Envoy fuzzing improvements

presented by

in collaboration with

Authors

David Korczynski <david@adalogics.com>
Adam Korczynski <adam@adalogics.com>
Date: 29th april, 2021

Table of Contents

Executive summary

In this engagement we performed two tasks. The first was optimising the performance of end-to-end fuzzers and the second was developing a fuzzer for the UDP listener code of Envoy.

Regarding optimisation, we find that the cause of performance issues in the end-to-end fuzzers is due to the large amount of code that is instrumented for coverage-guiding the Envoy fuzzers. Specifically, the Envoy fuzzers run with a large amount of inline 8-bit counters which is a counter inserted by SanitizerCoverage on every edge of the target application. libFuzzer uses the counters by iterating through all of the counters after each fuzz iteration in order to measure code coverage. Thus, the more counters there are, the more effort libFuzzer has to spend iterating through all counters on each fuzz iteration. We show how this impacts Envoy performance in four different ways:

1. The exact same empty fuzzer compiled with fuzzer instrumentation runs 419 times slower when also compiled with Envoy instrumentation.
2. The h2_capture_persistent_fuzz_test fuzzer spends only 20% of the actual execution inside LLVMFuzzerTestOneInput.
3. Profilers reveal an excessive amount of time is spent inside coverage-collection code.
4. The end-to-end fuzzers contain around 1.2 million inline 8-bit counters, which is huge in comparison to other fuzzers, e.g. Lua fuzzers contain 6000 inline 8-bit counters.

We show how limiting the coverage instrumentation improves fuzzer performance and observe the same effect on OSS-Fuzz statistics.

Regarding UDP fuzzing, we develop a UDP fuzzer that targets the code requested by the Envoy team, namely handleReadCallback. We demonstrate code coverage by observing the target code in OSS-Fuzz reports, and also demonstrates that the fuzzer catches the DOS issue previously reported.

In this report we go through both of these tasks and spend considerable detail on the coverage instrumentation part.

1 Envoy fuzzing optimisations

The goal of this task was to improve performance of the end-to-end fuzzers of Envoy. In this section we go through how we achieved this by first giving background information on how libFuzzer uses coverage instrumentation, then we proceed to show how this instrumentation affects the Envoy fuzzers, and finally we show how to improve the performance of the fuzzers by reducing instrumentation.

Executive summary:
The instrumentation of Envoy has a significant slowdown of fuzzer execution. For example, a fuzzer with a single branch runs 419 times slower with Envoy instrumentation than without. Limiting the amount of instrumentation used in the Envoy project shows to increase performance of the fuzzers. This has been added to the OSS-Fuzz build of Envoy, which has shown to increase performance on the OSS-Fuzz logs. There is future work for the Envoy maintainers on refining the instrumentation of the fuzzers.

1.1 Background on how coverage collection in libFuzzer works.

In this section we outline how libFuzzer uses coverage instrumentation to track fuzzer progress. This is not related to Envoy explicitly, however, the content described in this section gives an improved understanding as to why the Envoy fuzzers lack performance.

In the LLVM source code, the file llvm-project/compiler-rt/lib/fuzzer/FuzzerTracePC.cpp contains the logic for tracing coverage. At the bottom of this file you find a set of functions:

```
void __sanitizer_cov_trace_pc_guard(uint32_t *Guard)
void __sanitizer_cov_trace_pc()
void __sanitizer_cov_trace_pc_guard_init(uint32_t *Start, uint32_t *Stop)
void __sanitizer_cov_8bit_counters_init(uint8_t *Start, uint8_t *Stop)
void __sanitizer_cov_pcs_init(const uintptr_t *pcs_beg, const uintptr_t *pcs_end)
void __sanitizer_cov_trace_pc_indir(uintptr_t Callee)
void __sanitizer_cov_trace_cmp8(uint64_t Arg1, uint64_t Arg2)
void __sanitizer_cov_trace_const_cmp8(uint64_t Arg1, uint64_t Arg2)
void __sanitizer_cov_trace_cmp4(uint32_t Arg1, uint32_t Arg2)
void __sanitizer_cov_trace_const_cmp4(uint32_t Arg1, uint32_t Arg2)
void __sanitizer_cov_trace_cmp2(uint16_t Arg1, uint16_t Arg2)
void __sanitizer_cov_trace_const_cmp2(uint16_t Arg1, uint16_t Arg2)
void __sanitizer_cov_trace_cmp1(uint8_t Arg1, uint8_t Arg2)
void __sanitizer_cov_trace_const_cmp1(uint8_t Arg1, uint8_t Arg2)
void __sanitizer_cov_trace_switch(uint64_t Val, uint64_t *Cases)
void __sanitizer_cov_trace_div4(uint32_t Val)
void __sanitizer_cov_trace_div8(uint64_t Val)
void __sanitizer_cov_trace_gep(uintptr_t Idx)
```

These are all callbacks that SanitizerCoverage uses (https://clang.llvm.org/docs/SanitizerCoverage.html#id2) and from a high level perspective the callbacks are used to track execution of code, e.g. basic blocks, in the target. One of the

important callback functions is `__sanitizer_cov_8bit_counters_init` which is described here https://clang.llvm.org/docs/SanitizerCoverage.html#inline-8bit-counters. This callback is used during fuzzing compilation and instructs the compiler to insert inline counter increments on every code edge. The idea is then that these counters can be used to track how many times an edge was hit during execution.

The way libFuzzer uses these 8bit-counters is by iterating through all of them, i.e. iteration through a sequence of counters corresponding to the number of edges in the target application, after each fuzz iteration in order to determine if new coverage has been achieved.

In the `RunOne` function inside the FuzzerLoop.cpp (https://github.com/llvm/llvm-project/blob/28ab7ff2d732fb0580486baa02b1383a72cec0cb/compiler-rt/lib/fuzzer/FuzzerLoop.cpp#L506) source we observe the following code:

```
ExecuteCallback(Data, Size);
auto TimeOfUnit = duration_cast<microseconds>(UnitStopTime - UnitStartTime);

UniqFeatureSetTmp.clear();
size_t FoundUniqFeaturesOfII = 0;
size_t NumUpdatesBefore = Corpus.NumFeatureUpdates();
TPC.CollectFeatures([&](uint32_t Feature) {
  if (Corpus.AddFeature(Feature, static_cast<uint32_t>(Size), Options.Shrink))
    UniqFeatureSetTmp.push_back(Feature);
  if (Options.Entropic)
    Corpus.UpdateFeatureFrequency(II, Feature);
  if (Options.ReduceInputs && II && !II->NeverReduce)
    if (std::binary_search(II->UniqFeatureSet.begin(),
                           II->UniqFeatureSet.end(), Feature))
      FoundUniqFeaturesOfII++;
});
```

This code snippet has two purposes. First, to run `ExecuteCallback` which calls into the fuzzing entry point function (`LLVMFuzzerTestOneInput`). Second,
to gather the code coverage of the target follow the fuzzer's single execution, which is done by `TPC.CollectFeatures`. The `TPC.CollectFeatures` code begins as follows:

```
template <class Callback> // void Callback(uint32_t Feature)
ATTRIBUTE_NO_SANITIZE_ADDRESS ATTRIBUTE_NOINLINE size_t
TracePC::CollectFeatures(Callback HandleFeature) const {
  auto Handle8bitCounter = [&](size_t FirstFeature,
                               size_t Idx, uint8_t Counter) {
    if (UseCounters)
      HandleFeature(static_cast<uint32_t>(FirstFeature + Idx * 8 +
                                          CounterToFeature(Counter)));
    else
      HandleFeature(static_cast<uint32_t>(FirstFeature + Idx));
  };
```

```
        size_t FirstFeature = 0;

        for (size_t i = 0; i < NumModules; i++) {
          for (size_t r = 0; r < Modules[i].NumRegions; r++) {
            if (!Modules[i].Regions[r].Enabled) continue;
            FirstFeature += 8 * ForEachNonZeroByte(Modules[i].Regions[r].Start,
                                                   Modules[i].Regions[r].Stop,
                                                   FirstFeature, Handle8bitCounter);
          }
        }
```

