally, that information can be interpreted with afl-whatsup.

In principle, you can use the status screen of the master (-M) instance to monitor the overall fuzzing progress and decide when to stop. In this mode, the most important signal is just that no new paths are being found for a longer while. If you do not have a master instance, just pick any single secondary instance to watch and go by that.

You can also rely on that instance's output directory to collect the synthesized corpus that covers all the noteworthy paths discovered anywhere within the fleet. Secondary (-S) instances do not require any special monitoring, other than just making sure that they are up.

Keep in mind that crashing inputs are *not* automatically propagated to the master instance, so you may still want to monitor for crashes fleet-wide from within your synchronization or health checking scripts (see afl-whatsup).

5) Asymmetric setups

It is perhaps worth noting that all of the following is permitted:

- Running afl-fuzz with conjunction with other guided tools that can extend coverage (e.g., via concolic execution). Third-party tools simply need to follow the protocol described above for pulling new test cases from **`out_dir/<fuzzer_id>/queue/*`** and writing their own finds to sequentially numbered id:nnnnnn files in **`out_dir/<ext_tool_id>/queue/*`**.
- Running some of the synchronized fuzzers with different (but related) target binaries. For example, simultaneously stress-testing several different JPEG parsers (say, IJG jpeg and libjpeg-turbo) while sharing the discovered test cases can have synergistic effects and improve the overall coverage.

  (In this case, running one -M instance per each binary is a good plan.)
- Having some of the fuzzers invoke the binary in different ways. For example, 'djpeg' supports several DCT modes, configurable with a command-line flag, while 'dwebp' supports incremental and one-shot decoding. In some scenarios, going after multiple distinct modes and then pooling test cases will improve coverage.
- Much less convincingly, running the synchronized fuzzers with different starting test cases (e.g., progressive and standard JPEG) or dictionaries. The synchronization mechanism ensures that the test sets will get fairly homogeneous over time, but it introduces some initial variability.

1.7 Using ASAN with AFL

This file discusses some of the caveats for fuzzing under ASAN, and suggests a handful of alternatives.

1.7.1 Short version

ASAN on 64-bit systems requests a lot of memory in a way that can't be easily distinguished from a misbehaving program bent on crashing your system.

Because of this, fuzzing with ASAN is recommended only in four scenarios:

- On 32-bit systems, where we can always enforce a reasonable memory limit (-m 800 or so is a good starting point),
- On 64-bit systems only if you can do one of the following:

- Compile the binary in 32-bit mode (**gcc -m32**),
- Precisely gauge memory needs using http://jwilk.net/software/recidivm .
- Limit the memory available to process using cgroups on Linux (see **experimental/asan_cgroups**).

To compile with ASAN, set **AFL_USE_ASAN=1** before calling **make clean all**. The **afl-gcc** / **afl-clang** wrappers will pick that up and add the appropriate flags. Note that ASAN is incompatible with **-static**, so be mindful of that.

(You can also use **AFL_USE_MSAN=1** to enable MSAN instead.)

There is also the option of generating a corpus using a non-ASAN binary, and then feeding it to an ASAN-instrumented one to check for bugs. This is faster, and can give you somewhat comparable results. You can also try using **libdislocator** (see **libdislocator/README.dislocator** in the parent directory) as a lightweight and hassle-free (but less thorough) alternative.

1.7.2 Long version

ASAN allocates a huge region of virtual address space for bookkeeping purposes. Most of this is never actually accessed, so the OS never has to allocate any real pages of memory for the process, and the VM grabbed by ASAN is essentially "free" - but the mapping counts against the standard OS-enforced limit (RLIMIT_AS, aka ulimit -v).

On our end, afl-fuzz tries to protect you from processes that go off-rails and start consuming all the available memory in a vain attempt to parse a malformed input file. This happens surprisingly often, so enforcing such a limit is important for almost any fuzzer: the alternative is for the kernel OOM handler to step in and start killing random processes to free up resources. Needless to say, that's not a very nice prospect to live with.

Unfortunately, un*x systems offer no portable way to limit the amount of pages actually given to a process in a way that distinguishes between that and the harmless "land grab" done by ASAN. In principle, there are three standard ways to limit the size of the heap:

- The RLIMIT_AS mechanism (ulimit -v) caps the size of the virtual space - but as noted, this pays no attention to the number of pages actually in use by the process, and doesn't help us here.
- The RLIMIT_DATA mechanism (ulimit -d) seems like a good fit, but it applies only to the traditional sbrk() / brk() methods of requesting heap space; modern allocators, including the one in glibc, routinely rely on mmap() instead, and circumvent this limit completely.
- Finally, the RLIMIT_RSS limit (ulimit -m) sounds like what we need, but doesn't work on Linux - mostly because nobody felt like implementing it.

There are also cgroups, but they are Linux-specific, not universally available even on Linux systems, and they require root permissions to set up; I'm a bit hesitant to make afl-fuzz require root permissions just for that. That said, if you are on Linux and want to use cgroups, check out the contributed script that ships in experimental/asan_cgroups/.

In settings where cgroups aren't available, we have no nice, portable way to avoid counting the ASAN allocation toward the limit. On 32-bit systems, or for binaries compiled in 32-bit mode (-m32), this is not a big deal: ASAN needs around 600-800 MB or so, depending on the compiler - so all you need to do is to specify -m that is a bit higher than that.

On 64-bit systems, the situation is more murky, because the ASAN allocation is completely outlandish - around 17.5 TB in older versions, and closer to 20 TB with newest ones. The actual amount of memory on your system is (probably!) just a tiny fraction of that - so unless you dial the limit with surgical precision, you will get no protection from OOM bugs.

On my system, the amount of memory grabbed by ASAN with a slightly older version of gcc is around 17,825,850 MB; for newest clang, it's 20,971,600. But there is no guarantee that these numbers are stable, and if you get them wrong by "just" a couple gigs or so, you will be at risk.

To get the precise number, you can use the recidivm tool developed by Jakub Wilk (http://jwilk.net/software/recidivm). In absence of this, ASAN is *not* recommended when fuzzing 64-bit binaries, unless you are confident that they are robust and enforce reasonable memory limits (in which case, you can specify '-m none' when calling afl-fuzz).

Using recidivm or running with no limits aside, there are two other decent alternatives: build a corpus of test cases using a non-ASAN binary, and then examine them with ASAN, Valgrind, or other heavy-duty tools in a more controlled setting; or compile the target program with -m32 (32-bit mode) if your system supports that.

1.7.3 Interactions with the QEMU mode

ASAN, MSAN, and other sanitizers appear to be incompatible with QEMU user emulation, so please do not try to use them with the -Q option; QEMU doesn't seem to appreciate the shadow VM trick used by these tools, and will likely just allocate all your physical memory, then crash.

1.7.4 ASAN and OOM crashes

By default, ASAN treats memory allocation failures as fatal errors, immediately causing the program to crash. Since this is a departure from normal POSIX semantics (and creates the appearance of security issues in otherwise properly-behaving programs), we try to disable this by specifying allocator_may_return_null=1 in ASAN_OPTIONS.

Unfortunately, it's been reported that this setting still causes ASAN to trigger phantom crashes in situations where the standard allocator would simply return NULL. If this is interfering with your fuzzing jobs, you may want to cc: yourself on this bug:

> https://bugs.llvm.org/show_bug.cgi?id=22026

1.7.5 What about UBSAN?

Some folks expressed interest in fuzzing with UBSAN. This isn't officially supported, because many installations of UBSAN don't offer a consistent way to abort() on fault conditions or to terminate with a distinctive exit code.

That said, some versions of the library can be binary-patched to address this issue, while newer releases support explicit compile-time flags - see this mailing list thread for tips:

> https://groups.google.com/forum/#!topic/afl-users/GyeSBJt4M38

1.8 Tips

1.8.1 Performance Tips

> This file provides tips for troubleshooting slow or wasteful fuzzing jobs.

1) Keep your test cases small

This is probably the single most important step to take! Large test cases do not merely take more time and memory to be parsed by the tested binary, but also make the fuzzing process dramatically less efficient in several other ways.

To illustrate, let's say that you're randomly flipping bits in a file, one bit at a time. Let's assume that if you flip bit #47, you will hit a security bug; flipping any other bit just results in an invalid document.

Now, if your starting test case is 100 bytes long, you will have a 71% chance of triggering the bug within the first 1,000 execs - not bad! But if the test case is 1 kB long, the probability that we will randomly hit the right pattern in the same timeframe goes down to 11%. And if it has 10 kB of non-essential cruft, the odds plunge to 1%.

On top of that, with larger inputs, the binary may be now running 5-10x times slower than before - so the overall drop in fuzzing efficiency may be easily as high as 500x or so.

In practice, this means that you shouldn't fuzz image parsers with your vacation photos. Generate a tiny 16x16 picture instead, and run it through jpegtran or pngcrunch for good measure. The same goes for most other types of documents.

There's plenty of small starting test cases in `../testcases/*` - try them out or submit new ones!

If you want to start with a larger, third-party corpus, run afl-cmin with an aggressive timeout on that data set first.

2) Use a simpler target

Consider using a simpler target binary in your fuzzing work. For example, for image formats, bundled utilities such as djpeg, readpng, or gifhisto are considerably (10-20x) faster than the convert tool from ImageMagick - all while exercising roughly the same library-level image parsing code.

Even if you don't have a lightweight harness for a particular target, remember that you can always use another, related library to generate a corpus that will be then manually fed to a more resource-hungry program later on.

3) Use LLVM instrumentation

When fuzzing slow targets, you can gain 2x performance improvement by using the LLVM-based instrumentation mode described in llvm_mode/README.llvm. Note that this mode requires the use of clang and will not work with GCC.

The LLVM mode also offers a "persistent", in-process fuzzing mode that can work well for certain types of self-contained libraries, and for fast targets, can offer performance gains up to 5-10x; and a "deferred fork server" mode that can offer huge benefits for programs with high startup overhead. Both modes require you to edit the source code of the fuzzed program, but the changes often amount to just strategically placing a single line or two.

4) Profile and optimize the binary

Check for any parameters or settings that obviously improve performance. For example, the djpeg utility that comes with IJG jpeg and libjpeg-turbo can be called with:

```
-dct fast -nosmooth -onepass -dither none -scale 1/4
```

. . . and that will speed things up. There is a corresponding drop in the quality of decoded images, but it's probably not something you care about.

In some programs, it is possible to disable output altogether, or at least use an output format that is computationally inexpensive. For example, with image transcoding tools, converting to a BMP file will be a lot faster than to PNG.

With some laid-back parsers, enabling "strict" mode (i.e., bailing out after first error) may result in smaller files and improved run time without sacrificing coverage; for example, for sqlite, you may want to specify -bail.

If the program is still too slow, you can use strace -tt or an equivalent profiling tool to see if the targeted binary is doing anything silly. Sometimes, you can speed things up simply by specifying /dev/null as the config file, or disabling some compile-time features that aren't really needed for the job (try ./configure –help). One of the notoriously resource-consuming things would be calling other utilities via exec*(), popen(), system(), or equivalent calls; for example, tar can invoke external decompression tools when it decides that the input file is a compressed archive.

Some programs may also intentionally call **sleep()**, **usleep()**, or **nanosleep()**; vim is a good example of that. Other programs may attempt **fsync()** and so on. There are third-party libraries that make it easy to get rid of such code, e.g.:

https://launchpad.net/libeatmydata

In programs that are slow due to unavoidable initialization overhead, you may want to try the LLVM deferred fork-server mode (see llvm_mode/README.llvm), which can give you speed gains up to 10x, as mentioned above.

Last but not least, if you are using ASAN and the performance is unacceptable, consider turning it off for now, and manually examining the generated corpus with an ASAN-enabled binary later on.

5) Instrument just what you need

Instrument just the libraries you actually want to stress-test right now, one at a time. Let the program use system-wide, non-instrumented libraries for any functionality you don't actually want to fuzz. For example, in most cases, it doesn't make to instrument libgmp just because you're testing a crypto app that relies on it for bignum math.

Beware of programs that come with oddball third-party libraries bundled with their source code (Spidermonkey is a good example of this). Check ./configure options to use non-instrumented system-wide copies instead.

6) Parallelize your fuzzers

The fuzzer is designed to need ~1 core per job. This means that on a, say, 4-core system, you can easily run four parallel fuzzing jobs with relatively little performance hit. For tips on how to do that, see *Tips for parallel fuzzing*.

The **afl-gotcpu** utility can help you understand if you still have idle CPU capacity on your system. (It won't tell you about memory bandwidth, cache misses, or similar factors, but they are less likely to be a concern.)

7) Keep memory use and timeouts in check

If you have increased the -m or -t limits more than truly necessary, consider dialing them back down.

For programs that are nominally very fast, but get sluggish for some inputs, you can also try setting -t values that are more punishing than what afl-fuzz dares to use on its own. On fast and idle machines, going down to -t 5 may be a viable plan.

The -m parameter is worth looking at, too. Some programs can end up spending a fair amount of time allocating and initializing megabytes of memory when presented with pathological inputs. Low -m values can make them give up sooner and not waste CPU time.

8) Check OS configuration

There are several OS-level factors that may affect fuzzing speed:

- High system load. Use idle machines where possible. Kill any non-essential CPU hogs (idle browser windows, media players, complex screensavers, etc).
- Network filesystems, either used for fuzzer input / output, or accessed by the fuzzed binary to read configuration files (pay special attention to the home directory - many programs search it for dot-files).
- On-demand CPU scaling. The Linux 'ondemand' governor performs its analysis on a particular schedule and is known to underestimate the needs of short-lived processes spawned by afl-fuzz (or any other fuzzer). On Linux, this can be fixed with:

```
cd /sys/devices/system/cpu
echo performance | tee cpu*/cpufreq/scaling_governor
```

On other systems, the impact of CPU scaling will be different; when fuzzing, use OS-specific tools to find out if all cores are running at full speed.

- Transparent huge pages. Some allocators, such as jemalloc, can incur a heavy fuzzing penalty when transparent huge pages (THP) are enabled in the kernel. You can disable this via:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

- Suboptimal scheduling strategies. The significance of this will vary from one target to another, but on Linux, you may want to make sure that the following options are set:

```
echo 1 >/proc/sys/kernel/sched_child_runs_first
echo 1 >/proc/sys/kernel/sched_autogroup_enabled
```

Setting a different scheduling policy for the fuzzer process - say SCHED_RR - can usually speed things up, too, but needs to be done with care.

9) If all other options fail, use -d

For programs that are genuinely slow, in cases where you really can't escape using huge input files, or when you simply want to get quick and dirty results early on, you can always resort to the **`-d`** mode.

The mode causes afl-fuzz to skip all the deterministic fuzzing steps, which makes output a lot less neat and can ultimately make the testing a bit less in-depth, but it will give you an experience more familiar from other fuzzing tools.

1.8.2 AFL "Life Pro Tips"

Bite-sized advice for those who understand the basics, but can't be bothered to read or memorize every other piece of documentation for AFL.

