==================================
Technical "whitepaper" for afl-fuzz
==================================

This document provides a quick overview of the guts of American Fuzzy Lop.
See README for the general instruction manual; and for a discussion of
motivations and design goals behind AFL, see historical_notes.txt.

0) 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.

1) 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.

2) 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 are discarded, even if their overall

instrumentation output pattern 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 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.

3) 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.

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

Some of the earlier work on evolutionary fuzzing suggested maintaining just a single test case and selecting for mutations that improve coverage. At least in the tests described above, this "greedy" method appeared to offer no substantial benefits over blind fuzzing.

4) 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.

5) 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 perf_tips.txt.

The possibility of a bad starting corpus provided by the user 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. 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.

6) 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 non-deterministic steps include stacked bit flips, insertions, deletions, arithmetics, and splicing of different test cases.

Their relative yields and execve() costs have been investigated and are

discussed in the aforementioned blog post.

For the reasons discussed in historical_notes.txt (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 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 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.

7) 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.

8) 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.

9) 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.

10) 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. With some targets, this 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. As with the previous mode, this requires custom modifications, but can improve the performance of fast targets by a factor of 5 or more - approximating the benefits of in-process fuzzing jobs.

11) 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 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 parallel_fuzzing.txt.

12) 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.

13) 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.