The nested for loop calls into `ForEachNonZeroByte` with two pointers as the first two arguments. These arguments point to the areas in the code where a page of inline 8-bit counters exist, namely, each region in the `Modules` vector encapsulates a page of 8bit counters. The implementation of `ForEachNonZeroByte` is defined here https://github.com/llvm/llvm-project/blob/e60d6e91e196d91a1b9bfcc93d9f43946ea29299/compiler-rt/lib/fuzzer/FuzzerTracePC.h#L184 and essentially the function will call the callback (`Handle8bitCounter`) for each nonzero byte in the memory region. In the case of the `RunOne` function the result is to call into the function provided by the call to `TPC.CollectFeatures`, which eventually calls into various functions related to the corpus, e.g. `Corpus.AddFeature` and `Corpus.UpdateFeatureFrequency` which will trigger changes in the Corpus representation if the features indicate that a new piece of code has been executed.

The main point to get across in the above section is that nested for-loop in `CollectFeatures` iterates through the regions with inline 8-bit counters, and what we will observe in the Envoy fuzzers is that the number of inline 8-bit counters is significantly large.

1.1 is it possible to disable certain 8-bit counters?

A question that is relevant about how libFuzzer uses the instrumentation is whether you can tell libFuzzer to disable checking of inline counters. Indeed, in the `TPC.CollectFeatures` function the loop that is extensive (shown above), namely:

```
for (size_t i = 0; i < NumModules; i++) {
    for (size_t r = 0; r < Modules[i].NumRegions; r++) {
      if (!Modules[i].Regions[r].Enabled) continue;
      FirstFeature += 8 * ForEachNonZeroByte(Modules[i].Regions[r].Start,
                                             Modules[i].Regions[r].Stop,
                                             FirstFeature, Handle8bitCounter);
    }
  }
```

only iterates through the inline 8-bit counters if a given region is “`Enabled`”. Unfortunately, the `Enabled` boolean will always be true for all regions as all regions are initialised to true

(https://github.com/llvm/llvm-project/blob/ab5823867c4aee7f3e02ddfaa217905c87471bf9/compiler-rt/lib/fuzzer/FuzzerTracePC.cpp#L59) and never set to false.

1.2 How does coverage instrumentation impact fuzzer execution speed in Envoy?

In this section we show three different ways of observing the impact of the current instrumentation set up in Envoy.

1.2.1 Execution speed of an empty fuzzer.

We assess the slowdown of the instrumentation by creating the simplest fuzzer we can. We do this by having an empty fuzzer and compiling it in two different ways. First, we compile it without linking it to any Envoy code. Second, we compile it in the exact same way as `h2_capture_fuzz` namely by linking it to the exact same code in the Envoy code base, which corresponds to essentially all of the Envoy code. We define the new fuzzer as follows:

```
#include "test/integration/h2_fuzz.h"
DEFINE_FUZZER(const uint8_t* buf, size_t len) {
    if (len == 1234123412) {
        std::cout << "hello " << buf << "\n" ;
    }
}
```

and enable it by extending test/integration/BUILD with the following

```
envoy_cc_fuzz_test(
    name = "h2_empty_fuzz_test",
    srcs = ["h2_empty_fuzz_test.cc"],
    copts = ["-DPERSISTENT_FUZZER"],
    corpus = "h2_corpus",
    deps = [":h2_fuzz_persistent_lib"],
)
```

We then compile and run the fuzzer, and observe the following output:

```
$ h2_empty_fuzz_test
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2391912932
INFO: Loaded 1 modules   (1282530 inline 8-bit counters): 1282530 [0xa51f110,
0xa6582f2),
INFO: Loaded 1 PC tables (1282530 PCs): 1282530 [0xa6582f8,0xb9ea118),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 3 ft: 4 corp: 1/1b exec/s: 0 rss: 170Mb
```

```
#8192   pulse  cov: 3 ft: 4 corp: 1/1b lim: 80 exec/s: 4096 rss: 170Mb
#16384  pulse  cov: 3 ft: 4 corp: 1/1b lim: 163 exec/s: 4096 rss: 171Mb
#32768  pulse  cov: 3 ft: 4 corp: 1/1b lim: 325 exec/s: 4096 rss: 172Mb
#65536  pulse  cov: 3 ft: 4 corp: 1/1b lim: 652 exec/s: 3855 rss: 175Mb
```

If we create the same empty fuzzer but outside the envoy build environment but following the Envoy fuzzer set up https://github.com/envoyproxy/envoy/blob/1d1b708c7bf6efa02c41d9ce22cbf1e4a1aeec2c/test/fuzz/fuzz_runner.h#L71:

```
#include <iostream>

#define DEFINE_TEST_ONE_INPUT_IMPL                                          \
  extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {  \
    EnvoyTestOneInput(data, size);                                          \
    return 0;                                                               \
  }

#define DEFINE_FUZZER                                                       \
  static void EnvoyTestOneInput(const uint8_t* buf, size_t len);            \
  DEFINE_TEST_ONE_INPUT_IMPL                                                \
  static void EnvoyTestOneInput

DEFINE_FUZZER(const uint8_t* buf, size_t len) {
    if (len == 1234123412) {
        std::cout << "hello " << buf << "\n" ;
    }
}
```

and run the fuzzer, then we observe the output:

```
$ $CXX $CXXFLAGS $LIB_FUZZING_ENGINE ./empty_fuzzer.cc
$ ./a.out
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1703773772
INFO: Loaded 1 modules   (126 inline 8-bit counters): 126 [0x84c0f8, 0x84c176),
INFO: Loaded 1 PC tables (126 PCs): 126 [0x5e6398,0x5e6b78),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 3 ft: 4 corp: 1/1b exec/s: 0 rss: 29Mb
#4194304        pulse  cov: 3 ft: 4 corp: 1/1b lim: 4096 exec/s: 2097152 rss: 343Mb
#8388608        pulse  cov: 3 ft: 4 corp: 1/1b lim: 4096 exec/s: 1677721 rss: 611Mb
#16777216       pulse  cov: 3 ft: 4 corp: 1/1b lim: 4096 exec/s: 1677721 rss: 614Mb
```

We verify the disassembly of `LLVMFuzzerTestOneInput` is similar in both of the fuzzers (see Appendix A.0), specifically that they each execute the same amount of code in the `LLVMFuzzerTestOneInput`. We also note here that the coverage achieved by both fuzzers is equal (namely 3) as shown by the output of libFuzzer.