Tip: Get more bang for your buck by using fuzzing dictionaries. See **`dictionaries/README.dictionaries`** to learn how.

Tip: You can get the most out of your hardware by parallelizing AFL jobs. See *Tips for parallel fuzzing* for step-by-step tips.

Tip: Improve the odds of spotting memory corruption bugs with libdislocator.so! It's easy. Consult libdislocator/README.dislocator for usage tips.

Tip: Want to understand how your target parses a particular input file? Try the bundled afl-analyze tool; it's got colors and all!

Tip: You can visually monitor the progress of your fuzzing jobs. Run the bundled afl-plot utility to generate browser-friendly graphs.

Tip: Need to monitor AFL jobs programmatically? Check out the fuzzer_stats file in the AFL output dir or try afl-whatsup.

Tip: Puzzled by something showing up in red or purple in the AFL UI? It could be important - consult *Understanding the status screen* right away!

Tip: Know your target? Convert it to persistent mode for a huge performance gain! Consult section #5 in llvm_mode/README.llvm for tips.

Tip: Using clang? Check out llvm_mode/ for a faster alternative to afl-gcc!

Tip: Did you know that AFL can fuzz closed-source or cross-platform binaries? Check out qemu_mode/README.qemu for more.

Tip: Did you know that afl-fuzz can minimize any test case for you? Try the bundled afl-tmin tool - and get small repro files fast!

Tip: Not sure if a crash is exploitable? AFL can help you figure it out. Specify -C to enable the peruvian were-rabbit mode. See *Crash triage* for more.

Tip: Trouble dealing with a machine uprising? Relax, we've all been there. Find essential survival tips at http://lcamtuf.coredump.cx/prep/.

Tip: AFL-generated corpora can be used to power other testing processes. See *The AFL approach* for inspiration - it tends to pay off!

Tip: Want to automatically spot non-crashing memory handling bugs? Try running an AFL-generated corpus through ASAN, MSAN, or Valgrind.

Tip: Good selection of input files is critical to a successful fuzzing job. See section *Fuzzing with afl-fuzz* in README (or *Performance Tips*) for pro tips.

Tip: You can improve the odds of automatically spotting stack corruption issues. Specify **`AFL_HARDEN=1`** in the environment to enable hardening flags.

Tip: Bumping into problems with non-reproducible crashes? It happens, but usually isn't hard to diagnose. See the bottom of *Interpreting output* for tips.

Tip: Fuzzing is not just about memory corruption issues in the codebase. Add some sanity-checking assert() / abort() statements to effortlessly catch logic bugs.

Tip: Hey kid... pssst... want to figure out how AFL really works? Check out *How AFL works* for all the gory details in one place!

Tip: There's a ton of third-party helper tools designed to work with AFL! Be sure to check out *Related projects* before writing your own.

Tip: Need to fuzz the command-line arguments of a particular program? You can find a simple solution in experimental/argv_fuzzing.

Tip: Attacking a format that uses checksums? Remove the checksum-checking code or use a postprocessor! See experimental/post_library/ for more.

Tip: Dealing with a very slow target or hoping for instant results? Specify -d when calling afl-fuzz!

1.9 Limitations

Here are some of the most important caveats for AFL:

- AFL detects faults by checking for the first spawned process dying due to a signal (SIGSEGV, SIGABRT, etc). Programs that install custom handlers for these signals may need to have the relevant code commented out. In the same vein, faults in child processed spawned by the fuzzed target may evade detection unless you manually add some code to catch that.
- As with any other brute-force tool, the fuzzer offers limited coverage if encryption, checksums, cryptographic signatures, or compression are used to wholly wrap the actual data format to be tested.

  To work around this, you can comment out the relevant checks (see **`experimental/libpng_no_checksum/`** for inspiration); if this is not possible, you can also write a postprocessor, as explained in **`experimental/post_library/`**.
- There are some unfortunate trade-offs with ASAN and 64-bit binaries. This isn't due to any specific fault of afl-fuzz; see *Using ASAN with AFL* for tips.

- There is no direct support for fuzzing network services, background daemons, or interactive apps that require UI interaction to work. You may need to make simple code changes to make them behave in a more traditional way. Preeny may offer a relatively simple option, too - see: https://github.com/zardus/preeny

  Some useful tips for modifying network-based services can be also found at: https://www.fastly.com/blog/how-to-fuzz-server-american-fuzzy-lop

- AFL doesn't output human-readable coverage data. If you want to monitor coverage, use afl-cov from Michael Rash: https://github.com/mrash/afl-cov

- Occasionally, sentient machines rise against their creators. If this happens to you, please consult http://lcamtuf.coredump.cx/prep/.

Beyond this, see INSTALL for platform-specific tips.

1.9.1 Risks

Please keep in mind that, similarly to many other computationally-intensive tasks, fuzzing may put strain on your hardware and on the OS. In particular:

- Your CPU will run hot and will need adequate cooling. In most cases, if cooling is insufficient or stops working properly, CPU speeds will be automatically throttled. That said, especially when fuzzing on less suitable hardware (laptops, smartphones, etc), it's not entirely impossible for something to blow up.

- Targeted programs may end up erratically grabbing gigabytes of memory or filling up disk space with junk files. AFL tries to enforce basic memory limits, but can't prevent each and every possible mishap. The bottom line is that you shouldn't be fuzzing on systems where the prospect of data loss is not an acceptable risk.

- Fuzzing involves billions of reads and writes to the filesystem. On modern systems, this will be usually heavily cached, resulting in fairly modest "physical" I/O - but there are many factors that may alter this equation. It is your responsibility to monitor for potential trouble; with very heavy I/O, the lifespan of many HDDs and SSDs may be reduced.

  A good way to monitor disk I/O on Linux is the 'iostat' command:

```
$ iostat -d 3 -x -k [...optional disk ID...]
```

1.10 More about AFL

1.10.1 History

This doc talks about the rationale of some of the high-level design decisions for American Fuzzy Lop. It's adopted from a discussion with Rob Graham.

Influences

In short, afl-fuzz is inspired chiefly by the work done by Tavis Ormandy back in 2007. Tavis did some very persuasive experiments using gcov block coverage to select optimal test cases out of a large corpus of data, and then using them as a starting point for traditional fuzzing workflows.

(By "persuasive", I mean: netting a significant number of interesting vulnerabilities.)

In parallel to this, both Tavis and I were interested in evolutionary fuzzing. Tavis had his experiments, and I was working on a tool called bunny-the-fuzzer, released somewhere in 2007.

Bunny used a generational algorithm not much different from afl-fuzz, but also tried to reason about the relationship between various input bits and the internal state of the program, with hopes of deriving some additional value from that. The reasoning / correlation part was probably in part inspired by other projects done around the same time by Will Drewry and Chris Evans.

The state correlation approach sounded very sexy on paper, but ultimately, made the fuzzer complicated, brittle, and cumbersome to use; every other target program would require a tweak or two. Because Bunny didn't fare a whole lot better than less sophisticated brute-force tools, I eventually decided to write it off. You can still find its original documentation at:

https://code.google.com/p/bunny-the-fuzzer/wiki/BunnyDoc

There has been a fair amount of independent work, too. Most notably, a few weeks earlier that year, Jared DeMott had a Defcon presentation about a coverage-driven fuzzer that relied on coverage as a fitness function.

Jared's approach was by no means identical to what afl-fuzz does, but it was in the same ballpark. His fuzzer tried to explicitly solve for the maximum coverage with a single input file; in comparison, afl simply selects for cases that do something new (which yields better results - see *How AFL works*).

A few years later, Gabriel Campana released fuzzgrind, a tool that relied purely on Valgrind and a constraint solver to maximize coverage without any brute-force bits; and Microsoft Research folks talked extensively about their still non-public, solver-based SAGE framework.

In the past six years or so, I've also seen a fair number of academic papers that dealt with smart fuzzing (focusing chiefly on symbolic execution) and a couple papers that discussed proof-of-concept applications of genetic algorithms with the same goals in mind. I'm unconvinced how practical most of these experiments were; I suspect that many of them suffer from the bunny-the-fuzzer's curse of being cool on paper and in carefully designed experiments, but failing the ultimate test of being able to find new, worthwhile security bugs in otherwise well-fuzzed, real-world software.

In some ways, the baseline that the "cool" solutions have to compete against is a lot more impressive than it may seem, making it difficult for competitors to stand out. For a singular example, check out the work by Gynvael and Mateusz Jurczyk, applying "dumb" fuzzing to ffmpeg, a prominent and security-critical component of modern browsers and media players:

http://googleonlinesecurity.blogspot.com/2014/01/ffmpeg-and-thousand-fixes.html

Effortlessly getting comparable results with state-of-the-art symbolic execution in equally complex software still seems fairly unlikely, and hasn't been demonstrated in practice so far.

But I digress; ultimately, attribution is hard, and glorying the fundamental concepts behind AFL is probably a waste of time. The devil is very much in the often-overlooked details, which brings us to...

Design goals for afl-fuzz

In short, I believe that the current implementation of afl-fuzz takes care of several itches that seemed impossible to scratch with other tools:

1) Speed. It's genuinely hard to compete with brute force when your "smart" approach is resource-intensive. If your instrumentation makes it 10x more likely to find a bug, but runs 100x slower, your users are getting a bad deal.

   To avoid starting with a handicap, afl-fuzz is meant to let you fuzz most of the intended targets at roughly their native speed - so even if it doesn't add value, you do not lose much.

   On top of this, the tool leverages instrumentation to actually reduce the amount of work in a couple of ways: for example, by carefully trimming the corpus or skipping non-functional but non-trimmable regions in the input files.

2) Rock-solid reliability. It's hard to compete with brute force if your approach is brittle and fails unexpectedly. Automated testing is attractive because it's simple to use and scalable; anything that goes against these principles is an unwelcome trade-off and means that your tool will be used less often and with less consistent results.

   Most of the approaches based on symbolic execution, taint tracking, or complex syntax-aware instrumentation are currently fairly unreliable with real-world targets. Perhaps more importantly, their failure modes can render them strictly worse than "dumb" tools, and such degradation can be difficult for less experienced users to notice and correct.

   In contrast, afl-fuzz is designed to be rock solid, chiefly by keeping it simple. In fact, at its core, it's designed to be just a very good traditional fuzzer with a wide range of interesting, well-researched strategies to go by. The fancy parts just help it focus the effort in places where it matters the most.

3) Simplicity. The author of a testing framework is probably the only person who truly understands the impact of all the settings offered by the tool - and who can dial them in just right. Yet, even the most rudimentary fuzzer frameworks often come with countless knobs and fuzzing ratios that need to be guessed by the operator ahead of the time. This can do more harm than good.

   AFL is designed to avoid this as much as possible. The three knobs you can play with are the output file, the memory limit, and the ability to override the default, auto-calibrated timeout. The rest is just supposed to work. When it doesn't, user-friendly error messages outline the probable causes and workarounds, and get you back on track right away.

4) Chainability. Most general-purpose fuzzers can't be easily employed against resource-hungry or interaction-heavy tools, necessitating the creation of custom in-process fuzzers or the investment of massive CPU power (most of which is wasted on tasks not directly related to the code we actually want to test).

   AFL tries to scratch this itch by allowing users to use more lightweight targets (e.g., standalone image parsing libraries) to create small corpora of interesting test cases that can be fed into a manual testing process or a UI harness later on.

As mentioned in *How AFL works*, AFL does all this not by systematically applying a single overarching CS concept, but by experimenting with a variety of small, complementary methods that were shown to reliably yields results better than chance. The use of instrumentation is a part of that toolkit, but is far from being the most important one.

Ultimately, what matters is that afl-fuzz is designed to find cool bugs - and has a pretty robust track record of doing just that.

1.10.2 How AFL works

Technical "whitepaper" for afl-fuzz.

This document provides a quick overview of the guts of American Fuzzy Lop.

Design statement

American Fuzzy Lop does its best not to focus on any singular principle of operation and not be a proof-of-concept for any specific theory. The tool can be thought of as a collection of hacks that have been tested in practice, found to be surprisingly effective, and have been implemented in the simplest, most robust way I could think of at the time.

Many of the resulting features are made possible thanks to the availability of lightweight instrumentation that served as a foundation for the tool, but this mechanism should be thought of merely as a means to an end. The only true governing principles are speed, reliability, and ease of use.

Coverage measurements

The instrumentation injected into compiled programs captures branch (edge) coverage, along with coarse branch-taken hit counts. The code injected at branch points is essentially equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

The cur_location value is generated randomly to simplify the process of linking complex projects and keep the XOR output distributed uniformly.

The shared_mem[] array is a 64 kB SHM region passed to the instrumented binary by the caller. Every byte set in the output map can be thought of as a hit for a particular (branch_src, branch_dst) tuple in the instrumented code.

The size of the map is chosen so that collisions are sporadic with almost all of the intended targets, which usually sport between 2k and 10k discoverable branch points:

Branch cnt	Colliding tuples	Example targets
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	•

At the same time, its size is small enough to allow the map to be analyzed in a matter of microseconds on the receiving end, and to effortlessly fit within L2 cache.

This form of coverage provides considerably more insight into the execution path of the program than simple block coverage. In particular, it trivially distinguishes between the following execution traces:

```
A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)
A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)
```

This aids the discovery of subtle fault conditions in the underlying code, because security vulnerabilities are more often associated with unexpected or incorrect state transitions than with merely reaching a new basic block.

The reason for the shift operation in the last line of the pseudocode shown earlier in this section is to preserve the directionality of tuples (without this, A ^ B would be indistinguishable from B ^ A) and to retain the identity of tight loops (otherwise, A ^ A would be obviously equal to B ^ B).

The absence of simple saturating arithmetic opcodes on Intel CPUs means that the hit counters can sometimes wrap around to zero. Since this is a fairly unlikely and localized event, it's seen as an acceptable performance trade-off.

Detecting new behaviors

The fuzzer maintains a global map of tuples seen in previous executions; this data can be rapidly compared with individual traces and updated in just a couple of dword- or qword-wide instructions and a simple loop.

When a mutated input produces an execution trace containing new tuples, the corresponding input file is preserved and routed for additional processing later on (see section #3). Inputs that do not trigger new local-scale state transitions in the execution trace (i.e., produce no new tuples) are discarded, even if their overall control flow sequence is unique.

This approach allows for a very fine-grained and long-term exploration of program state while not having to perform any computationally intensive and fragile global comparisons of complex execution traces, and while avoiding the scourge of path explosion.

To illustrate the properties of the algorithm, consider that the second trace shown below would be considered substantially new because of the presence of new tuples (CA, AE):

```
#1: A -> B -> C -> D -> E
#2: A -> B -> C -> A -> E
```

At the same time, with #2 processed, the following pattern will not be seen as unique, despite having a markedly different overall execution path:

```
#3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E
```

In addition to detecting new tuples, the fuzzer also considers coarse tuple hit counts. These are divided into several buckets:

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

To some extent, the number of buckets is an implementation artifact: it allows an in-place mapping of an 8-bit counter generated by the instrumentation to an 8-position bitmap relied on by the fuzzer executable to keep track of the already-seen execution counts for each tuple.

