Efficient Fuzzer Guide

This document describes ways to determine efficiency of a fuzz target and ways to improve it.

Overview

Being a coverage-driven fuzzing engine, libFuzzer considers a certain input *interesting* if it results in new code coverage, i.e. it reaches a code that has not been reached before. The set of all interesting inputs is called *corpus*.

Items in corpus are constantly mutated in search of new interesting inputs. Corpus can be shared across fuzzer runs and grows over time as new code is reached.

There are several metrics you should look at to determine effectiveness of your fuzz target:

You can collect these metrics manually or take them from ClusterFuzz status pages after a fuzz target is checked in Chromium repository.

The following things are extremely useful for improving fuzzing efficiency, so we *strongly recommend* them for any fuzz target:

- Seed Corpus
- Fuzzer Dictionary

There are other ways that are useful in some cases, but not always applicable:

- Custom Options
- Custom Build

Execution Speed

Fuzz target speed is calculated in executions per second. It is printed while a fuzz target is running:

```
#19346  NEW    cov: 2815 bits: 1082 indir: 43 units: 150 exec/s: 19346 L: 62
```

Because libFuzzer performs randomized mutations, it is critical to have it run as fast as possible to navigate through the large search space efficiently and find interesting code paths. You should try to get to at least 1,000 exec/s from your fuzz target locally before submitting it to the Chromium repository.

Initialization/Cleanup

Try to keep `LLVMFuzzerTestOneInput` function as simple as possible. If your fuzzing function is too complex, it can bring down fuzzer execution speed OR it can target very specific usecases and fail to account for unexpected scenarios.

Prefer to use static initialization and shared resources rather than performing setup and teardown on every single input. Checkout example on startup initialization in libFuzzer documentation.

You can skip freeing static resources. However, all resources allocated within `LLVMFuzzerTestOneInput` function should be de-allocated since this function is called millions of times during a fuzzing session. Otherwise, we will hit OOMs frequently and reduce overall fuzzing efficiency.

Memory Usage

Avoid allocation of dynamic memory wherever possible. Memory instrumentation works faster for stack-based and static objects, than for heap allocated ones.

It is always a good idea to try different variants for your fuzz target locally, and then submit the fastest implementation.

Code Coverage

ClusterFuzz status page provides source-level coverage report for fuzz targets from recent runs. Looking at the report might provide an insight on how to improve code coverage of a fuzz target.

You can also generate source-level coverage report locally by running the coverage script stored in Chromium repository. The script provides detailed instructions as well as an usage example.

Note that code coverage of a fuzz target **depends heavily** on the corpus provided when running the target, i.e. code coverage report generated by a fuzz target launched without any corpus would not make much sense.

We encourage you to try out the coverage script, as it usually generates a better code coverage visualization compared to the coverage report hosted on ClusterFuzz. *NOTE: This is an experimental feature and an active area of work. We are working on improving this process.*

Corpus Size

After running for a while, a fuzz target would reach a plateau and may stop discovering new interesting inputs. Corpus for a reasonably complex target should contain hundreds (if not thousands) of items.

Too small of a corpus size indicates that fuzz target is hitting a code barrier and is unable to get past it. Common cases of such issues include: checksums, magic numbers, etc. The easiest way to diagnose this problem is to generate and analyze a coverage report. To fix the issue, you can:

Seed Corpus

Seed corpus is a set of *valid* and *interesting* inputs that serve as starting points for a fuzz target. If one is not provided, a fuzzing engine would have to guess these inputs from scratch, which can take an indefinite amount of time depending on size of the inputs and complexity of the target format.

Seed corpus works especially well for strictly defined file formats and data transmission protocols.

- For file format parsers, add valid files from your test suite.
- For protocol parsers, add valid raw streams from test suite into separate files.

Other examples include a graphics library seed corpus, which would be a variety of small PNG/JPG/GIF files.

If you are running a fuzz target locally, you can pass a corpus directory as an argument:

```
./out/libfuzzer/my_fuzzer ~/tmp/my_fuzzer_corpus
```

The fuzzer would store all the interesting inputs it finds in that directory.

While libFuzzer can start with an empty corpus, seed corpus is always useful and in many cases is able to increase code coverage by an order of magnitude.