The execution speed and the number of inline 8-bit counters are very different in the two cases. Specifically, we extract two important conclusions from the above observations

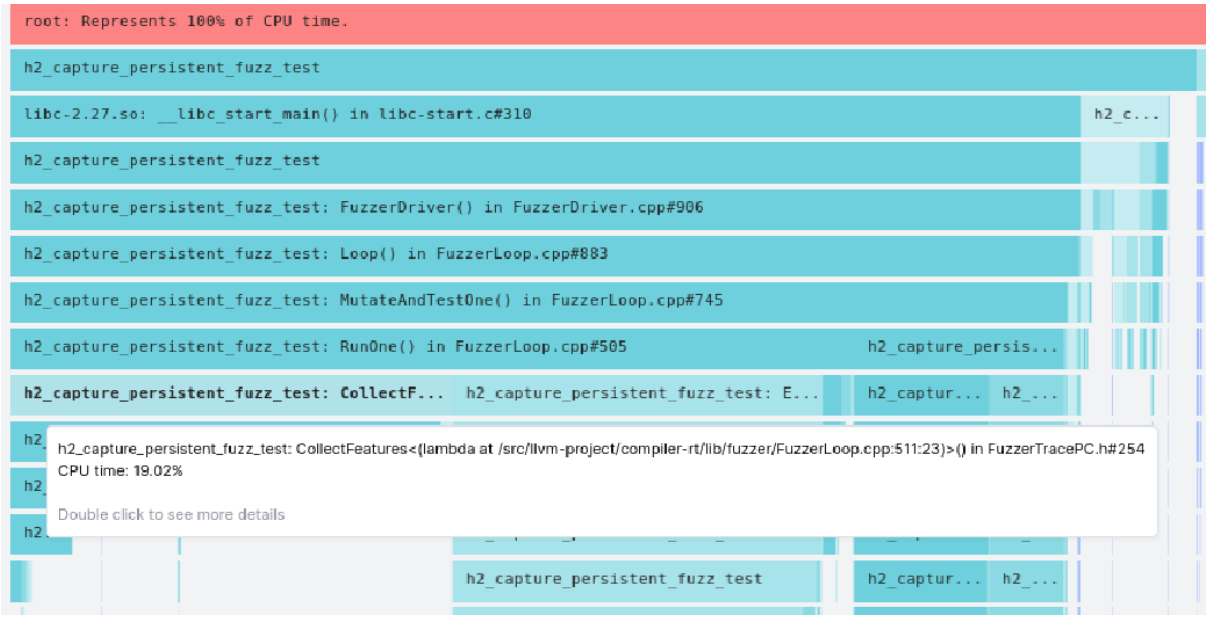
1. The fuzzer that is compiled outside of the Envoy fuzzer environment executes with roughly 1.67 million executions per second, whereas the one in the Envoy environment executes with roughly 4000 executions per second. The envoy instrumentation slows down the fuzzer by roughly 419x.
2. The number of inline 8-bit counters in the Envoy fuzzer is 1282530 whereas it is 126 in the empty fuzzer (these numbers can be observed by the output from LibFuzzer given above). This corresponds to iterating through 126 bytes versus 1282530 bytes after each fuzz iteration.

1.2.2 Using Prodfiler to observe the impact.

Another way we tried to assess the execution speed of the fuzzers was by using profiling tools to tell us where execution is spent. To do this, we relied on Prodifiler https://github.com/optimyze/prodfiler-documentation to profile the entire machine in which we ran the fuzzer. Prodfiler reported that 25.8% of all function samples observed happened

Rank	Function	Samples	Percentage
1	h2_capture_persistent_fuzz_test: ForEachNonZeroByte<(lam /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerTracePC.h	5857	25.8%
2	h2_capture_persistent_fuzz_test: AddFeature /src/llvm-project/compiler-rt/lib/fuzzer/FuzzerCorpus.h	1202	5.3%

inside the `ForEachNonZeroByte` function**.** In addition to this, looking at the flame graphs provided by Prodfiler we see that around 19% of the entire machine execution was spent inside of the `CollectFeatures` function.



1.2.3 Using fuzzer instrumentation to assess the overall time spent with fuzzing.

Another thing we were interested in understanding was how much of the fuzzing execution was actually spent in the target and how much was spent in the fuzzing engine itself. To do this, we added some simple instrumentation to the h2_capture_fuzz.cc and h2_fuzz.cc files, such that the h2_capture_fuzz fuzzer will log timestamps as follows:

```
DEFINE_PROTO_FUZZER(..H2CaptureFuzzTestCase input)
  ADD_TIME_STAMP(F1)
  H2FuzzIntegrationTest h2_var;
  h2_var.replay(input)
    --> H2FuzzIntegrationTest::replay(... H2CaptureFuzzTestCase& input,)
         ...
         ADD_TIME_STAMP(R4)
         IntegrationTcpClientPtr tcp_client = makeTcpConnection(lookupPort("http"));
         ...
         ADD_TIME_STAMP(R5)
         for (int i = 0; i < input.events().size(); ++i) {
                   if (stop_further_inputs) {
                      break;
                    }
              INC_GLOBAL(PACKET_COUNT)
              // send packet
         } // endloop
         ADD_TIME_STAMP(R6)
         ...
    <--
  ADD_TIME_STAMP(F2)
  <--
```

With these timestamps available we set up an experiment to measure the following metrics:

- Total time spent fuzzing
- Total time spent in code between the following timestamps:
  - F1-F2: This is the total time spent in the fuzzer code.
  - R4-R6: This is the total time spent in the replay function.
  - R5-R6: This is the total time spent in the loop sending fuzz inputs to the client
- Number of total fuzz inputs to the client
- Various proportionate metrics, e.g. inputs sent to the client.

We executed **h2_capture_persistent_fuzz** with the seeds provided, and observed the following results:

Metric	**Result**
Total execution time	1200 sec
F1-F2	244 sec
R4-R6	239 sec
R5-R6	226 sec
F1-F2/Total time	**20%**
R4-R6/Total time	19.9%
R5-R6/Total time	18.8%
Total number of client fuzz inputs	82797
Total fuzz inputs to client / (F1-F2)	339 inputs/sec
Total fuzz inputs to client / (R4-R6)	346 inputs/sec
Total fuzz inputs to client / (R5-R6)	366 inputs/sec
Total number of fuzz iterations	27761

We observe that only 20% of the fuzz time is spent in the actual fuzzing code, and thus 80% of the time is spent in the libFuzzer engine.

1.2.4 Counting the number of 8-bit inline counters in each Envoy fuzzer.

Finally, we wanted to understand how many 8-bit inline counters are actually used by the Envoy fuzzers. This number is reported by the fuzzers when they are initiated, so the only thing we have to do in this case is build the fuzzers, launch them and then count the number of inline 8-bit counters. We do this using a build with AddressSanitizer, and we see the following results:

Fuzzer name	**inline 8-bit counters**
h2_capture_persistent_fuzz_test	1281655
h2_capture_fuzz_test	1281649

h1_capture_fuzz_test	1268160
h1_capture_persistent_fuzz_test	1268166
codec_fuzz_test	1030067
hash_fuzz_test	570594
get_sha_256_digest_fuzz_test	579995
evaluator_fuzz_test	1100639
dns_filter_fuzz_test	1139928

For comparison purposes the following table gives examples of 8-bit counters in various open source projects that we have fuzzed.

Project fuzzer targets	**Inline 8-bit counters**
Lua	6322
PugiXML	4871
LevelDB	5948
RocksDB	137913

It is clear that the number of inline 8-bit counters in Envoy is significantly larger than any of the other projects. For example, our fuzzer targeted the core library (liblua.a) for the Lua programming language has a total of 6322 inline 8-bit counters whereas the h2 end-to-end Envoy fuzzer has a total of 1.2 million.

1.2.5 Conclusions

The instrumentation set up in Envoy is a significant performance bottleneck for the fuzzers. The main reason for the performance is the post-processing that occurs after each fuzz execution in terms of collecting coverage features to assess whether the execution discovered new code execution. An empty fuzzer without Envoy instrumentation runs 419x faster than an empty fuzzer with Envoy instrumentation.

We want to emphasize here that the fuzzing infrastructure of Envoy is of high quality and the team behind it has shown high competencies in fuzzing Envoy. In fact, from a certain perspective it is positive that Envoy builds all code related to Envoy with sanitizers and this should be considered a great achievement as well as having the right intentions. It is due to the sheer amount of code in the resulting Envoy binaries, which is due to linking of many third-party libraries in addition to the large Envoy code base in and of itself, that the coverage collection ends up having a significant slowdown. In the vast majority of projects we would not advise limiting the coverage instrumentation across a fuzzing set up.

Furthermore, the observations we made in this section are not common knowledge and it is not to be expected from non-fuzzing experts to consider this during fuzzer development.