Changes within the range of a single bucket are ignored; transition from one bucket to another is flagged as an interesting change in program control flow, and is routed to the evolutionary process outlined in the section below.

The hit count behavior provides a way to distinguish between potentially interesting control flow changes, such as a block of code being executed twice when it was normally hit only once. At the same time, it is fairly insensitive to empirically less notable changes, such as a loop going from 47 cycles to 48. The counters also provide some degree of "accidental" immunity against tuple collisions in dense trace maps.

The execution is policed fairly heavily through memory and execution time limits; by default, the timeout is set at 5x the initially-calibrated execution speed, rounded up to 20 ms. The aggressive timeouts are meant to prevent dramatic fuzzer performance degradation by descending into tarpits that, say, improve coverage by 1% while being 100x slower; we pragmatically reject them and hope that the fuzzer will find a less expensive way to reach the same code. Empirical testing strongly suggests that more generous time limits are not worth the cost.

Evolving the input queue

Mutated test cases that produced new state transitions within the program are added to the input queue and used as a starting point for future rounds of fuzzing. They supplement, but do not automatically replace, existing finds.

In contrast to more greedy genetic algorithms, this approach allows the tool to progressively explore various disjoint and possibly mutually incompatible features of the underlying data format, as shown in this image:

http://lcamtuf.coredump.cx/afl/afl_gzip.png

Several practical examples of the results of this algorithm are discussed here:

http://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html http://lcamtuf.blogspot.com/2014/11/afl-fuzz-nobody-expects-cdata-sections.html

The synthetic corpus produced by this process is essentially a compact collection of "hmm, this does something new!" input files, and can be used to seed any other testing processes down the line (for example, to manually stress-test resource-intensive desktop apps).

With this approach, the queue for most targets grows to somewhere between 1k and 10k entries; approximately 10-30% of this is attributable to the discovery of new tuples, and the remainder is associated with changes in hit counts.

The following table compares the relative ability to discover file syntax and explore program states when using several different approaches to guided fuzzing. The instrumented target was GNU patch 2.7.3 compiled with -O3 and seeded with a dummy text file; the session consisted of a single pass over the input queue with afl-fuzz:

Fuzzer guidance strategy used	Blocks reached	Edges reached	Edge hit cnt var	Highest-coverage test case generated
(Initial file)	156	163	1.00	(none)
Blind fuzzing S	182	205	2.23	First 2 B of RCS diff
Blind fuzzing L	228	265	2.23	First 4 B of -c mode diff
Block coverage	855	1,130	1.57	Almost-valid RCS diff
Edge coverage	1,452	2,070	2.18	One-chunk -c mode diff
AFL model	1,765	2,597	4.99	Four-chunk -c mode diff

The first entry for blind fuzzing ("S") corresponds to executing just a single round of testing; the second set of figures ("L") shows the fuzzer running in a loop for a number of execution cycles comparable with that of the instrumented runs, which required more time to fully process the growing queue.

Roughly similar results have been obtained in a separate experiment where the fuzzer was modified to compile out all the random fuzzing stages and leave just a series of rudimentary, sequential operations such as walking bit flips. Because this mode would be incapable of altering the size of the input file, the sessions were seeded with a valid unified diff:

Queue extension strategy used	Blocks reached	Edges reached	Edge hit cnt var	Number of unique crashes found
(Initial file)	624	717	1.00	•
Blind fuzzing	1,101	1,409	1.60	0
Block coverage	1,255	1,649	1.48	0
Edge coverage	1,259	1,734	1.72	0
AFL model	1,452	2,040	3.16	1

At noted earlier on, some of the prior work on genetic fuzzing relied on maintaining a single test case and evolving it to maximize coverage. At least in the tests described above, this "greedy" approach appears to confer no substantial benefits over blind fuzzing strategies.

Culling the corpus

The progressive state exploration approach outlined above means that some of the test cases synthesized later on in the game may have edge coverage that is a strict superset of the coverage provided by their ancestors.

To optimize the fuzzing effort, AFL periodically re-evaluates the queue using a fast algorithm that selects a smaller subset of test cases that still cover every tuple seen so far, and whose characteristics make them particularly favorable to the tool.

The algorithm works by assigning every queue entry a score proportional to its execution latency and file size; and then selecting lowest-scoring candidates for each tuple.

The tuples are then processed sequentially using a simple workflow:

1) Find next tuple not yet in the temporary working set,

2) Locate the winning queue entry for this tuple,

3) Register *all* tuples present in that entry's trace in the working set,

4) Go to #1 if there are any missing tuples in the set.

The generated corpus of "favored" entries is usually 5-10x smaller than the starting data set. Non-favored entries are not discarded, but they are skipped with varying probabilities when encountered in the queue:

- If there are new, yet-to-be-fuzzed favorites present in the queue, 99% of non-favored entries will be skipped to get to the favored ones.
- If there are no new favorites:
  - If the current non-favored entry was fuzzed before, it will be skipped 95% of the time.
  - If it hasn't gone through any fuzzing rounds yet, the odds of skipping drop down to 75%.

Based on empirical testing, this provides a reasonable balance between queue cycling speed and test case diversity.

Slightly more sophisticated but much slower culling can be performed on input or output corpora with afl-cmin. This tool permanently discards the redundant entries and produces a smaller corpus suitable for use with afl-fuzz or external tools.

Trimming input files

File size has a dramatic impact on fuzzing performance, both because large files make the target binary slower, and because they reduce the likelihood that a mutation would touch important format control structures, rather than redundant data blocks. This is discussed in more detail in *Performance Tips*.

The possibility that the user will provide a low-quality starting corpus aside, some types of mutations can have the effect of iteratively increasing the size of the generated files, so it is important to counter this trend.

Luckily, the instrumentation feedback provides a simple way to automatically trim down input files while ensuring that the changes made to the files have no impact on the execution path.

The built-in trimmer in afl-fuzz attempts to sequentially remove blocks of data with variable length and stepover; any deletion that doesn't affect the checksum of the trace map is committed to disk. The trimmer is not designed to be particularly thorough; instead, it tries to strike a balance between precision and the number of execve() calls spent on the process, selecting the block size and stepover to match. The average per-file gains are around 5-20%.

The standalone afl-tmin tool uses a more exhaustive, iterative algorithm, and also attempts to perform alphabet normalization on the trimmed files. The operation of afl-tmin is as follows.

First, the tool automatically selects the operating mode. If the initial input crashes the target binary, afl-tmin will run in non-instrumented mode, simply keeping any tweaks that produce a simpler file but still crash the target. If the target is non-crashing, the tool uses an instrumented mode and keeps only the tweaks that produce exactly the same execution path.

The actual minimization algorithm is:

1) Attempt to zero large blocks of data with large stepovers. Empirically, this is shown to reduce the number of execs by preempting finer-grained efforts later on.

2) Perform a block deletion pass with decreasing block sizes and stepovers, binary-search-style.

3) Perform alphabet normalization by counting unique characters and trying to bulk-replace each with a zero value.

4) As a last result, perform byte-by-byte normalization on non-zero bytes.

Instead of zeroing with a 0x00 byte, afl-tmin uses the ASCII digit '0'. This is done because such a modification is much less likely to interfere with text parsing, so it is more likely to result in successful minimization of text files.

The algorithm used here is less involved than some other test case minimization approaches proposed in academic work, but requires far fewer executions and tends to produce comparable results in most real-world applications.

Fuzzing strategies

The feedback provided by the instrumentation makes it easy to understand the value of various fuzzing strategies and optimize their parameters so that they work equally well across a wide range of file types. The strategies used by afl-fuzz are generally format-agnostic and are discussed in more detail here:

http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html

It is somewhat notable that especially early on, most of the work done by afl-fuzz is actually highly deterministic, and progresses to random stacked modifications and test case splicing only at a later stage. The deterministic strategies include:

- Sequential bit flips with varying lengths and stepovers,
- Sequential addition and subtraction of small integers,
- Sequential insertion of known interesting integers (0, 1, INT_MAX, etc),

The purpose of opening with deterministic steps is related to their tendency to produce compact test cases and small diffs between the non-crashing and crashing inputs.

With deterministic fuzzing out of the way, the non-deterministic steps include stacked bit flips, insertions, deletions, arithmetics, and splicing of different test cases.

The relative yields and execve() costs of all these strategies have been investigated and are discussed in the aforementioned blog post.

For the reasons discussed in *History* (chiefly, performance, simplicity, and reliability), AFL generally does not try to reason about the relationship between specific mutations and program states; the fuzzing steps are nominally blind, and are guided only by the evolutionary design of the input queue.

That said, there is one (trivial) exception to this rule: when a new queue entry goes through the initial set of deterministic fuzzing steps, and tweaks to some regions in the file are observed to have no effect on the checksum of the execution path, they may be excluded from the remaining phases of deterministic fuzzing - and the fuzzer may proceed straight to random tweaks. Especially for verbose, human-readable data formats, this can reduce the number of execs by 10-40% or so without an appreciable drop in coverage. In extreme cases, such as normally block-aligned tar archives, the gains can be as high as 90%.

Because the underlying "effector maps" are local every queue entry and remain in force only during deterministic stages that do not alter the size or the general layout of the underlying file, this mechanism appears to work very reliably and proved to be simple to implement.

Dictionaries

The feedback provided by the instrumentation makes it easy to automatically identify syntax tokens in some types of input files, and to detect that certain combinations of predefined or auto-detected dictionary terms constitute a valid grammar for the tested parser.

A discussion of how these features are implemented within afl-fuzz can be found here:

http://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

In essence, when basic, typically easily-obtained syntax tokens are combined together in a purely random manner, the instrumentation and the evolutionary design of the queue together provide a feedback mechanism to differentiate between meaningless mutations and ones that trigger new behaviors in the instrumented code - and to incrementally build more complex syntax on top of this discovery.

The dictionaries have been shown to enable the fuzzer to rapidly reconstruct the grammar of highly verbose and complex languages such as JavaScript, SQL, or XML; several examples of generated SQL statements are given in the blog post mentioned above.

Interestingly, the AFL instrumentation also allows the fuzzer to automatically isolate syntax tokens already present in an input file. It can do so by looking for run of bytes that, when flipped, produce a consistent change to the program's execution path; this is suggestive of an underlying atomic comparison to a predefined value baked into the code. The fuzzer relies on this signal to build compact "auto dictionaries" that are then used in conjunction with other fuzzing strategies.

De-duping crashes

De-duplication of crashes is one of the more important problems for any competent fuzzing tool. Many of the naive approaches run into problems; in particular, looking just at the faulting address may lead to completely unrelated issues being clustered together if the fault happens in a common library function (say, strcmp, strcpy); while checksumming call stack backtraces can lead to extreme crash count inflation if the fault can be reached through a number of different, possibly recursive code paths.

The solution implemented in afl-fuzz considers a crash unique if any of two conditions are met:

- The crash trace includes a tuple not seen in any of the previous crashes,
- The crash trace is missing a tuple that was always present in earlier faults.

The approach is vulnerable to some path count inflation early on, but exhibits a very strong self-limiting effect, similar to the execution path analysis logic that is the cornerstone of afl-fuzz.

Investigating crashes

The exploitability of many types of crashes can be ambiguous; afl-fuzz tries to address this by providing a crash exploration mode where a known-faulting test case is fuzzed in a manner very similar to the normal operation of the fuzzer, but with a constraint that causes any non-crashing mutations to be thrown away.

A detailed discussion of the value of this approach can be found here:

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

The method uses instrumentation feedback to explore the state of the crashing program to get past the ambiguous faulting condition and then isolate the newly-found inputs for human review.

On the subject of crashes, it is worth noting that in contrast to normal queue entries, crashing inputs are *not* trimmed; they are kept exactly as discovered to make it easier to compare them to the parent, non-crashing entry in the queue. That said, afl-tmin can be used to shrink them at will.

The fork server

To improve performance, afl-fuzz uses a "fork server", where the fuzzed process goes through execve(), linking, and libc initialization only once, and is then cloned from a stopped process image by leveraging copy-on-write. The implementation is described in more detail here:

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

The fork server is an integral aspect of the injected instrumentation and simply stops at the first instrumented function to await commands from afl-fuzz.

With fast targets, the fork server can offer considerable performance gains, usually between 1.5x and 2x. It is also possible to:

- Use the fork server in manual ("deferred") mode, skipping over larger, user-selected chunks of initialization code. It requires very modest code changes to the targeted program, and With some targets, can produce 10x+ performance gains.

- Enable "persistent" mode, where a single process is used to try out multiple inputs, greatly limiting the overhead of repetitive fork() calls. This generally requires some code changes to the targeted program, but can improve the performance of fast targets by a factor of 5 or more - approximating the benefits of in-process fuzzing jobs while still maintaining very robust isolation between the fuzzer process and the targeted binary.

Parallelization

The parallelization mechanism relies on periodically examining the queues produced by independently-running instances on other CPU cores or on remote machines, and then selectively pulling in the test cases that, when tried out locally, produce behaviors not yet seen by the fuzzer at hand.

This allows for extreme flexibility in fuzzer setup, including running synced instances against different parsers of a common data format, often with synergistic effects.

For more information about this design, see *Tips for parallel fuzzing*.

Binary-only instrumentation

Instrumentation of black-box, binary-only targets is accomplished with the help of a separately-built version of QEMU in "user emulation" mode. This also allows the execution of cross-architecture code - say, ARM binaries on x86.

QEMU uses basic blocks as translation units; the instrumentation is implemented on top of this and uses a model roughly analogous to the compile-time hooks:

```
if (block_address > elf_text_start && block_address < elf_text_end) {

  cur_location = (block_address >> 4) ^ (block_address << 8);
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;

}
```

The shift-and-XOR-based scrambling in the second line is used to mask the effects of instruction alignment.

The start-up of binary translators such as QEMU, DynamoRIO, and PIN is fairly slow; to counter this, the QEMU mode leverages a fork server similar to that used for compiler-instrumented code, effectively spawning copies of an already-initialized process paused at _start.

First-time translation of a new basic block also incurs substantial latency. To eliminate this problem, the AFL fork server is extended by providing a channel between the running emulator and the parent process. The channel is used to notify the parent about the addresses of any newly-encountered blocks and to add them to the translation cache that will be replicated for future child processes.

As a result of these two optimizations, the overhead of the QEMU mode is roughly 2-5x, compared to 100x+ for PIN.

The afl-analyze tool

The file format analyzer is a simple extension of the minimization algorithm discussed earlier on; instead of attempting to remove no-op blocks, the tool performs a series of walking byte flips and then annotates runs of bytes in the input file.

It uses the following classification scheme:

- "No-op blocks" - segments where bit flips cause no apparent changes to control flow. Common examples may be comment sections, pixel data within a bitmap file, etc.