ClusterFuzz uses seed corpus defined in Chromium source repository. You need to add a `seed_corpus` attribute to your `fuzzer_test` definition in BUILD.gn file:

```
fuzzer_test("my_protocol_fuzzer") {
  ...
  seed_corpus = "test/fuzz/testcases"
  ...
}
```

You may specify multiple seed corpus directories via `seed_corpuses` attribute:

```
fuzzer_test("my_protocol_fuzzer") {
  ...
  seed_corpuses = [ "test/fuzz/testcases", "test/unittest/data" ]
  ...
}
```

All files found in these directories and their subdirectories will be archived into a `<my_fuzzer_name>_seed_corpus.zip` output archive.

If you can't store seed corpus in Chromium repository (e.g. it is too large, cannot be open sourced, etc), you can upload the corpus to Google Cloud Storage bucket used by ClusterFuzz:

1. Go to Corpus GCS Bucket.
2. Open directory named `<my_fuzzer_name>_static` . If the directory does not exist, please create it.

3. Upload corpus files into the directory.

Alternative and faster way is to use gsutil command line tool:

```
gsutil -m rsync <path_to_corpus> gs://clusterfuzz-
corpus/libfuzzer/<my_fuzzer_name>_static
```

Note that if you upload the corpus to GCS, the `seed_corpus` attribute is not needed in your `fuzzer_test` definition.

Corpus Minimization

It's important to minimize seed corpus to a *small set of interesting inputs* before uploading. The reason being that seed corpus is synced to all fuzzing bots for every iteration, so it is important to keep it small both for fuzzing efficiency and to prevent our bots from running out of disk space.

The minimization can be done using `-merge=1` option of libFuzzer:

```
# Create an empty directory.
mkdir seed_corpus_minimized

# Run the fuzzer with -merge=1 flag.
./my_fuzzer -merge=1 ./seed_corpus_minimized ./seed_corpus
```

After running the command above, `seed_corpus_minimized` directory will contain a minimized corpus that gives the same code coverage as the initial `seed_corpus` directory.

Fuzzer Dictionary

It is very useful to provide fuzz target with a set of *common words or values* that you expect to find in the input. Adding a dictionary highly improves the efficiency of finding new units and works especially well in certain usecases (e.g. fuzzing file format decoders or text based protocols like XML).

To add a dictionary, first create a dictionary file. This is a flat text file where tokens are listed one per line in the format of `name="value"` , where `name` is optional and can be omitted, although it is a convenient way to document the meaning of a particular token. The value must appear in quotes, with hex escaping (\xNN) applied to all non-printable, high-bit, or otherwise problematic characters (\ and " shorthands are recognized too). This syntax is similar to the one used by AFL fuzzing engine (-x option).

An example dictionary looks like:

```
# Lines starting with '#' and empty lines are ignored.

# Adds "blah" word (w/o quotes) to the dictionary.
kw1="blah"
# Use \\ for backslash and \" for quotes.
kw2="\"ac\\dc\""
# Use \xAB for hex values.
kw3="\xF7\xF8"
# Key name before '=' can be omitted:
"foo\x0Abar"
```

Make sure to test your dictionary by running your fuzz target locally:

```
./out/libfuzzer/my_protocol_fuzzer -dict=<path_to_dict> <path_to_corpus>
```

If the dictionary is effective, you should see new units discovered in the output.

To submit a dictionary to Chromium repository:

1. Add the dictionary file in the same directory as your fuzz target
2. Add `dict` attribute to `fuzzer_test` definition in BUILD.gn file:

```
fuzzer_test("my_protocol_fuzzer") {
  ...
  dict = "my_protocol_fuzzer.dict"
}
```

The dictionary will be used automatically by ClusterFuzz once it picks up a new revision build.

Custom Options

Custom options help to fine tune libFuzzer execution parameters and will also override the default values used by ClusterFuzz. Please read libFuzzer options page for detailed documentation on how these work.

Add the options needed in `libfuzzer_options` attribute to your `fuzzer_test` definition in BUILD.gn file:

```
fuzzer_test("my_protocol_fuzzer") {
  ...
  libfuzzer_options = [
    "max_len=2048",
    "use_traces=1",
  ]
}
```

Please note that `dict` parameter should be provided separately. All other options can be passed using `libfuzzer_options` property.

Custom Build

If you need to change the code being tested by your fuzz target, you may use an `#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION` macro in your target code.

Note that patching target code is not a preferred way of improving corresponding fuzz target, but in some cases that might be the only way possible, e.g. when there is no intended API to disable checksum verification, or when target code uses random generator that affects reproducibility of crashes.