1.3 Improving Envoy fuzzer execution by reducing instrumentation

1.3.1 Disabling instrumentation

The crucial task for improving fuzzing performance in Envoy is to reduce the amount of coverage instrumentation in the target application. Specifically, the goal is to reduce the amount of inline 8-bit counters. The main way to this is by including the compilation flags:

```
-fsanitize-coverage=0
-fno-sanitize=all
```

The first flag `-fsanitize-coverage=0` disablescoverage instrumentation, whereas the second flag also disables bug-finding sanitizers, e.g. AddressSanitizer. The first flag should be removed from code that can avoid being explored or are potentially fuzzed elsewhere. The second flag, namely disabling bug-finding sanitizers, should be disabled with more care.

In order to disable instrumentation, we tried three things:

Bazel builds: First, we tried controlling instrumentation details inside of the Envoy BUILD scripts. In particular, adding `copts` definitions to various places in the BUILD scripts of ENVOY. This works for the majority of bazel rules, however, it does not work for any rule defined by way of `proto_library`. This is because `proto_library` does not allow us to propagate compilation options to the protobuf compiler, and thus it does not allow us to include the flags to disable instrumentation on a per-target basis. We reference to issues in the bazelbuild/rules_proto repository here:
https://github.com/bazelbuild/rules_proto/issues/85
https://github.com/bazelbuild/rules_proto/issues/41

Passing flags to clang: Second, we tried using the `--per_file_copt` command line option for bazel builds. This option accepts a regular expression that encapsulates a set of files used during compilation as well as a string that will be passed to the compiler during compilation of files that match the regular expression. For example, the following line:

```
--per_file_copt=^.*\.pb\.cc$@-fsanitize-coverage=0,-fno-sanitize=all
```

will pass the flags `-fsanitizer-coverage=0` and `-fno-sanitize=all` to all files compiled in the bazel build with `.pb.cc` extensions. This option worked well.

Using a blocklist for SanitizerCoverage: Third, using the `-fsanitize-coverage-blocklist=/src/block_list.txt` also by way of `per_file_copt`. This option disables coverage instrumentation based on contents of the file `/src/block_list.txt` in which it is possible to disable instrumentation based on file names or (mangled) function names.

1.3.2 Results from disabling instrumentation

Measuring different configurations

The next was to identify which parts of the code yields best results when disabling coverage instrumentation. To do this, we set up a set of different build configurations following the approaches (2 and 3) described above and measured the following data:

1. The number of 8-bit inline counters in the h1 e2e and h2 2e2 fuzzers;
2. The total number of fuzz iterations achieved over a 5 minute run with the seeds from the OSS-Fuzz repository.

In total, we compiled the Envoy fuzzers with 6 different configurations and the configurations are shown in Appendix A.1. The following table shows the results.

Fuzzer	**Setting**	**inline 8-bit counters**	**Total execs (300 sec)**	**Exec/sec**
http2 e2e	Configuration 1 (regular build)	1281922	4527	15.1
http1 e2e	Configuration 1 (regular build)	1268433	7123	23.7
http2 e2e	Configuration 2	271239	7372	24.6
http1 e2e	Configuration 2	272107	25370	84.6
http2 e2e	Configuration 3	313473	6864	22.9
http1 e2e	Configuration 3	314363	17640	58.9
http2 e2e	Configuration 4	313545	5291	17.6
http1 e2e	Configuration 4	314438	16530	55.1
http2 e2e	Configuration 5	258955	6522	21.7
http1 e2e	Configuration 5	259216	20639	68.8
http2 e2e	Configuration 6 **(includes blocked list of functions)**	273673	8055	26.85
http1 e2e	Configuration 6	274564	19932	66.4

	(includes blocked list of functions)			

Finally, we note that the executions per second increases if the fuzzer is run for a longer period of time. For example, in our experiments we have observed above 100 exec/sec in the http2 end-to-end fuzzer with seeds in longer fuzz runs. The data of these runs have been shared with the Envoy team internally.

Deploying on OSS-Fuzz.

We integrated some of the above efforts into the OSS-Fuzz build script of Envoy and the following figure shows impact.

date	perf_report	tests_executed	new_crashes	edge_coverage	cov_report	corpus_size	avg_exec_per_sec
Apr 25, 2021	Performance	32,056,830	0	1.89% (11267/596450)	Coverage	7366 (50 MB)	37.4
Apr 24, 2021	Performance	27,511,207	0	1.89% (11267/596450)	Coverage	7118 (41 MB)	35.8
Apr 23, 2021	Performance	34,922,637	0	1.89% (11267/596453)	Coverage	5189 (39 MB)	40.6
Apr 22, 2021	Performance	26,497,485	0	1.89% (11267/595441)	Coverage	4510 (41 MB)	32.9
Apr 21, 2021	Performance	22,649,429	0	1.89% (11267/595441)	Coverage	7202 (52 MB)	32.5
Apr 20, 2021	Performance	27,129,347	0	1.89% (11267/595299)	Coverage	7702 (49 MB)	36.4
Apr 19, 2021	Performance	23,931,695	0	1.89% (11267/595167)	Coverage	--	35.1
Apr 18, 2021	Performance	26,858,142	0	1.89% (11267/595167)	Coverage	7092 (50 MB)	36.3
Apr 17, 2021	Performance	30,912,970	0	1.89% (11267/595167)	Coverage	6885 (32 MB)	41.1
Apr 16, 2021	Performance	17,195,865	0	1.89% (11262/594796)	Coverage	6122 (41 MB)	29.1
Apr 15, 2021	Performance	4,571,168	0	8.38% (49838/594783)	Coverage	5883 (35 MB)	12.8
Apr 14, 2021	Performance	3,088,806	1	8.44% (50225/594784)	Coverage	7120 (50 MB)	12.9
Apr 13, 2021	Performance	1,999,456	0	8.39% (49927/594781)	Coverage	7197 (55 MB)	9.9

We can see here that a performance improvement happened on the 16th April 2021. Inspecting the logs, we notice on the 15th April the amount of inline 8-bit counters was ~600,000 (https://console.cloud.google.com/storage/browser/_details/envoy-logs.clusterfuzz-external.appspot.com/libFuzzer_envoy_h2_capture_persistent_fuzz_test/libfuzzer_asan_envoy/2021-04-15/00:06:53:330737.log) whereas on the 16th April it was ~300,000 (https://console.cloud.google.com/storage/browser/_details/envoy-logs.clusterfuzz-external.appspot.com/libFuzzer_envoy_h2_capture_persistent_fuzz_test/libfuzzer_asan_envoy/2021-04-16/01:54:36:943426.log). This decrease in inline 8-bit counters resulted in a 3x improvement of the fuzzer execution. We note here that in addition to the decrease in inline 8-bit counters the stability of the fuzzer was also improved on the same day through bug fixing, which may have impacted the performance as well.