- "Superficial content" - segments where some, but not all, bitflips produce some control flow changes. Examples may include strings in rich documents (e.g., XML, RTF).
- "Critical stream" - a sequence of bytes where all bit flips alter control flow in different but correlated ways. This may be compressed data, non-atomically compared keywords or magic values, etc.
- "Suspected length field" - small, atomic integer that, when touched in any way, causes a consistent change to program control flow, suggestive of a failed length check.
- "Suspected cksum or magic int" - an integer that behaves similarly to a length field, but has a numerical value that makes the length explanation unlikely. This is suggestive of a checksum or other "magic" integer.
- "Suspected checksummed block" - a long block of data where any change always triggers the same new execution path. Likely caused by failing a checksum or a similar integrity check before any subsequent parsing takes place.
- "Magic value section" - a generic token where changes cause the type of binary behavior outlined earlier, but that doesn't meet any of the other criteria. May be an atomically compared keyword or so.

1.11 Related projects

This doc lists some of the projects that are inspired by, derived from, designed for, or meant to integrate with AFL. See README for the general instruction manual.

1.11.1 Support for other languages / environments:

Python AFL (Jakub Wilk)

Allows fuzz-testing of Python programs. Uses custom instrumentation and its own forkserver.

http://jwilk.net/software/python-afl

Go-fuzz (Dmitry Vyukov)

AFL-inspired guided fuzzing approach for Go targets:

https://github.com/dvyukov/go-fuzz

afl.rs (Keegan McAllister)

Allows Rust features to be easily fuzzed with AFL (using the LLVM mode).

https://github.com/kmcallister/afl.rs

OCaml support (KC Sivaramakrishnan)

Adds AFL-compatible instrumentation to OCaml programs.

https://github.com/ocamllabs/opam-repo-dev/pull/23 http://canopy.mirage.io/Posts/Fuzzing

AFL for GCJ Java and other GCC frontends (-)

GCC Java programs are actually supported out of the box - simply rename afl-gcc to afl-gcj. Unfortunately, by default, unhandled exceptions in GCJ do not result in abort() being called, so you will need to manually add a top-level exception handler that exits with SIGABRT or something equivalent.

Other GCC-supported languages should be fairly easy to get working, but may face similar problems. See https://gcc.gnu.org/frontends.html for a list of options.

AFL-style in-process fuzzer for LLVM (Kostya Serebryany)

Provides an evolutionary instrumentation-guided fuzzing harness that allows some programs to be fuzzed without the fork / execve overhead. (Similar functionality is now available as the "persistent" feature described in ../llvm_mode/README.llvm.)

http://llvm.org/docs/LibFuzzer.html

AFL fixup shim (Ben Nagy)

Allows AFL_POST_LIBRARY postprocessors to be written in arbitrary languages that don't have C / .so bindings. Includes examples in Go.

https://github.com/bnagy/aflfix

TriforceAFL (Tim Newsham and Jesse Hertz)

Leverages QEMU full system emulation mode to allow AFL to target operating systems and other alien worlds:

https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/

WinAFL (Ivan Fratric)

As the name implies, allows you to fuzz Windows binaries (using DynamoRio).

https://github.com/ivanfratric/winafl

Another Windows alternative may be:

https://github.com/carlosgprado/BrundleFuzz/

1.11.2 Network fuzzing:

Preeny (Yan Shoshitaishvili)

Provides a fairly simple way to convince dynamically linked network-centric programs to read from a file or not fork. Not AFL-specific, but described as useful by many users. Some assembly required.

https://github.com/zardus/preeny

1.11.3 Distributed fuzzing and related automation:

roving (Richo Healey)

A client-server architecture for effortlessly orchestrating AFL runs across a fleet of machines. You don't want to use this on systems that face the Internet or live in other untrusted environments.

https://github.com/richo/roving

Distfuzz-AFL (Martijn Bogaard)

Simplifies the management of afl-fuzz instances on remote machines. The author notes that the current implementation isn't secure and should not be exposed on the Internet.

https://github.com/MartijnB/disfuzz-afl

AFLDFF (quantumvm)

A nice GUI for managing AFL jobs.

https://github.com/quantumvm/AFLDFF

afl-launch (Ben Nagy)

Batch AFL launcher utility with a simple CLI.

https://github.com/bnagy/afl-launch

AFL Utils (rc0r)

Simplifies the triage of discovered crashes, start parallel instances, etc.

https://github.com/rc0r/afl-utils

Another crash triage tool:

https://github.com/floyd-fuh/afl-crash-analyzer

afl-fuzzing-scripts (Tobias Ospelt)

Simplifies starting up multiple parallel AFL jobs.

https://github.com/floyd-fuh/afl-fuzzing-scripts/

afl-sid (Jacek Wielemborek)

Allows users to more conveniently build and deploy AFL via Docker.

https://github.com/d33tah/afl-sid

Another Docker-related project:

https://github.com/ozzyjohnson/docker-afl

afl-monitor (Paul S. Ziegler)

Provides more detailed and versatile statistics about your running AFL jobs.

https://github.com/reflare/afl-monitor

1.11.4 Crash triage, coverage analysis, and other companion tools:

afl-crash-analyzer (Tobias Ospelt)

Makes it easier to navigate and annotate crashing test cases.

https://github.com/floyd-fuh/afl-crash-analyzer/

Crashwalk (Ben Nagy)

AFL-aware tool to annotate and sort through crashing test cases.

https://github.com/bnagy/crashwalk

afl-cov (Michael Rash)

Produces human-readable coverage data based on the output queue of afl-fuzz.

https://github.com/mrash/afl-cov

afl-sancov (Bhargava Shastry)

Similar to afl-cov, but uses clang sanitizer instrumentation.

https://github.com/bshastry/afl-sancov

RecidiVM (Jakub Wilk)

Makes it easy to estimate memory usage limits when fuzzing with ASAN or MSAN.

http://jwilk.net/software/recidivm

aflize (Jacek Wielemborek)

Automatically build AFL-enabled versions of Debian packages.

https://github.com/d33tah/aflize

afl-ddmin-mod (Markus Teufelberger)

A variant of afl-tmin that uses a more sophisticated (but slower) minimization algorithm.

https://github.com/MarkusTeufelberger/afl-ddmin-mod

afl-kit (Kuang-che Wu)

Replacements for afl-cmin and afl-tmin with additional features, such as the ability to filter crashes based on stderr patterns.

https://github.com/kcwu/afl-kit

1.11.5 Narrow-purpose or experimental:

Cygwin support (Ali Rizvi-Santiago)

Pretty self-explanatory. As per the author, this "mostly" ports AFL to Windows. Field reports welcome!

https://github.com/arizvisa/afl-cygwin

Pause and resume scripts (Ben Nagy)

Simple automation to suspend and resume groups of fuzzing jobs.

https://github.com/bnagy/afl-trivia

Static binary-only instrumentation (Aleksandar Nikolich)

Allows black-box binaries to be instrumented statically (i.e., by modifying the binary ahead of the time, rather than translating it on the run). Author reports better performance compared to QEMU, but occasional translation errors with stripped binaries.

https://github.com/vrtadmin/moflow/tree/master/afl-dyninst

AFL PIN (Parker Thompson)

Early-stage Intel PIN instrumentation support (from before we settled on faster-running QEMU).

https://github.com/mothran/aflpin

AFL-style instrumentation in llvm (Kostya Serebryany)

Allows AFL-equivalent instrumentation to be injected at compiler level. This is currently not supported by AFL as-is, but may be useful in other projects.

https://code.google.com/p/address-sanitizer/wiki/AsanCoverage#Coverage_counters

AFL JS (Han Choongwoo)

One-off optimizations to speed up the fuzzing of JavaScriptCore (now likely superseded by LLVM deferred forkserver init - see llvm_mode/README.llvm).

https://github.com/tunz/afl-fuzz-js

AFL harness for fwknop (Michael Rash)

An example of a fairly involved integration with AFL.

https://github.com/mrash/fwknop/tree/master/test/afl

Building harnesses for DNS servers (Jonathan Foote, Ron Bowes)

Two articles outlining the general principles and showing some example code.

https://www.fastly.com/blog/how-to-fuzz-server-american-fuzzy-lop https://goo.gl/j9EgFf

Fuzzer shell for SQLite (Richard Hipp)

A simple SQL shell designed specifically for fuzzing the underlying library.

http://www.sqlite.org/src/artifact/9e7e273da2030371

Support for Python mutation modules (Christian Holler)

https://github.com/choller/afl/blob/master/docs/mozilla/python_modules.txt

Support for selective instrumentation (Christian Holler)

https://github.com/choller/afl/blob/master/docs/mozilla/partial_instrumentation.txt

Kernel fuzzing (Dmitry Vyukov)

A similar guided approach as applied to fuzzing syscalls:

https://github.com/google/syzkaller/wiki/Found-Bugs https://github.com/dvyukov/linux/commit/33787098ffaaa83b8a7ccf519913ac5fd6125931 http://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing%2C%20Vault%202016_0.pdf

Android support (ele7enxxh)

Based on a somewhat dated version of AFL:

https://github.com/ele7enxxh/android-afl

CGI wrapper (floyd)

Facilitates the testing of CGI scripts.

https://github.com/floyd-fuh/afl-cgi-wrapper

Fuzzing difficulty estimation (Marcel Boehme)

A fork of AFL that tries to quantify the likelihood of finding additional paths or crashes at any point in a fuzzing job.

https://github.com/mboehme/pythia

CHAPTER 2

ChangeLog

This is the list of all noteworthy changes made in every public release of the tool. See README for the general instruction manual.

2.1 Staying informed

Want to stay in the loop on major new features? Join our mailing list by sending a mail to <afl-users+subscribe@googlegroups.com>.

Not sure if you should upgrade? The lowest currently recommended version is 2.41b. If you're stuck on an earlier release, it's strongly advisable to get on with the times.

2.2 Version 2.53b (2019-07-25):

- No functional changes. Updated some comments and license headers to comply with the open sourcing guidelines and publish the source code on GitHub.

2.3 Version 2.52b (2017-11-04):

- Upgraded QEMU patches from 2.3.0 to 2.10.0. Required troubleshooting several weird issues. All the legwork done by Andrew Griffiths.
- Added setsid to afl-showmap. See the notes for 2.51b.
- Added target mode (deferred, persistent, qemu, etc) to fuzzer_stats. Requested by Jakub Wilk.
- afl-tmin should now save a partially minimized file when Ctrl-C is pressed. Suggested by Jakub Wilk.
- Added an option for afl-analyze to dump offsets in hex. Suggested by Jakub Wilk.
- Added support for parameters in triage_crashes.sh. Patch by Adam of DC949.

2.4 Version 2.51b (2017-08-30):

- Made afl-tmin call setsid to prevent glibc traceback junk from showing up on the terminal in some distros. Suggested by Jakub Wilk.

2.5 Version 2.50b (2017-08-19):

- Fixed an interesting timing corner case spotted by Jakub Wilk.
- Addressed a libtokencap / pthreads incompatibility issue. Likewise, spotted by Jakub Wilk.
- Added a mention of afl-kit and Pythia.
- Added AFL_FAST_CAL.
- In-place resume now preserves .synced. Suggested by Jakub Wilk.

2.6 Version 2.49b (2017-07-18):

- Added AFL_TMIN_EXACT to allow path constraint for crash minimization.
- Added dates for releases (retroactively for all of 2017).

2.7 Version 2.48b (2017-07-17):

- Added AFL_ALLOW_TMP to permit some scripts to run in /tmp.
- Fixed cwd handling in afl-analyze (similar to the quirk in afl-tmin).
- Made it possible to point -o and -f to the same file in afl-tmin.

2.8 Version 2.47b (2017-07-14):

- Fixed cwd handling in afl-tmin. Spotted by Jakub Wilk.

2.9 Version 2.46b (2017-07-10):

- libdislocator now supports AFL_LD_NO_CALLOC_OVER for folks who do not want to abort on calloc() overflows.
- Made a minor fix to libtokencap. Reported by Daniel Stender.
- Added a small JSON dictionary, inspired on a dictionary done by Jakub Wilk.

2.10 Version 2.45b (2017-07-04):

- Added strstr, strcasestr support to libtokencap. Contributed by Daniel Hodson.
- Fixed a resumption offset glitch spotted by Jakub Wilk.
- There are definitely no bugs in afl-showmap -c now.

2.11 Version 2.44b (2017-06-28):

- Added a visual indicator of ASAN / MSAN mode when compiling. Requested by Jakub Wilk.
- Added support for afl-showmap coredumps (-c). Suggested by Jakub Wilk.
- Added LD_BIND_NOW=1 for afl-showmap by default. Although not really useful, it reportedly helps reproduce some crashes. Suggested by Jakub Wilk.
- Added a note about allocator_may_return_null=1 not always working with ASAN. Spotted by Jakub Wilk.

2.12 Version 2.43b (2017-06-16):

- Added AFL_NO_ARITH to aid in the fuzzing of text-based formats. Requested by Jakub Wilk.

2.13 Version 2.42b (2017-06-02):

- Renamed the R() macro to avoid a problem with llvm_mode in the latest versions of LLVM. Fix suggested by Christian Holler.

2.14 Version 2.41b (2017-04-12):

- Addressed a major user complaint related to timeout detection. Timing out inputs are now binned as "hangs" only if they exceed a far more generous time limit than the one used to reject slow paths.

2.15 Version 2.40b (2017-04-02):

- Fixed a minor oversight in the insertion strategy for dictionary words. Spotted by Andrzej Jackowski.
- Made a small improvement to the havoc block insertion strategy.
- Adjusted color rules for "is it done yet?" indicators.

2.16 Version 2.39b (2017-02-02):

- Improved error reporting in afl-cmin. Suggested by floyd.
- Made a minor tweak to trace-pc-guard support. Suggested by kcc.
- Added a mention of afl-monitor.

2.17 Version 2.38b (2017-01-22):

- Added -mllvm -sanitizer-coverage-block-threshold=0 to trace-pc-guard mode, as suggested by Kostya Serebryany.

2.18 Version 2.37b (2017-01-22):

- Fixed a typo. Spotted by Jakub Wilk.
- Fixed support for make install when using trace-pc. Spotted by Kurt Roeckx.
- Switched trace-pc to trace-pc-guard, which should be considerably faster and is less quirky. Kudos to Konstantin Serebryany (and sorry for dragging my feet).

  Note that for some reason, this mode doesn't perform as well as "vanilla" afl-clang-fast / afl-clang.

2.19 Version 2.36b (2017-01-14):

- Fixed a cosmetic bad free() bug when aborting -S sessions. Spotted by Johannes S.
- Made a small change to afl-whatsup to sort fuzzers by name.
- Fixed a minor issue with malloc(0) in libdislocator. Spotted by Rene Freingruber.
- Changed the clobber pattern in libdislocator to a slightly more reliable one. Suggested by Rene Freingruber.
- Added a note about THP performance. Suggested by Sergey Davidoff.
- Added a somewhat unofficial support for running afl-tmin with a baseline "mask" that causes it to minimize only for edges that are unique to the input file, but not to the "boring" baseline. Suggested by Sami Liedes.
- "Fixed" a getPassName() problem with newer versions of clang. Reported by Craig Young and several other folks.