2 Fuzzing the Envoy UdpListener

The second task of the Engagement was to create a new fuzzer that targets the UDP listener of Envoy. The goal of this fuzzer was to enable catching the bug described in this Github issue https://github.com/envoyproxy/envoy/issues/14113. The fuzzer we created was merged into Envoy and is available at this location in the repository: https://github.com/envoyproxy/envoy/blob/main/test/common/network/udp_fuzz.cc

The goal of the fuzzer is to send an arbitrary number of arbitrary packets to the Envoy UDP listener. To do this we rely on the primitives provided by `Network::Test::UdpSyncPeer` and the core of the fuzzer is encapsulated by the following loop (https://github.com/envoyproxy/envoy/blob/58a13570f3f4dea9bad8b8fa5e1221d7ed5056de/test/common/network/udp_fuzz.cc#L92):

```
    // Now do all of the fuzzing
    static const int MaxPackets = 15;
    total_packets_ = provider.ConsumeIntegralInRange<uint16_t>(1, MaxPackets);
    Network::Test::UdpSyncPeer client_(ip_version_);
    for (uint16_t i = 0; i < total_packets_; i++) {
      std::string packet_ =
          provider.ConsumeBytesAsString(provider.ConsumeIntegralInRange<uint32_t>(1,
3000));
      if (packet_.empty()) {
        packet_ = "EMPTY_PACKET";
      }
      client_.write(packet_, *send_to_addr_);
    }
    dispatcher_->run(Event::Dispatcher::RunType::Block);
```

The fuzzer supports communicating by way of `GRO`, `MMSG` as well as regular `recvmsg`, which is controlled by the following code (https://github.com/envoyproxy/envoy/blob/58a13570f3f4dea9bad8b8fa5e1221d7ed5056de/test/common/network/udp_fuzz.cc#L73):

```
    FuzzedDataProvider provider(buf, len);
    uint16_t SocketType = provider.ConsumeIntegralInRange<uint16_t>(0, 2);
    if (SocketType == 0) {
      config.mutable_prefer_gro()->set_value(true);
      ON_CALL(override_syscall_, supportsUdpGro()).WillByDefault(Return(true));
    } else if (SocketType == 1) {
      ON_CALL(override_syscall_, supportsMmsg()).WillByDefault(Return(true));
    } else {
      ON_CALL(override_syscall_, supportsMmsg()).WillByDefault(Return(false));
      ON_CALL(override_syscall_, supportsUdpGro()).WillByDefault(Return(false));
    }
```

In order to verify the fuzzer finds the original crash of the code we ran the fuzzer

against the Envoy codebase with the changes applied by the commit that fixed the UDP bug (https://github.com/envoyproxy/envoy/pull/14122/files). More specifically, if we avoid the use of the condition

```
      if (output.msg_[i].truncated_and_dropped_) {
        continue;
      }
```

in this commit https://github.com/envoyproxy/envoy/pull/14122/files#diff-c9702469f313c70572e261f4af73687873ab8247264eb47fd05300098067c5abR632 then we get the following result:

```
$ /out/udp_fuzz ./seeds/
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 185475315
INFO: Loaded 1 modules   (1087563 inline 8-bit counters): 1087563 [0x861fcb0,
0x87294fb),
INFO: Loaded 1 PC tables (1087563 PCs): 1087563 [0x8729500,0x97c19b0),
INFO:       55 files found in ./seeds/
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: seed corpus: files: 55 min: 1b max: 2251b total: 4519b rss: 150Mb
AddressSanitizer:DEADLYSIGNAL
=================================================================
==28367==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc
0x000005872691 bp 0x7fff415ade70 sp 0x7fff415ad980 T0)
==28367==The signal is caused by a READ memory access.
==28367==Hint: address points to the zero page.
    #0 0x5872691 in Envoy::Network::passPayloadToProcessor(unsigned long,
std::__1::unique_ptr<Envoy::Buffer::Instance,
std::__1::default_delete<Envoy::Buffer::Instance> >,
std::__1::shared_ptr<Envoy::Network::Address::Instance const>,
std::__1::shared_ptr<Envoy::Network::Address::Instance const>,
Envoy::Network::UdpPacketProcessor&,
std::__1::chrono::time_point<std::__1::chrono::steady_clock,
std::__1::chrono::duration<long long, std::__1::ratio<1l, 1000000000l> > >)
/proc/self/cwd/source/common/network/utility.cc:562:3
    #1 0x587501e in Envoy::Network::Utility::readFromSocket(Envoy::Network::IoHandle&,
Envoy::Network::Address::Instance const&, Envoy::Network::UdpPacketProcessor&,
std::__1::chrono::time_point<std::__1::chrono::steady_clock,
std::__1::chrono::duration<long long, std::__1::ratio<1l, 1000000000l> > >, bool,
unsigned int*) /proc/self/cwd/source/common/network/utility.cc:669:7
    #2 0x5877d3a in
Envoy::Network::Utility::readPacketsFromSocket(Envoy::Network::IoHandle&,
Envoy::Network::Address::Instance const&, Envoy::Network::UdpPacketProcessor&,
Envoy::TimeSource&, bool, unsigned int&)
/proc/self/cwd/source/common/network/utility.cc:702:38
    #3 0x552b14d in Envoy::Network::UdpListenerImpl::handleReadCallback()
/proc/self/cwd/source/common/network/udp_listener_impl.cc:75:34
    #4 0x552a15f in Envoy::Network::UdpListenerImpl::onSocketEvent(short)
/proc/self/cwd/source/common/network/udp_listener_impl.cc:64:5
```

```
    #5 0x5531e26 in
Envoy::Network::UdpListenerImpl::UdpListenerImpl(Envoy::Event::DispatcherImpl&,
std::__1::shared_ptr<Envoy::Network::Socket>, Envoy::Network::UdpListenerCallbacks&,
Envoy::TimeSource&, envoy::config::core::v3::UdpSocketConfig
const&)::$_0::operator()(unsigned int) const
/proc/self/cwd/source/common/network/udp_listener_impl.cc:38:53
....
....
```

To confirm the fuzzer targets the code that was asked for by the Envoy team we observe coverage of the fuzzer in OSS-Fuzz. Consider the following links showing the UDP Listener code is being analysed:

- `Envoy::Network::UdpListenerImpl::handleReadCallback` https://storage.googleapis.com/oss-fuzz-coverage/envoy/reports/20210423/linux/proc/self/cwd/source/common/network/udp_listener_impl.cc.html#L72
- `Envoy::Network::Utility::readFromSocket` https://storage.googleapis.com/oss-fuzz-coverage/envoy/reports/20210423/linux/proc/self/cwd/source/common/network/utility.cc.html#L576

In addition to this, consider the figures in Appendix A.2 showing the coverage is being achieved.

3 Future advice

We consider there to be three problems that need solving for the Envoy team.

First, enable the ability to disable instrumentation of protobuf code included in header files, which in themselves can have a huge amount of code. This is because a lot of protobuf code is included by way of header files in the Envoy source code:

```
envoy/source/common$ grep -rn "pb.h" ./ | wc -l
388
```

The unfortunate consequence of this is that the code in the header file could avoid being instrumented but because it is included by important code that has to be instrumented, the protobuf code consequently will also be instrumented. Some refactoring that makes it easy to control the instrumentation parameters of the protobuf code would likely have a large impact on the execution speed.

An option for solving the protobuf problems above is to disable instrumentation on a namespace or function level. We can do this by performing partial instrumentation with SanitizerCoverage as described here: https://clang.llvm.org/docs/SanitizerCoverage.html#partially-disabling-instrumentation

Specifically, consider the file *block_list.txt* with the following contents:

```
fun:*nocoveragepls*
```

This will disable coverage instrumentation for any function that contains `nocoveragepls` in its function name, which includes the namespace name. As such, a way to disable instrumentation in a smart manner is to include a file *blocked_list.txt* in the Envoy repository, which we will then use during Envoy fuzzing compilation as follows:
`--per_file_copt="-fsanitize-coverage-blocklist=envoy_code/block_list.txt"`
We did an initial experimentation of this by way of configuration #6 discussed in section 1.3.2.

Second, add the ability to do per-target instrumentation. It is clear that the fuzz targets in the Envoy code target separate parts of the code base. Ideally instrumentation should be made on a target-specific basis. The goal should be to have less than 100,000 inline 8-bit counters per fuzzer. Notice that the number of counters is shown in the libFuzzer logs (as shown above section 1.2.1) which are also available on OSS-Fuzz for each of the OSS-Fuzz runs.

We leave the following instrumentation-specific advice:

- It can be beneficial to reduce coverage instrumentation also from a perspective of letting the fuzzer focus on code that matters, since the fuzzer will be guided by improvements in relevant code and not get “lost” in irrelevant code.
- Large chunks of sequential code does not need to be instrumented with coverage, including Envoy critical code. We advise to keep bug-finding sanitizers on nonetheless.

Third, during the engagement we observed that several of the fuzzers had a high crashing percentage (above 90%) for several months. In fact, the http2 end-to-end fuzzer had an input in its corpus that caused the fuzzer to crash. Effectively, the fuzzer had not explored new code for months because this input blocked the fuzzer from continuing. We think it is crucial for the Envoy team to ensure the fuzzers of Envoy are running properly and this should be considered higher priority than ensuring the fuzzers run fast.

4 Conclusions

In this engagement we improved Envoy fuzzing by identifying the performance bottleneck in the Envoy fuzzing set up and creating a new UDP fuzzer that targets previously buggy code. Our findings identify that the performance bottleneck in Envoy is due to large amounts of code instrumentation which causes LibFuzzer to spend significant effort in determining if each fuzz iteration caused new code to execute. We show how this affects the Envoy fuzzers, for example by highlighting an empty fuzzer runs 419 times slower with Envoy’s instrumentation approach. We show how to limit the amount of instrumentation in Envoy and how it improves performance.

5 Appendix

A.0 Disassembly of empty fuzzers

Appendix showing the disassembly of *LLVMFuzzerTestOneInput* in the two empty fuzzers is similar.
Disassembly of **h2_empty_fuzz_test**, namely the empty fuzzer compiled with Envoy instrumentation.

```
$ gdb /out/h2_empty_fuzz_test
Reading symbols from /out/h2_empty_fuzz_test...done.
(gdb) set disassembly-flavor intel
(gdb) disass LLVMFuzzerTestOneInput
Dump of assembler code for function LLVMFuzzerTestOneInput:
   0x0000000003235e90 <+0>:     push   rbp
   0x0000000003235e91 <+1>:     mov    rbp,rsp
   0x0000000003235e94 <+4>:     add    BYTE PTR [rip+0x72d9dfc],0x1        # 0xa50fc97
   0x0000000003235e9b <+11>:    mov    rax,0xfffffffffffedc8
   0x0000000003235ea2 <+18>:    cmp    rbp,QWORD PTR fs:[rax]
   0x0000000003235ea6 <+22>:    jae    0x3235eac <LLVMFuzzerTestOneInput+28>
   0x0000000003235ea8 <+24>:    mov    QWORD PTR fs:[rax],rbp
   0x0000000003235eac <+28>:    call   0x3235ec0 <_ZL17EnvoyTestOneInputPKhm>
   0x0000000003235eb1 <+33>:    xor    eax,eax
   0x0000000003235eb3 <+35>:    pop    rbp
   0x0000000003235eb4 <+36>:    ret
End of assembler dump.
(gdb) disass _ZL17EnvoyTestOneInputPKhm
Dump of assembler code for function _ZL17EnvoyTestOneInputPKhm:
   0x0000000003235ec0 <+0>:     push   rbp
   0x0000000003235ec1 <+1>:     mov    rbp,rsp
   0x0000000003235ec4 <+4>:     push   r14
   0x0000000003235ec6 <+6>:     push   rbx
   0x0000000003235ec7 <+7>:     add    BYTE PTR [rip+0x72d9dca],0x1        # 0xa50fc98
   0x0000000003235ece <+14>:    mov    rbx,rsi
   0x0000000003235ed1 <+17>:    mov    r14,rdi
   0x0000000003235ed4 <+20>:    mov    rax,0xfffffffffffedc8
   0x0000000003235edb <+27>:    cmp    rbp,QWORD PTR fs:[rax]
   0x0000000003235edf <+31>:    jae    0x3235ee5 <_ZL17EnvoyTestOneInputPKhm+37>
   0x0000000003235ee1 <+33>:    mov    QWORD PTR fs:[rax],rbp
   0x0000000003235ee5 <+37>:    mov    edi,0x498f3a94
   0x0000000003235eea <+42>:    mov    rsi,rbx
   0x0000000003235eed <+45>:    call   0x3153860 <__sanitizer_cov_trace_const_cmp8()>
   0x0000000003235ef2 <+50>:    cmp    rbx,0x498f3a94
   0x0000000003235ef9 <+57>:    jne    0x3235f31 <_ZL17EnvoyTestOneInputPKhm+113>
   0x0000000003235efb <+59>:    add    BYTE PTR [rip+0x72d9d98],0x1        # 0xa50fc9a
   0x0000000003235f02 <+66>:    lea    rdi,[rip+0x908047f]        # 0xc2b6388 <_ZNSt3__14coutE>
   0x0000000003235f09 <+73>:    lea    rsi,[rip+0xfffffffffd3ba150]        # 0x5f0060 <.str.4>
   0x0000000003235f10 <+80>:    call   0x3235f40
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc>
   0x0000000003235f15 <+85>:    mov    rdi,rax
   0x0000000003235f18 <+88>:    mov    rsi,r14
   0x0000000003235f1b <+91>:    call   0x3235f80
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKh>
   0x0000000003235f20 <+96>:    lea    rsi,[rip+0xfffffffffd3ba179]        # 0x5f00a0 <.str.5>
   0x0000000003235f27 <+103>:   mov    rdi,rax
   0x0000000003235f2a <+106>:   call   0x3235f40
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc>
```

```
   0x0000000003235f2f <+111>:   jmp    0x3235f38 <_ZL17EnvoyTestOneInputPKhm+120>
   0x0000000003235f31 <+113>:   add    BYTE PTR [rip+0x72d9d61],0x1        # 0xa50fc99
   0x0000000003235f38 <+120>:   pop    rbx
   0x0000000003235f39 <+121>:   pop    r14
   0x0000000003235f3b <+123>:   pop    rbp
   0x0000000003235f3c <+124>:   ret
End of assembler dump.
(gdb)
```

Disassembly of empty fuzz test compiled without instrumentation:

```
# gdb -q ./a.out
Reading symbols from ./a.out...done.
(gdb) set disassembly-flavor intel
(gdb) disass LLVMFuzzerTestOneInput
Dump of assembler code for function LLVMFuzzerTestOneInput:
   0x0000000000557d40 <+0>:     push   rbp
   0x0000000000557d41 <+1>:     mov    rbp,rsp
   0x0000000000557d44 <+4>:     add    BYTE PTR [rip+0x2f43ad],0x1        # 0x84c0f8
   0x0000000000557d4b <+11>:    mov    rax,QWORD PTR [rip+0x2f1246]        # 0x848f98
   0x0000000000557d52 <+18>:    cmp    rbp,QWORD PTR fs:[rax]
   0x0000000000557d56 <+22>:    jae    0x557d5c <LLVMFuzzerTestOneInput+28>
   0x0000000000557d58 <+24>:    mov    QWORD PTR fs:[rax],rbp
   0x0000000000557d5c <+28>:    call   0x557d70 <_ZL17EnvoyTestOneInputPKhm>
   0x0000000000557d61 <+33>:    xor    eax,eax
   0x0000000000557d63 <+35>:    pop    rbp
   0x0000000000557d64 <+36>:    ret
End of assembler dump.
(gdb) disass _ZL17EnvoyTestOneInputPKhm
Dump of assembler code for function _ZL17EnvoyTestOneInputPKhm:
   0x0000000000557d70 <+0>:     push   rbp
   0x0000000000557d71 <+1>:     mov    rbp,rsp
   0x0000000000557d74 <+4>:     push   r14
   0x0000000000557d76 <+6>:     push   rbx
   0x0000000000557d77 <+7>:     add    BYTE PTR [rip+0x2f437b],0x1        # 0x84c0f9
   0x0000000000557d7e <+14>:    mov    rbx,rsi
   0x0000000000557d81 <+17>:    mov    r14,rdi
   0x0000000000557d84 <+20>:    mov    rax,QWORD PTR [rip+0x2f120d]        # 0x848f98
   0x0000000000557d8b <+27>:    cmp    rbp,QWORD PTR fs:[rax]
   0x0000000000557d8f <+31>:    jae    0x557d95 <_ZL17EnvoyTestOneInputPKhm+37>
   0x0000000000557d91 <+33>:    mov    QWORD PTR fs:[rax],rbp
   0x0000000000557d95 <+37>:    mov    edi,0x498f3a94
   0x0000000000557d9a <+42>:    mov    rsi,rbx
   0x0000000000557d9d <+45>:    call   0x473370 <__sanitizer_cov_trace_const_cmp8()>
   0x0000000000557da2 <+50>:    cmp    rbx,0x498f3a94
   0x0000000000557da9 <+57>:    jne    0x557ddb <_ZL17EnvoyTestOneInputPKhm+107>
   0x0000000000557dab <+59>:    add    BYTE PTR [rip+0x2f4349],0x1        # 0x84c0fb
   0x0000000000557db2 <+66>:    mov    edi,0x11a77e0
   0x0000000000557db7 <+71>:    mov    esi,0x5e03a0
   0x0000000000557dbc <+76>:    call   0x557df0
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc>
   0x0000000000557dc1 <+81>:    mov    rdi,rax
   0x0000000000557dc4 <+84>:    mov    rsi,r14
   0x0000000000557dc7 <+87>:    call   0x557e30
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKh>
   0x0000000000557dcc <+92>:    mov    esi,0x5e03e0
   0x0000000000557dd1 <+97>:    mov    rdi,rax
   0x0000000000557dd4 <+100>:   call   0x557df0
```

```
<_ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc>
   0x0000000000557dd9 <+105>:   jmp    0x557de2 <_ZL17EnvoyTestOneInputPKhm+114>
   0x0000000000557ddb <+107>:   add    BYTE PTR [rip+0x2f4318],0x1        # 0x84c0fa
   0x0000000000557de2 <+114>:   pop    rbx
   0x0000000000557de3 <+115>:   pop    r14
   0x0000000000557de5 <+117>:   pop    rbp
   0x0000000000557de6 <+118>:   ret
End of assembler dump.
```