Yep, I know I have a backlog on several other feature requests. Stay tuned!

2.20 Version 2.35b:

- Fixed a minor cmdline reporting glitch, spotted by Leo Barnes.
- Fixed a silly bug in libdislocator. Spotted by Johannes Schultz.

2.21 Version 2.34b:

- Added a note about afl-tmin to technical_details.txt.
- Added support for AFL_NO_UI, as suggested by Leo Barnes.

2.22 Version 2.33b:

- Added code to strip -Wl,-z,defs and -Wl,–no-undefined for afl-clang-fast, since they interfere with -shared. Spotted and diagnosed by Toby Hutton.
- Added some fuzzing tips for Android.

2.23 Version 2.32b:

- Added a check for AFL_HARDEN combined with AFL_USE_*SAN. Suggested by Hanno Boeck.
- Made several other cosmetic adjustments to cycle timing in the wake of the big tweak made in 2.31b.

2.24 Version 2.31b:

- Changed havoc cycle counts for a marked performance boost, especially with -S / -d. See the discussion of FidgetyAFL in:

  https://groups.google.com/forum/#!topic/afl-users/fOPeb62FZUg

  While this does not implement the approach proposed by the authors of the CCS paper, the solution is a result of digging into that research; more improvements may follow as I do more experiments and get more definitive data.

2.25 Version 2.30b:

- Made minor improvements to persistent mode to avoid the remote possibility of "no instrumentation detected" issues with very low instrumentation densities.
- Fixed a minor glitch with a leftover process in persistent mode. Reported by Jakub Wilk and Daniel Stender.
- Made persistent mode bitmaps a bit more consistent and adjusted the way this is shown in the UI, especially in persistent mode.

2.26 Version 2.29b:

- Made a minor #include fix to llvm_mode. Suggested by Jonathan Metzman.
- Made cosmetic updates to the docs.

2.27 Version 2.28b:

- Added "life pro tips" to docs/.
- Moved testcases/_extras/ to dictionaries/ for visibility.
- Made minor improvements to install scripts.
- Added an important safety tip.

2.28 Version 2.27b:

- Added libtokencap, a simple feature to intercept strcmp / memcmp and generate dictionary entries that can help extend coverage.
- Moved libdislocator to its own dir, added README.
- The demo in experimental/instrumented_cmp is no more.

2.29 Version 2.26b:

- Made a fix for libdislocator.so to compile on MacOS X.
- Added support for DYLD_INSERT_LIBRARIES.
- Renamed AFL_LD_PRELOAD to AFL_PRELOAD.

2.30 Version 2.25b:

- Made some cosmetic updates to libdislocator.so, renamed one env variable.

2.31 Version 2.24b:

- Added libdislocator.so, an experimental, abusive allocator. Try it out with AFL_LD_PRELOAD=/path/to/libdislocator.so when running afl-fuzz.

2.32 Version 2.23b:

- Improved the stability metric for persistent mode binaries. Problem spotted by Kurt Roeckx.
- Made a related improvement that may bring the metric to 100% for those targets.

2.33 Version 2.22b:

- Mentioned the potential conflicts between MSAN / ASAN and FORTIFY_SOURCE. There is no automated check for this, since some distros may implicitly set FORTIFY_SOURCE outside of the compiler's argv[].
- Populated the support for AFL_LD_PRELOAD to all companion tools.
- Made a change to the handling of ./afl-clang-fast -v. Spotted by Jan Kneschke.

2.34 Version 2.21b:

- Added some crash reporting notes for Solaris in docs/INSTALL, as investigated by Martin Carpenter.
- Fixed a minor UI mix-up with havoc strategy stats.

2.35 Version 2.20b:

- Revamped the handling of variable paths, replacing path count with a "stability" score to give users a much better signal. Based on the feedback from Vegard Nossum.
- Made a stability improvement to the syncing behavior with resuming fuzzers. Based on the feedback from Vegard.
- Changed the UI to include current input bitmap density along with total density. Ditto.
- Added experimental support for parallelizing -M.

2.36 Version 2.19b:

- Made a fix to make sure that auto CPU binding happens at non-overlapping times.

2.37 Version 2.18b:

- Made several performance improvements to has_new_bits() and classify_counts(). This should offer a robust performance bump with fast targets.

2.38 Version 2.17b:

- Killed the error-prone and manual -Z option. On Linux, AFL will now automatically bind to the first free core (or complain if there are no free cores left).
- Made some doc updates along these lines.

2.39 Version 2.16b:

- Improved support for older versions of clang (hopefully without breaking anything).
- Moved version data from Makefile to config.h. Suggested by Jonathan Metzman.

2.40 Version 2.15b:

- Added a README section on looking for non-crashing bugs.
- Added license data to several boring files. Contributed by Jonathan Metzman.

2.41 Version 2.14b:

- Added FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION as a macro defined when compiling with afl-gcc and friends. Suggested by Kostya Serebryany.
- Refreshed some of the non-x86 docs.

2.42 Version 2.13b:

- Fixed a spurious build test error with trace-pc and llvm_mode/Makefile. Spotted by Markus Teufelberger.
- Fixed a cosmetic issue with afl-whatsup. Spotted by Brandon Perry.

2.43 Version 2.12b:

- Fixed a minor issue in afl-tmin that can make alphabet minimization less efficient during passes > 1. Spotted by Daniel Binderman.

2.44 Version 2.11b:

- Fixed a minor typo in instrumented_cmp, spotted by Hanno Eissfeldt.
- Added a missing size check for deterministic insertion steps.
- Made an improvement to afl-gotcpu when -Z not used.
- Fixed a typo in post_library_png.so.c in experimental/. Spotted by Kostya Serebryany.

2.45 Version 2.10b:

- Fixed a minor core counting glitch, reported by Tyler Nighswander.

2.46 Version 2.09b:

- Made several documentation updates.
- Added some visual indicators to promote and simplify the use of -Z.

2.47 Version 2.08b:

- Added explicit support for -m32 and -m64 for llvm_mode. Inspired by a request from Christian Holler.
- Added a new benchmarking option, as requested by Kostya Serebryany.

2.48 Version 2.07b:

- Added CPU affinity option (-Z) on Linux. With some caution, this can offer a significant (10%+) performance bump and reduce jitter. Proposed by Austin Seipp.
- Updated afl-gotcpu to use CPU affinity where supported.
- Fixed confusing CPU_TARGET error messages with QEMU build. Spotted by Daniel Komaromy and others.

2.49 Version 2.06b:

- Worked around LLVM persistent mode hiccups with -shared code. Contributed by Christian Holler.
- Added __AFL_COMPILER as a convenient way to detect that something is built under afl-gcc / afl-clang / afl-clang-fast and enable custom optimizations in your code. Suggested by Pedro Corte-Real.
- Upstreamed several minor changes developed by Franjo Ivancic to allow AFL to be built as a library. This is fairly use-specific and may have relatively little appeal to general audiences.

2.50 Version 2.05b:

- Put __sanitizer_cov_module_init & co behind #ifdef to avoid problems with ASAN. Spotted by Christian Holler.

2.51 Version 2.04b:

- Removed indirect-calls coverage from -fsanitize-coverage (since it's redundant). Spotted by Kostya Serebryany.

2.52 Version 2.03b:

- Added experimental -fsanitize-coverage=trace-pc support that goes with some recent additions to LLVM, as implemented by Kostya Serebryany. Right now, this is cumbersome to use with common build systems, so the mode remains undocumented.
- Made several substantial improvements to better support non-standard map sizes in LLVM mode.
- Switched LLVM mode to thread-local execution tracing, which may offer better results in some multithreaded apps.
- Fixed a minor typo, reported by Heiko Eissfeldt.
- Force-disabled symbolization for ASAN, as suggested by Christian Holler.
- AFL_NOX86 renamed to AFL_NO_X86 for consistency.
- Added AFL_LD_PRELOAD to allow LD_PRELOAD to be set for targets without affecting AFL itself. Suggested by Daniel Godas-Lopez.

2.53 Version 2.02b:

- Fixed a "lcamtuf can't count to 16" bug in the havoc stage. Reported by Guillaume Endignoux.

2.54 Version 2.01b:

- Made an improvement to cycle counter color coding, based on feedback from Shai Sarfaty.
- Added a mention of aflize to sister_projects.txt.
- Fixed an installation issue with afl-as, as spotted by ilovezfs.

2.55 Version 2.00b:

- Cleaned up color handling after a minor snafu in 1.99b (affecting some terminals).
- Made minor updates to the documentation.

2.56 Version 1.99b:

- Substantially revamped the output and the internal logic of afl-analyze.
- Cleaned up some of the color handling code and added support for background colors.
- Removed some stray files (oops).
- Updated docs to better explain afl-analyze.

2.57 Version 1.98b:

- Improved to "boring string" detection in afl-analyze.
- Added technical_details.txt for afl-analyze.

2.58 Version 1.97b:

- Added afl-analyze, a nifty tool to analyze the structure of a file based on the feedback from AFL instrumentation. This is kinda experimental, so field reports welcome.
- Added a mention of afl-cygwin.
- Fixed a couple of typos, as reported by Jakub Wilk and others.

2.59 Version 1.96b:

- Added -fpic to CFLAGS for the clang plugin, as suggested by Hanno Boeck.
- Made another clang change (IRBuilder) suggested by Jeff Trull.
- Fixed several typos, spotted by Jakub Wilk.
- Added support for AFL_SHUFFLE_QUEUE, based on discussions with Christian Holler.

2.60 Version 1.95b:

- Fixed a harmless bug when handling -B. Spotted by Jacek Wielemborek.
- Made the exit message a bit more accurate when AFL_EXIT_WHEN_DONE is set.
- Added some error-checking for old-style forkserver syntax. Suggested by Ben Nagy.
- Switched from exit() to _exit() in injected code to avoid snafus with destructors in C++ code. Spotted by sunblate.

- Made a change to avoid spuriously setting __AFL_SHM_ID when AFL_DUMB_FORKSRV is set in conjunction with -n. Spotted by Jakub Wilk.

2.61 Version 1.94b:

- Changed allocator alignment to improve support for non-x86 systems (now that llvm_mode makes this more feasible).
- Fixed a minor typo in afl-cmin. Spotted by Jonathan Neuschafer.
- Fixed an obscure bug that would affect people trying to use afl-gcc with $TMP set but $TMPDIR absent. Spotted by Jeremy Barnes.

2.62 Version 1.93b:

- Hopefully fixed a problem with MacOS X and persistent mode, spotted by Leo Barnes.

2.63 Version 1.92b:

- Made yet another C++ fix (namespaces). Reported by Daniel Lockyer.

2.64 Version 1.91b:

- Made another fix to make 1.90b actually work properly with C++ (d'oh). Problem spotted by Daniel Lockyer.

2.65 Version 1.90b:

- Fixed a minor typo spotted by Kai Zhao; and made several other minor updates to docs.
- Updated the project URL for python-afl. Requested by Jakub Wilk.
- Fixed a potential problem with deferred mode signatures getting optimized out by the linker (with –gc-sections).

2.66 Version 1.89b:

- Revamped the support for persistent and deferred forkserver modes. Both now feature simpler syntax and do not require companion env variables. Suggested by Jakub Wilk.
- Added a bit more info about afl-showmap. Suggested by Jacek Wielemborek.

2.67 Version 1.88b:

- Made AFL_EXIT_WHEN_DONE work in non-tty mode. Issue spotted by Jacek Wielemborek.

2.68 Version 1.87b:

- Added QuickStartGuide.txt, a one-page quick start doc.
- Fixed several typos spotted by Dominique Pelle.
- Revamped several parts of README.

2.69 Version 1.86b:

- Added support for AFL_SKIP_CRASHES, which is a very hackish solution to the problem of resuming sessions with intermittently crashing inputs.
- Removed the hard-fail terminal size check, replaced with a dynamic warning shown in place of the UI. Based on feedback from Christian Holler.
- Fixed a minor typo in show_stats. Spotted by Dingbao Xie.

2.70 Version 1.85b:

- Fixed a garbled sentence in notes on parallel fuzzing. Thanks to Jakub Wilk.
- Fixed a minor glitch in afl-cmin. Spotted by Jonathan Foote.

2.71 Version 1.84b:

- Made SIMPLE_FILES behave as expected when naming backup directories for crashes and hangs.
- Added the total number of favored paths to fuzzer_stats. Requested by Ben Nagy.
- Made afl-tmin, afl-fuzz, and afl-cmin reject negative values passed to -t and -m, since they generally won't work as expected.
- Made a fix for no lahf / sahf support on older versions of FreeBSD. Patch contributed by Alex Moneger.

2.72 Version 1.83b:

- Fixed a problem with xargs -d on non-Linux systems in afl-cmin. Spotted by teor2345 and Ben Nagy.
- Fixed an implicit declaration in LLVM mode on MacOS X. Reported by Kai Zhao.

2.73 Version 1.82b:

- Fixed a harmless but annoying race condition in persistent mode - signal delivery is a bit more finicky than I thought.
- Updated the documentation to explain persistent mode a bit better.
- Tweaked AFL_PERSISTENT to force AFL_NO_VAR_CHECK.

2.74 Version 1.81b:

- Added persistent mode for in-process fuzzing. See llvm_mode/README.llvm. Inspired by Kostya Serebryany and Christian Holler.
- Changed the in-place resume code to preserve crashes/README.txt. Suggested by Ben Nagy.
- Included a potential fix for LLVM mode issues on MacOS X, based on the investigation done by teor2345.

2.75 Version 1.80b:

- Made afl-cmin tolerant of whitespaces in filenames. Suggested by Jonathan Neuschafer and Ketil Froyn.
- Added support for AFL_EXIT_WHEN_DONE, as suggested by Michael Rash.

2.76 Version 1.79b:

- Added support for dictionary levels, see testcases/README.testcases.
- Reworked the SQL dictionary to use levels.
- Added a note about Preeny.

2.77 Version 1.78b:

- Added a dictionary for PDF, contributed by Ben Nagy.
- Added several references to afl-cov, a new tool by Michael Rash.
- Fixed a problem with crash reporter detection on MacOS X, as reported by Louis Dassy.

2.78 Version 1.77b:

- Extended the -x option to support single-file dictionaries.
- Replaced factory-packaged dictionaries with file-based variants.
- Removed newlines from HTML keywords in testcases/_extras/html/.

2.79 Version 1.76b:

- Very significantly reduced the number of duplicate execs during deterministic checks, chiefly in int16 and int32 stages. Confirmed identical path yields. This should improve early-stage efficiency by around 5-10%.
- Reduced the likelihood of duplicate non-deterministic execs by bumping up lowest stacking factor from 1 to 2. Quickly confirmed that this doesn't seem to have significant impact on coverage with libpng.
- Added a note about integrating afl-fuzz with third-party tools.

2.80 Version 1.75b:

- Improved argv_fuzzing to allow it to emit empty args. Spotted by Jakub Wilk.
- afl-clang-fast now defines __AFL_HAVE_MANUAL_INIT. Suggested by Jakub Wilk.
- Fixed a libtool-related bug with afl-clang-fast that would make some ./configure invocations generate incorrect output. Spotted by Jakub Wilk.
- Removed flock() on Solaris. This means no locking on this platform, but so be it. Problem reported by Martin Carpenter.
- Fixed a typo. Reported by Jakub Wilk.

2.81 Version 1.74b:

- Added an example argv[] fuzzing wrapper in experimental/argv_fuzzing. Reworked the bash example to be faster, too.
- Clarified llvm_mode prerequisites for FreeBSD.
- Improved afl-tmin to use /tmp if cwd is not writeable.
- Removed redundant includes for sys/fcntl.h, which caused warnings with some nitpicky versions of libc.
- Added a corpus of basic HTML tags that parsers are likely to pay attention to (no attributes).
- Added EP_EnabledOnOptLevel0 to llvm_mode, so that the instrumentation is inserted even when AFL_DONT_OPTIMIZE=1 is set.
- Switched qemu_mode to use the newly-released QEMU 2.3.0, which contains a couple of minor bugfixes.

2.82 Version 1.73b:

- Fixed a bug in effector maps that could sometimes cause AFL to fuzz slightly more than necessary; and in very rare circumstances, could lead to SEGV if eff_map is aligned with page boundary and followed by an unmapped page. Spotted by Jonathan Gray.

2.83 Version 1.72b:

- Fixed a glitch in non-x86 install, spotted by Tobias Ospelt.
- Added a minor safeguard to llvm_mode Makefile following a report from Kai Zhao.

2.84 Version 1.71b:

- Fixed a bug with installed copies of AFL trying to use QEMU mode. Spotted by G.M. Lime.
- Added last path / crash / hang times to fuzzer_stats, suggested by Richard Hipp.
- Fixed a typo, thanks to Jakub Wilk.

2.85 Version 1.70b:

- Modified resumption code to reuse the original timeout value when resuming a session if -t is not given. This prevents timeout creep in continuous fuzzing.
- Added improved error messages for failed handshake when AFL_DEFER_FORKSRV is set.
- Made a slight improvement to llvm_mode/Makefile based on feedback from Jakub Wilk.
- Refreshed several bits of documentation.
- Added a more prominent note about the MacOS X trade-offs to Makefile.

2.86 Version 1.69b:

- Added support for deferred initialization in LLVM mode. Suggested by Richard Godbee.

2.87 Version 1.68b:

- Fixed a minor PRNG glitch that would make the first seconds of a fuzzing job deterministic. Thanks to Andreas Stieger.
- Made tmp[] static in the LLVM runtime to keep Valgrind happy (this had no impact on anything else). Spotted by Richard Godbee.
- Clarified the footnote in README.

2.88 Version 1.67b:

- Made one more correction to llvm_mode Makefile, spotted by Jakub Wilk.

2.89 Version 1.66b:

- Added CC / CXX support to llvm_mode Makefile. Requested by Charlie Eriksen.
- Fixed 'make clean' with gmake. Suggested by Oliver Schneider.
- Fixed 'make -j n clean all'. Suggested by Oliver Schneider.
- Removed build date and time from banners to give people deterministic builds. Requested by Jakub Wilk.

2.90 Version 1.65b:

- Fixed a snafu with some leftover code in afl-clang-fast.
- Corrected even moar typos.

2.91 Version 1.64b:

- Further simplified afl-clang-fast runtime by reverting .init_array to __attribute__((constructor(0)). This should improve compatibility with non-ELF platforms.
- Fixed a problem with afl-clang-fast and -shared libraries. Simplified the code by getting rid of .preinit_array and replacing it with a .comm object. Problem reported by Charlie Eriksen.
- Removed unnecessary instrumentation density adjustment for the LLVM mode. Reported by Jonathan Neuschafer.

2.92 Version 1.63b:

- Updated cgroups_asan/ with a new version from Sam, made a couple changes to streamline it and keep parallel afl instances in separate groups.
- Fixed typos, thanks to Jakub Wilk.

2.93 Version 1.62b:

- Improved the handling of -x in afl-clang-fast,
- Improved the handling of low AFL_INST_RATIO settings for QEMU and LLVM modes.
- Fixed the llvm-config bug for good (thanks to Tobias Ospelt).

2.94 Version 1.61b:

- Fixed an obscure bug compiling OpenSSL with afl-clang-fast. Patch by Laszlo Szekeres.
- Fixed a 'make install' bug on non-x86 systems, thanks to Tobias Ospelt.
- Fixed a problem with half-broken llvm-config on Odroid, thanks to Tobias Ospelt. (There is another odd bug there that hasn't been fully fixed - TBD).

2.95 Version 1.60b:

- Allowed experimental/llvm_instrumentation/ to graduate to llvm_mode/.
- Removed experimental/arm_support/, since it's completely broken and likely unnecessary with LLVM support in place.
- Added ASAN cgroups script to experimental/asan_cgroups/, updated existing docs. Courtesy Sam Hakim and David A. Wheeler.
- Refactored afl-tmin to reduce the number of execs in common use cases. Ideas from Jonathan Neuschafer and Turo Lamminen.
- Added a note about CLAs at the bottom of README.
- Renamed testcases_readme.txt to README.testcases for some semblance of consistency.
- Made assorted updates to docs.

- Added MEM_BARRIER() to afl-showmap and afl-tmin, just to be safe.

2.96 Version 1.59b:

- Imported Laszlo Szekeres' experimental LLVM instrumentation into experimental/llvm_instrumentation. I'll work on including it in the "mainstream" version soon.
- Fixed another typo, thanks to Jakub Wilk.

2.97 Version 1.58b:

- Added a workaround for abort() behavior in -lpthread programs in QEMU mode. Spotted by Aidan Thornton.
- Made several documentation updates, including links to the static instrumentation tool (sister_projects.txt).

2.98 Version 1.57b:

- Fixed a problem with exception handling on some versions of MacOS X. Spotted by Samir Aguiar and Anders Wang Kristensen.
- Tweaked afl-gcc to use BIN_PATH instead of a fixed string in help messages.

2.99 Version 1.56b:

- Renamed related_work.txt to historical_notes.txt.
- Made minor edits to the ASAN doc.
- Added docs/sister_projects.txt with a list of inspired or closely related utilities.

2.100 Version 1.55b:

- Fixed a glitch with afl-showmap opening /dev/null with O_RDONLY when running in quiet mode. Spotted by Tyler Nighswander.

2.101 Version 1.54b:

- Added another postprocessor example for PNG.
- Made a cosmetic fix to realloc() handling in experimental/post_library/, suggested by Jakub Wilk.
- Improved -ldl handling. Suggested by Jakub Wilk.

2.102 Version 1.53b:

- Fixed an -l ordering issue that is apparently still a problem on Ubuntu. Spotted by William Robinet.

2.103 Version 1.52b:

- Added support for file format postprocessors. Requested by Ben Nagy. This feature is intentionally buried, since it's fairly easy to misuse and useful only in some scenarios. See experimental/post_library/.

2.104 Version 1.51b:

- Made it possible to properly override LD_BIND_NOW after one very unusual report of trouble.
- Cleaned up typos, thanks to Jakub Wilk.
- Fixed a bug in AFL_DUMB_FORKSRV.

2.105 Version 1.50b:

- Fixed a flock() bug that would prevent dir reuse errors from kicking in every now and then.
- Renamed references to ppvm (the project is now called recidivm).
- Made improvements to file descriptor handling to avoid leaving some fds unnecessarily open in the child process.
- Fixed a typo or two.

2.106 Version 1.49b:

- Added code to save original command line in fuzzer_stats and crashes/README.txt. Also saves fuzzer version in fuzzer_stats. Requested by Ben Nagy.

2.107 Version 1.48b:

- Fixed a bug with QEMU fork server crashes when translation is attempted after a jump to an invalid pointer in the child process (i.e., after bumping into a particularly nasty security bug in the tested binary). Reported by Tyler Nighswander.

2.108 Version 1.47b:

- Fixed a bug with afl-cmin in -Q mode complaining about binary being not instrumented. Thanks to Jonathan Neuschafer for the bug report.
- Fixed another bug with argv handling for afl-fuzz in -Q mode. Reported by Jonathan Neuschafer.
- Improved the use of colors when showing crash counts in -C mode.

2.109 Version 1.46b:

- Improved instrumentation performance on 32-bit systems by getting rid of xor-swap (oddly enough, xor-swap is still faster on 64-bit) and tweaking alignment.
- Made path depth numbers more accurate with imported test cases.

2.110 Version 1.45b:

- Added support for SIMPLE_FILES in config.h for folks who don't like descriptive file names. Generates very simple names without colons, commas, plus signs, dashes, etc.
- Replaced zero-sized files with symlinks in the variable behavior state dir to simplify examining the relevant test cases.
- Changed the period of limited-range block ops from 5 to 10 minutes based on a couple of experiments. The basic goal of this delay timer behavior is to better support jobs that are seeded with completely invalid files, in which case, the first few queue cycles may be completed very quickly without discovering new paths. Should have no effect on well-seeded jobs.
- Made several minor updates to docs.

2.111 Version 1.44b:

- Corrected two bungled attempts to get the -C mode work properly with afl-cmin (accounting for the short-lived releases tagged 1.42 and 1.43b) - sorry.
- Removed AFL_ALLOW_CRASHES in favor of the -C mode in said tool.
- Said goodbye to Hello Kitty, as requested by Padraig Brady.

2.112 Version 1.41b:

- Added AFL_ALLOW_CRASHES=1 to afl-cmin. Allows crashing inputs in the output corpus. Changed the default behavior to disallow it.
- Made the afl-cmin output dir default to 0700, not 0755, to be consistent with afl-fuzz; documented the rationale for 0755 in afl-plot.
- Lowered the output dir reuse time limit to 25 minutes as a dice-roll compromise after a discussion on afl-users@.
- Made afl-showmap accept -o /dev/null without borking out.
- Added support for crash / hang info in exit codes of afl-showmap.
- Tweaked block operation scaling to also factor in ballpark run time in cases where queue passes take very little time.
- Fixed typos and made improvements to several docs.

2.113 Version 1.40b:

- Switched to smaller block op sizes during the first passes over the queue. Helps keep test cases small.
- Added memory barrier for run_target(), just in case compilers get smarter than they are today.
- Updated a bunch of docs.

2.114 Version 1.39b:

- Added the ability to skip inputs by sending SIGUSR1 to the fuzzer.
- Reworked several portions of the documentation.
- Changed the code to reset splicing perf scores between runs to keep them closer to intended length.
- Reduced the minimum value of -t to 5 for afl-fuzz (~200 exec/sec) and to 10 for auxiliary tools (due to the absence of a fork server).
- Switched to more aggressive default timeouts (rounded up to 25 ms versus 50 ms - ~40 execs/sec) and made several other cosmetic changes to the timeout code.

2.115 Version 1.38b:

- Fixed a bug in the QEMU build script, spotted by William Robinet.
- Improved the reporting of skipped bitflips to keep the UI counters a bit more accurate.
- Cleaned up related_work.txt and added some non-goals.
- Fixed typos, thanks to Jakub Wilk.

2.116 Version 1.37b:

- Added effector maps, which detect regions that do not seem to respond to bitflips and subsequently exclude them from more expensive steps (arithmetics, known ints, etc). This should offer significant performance improvements with quite a few types of text-based formats, reducing the number of deterministic execs by a factor of 2 or so.
- Cleaned up mem limit handling in afl-cmin.
- Switched from uname -i to uname -m to work around Gentoo-specific issues with coreutils when building QEMU. Reported by William Robinet.
- Switched from PID checking to flock() to detect running sessions. Problem, against all odds, bumped into by Jakub Wilk.
- Added SKIP_COUNTS and changed the behavior of COVERAGE_ONLY in config.h. Useful only for internal benchmarking.
- Made improvements to UI refresh rates and exec/sec stats to make them more stable.
- Made assorted improvements to the documentation and to the QEMU build script.
- Switched from perror() to strerror() in error macros, thanks to Jakub Wilk for the nag.

- Moved afl-cmin back to bash, wasn't thinking straight. It has to stay on bash because other shells may have restrictive limits on array sizes.

2.117 Version 1.36b:

- Switched afl-cmin over to /bin/sh. Thanks to Jonathan Gray.
- Fixed an off-by-one bug in queue limit check when resuming sessions (could cause NULL ptr deref if you are *really* unlucky).
- Fixed the QEMU script to tolerate i686 if returned by uname -i. Based on a problem report from Sebastien Duquette.
- Added multiple references to Jakub's ppvm tool.
- Made several minor improvements to the Makefile.
- Believe it or not, fixed some typos. Thanks to Jakub Wilk.

2.118 Version 1.35b:

- Cleaned up regular expressions in some of the scripts to avoid errors on *BSD systems. Spotted by Jonathan Gray.

2.119 Version 1.34b:

- Performed a substantial documentation and program output cleanup to better explain the QEMU feature.

2.120 Version 1.33b:

- Added support for AFL_INST_RATIO and AFL_INST_LIBS in the QEMU mode.
- Fixed a stack allocation crash in QEMU mode (bug in QEMU, fixed with an extra patch applied to the downloaded release).
- Added code to test the QEMU instrumentation once the afl-qemu-trace binary is built.
- Modified afl-tmin and afl-showmap to search $PATH for binaries and to better handle QEMU support.
- Added a check for instrumented binaries when passing -Q to afl-fuzz.

2.121 Version 1.32b:

- Fixed 'make install' following the QEMU changes. Spotted by Hanno Boeck.
- Fixed EXTRA_PAR handling in afl-cmin.

2.122 Version 1.31b:

- Hallelujah! Thanks to Andrew Griffiths, we now support very fast, black-box instrumentation of binary-only code. See qemu_mode/README.qemu.

  To use this feature, you need to follow the instructions in that directory and then run afl-fuzz with -Q.

2.123 Version 1.30b:

- Added -s (summary) option to afl-whatsup. Suggested by Jodie Cunningham.
- Added a sanity check in afl-tmin to detect minimization to zero len or excess hangs.
- Fixed alphabet size counter in afl-tmin.
- Slightly improved the handling of -B in afl-fuzz.
- Fixed process crash messages with -m none.

2.124 Version 1.29b:

- Improved the naming of test cases when orig: is already present in the file name.
- Made substantial improvements to technical_details.txt.

2.125 Version 1.28b:

- Made a minor tweak to the instrumentation to preserve the directionality of tuples (i.e., A -> B != B -> A) and to maintain the identity of tight loops (A -> A). You need to recompile targeted binaries to leverage this.
- Cleaned up some of the afl-whatsup stats.
- Added several sanity checks to afl-cmin.

2.126 Version 1.27b:

- Made afl-tmin recursive. Thanks to Hanno Boeck for the tip.
- Added docs/technical_details.txt.
- Changed afl-showmap search strategy in afl-cmap to just look into the same place that afl-cmin is executed from. Thanks to Jakub Wilk.
- Removed current_todo.txt and cleaned up the remaining docs.