A.1 Build configurations

Configuration 1. Regular build
This is simply a regular build.

Configuration 2.
This configuration disables:

- Coverage and bug-finding sanitizers in many of the external libraries
- Coverage instrumentation of various folders under source/common and source/extensions
- Coverage instrumentation of code in test/ directory.
- Coverage instrumentation of all .pb.cc files and all (.cc) files in the bazel-out directory

```
declare -r DI="$(
if [ "$SANITIZER" != "coverage" ]
then
# Envoy code. Disable coverage instrumentation
  echo " --per_file_copt=^.*source/extensions/access_loggers/.*\.cc\$@-fsanitize-coverage=0"
  echo " --per_file_copt=^.*source/extensions/filters/.*\.cc\$@-fsanitize-coverage=0"
  echo " --per_file_copt=^.*source/extensions/.*\.cc\$@-fsanitize-coverage=0"
  echo " --per_file_copt=^.*source/common/protobuf/.*\.cc\$@-fsanitize-coverage=0"
  echo " --per_file_copt=^.*source/common/network.*\.cc\$@-fsanitize-coverage=0"
  echo " --per_file_copt=^.*source/common/runtime.*\.cc\$@-fsanitize-coverage=0"

# Envoy test code. Disable coverage instrumentation
  echo " --per_file_copt=^.*test/.*\.cc\$@-fsanitize-coverage=0"

# Disable external libraries.
  echo " --per_file_copt=^.*antlr4_runtimes.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_alibaba_hessian2_codec.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*io_opencensus_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_protobuf.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_absl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*googletest.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_code_re2.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*upb.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_jbeder_yaml_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*proxy_wasm_cpp_host/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_google_libprotobuf_mutator/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
```

```
  echo " --per_file_copt=^.*com_googlesource_googleurl/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_datadog_dd_opentracing_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"

# All protobuf code and code in bazel-out
  echo " --per_file_copt=^.*\.pb\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*bazel-out/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
fi
)"
```

Configuration 3.

This configuration disables everything that configuration 2 disables **except**:

- Code under source/common and source/extensions.

```
declare -r DI="$(
if [ "$SANITIZER" != "coverage" ]
then
# Envoy code. Disable coverage instrumentation

# Envoy test code. Disable coverage instrumentation
  echo " --per_file_copt=^.*test/.*\.cc\$@-fsanitize-coverage=0"

# Disable celcpp and grpc correctly.
  echo " --per_file_copt=^.*antlr4_runtimes.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_alibaba_hessian2_codec.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*io_opencensus_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_protobuf.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_absl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*googletest.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_code_re2.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*upb.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_jbeder_yaml_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*proxy_wasm_cpp_host/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_google_libprotobuf_mutator/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_googleurl/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_datadog_dd_opentracing_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_envoyproxy_sqlparser.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"

# All protobuf code and code in bazel-out
  echo " --per_file_copt=^.*\.pb\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*bazel-out/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
fi
)"
```

Configuration 4

This configuration disables everything that configuration 3 disables **and also**:

- Disabling bug-finding instrumentation in files under the test/ directory.

```
declare -r DI="$(
if [ "$SANITIZER" != "coverage" ]
then
# Envoy code. Disable coverage instrumentation

# Envoy test code. Disable coverage instrumentation
  echo " --per_file_copt=^.*test/.*\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"

# Disable celcpp and grpc correctly.
  echo " --per_file_copt=^.*antlr4_runtimes.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_alibaba_hessian2_codec.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*io_opencensus_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_protobuf.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_absl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*googletest.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_code_re2.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*upb.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_jbeder_yaml_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*proxy_wasm_cpp_host/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_google_libprotobuf_mutator/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_googleurl/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_datadog_dd_opentracing_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_envoyproxy_sqlparser.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"

# All protobuf code and code in bazel-out
  echo " --per_file_copt=^.*\.pb\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*bazel-out/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
fi
)"
```

Configuration 5

This configuration disables everything that configuration 4 disables **and also**:

- Disabling coverage instrumentation of all .cc files in source/server folder.

```
declare -r DI="$(
if [ "$SANITIZER" != "coverage" ]
then
# Envoy code. Disable coverage instrumentation
  echo " --per_file_copt=^.*source/server.*\.cc\$@-fsanitize-coverage=0"
# Envoy test code. Disable coverage instrumentation
  echo " --per_file_copt=^.*test/.*\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"

# Disable celcpp and grpc correctly.
  echo " --per_file_copt=^.*antlr4_runtimes.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_alibaba_hessian2_codec.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*io_opencensus_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_protobuf.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_absl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*googletest.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
```

```
  echo " --per_file_copt=^.*com_googlesource_code_re2.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*upb.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_jbeder_yaml_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*proxy_wasm_cpp_host/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_google_libprotobuf_mutator/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_googleurl/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_datadog_dd_opentracing_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_envoyproxy_sqlparser.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"

# All protobuf code and code in bazel-out
  echo " --per_file_copt=^.*\.pb\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*bazel-out/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
fi
)"
```

Configuration 6

This configuration is similar to configuration #4, however, it also includes the use of a file with a function name regex on the path /src/blog_list.txt. This file is used by SanitizerCoverage to disable coverage instrumentation in functions that match the regular expression, the contents of this file is:

```
fun:*envoy*
```