2.127 Version 1.26b:

- Added total execs/sec stat for afl-whatsup.
- afl-cmin now auto-selects between cp or ln. Based on feedback from Even Huus.

- Fixed a typo. Thanks to Jakub Wilk.
- Made afl-gotcpu a bit more accurate by using getrusage instead of times. Thanks to Jakub Wilk.
- Fixed a memory limit issue during the build process on NetBSD-current. Reported by Thomas Klausner.

2.128 Version 1.25b:

- Introduced afl-whatsup, a simple tool for querying the status of local synced instances of afl-fuzz.
- Added -x compiler to clang options on Darwin. Suggested by Filipe Cabecinhas.
- Improved exit codes for afl-gotcpu.
- Improved the checks for -m and -t values in afl-cmin. Bug report from Evan Huus.

2.129 Version 1.24b:

- Introduced afl-getcpu, an experimental tool to empirically measure CPU preemption rates. Thanks to Jakub Wilk for the idea.

2.130 Version 1.23b:

- Reverted one change to afl-cmin that actually made it slower.

2.131 Version 1.22b:

- Reworked afl-showmap.c to support normal options, including -o, -q, -e. Also added support for timeouts and memory limits.
- Made changes to afl-cmin and other scripts to accommodate the new semantics.
- Officially retired AFL_EDGES_ONLY.
- Fixed another typo in afl-tmin, courtesy of Jakub Wilk.

2.132 Version 1.21b:

- Graduated minimize_corpus.sh to afl-cmin. It is now a first-class utility bundled with the fuzzer.
- Made significant improvements to afl-cmin to make it faster, more robust, and more versatile.
- Refactored some of afl-tmin code to make it a bit more readable.
- Made assorted changes to the doc to document afl-cmin and other stuff.

2.133 Version 1.20b:

- Added AFL_DUMB_FORKSRV, as requested by Jakub Wilk. This works only in -n mode and allows afl-fuzz to run with "dummy" fork servers that don't output any instrumentation, but follow the same protocol.
- Renamed AFL_SKIP_CHECKS to AFL_SKIP_BIN_CHECK to make it at least somewhat descriptive.
- Switched to using clang as the default assembler on MacOS X to work around Xcode issues with newer builds of clang. Testing and patch by Nico Weber.
- Fixed a typo (via Jakub Wilk).

2.134 Version 1.19b:

- Improved exec failure detection in afl-fuzz and afl-showmap.
- Improved Ctrl-C handling in afl-showmap.
- Added afl-tmin, a handy instrumentation-enabled minimizer.

2.135 Version 1.18b:

- Fixed a serious but short-lived bug in the resumption behavior introduced in version 1.16b.
- Added -t nn+ mode for soft-skipping timing-out paths.

2.136 Version 1.17b:

- Fixed a compiler warning introduced in 1.16b for newer versions of GCC. Thanks to Jakub Wilk and Ilfak Guilfanov.
- Improved the consistency of saving fuzzer_stats, bitmap info, and auto-dictionaries when aborting fuzzing sessions.
- Made several noticeable performance improvements to deterministic arith and known int steps.

2.137 Version 1.16b:

- Added a bit of code to make resumption pick up from the last known offset in the queue, rather than always rewinding to the start. Suggested by Jakub Wilk.
- Switched to tighter timeout control for slow programs (3x rather than 5x average exec speed at init).

2.138 Version 1.15b:

- Added support for AFL_NO_VAR_CHECK to speed up resumption and inhibit variable path warnings for some programs.
- Made the trimmer run even for variable paths, since there is no special harm in doing so and it can be very beneficial if the trimming still pans out.

- Made the UI a bit more descriptive by adding "n/a" instead of "0" in a couple of corner cases.

2.139 Version 1.14b:

- Added a (partial) dictionary for JavaScript.
- Added AFL_NO_CPU_RED, as suggested by Jakub Wilk.
- Tweaked the havoc scaling logic added in 1.12b.

2.140 Version 1.13b:

- Improved the performance of minimize_corpus.sh by switching to a sort-based approach.
- Made several minor revisions to the docs.

2.141 Version 1.12b:

- Made an improvement to dictionary generation to avoid runs of identical bytes.
- Added havoc cycle scaling to help with slow binaries in -d mode. Based on a thread with Sami Liedes.
- Added AFL_SYNC_FIRST for afl-fuzz. This is useful for those who obsess over stats, no special purpose otherwise.
- Switched to more robust box drawing codes, suggested by Jakub Wilk.
- Created faster 64-bit variants of several critical-path bitmap functions (sorry, no difference on 32 bits).
- Fixed moar typos, as reported by Jakub Wilk.

2.142 Version 1.11b:

- Added a bit more info about dictionary strategies to the status screen.

2.143 Version 1.10b:

- Revised the dictionary behavior to use insertion and overwrite in deterministic steps, rather than just the latter. This improves coverage with SQL and the like.
- Added "*" in status_screen.txt, as suggested by Jakub Wilk.

2.144 Version 1.09b:

- Corrected a cosmetic problem with 'extras' stage count not always being accurate in the stage yields view.
- Fixed a typo reported by Jakub Wilk and made some minor documentation improvements.

2.145 Version 1.08b:

- Fixed a div-by-zero bug in the newly-added code when using a dictionary.

2.146 Version 1.07b:

- Added code that automatically finds and extracts syntax tokens from the input corpus.
- Fixed a problem with ld dead-code removal option on MacOS X, reported by Filipe Cabecinhas.
- Corrected minor typos spotted by Jakub Wilk.
- Added a couple of more exotic archive format samples.

2.147 Version 1.06b:

- Switched to slightly more accurate (if still not very helpful) reporting of short read and short write errors. These theoretically shouldn't happen unless you kill the forkserver or run out of disk space. Suggested by Jakub Wilk.
- Revamped some of the allocator and debug code, adding comments and cleaning up other mess.
- Tweaked the odds of fuzzing non-favored test cases to make sure that baseline coverage of all inputs is reached sooner.

2.148 Version 1.05b:

- Added a dictionary for WebP.
- Made some additional performance improvements to minimize_corpus.sh, getting deeper into the bash woods.

2.149 Version 1.04b:

- Made substantial performance improvements to minimize_corpus.sh with large datasets, albeit at the expense of having to switch back to bash (other shells may have limits on array sizes, etc).
- Tweaked afl-showmap to support the format used by the new script.

2.150 Version 1.03b:

- Added code to skip README.txt in the input directory to make the crash exploration mode work better. Suggested by Jakub Wilk.
- Added a dictionary for SQLite.

2.151 Version 1.02b:

- Reverted the ./ search path in minimize_corpus.sh because people did not like it.
- Added very explicit warnings not to run various shell scripts that read or write to /tmp/ (since this is generally a pretty bad idea on multi-user systems).
- Added a check for /tmp binaries and -f locations in afl-fuzz.

2.152 Version 1.01b:

- Added dictionaries for XML and GIF.

2.153 Version 1.00b:

- Slightly improved the performance of minimize_corpus.sh, especially on Linux.
- Made a couple of improvements to calibration timeouts for resumed scans.

2.154 Version 0.99b:

- Fixed minimize_corpus.sh to work with dash, as suggested by Jakub Wilk.
- Modified minimize_corpus.sh to try locate afl-showmap in $PATH and ./. The first part requested by Jakub Wilk.
- Added support for afl-as –version, as required by one funky build script. Reported by William Robinet.

2.155 Version 0.98b:

- Added a dictionary for TIFF.
- Fixed another cosmetic snafu with stage exec counts for -x.
- Switched afl-plot to /bin/sh, since it seems bashism-free. Also tried to remove any obvious bashisms from other experimental/ scripts, most notably including minimize_corpus.sh and triage_crashes.sh. Requested by Jonathan Gray.

2.156 Version 0.97b:

- Fixed cosmetic issues around the naming of -x strategy files.
- Added a dictionary for JPEG.
- Fixed a very rare glitch when running instrumenting 64-bit code that makes heavy use of xmm registers that are also touched by glibc.

2.157 Version 0.96b:

- Added support for extra dictionaries, provided testcases/_extras/png/ as a demo.
- Fixed a minor bug in number formatting routines used by the UI.
- Added several additional PNG test cases that are relatively unlikely to be hit by chance.
- Fixed afl-plot syntax for gnuplot 5.x. Reported by David Necas.

2.158 Version 0.95b:

- Cleaned up the OSX ReportCrash code. Thanks to Tobias Ospelt for help.
- Added some extra tips for AFL_NO_FORKSERVER on OSX.
- Refreshed the INSTALL file.

2.159 Version 0.94b:

- Added in-place resume (-i-) to address a common user complaint.
- Added an awful workaround for ReportCrash on MacOS X. Problem spotted by Joseph Gentle.

2.160 Version 0.93b:

- Fixed the link() workaround, as reported by Jakub Wilk.

2.161 Version 0.92b:

- Added support for reading test cases from another filesystem. Requested by Jakub Wilk.
- Added pointers to the mailing list.
- Added a sample PDF document.

2.162 Version 0.91b:

- Refactored minimize_corpus.sh to make it a bit more user-friendly and to select for smallest files, not largest bitmaps. Offers a modest corpus size improvement in most cases.
- Slightly improved the performance of splicing code.

2.163 Version 0.90b:

- Moved to an algorithm where paths are marked as preferred primarily based on size and speed, rather than bitmap coverage. This should offer noticeable performance gains in many use cases.

- Refactored path calibration code; calibration now takes place as soon as a test case is discovered, to facilitate better prioritization decisions later on.
- Changed the way of marking variable paths to avoid .state metadata inconsistencies.
- Made sure that calibration routines always create a new test case to avoid hypothetical problems with utilities that modify the input file.
- Added bitmap saturation to fuzzer stats and plot data.
- Added a testcase for JPEG XR.
- Added a tty check for the colors warning in Makefile, to keep distro build logs tidy. Suggested by Jakub Wilk.

2.164 Version 0.89b:

- Renamed afl-plot.sh to afl-plot, as requested by Padraig Brady.
- Improved the compatibility of afl-plot with older versions of gnuplot.
- Added banner information to fuzzer_stats, populated it to afl-plot.

2.165 Version 0.88b:

- Added support for plotting, with design and implementation based on a prototype design proposed by Michael Rash. Huge thanks!
- Added afl-plot.sh, which allows you to, well, generate a nice plot using this data.
- Refactored the code slightly to make more frequent updates to fuzzer_stats and to provide more detail about synchronization.
- Added an fflush(stdout) call for non-tty operation, as requested by Joonas Kuorilehto.
- Added some detail to fuzzer_stats for parity with plot_file.

2.166 Version 0.87b:

- Added support for MSAN, via AFL_USE_MSAN, same gotchas as for ASAN.

2.167 Version 0.86b:

- Added AFL_NO_FORKSRV, allowing the forkserver to be bypassed. Suggested by Ryan Govostes.
- Simplified afl-showmap.c to make use of the no-forkserver mode.
- Made minor improvements to crash_triage.sh, as suggested by Jakub Wilk.

2.168 Version 0.85b:

- Fixed the CPU counting code - no sysctlbyname() on OpenBSD, d'oh. Bug reported by Daniel Dickman.
- Made a slight correction to error messages - the advice on testing with ulimit was a tiny bit off by a factor of 1024.

2.169 Version 0.84b:

- Added support for the CPU widget on some non-Linux platforms (I hope). Based on feedback from Ryan Govostes.
- Cleaned up the changelog (very meta).

2.170 Version 0.83b:

- Added experimental/clang_asm_normalize/ and related notes in env_variables.txt and afl-as.c. Thanks to Ryan Govostes for the idea.
- Added advice on hardware utilization in README.

2.171 Version 0.82b:

- Made additional fixes for Xcode support, juggling -Q and -q flags. Thanks to Ryan Govostes.
- Added a check for __asm__ blocks and switches to .intel_syntax in assembly. Based on feedback from Ryan Govostes.

2.172 Version 0.81b:

- A workaround for Xcode 6 as -Q flag glitch. Spotted by Ryan Govostes.
- Improved Solaris build instructions, as suggested by Martin Carpenter.
- Fix for a slightly busted path scoring conditional. Minor practical impact.

2.173 Version 0.80b:

- Added a check for $PATH-induced loops. Problem noticed by Kartik Agaram.
- Added AFL_KEEP_ASSEMBLY for easier troubleshooting.
- Added an override for AFL_USE_ASAN if set at afl compile time. Requested by Hanno Boeck.

2.174 Version 0.79b:

- Made minor adjustments to path skipping logic.
- Made several documentation updates to reflect the path selection changes made in 0.78b.

2.175 Version 0.78b:

- Added a CPU governor check. Bug report from Joe Zbiciak.
- Favored paths are now selected strictly based on new edges, not hit counts. This speeds up the first pass by a factor of 3-6x without significantly impacting ultimate coverage (tested with libgif, libpng, libjpeg).

  It also allows some performance & memory usage improvements by making some of the in-memory bitmaps much smaller.
- Made multiple significant performance improvements to bitmap checking functions, plus switched to a faster hash.
- Owing largely to these optimizations, bumped the size of the bitmap to 64k and added a warning to detect older binaries that rely on smaller bitmaps.

2.176 Version 0.77b:

- Added AFL_SKIP_CHECKS to bypass binary checks when really warranted. Feature requested by Jakub Wilk.
- Fixed a couple of typos.
- Added a warning for runs that are aborted early on.

2.177 Version 0.76b:

- Incorporated another signal handling fix for Solaris. Suggestion submitted by Martin Carpenter.

2.178 Version 0.75b:

- Implemented a slightly more "elegant" kludge for the %llu glitch (see types.h).
- Relaxed CPU load warnings to stay in sync with reality.

2.179 Version 0.74b:

- Switched to more responsive exec speed averages and better UI speed scaling.
- Fixed a bug with interrupted reads on Solaris. Issue spotted by Martin Carpenter.

2.180 Version 0.73b:

- Fixed a stray memcpy() instead of memmove() on overlapping buffers. Mostly harmless but still dumb. Mistake spotted thanks to David Higgs.

2.181 Version 0.72b:

- Bumped map size up to 32k. You may want to recompile instrumented binaries (but nothing horrible will happen if you don't).
- Made huge performance improvements for bit-counting functions.
- Default optimizations now include -funroll-loops. This should have interesting effects on the instrumentation. Frankly, I'm just going to ship it and see what happens next. I have a good feeling about this.
- Made a fix for stack alignment crash on MacOS X 10.10; looks like the rhetorical question in the comments in afl-as.h has been answered. Tracked down by Mudge Zatko.

2.182 Version 0.71b:

- Added a fix for the nonsensical MacOS ELF check. Spotted by Mudge Zatko.
- Made some improvements to ASAN checks.

2.183 Version 0.70b:

- Added explicit detection of ASANified binaries.
- Fixed compilation issues on Solaris. Reported by Martin Carpenter.

2.184 Version 0.69b:

- Improved the detection of non-instrumented binaries.
- Made the crash counter in -C mode accurate.
- Fixed an obscure install bug that made afl-as non-functional with the tool installed to /usr/bin instead of /usr/local/bin. Found by Florian Kiersch.
- Fixed for a cosmetic SIGFPE when Ctrl-C is pressed while the fork server is spinning up.

2.185 Version 0.68b:

- Added crash exploration mode! Woot!

2.186 Version 0.67b:

- Fixed several more typos, the project is now cartified 100% typo-free. Thanks to Thomas Jarosch and Jakub Wilk.
- Made a change to write fuzzer_stats early on.
- Fixed a glitch when (not!) running on MacOS X as root. Spotted by Tobias Ospelt.
- Made it possible to override -O3 in Makefile. Suggested by Jakub Wilk.

2.187 Version 0.66b:

- Fixed a very obscure issue with build systems that use gcc as an assembler for hand-written .s files; this would confuse afl-as. Affected nss, reported by Hanno Boeck.
- Fixed a bug when cleaning up synchronized fuzzer output dirs. Issue reported by Thomas Jarosch.

2.188 Version 0.65b:

- Cleaned up shell printf escape codes in Makefile. Reported by Jakub Wilk.
- Added more color to fuzzer_stats, provided short documentation of the file format, and made several other stats-related improvements.

2.189 Version 0.64b:

- Enabled GCC support on MacOS X.

2.190 Version 0.63b:

- Provided a new, simplified way to pass data in files (@@). See README.
- Made additional fixes for 64-bit MacOS X, working around a crashing bug in their linker (umpf) and several other things. It's alive!
- Added a minor workaround for a bug in 64-bit FreeBSD (clang -m32 -g doesn't work on that platform, but clang -m32 does, so we no longer insert -g).
- Added a build-time warning for inverse video terminals and better instructions in status_screen.txt.

2.191 Version 0.62b:

- Made minor improvements to the allocator, as suggested by Tobias Ospelt.
- Added example instrumented memcmp() in experimental/instrumented_cmp.
- Added a speculative fix for MacOS X (clang detection, again).
- Fixed typos in parallel_fuzzing.txt. Problems spotted by Thomas Jarosch.

2.192 Version 0.61b:

- Fixed a minor issue with clang detection on systems with a clang cc wrapper, so that afl-gcc doesn't confuse it with GCC.
- Made cosmetic improvements to docs and to the CPU load indicator.
- Fixed a glitch with crash removal (README.txt left behind, d'oh).

2.193 Version 0.60b:

- Fixed problems with jump tables generated by exotic versions of GCC. This solves an outstanding problem on OpenBSD when using afl-gcc + PIE (not present with afl-clang).
- Fixed permissions on one of the sample archives.
- Added a lahf / sahf workaround for OpenBSD (their assembler doesn't know about these opcodes).
- Added docs/INSTALL.

2.194 Version 0.59b:

- Modified 'make install' to also install test cases.
- Provided better pointers to installed README in afl-fuzz.
- More work on RLIMIT_AS for OpenBSD.

2.195 Version 0.58b:

- Added a core count check on Linux.
- Refined the code for the lack-of-RLIMIT_AS case on OpenBSD.
- Added a rudimentary CPU utilization meter to help with optimal loading.

2.196 Version 0.57b:

- Made fixes to support FreeBSD and OpenBSD: use_64bit is now inferred if not explicitly specified when calling afl-as, and RLIMIT_AS is behind an #ifdef. Thanks to Fabian Keil and Jonathan Gray for helping troubleshoot this.
- Modified 'make install' to also install docs (in /usr/local/share/doc/afl).
- Fixed a typo in status_screen.txt.
- Made a couple of Makefile improvements as proposed by Jakub Wilk.

2.197 Version 0.56b:

- Added probabilistic instrumentation density reduction in ASAN mode. This compensates for ASAN-specific branches in a crude but workable way.
- Updated notes_for_asan.txt.

2.198 Version 0.55b:

- Implemented smarter out_dir behavior, automatically deleting directories that don't contain anything of special value. Requested by several folks, including Hanno Boeck.
- Added more detail in fuzzer_stats (start time, run time, fuzzer PID).
- Implemented support for configurable install prefixes in Makefile ($PREFIX), as requested by Luca Barbato.
- Made it possible to resume by doing -i <out_dir>, without having to specify -i <out_dir>/queue/.

2.199 Version 0.54b:

- Added a fix for -Wformat warning messages (oops, I thought this had been in place for a while).

2.200 Version 0.53b:

- Redesigned the crash & hang duplicate detection code to better deal with fault conditions that can be reached in a multitude of ways.

  The old approach could be compared to hashing stack traces to de-dupe crashes, a method prone to crash count inflation. The alternative I wanted to avoid would be equivalent to just looking at crash %eip, which can have false negatives in common functions such as memcpy().

  The middle ground currently used in afl-fuzz can be compared to looking at every line item in the stack trace and tagging crashes as unique if we see any function name that we haven't seen before (or if something that we have *always* seen there suddenly disappears). We do the comparison without paying any attention to ordering or hit counts. This can still cause some crash inflation early on, but the problem will quickly taper off. So, you may get 20 dupes instead of 5,000.
- Added a fix for harmless but absurd trim ratios shown if the first exec in the trimmer timed out. Spotted by @EspenGx.

2.201 Version 0.52b:

- Added a quick summary of the contents in experimental/.
- Made a fix to the process of writing fuzzer_stats.
- Slightly reorganized the .state/ directory, now recording redundant paths, too. Note that this breaks the ability to properly resume older sessions - sorry about that.

  (To fix this, simply move "<out_dir>/.state/" *from an older run to <out_dir>/.state/deterministic_done/.*)

2.202 Version 0.51b:

- Changed the search order for afl-as to avoid the problem with older copies installed system-wide; this also means that I can remove the Makefile check for that.
- Made it possible to set instrumentation ratio of 0%.
- Introduced some typos, fixed others.
- Fixed the test_prev target in Makefile, as reported by Ozzy Johnson.

2.203 Version 0.50b:

- Improved the 'make install' logic, as suggested by Padraig Brady.
- Revamped various bits of the documentation, especially around perf_tips.txt; based on the feedback from Alexander Cherepanov.
- Added AFL_INST_RATIO to afl-as. The only case where this comes handy is ffmpeg, at least as far as I can tell. (Trivia: the current version of ffmpeg ./configure also ignores CC and –cc, probably unintentionally).
- Added documentation for all environmental variables (env_variables.txt).
- Implemented a visual warning for excessive or insufficient bitmap density.
- Changed afl-gcc to add -O3 by default; use AFL_DONT_OPTIMIZE if you don't like that. Big speed gain for ffmpeg, so seems like a good idea.
- Made a regression fix to afl-as to ignore .LBB labels in gcc mode.

2.204 Version 0.49b:

- Fixed more typos, as found by Jakub Wilk.
- Added support for clang!
- Changed AFL_HARDEN to *not* include ASAN by default. Use AFL_USE_ASAN if needed. The reasons for this are in notes_for_asan.txt.
- Switched from configure auto-detection to isatty() to keep afl-as and afl-gcc quiet.
- Improved installation process to properly create symlinks, rather than copies of binaries.

2.205 Version 0.48b:

- Improved afl-fuzz to force-set ASAN_OPTIONS=abort_on_error=1. Otherwise, ASAN crashes wouldn't be caught at all. Reported by Hanno Boeck.
- Improved Makefile mkdir logic, as suggested by Hanno Boeck.
- Improved the 64-bit instrumentation to properly save r8-r11 registers in the x86 setup code. The old behavior could cause rare problems running *without* instrumentation when the first function called in a particular .o file has 5+ parameters. No impact on code running under afl-fuzz or afl-showmap. Issue spotted by Padraig Brady.

2.206 Version 0.47b:

- Fixed another Makefile bug for parallel builds of afl. Problem identified by Richard W. M. Jones.
- Added support for suffixes for -m.
- Updated the documentation and added notes_for_asan.txt. Based on feedback from Hanno Boeck, Ben Laurie, and others.
- Moved the project to http://lcamtuf.coredump.cx/afl/.

2.207 Version 0.46b:

- Cleaned up Makefile dependencies for parallel builds. Requested by Richard W. M. Jones.
- Added support for DESTDIR in Makefile. Once again suggested by Richard W. M. Jones :-)
- Removed all the USE_64BIT stuff; we now just auto-detect compilation mode. As requested by many callers to the show.
- Fixed rare problems with programs that use snippets of assembly and switch between .code32 and .code64. Addresses a glitch spotted by Hanno Boeck with compiling ToT gdb.

2.208 Version 0.45b:

- Implemented a test case trimmer. Results in 20-30% size reduction for many types of work loads, with very pronounced improvements in path discovery speeds.
- Added better warnings for various problems with input directories.
- Added a Makefile warning for older copies, based on counterintuitive behavior observed by Hovik Manucharyan.
- Added fuzzer_stats file for status monitoring. Suggested by @dronesec.
- Fixed moar typos, thanks to Alexander Cherepanov.
- Implemented better warnings for ASAN memory requirements, based on calls from several angry listeners.
- Switched to saner behavior with non-tty stdout (less output generated, no ANSI art).

2.209 Version 0.44b:

- Added support for AFL_CC and AFL_CXX, based on a patch from Ben Laurie.
- Replaced afl-fuzz -S -D with -M for simplicity.
- Added a check for .section .text; lack of this prevented main() from getting instrumented for some users. Reported by Tom Ritter.
- Reorganized the testcases/ directory.
- Added an extra check to confirm that the build is operational.
- Made more consistent use of color reset codes, as suggested by Oliver Kunz.

2.210 Version 0.43b:

- Fixed a bug with 64-bit gcc -shared relocs.
- Removed echo -e from Makefile for compatibility with dash. Suggested by Jakub Wilk.
- Added status_screen.txt.
- Added experimental/canvas_harness.
- Made a minor change to the Makefile GCC check. Suggested by Hanno Boeck.

2.211 Version 0.42b:

- Fixed a bug with red zone handling for 64-bit (oops!). Problem reported by Felix Groebert.
- Implemented horribly experimental ARM support in experimental/arm_support.
- Made several improvements to error messages.
- Added AFL_QUIET to silence afl-gcc and afl-as when using wonky build systems. Reported by Hanno Boeck.
- Improved check for 64-bit compilation, plus several sanity checks in Makefile.

2.212 Version 0.41b:

- Fixed a fork served bug for processes that call execve().
- Made minor compatibility fixes to Makefile, afl-gcc; suggested by Jakub Wilk.
- Fixed triage_crashes.sh to work with the new layout of output directories. Suggested by Jakub Wilk.
- Made multiple performance-related improvements to the injected instrumentation.
- Added visual indication of the number of imported paths.
- Fixed afl-showmap to make it work well with new instrumentation.
- Added much better error messages for crashes when importing test cases or otherwise calibrating the binary.

2.213 Version 0.40b:

- Added support for parallelized fuzzing. Inspired by earlier patch from Sebastian Roschke.
- Added an example in experimental/distributed_fuzzing/.

2.214 Version 0.39b:

- Redesigned status screen, now 90% more spiffy.
- Added more verbose and user-friendly messages for some common problems.
- Modified the resumption code to reconstruct path depth.
- Changed the code to inhibit core dumps and improve the ability to detect SEGVs.

- Added a check for redirection of core dumps to programs.
- Made a minor improvement to the handling of variable paths.
- Made additional performance tweaks to afl-fuzz, chiefly around mem limits.
- Added performance_tips.txt.

2.215 Version 0.38b:

- Fixed an fd leak and +cov tracking bug resulting from changes in 0.37b.
- Implemented auto-scaling for screen update speed.
- Added a visual indication when running in non-instrumented mode.

2.216 Version 0.37b:

- Added fuzz state tracking for more seamless resumption of aborted fuzzing sessions.
- Removed the -D option, as it's no longer necessary.
- Refactored calibration code and improved startup reporting.
- Implemented dynamically scaled timeouts, so that you don't need to play with -t except in some very rare cases.
- Added visual notification for slow binaries.
- Improved instrumentation to explicitly cover the other leg of every branch.

2.217 Version 0.36b:

- Implemented fork server support to avoid the overhead of execve(). A nearly-verbatim design from Jann Horn; still pending part 2 that would also skip initial setup steps (thinking about reliable heuristics now).
- Added a check for shell scripts used as fuzz targets.
- Added a check for fuzz jobs that don't seem to be finding anything.
- Fixed the way IGNORE_FINDS works (was a bit broken after adding splicing and path skip heuristics).

2.218 Version 0.35b:

- Properly integrated 64-bit instrumentation into afl-as.

2.219 Version 0.34b:

- Added a new exec count classifier (the working theory is that it gets meaningful coverage with fewer test cases spewed out).

2.220 Version 0.33b:

- Switched to new, somewhat experimental instrumentation that tries to target only arcs, rather than every line. May be fragile, but is a lot faster (2x+).
- Made several other cosmetic fixes and typo corrections, thanks to Jakub Wilk.

2.221 Version 0.32b:

- Another take at fixing the C++ exception thing. Reported by Jakub Wilk.

2.222 Version 0.31b:

- Made another fix to afl-as to address a potential problem with newer versions of GCC (introduced in 0.28b). Thanks to Jann Horn.

2.223 Version 0.30b:

- Added more detail about the underlying operations in file names.

2.224 Version 0.29b:

- Made some general improvements to chunk operations.

2.225 Version 0.28b:

- Fixed C++ exception handling in newer versions of GCC. Problem diagnosed by Eberhard Mattes.
- Fixed the handling of the overflow flag. Once again, thanks to Eberhard Mattes.

2.226 Version 0.27b:

- Added prioritization of new paths over the already-fuzzed ones.
- Included spliced test case ID in the output file name.
- Fixed a rare, cosmetic null ptr deref after Ctrl-C.
- Refactored the code to make copies of test cases in the output directory.
- Switched to better output file names, keeping track of stage and splicing sources.

2.227 Version 0.26b:

- Revamped storage of testcases, -u option removed,
- Added a built-in effort minimizer to get rid of potentially redundant inputs,
- Provided a testcase count minimization script in experimental/,
- Made miscellaneous improvements to directory and file handling.
- Fixed a bug in timeout detection.

2.228 Version 0.25b:

- Improved count-based instrumentation.
- Improved the hang deduplication logic.
- Added -cov prefixes for test cases.
- Switched from readdir() to scandir() + alphasort() to preserve ordering of test cases.
- Added a splicing strategy.
- Made various minor UI improvements and several other bugfixes.

2.229 Version 0.24b:

- Added program name to the status screen, plus the -T parameter to go with it.

2.230 Version 0.23b:

- Improved the detection of variable behaviors.
- Added path depth tracking,
- Improved the UI a bit,
- Switched to simplified (XOR-based) tuple instrumentation.

2.231 Version 0.22b:

- Refactored the handling of long bitflips and some swaps.
- Fixed the handling of gcc -pipe, thanks to anonymous reporter.

2.232 Version 0.21b (2013-11-12):

- Initial public release.