The important part of the build.sh script is as follows:

```
declare -r DI="$(
if [ "$SANITIZER" != "coverage" ]
then
# Envoy code. Disable coverage instrumentation

# Envoy test code. Disable coverage instrumentation
  echo " --per_file_copt=^.*test/.*\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"

# Disable celcpp and grpc correctly.
  echo " --per_file_copt=^.*antlr4_runtimes.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_alibaba_hessian2_codec.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*io_opencensus_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_protobuf.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_absl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*googletest.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_grpc_grpc.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*boringssl.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_googlesource_code_re2.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*upb.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*org_brotli.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_google_cel_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_jbeder_yaml_cpp.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*proxy_wasm_cpp_host/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_google_libprotobuf_mutator/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
```

```
  echo " --per_file_copt=^.*com_googlesource_googleurl/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_lightstep_tracer_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_datadog_dd_opentracing_cpp/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*com_github_envoyproxy_sqlparser.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*grpc_httpjson_transcoding.*\$@-fsanitize-coverage=0,-fno-sanitize=all"

# All protobuf code and code in bazel-out
  echo " --per_file_copt=^.*\.pb\.cc\$@-fsanitize-coverage=0,-fno-sanitize=all"
  echo " --per_file_copt=^.*bazel-out/.*\$@-fsanitize-coverage=0,-fno-sanitize=all"
fi
)"


# Benchmark about 3 GB per CPU (10 threads for 28.8 GB RAM)
# TODO(asraa): Remove deprecation warnings when Envoy and deps moves to C++17
bazel build -s --verbose_failures --dynamic_mode=off ${DI} \
  --per_file_copt="^.*\$"@"-Wno-error=unused-command-line-argument" \
  --per_file_copt="^.*\$"@"-fsanitize-coverage-blocklist=/src/block_list.txt" \
```

A.2 Envoy UDP fuzzer coverage

`Envoy::Network::UdpListenerImpl::handleReadCallback` : https://storage.googleapis.com/oss-fuzz-coverage/envoy/reports/20210423/linux/proc/self/cwd/source/common/network/udp_listener_impl.cc.html#L72

```
72   136 void UdpListenerImpl::handleReadCallback() {
73   136   ENVOY_UDP_LOG(trace, "handleReadCallback");
74   136   cb_.onReadReady();
75   136   const Api::IoErrorPtr result = Utility::readPacketsFromSocket(
76   136       socket_->ioHandle(), *socket_->addressProvider().localAddress(), *this, time_source_,
77   136       config_.prefer_gro_, packets_dropped_);
78         // TODO(mattklein123): Handle no error when we limit the number of packets read.
79   136   if (result->getErrorCode() != Api::IoError::IoErrorCode::Again) {
80           // TODO(mattklein123): When rate limited logging is implemented log this at error level
81           // on a periodic basis.
82     0     ENVOY_UDP_LOG(debug, "recvmsg result {}: {}", static_cast<int>(result->getErrorCode()),
83     0                   result->getErrorDetails());
84     0     cb_.onReceiveError(result->getErrorCode());
85     0   }
86   136 }
87
88         void UdpListenerImpl::processPacket(Address::InstanceConstSharedPtr local_address,
89                                             Address::InstanceConstSharedPtr peer_address,
90   796                                       Buffer::InstancePtr buffer, MonotonicTime receive_time) {
91         // UDP listeners are always configured with the socket option that allows pulling the local
92         // address. This should never be null.
93   796   ASSERT(local_address != nullptr);
94   796   UdpRecvData recvData{
95   796       {std::move(local_address), std::move(peer_address)}, std::move(buffer), receive_time};
96   796   cb_.onData(std::move(recvData));
97   796 }
98
99   136 void UdpListenerImpl::handleWriteCallback() {
100  136   ENVOY_UDP_LOG(trace, "handleWriteCallback");
101  136   cb_.onWriteReady(*socket_);
102  136 }
```

`Envoy::Network::Utility::readFromSocket`https://storage.googleapis.com/oss-fuzz-coverage/envoy/reports/20210423/linux/proc/self/cwd/source/common/network/utility.cc.html#L576

```
576          Api::IoCallUint64Result Utility::readFromSocket(IoHandle& handle,
577                                                          const Address::Instance& local_address,
578                                                          UdpPacketProcessor& udp_packet_processor,
579                                                          MonotonicTime receive_time, bool prefer_gro,
580      786                                                 uint32_t* packets_dropped) {
581
582      786   if (prefer_gro && handle.supportsUdpGro()) {
583      178     Buffer::InstancePtr buffer = std::make_unique<Buffer::OwnedImpl>();
584      178     IoHandle::RecvMsgOutput output(1, packets_dropped);
585
586              // TODO(yugant): Avoid allocating 24k for each read by getting memory from UdpPacketProcessor
587      178     const uint64_t max_rx_datagram_size_with_gro =
588      178         NUM_DATAGRAMS_PER_GRO_RECEIVE * udp_packet_processor.maxDatagramSize();
589      178     ENVOY_LOG_MISC(trace, "starting gro recvmsg with max={}", max_rx_datagram_size_with_gro);
590
591      178     Api::IoCallUint64Result result =
592      178         receiveMessage(max_rx_datagram_size_with_gro, buffer, output, handle, local_address);
593
594      178     if (!result.ok() || output.msg_[0].truncated_and_dropped_) {
595       24       return result;
596       24     }
597
598      154     const uint64_t gso_size = output.msg_[0].gso_size_;
599      154     ENVOY_LOG_MISC(trace, "gro recvmsg bytes {} with gso_size as {}", result.rc_, gso_size);
600
601              // Skip gso segmentation and proceed as a single payload.
602      154     if (gso_size == 0u) {
603      154       passPayloadToProcessor(result.rc_, std::move(buffer), std::move(output.msg_[0].peer_address_),
604      154                              std::move(output.msg_[0].local_address_), udp_packet_processor,
605      154                              receive_time);
606      154       return result;
607      154     }
608
609              // Segment the buffer read by the recvmsg syscall into gso_sized sub buffers.
610              // TODO(mattklein123): The following code should be optimized to avoid buffer copies, either by
611              // switching to slices or by using a CoW buffer type.
612        0     while (buffer->length() > 0) {
613        0       const uint64_t bytes_to_copy = std::min(buffer->length(), gso_size);
614        0       Buffer::InstancePtr sub_buffer = std::make_unique<Buffer::OwnedImpl>();
615        0       sub_buffer->move(*buffer, bytes_to_copy);
616        0       passPayloadToProcessor(bytes_to_copy, std::move(sub_buffer), output.msg_[0].peer_address_,
617        0                              output.msg_[0].local_address_, udp_packet_processor, receive_time);
618        0     }
619
620        0     return result;
621        0   }
622
623      608   if (handle.supportsMmsg()) {
624      126     const auto max_rx_datagram_size = udp_packet_processor.maxDatagramSize();
625
626              // Buffer::ReservationSingleSlice is always passed by value, and can only be constructed
627              // by Buffer::Instance::reserve(), so this is needed to keep a fixed array
628              // in which all elements are legally constructed.
629      126     struct BufferAndReservation {
630      126       BufferAndReservation(uint64_t max_rx_datagram_size)
631      126           : buffer_(std::make_unique<Buffer::OwnedImpl>()),
632    2.01k           reservation_(buffer_->reserveSingleSlice(max_rx_datagram_size, true)) {}
633
634      126       Buffer::InstancePtr buffer_;
635      126       Buffer::ReservationSingleSlice reservation_;
636      126     };
637      126     constexpr uint32_t num_slices_per_packet = 1u;
638      126     absl::InlinedVector<BufferAndReservation, NUM_DATAGRAMS_PER_MMSG_RECEIVE> buffers;
639      126     RawSliceArrays slices(NUM_DATAGRAMS_PER_MMSG_RECEIVE,
640      126                           absl::FixedArray<Buffer::RawSlice>(num_slices_per_packet));
641    2.14k     for (uint32_t i = 0; i < NUM_DATAGRAMS_PER_MMSG_RECEIVE; i++) {
642    2.01k       buffers.push_back(max_rx_datagram_size);
643    2.01k       slices[i][0] = buffers[i].reservation_.slice();
644    2.01k     }
645
646      126     IoHandle::RecvMsgOutput output(NUM_DATAGRAMS_PER_MMSG_RECEIVE, packets_dropped);
```