Chapter 41
Coverage Guided Fuzz Testing with AFL++

Revision 1.0, February 2026.

Objectives

- Learn about the history and background of AFL++
- Understand what AFL++ does and its functionality
- Understand how AFL++ works including structure and mechanisms
- See an example of AFL++'s use

41.1 Introduction

In Chapter 38, we introduced the idea of a coverage guided fuzz tester, i.e., a fuzz testing tool that was mutational and gray box. While many coverage-guided fuzzing tools exist, AFL++ is particularly popular and effective, making it a useful one to study. Understanding how AFL++ works will also help you understand other similar tools you may use.

AFL++ is an open source project based on the original AFL tool written by Michal Zalewsky. AFL was written while Zalesky was at Google and first released in November 2013. Since then, there have been many contributors to it and it has spawned many variations. In particular, AFL++ branched off from AFL in 2017, leading up to the latest release of version 4.35c. While AFL++ has made some significant advances over original AFL[1], the material in this chapter applies equally well to AFL and AFL++.

AFL stands for "American Fuzzy Lop". While you might be familiar with the AFL fuzz testing tool, you might not realize that AFL is not the original American Fuzzy Lop, so do not confuse the fuzz testing tool with the enormously cute breed of rabbit.

As we saw in Chapter 38, there are three important categories of variation on the design of a fuzz tester. AFL++, as it is most commonly used, is mutation-based, gray box, and unstructured.

To review these categories, this means that first, AFL++ starts with a set of programmer-provided inputs – seeds – and then mutates them to produce the test inputs. Second, it is gray box, in that it uses information about what parts

[1] Andrea Fioraldi, Dominik Maier, Heiko Eissfeldt and Marc Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research", *14th USENIX Workshop on Offensive Technologies (WOOT20),* online, August 2020.

of the program have executed to control the selection and transformation of the test inputs. Third, it is unstructured in that it has no knowledge of what comprises properly formatted input. The combination of these categories is often called *coverage guided testing.*

From a high level, we can talk about AFL++'s inputs and outputs. The basic input, of course, is the program to be tested. In addition, you have to provide one or more initial inputs to the program (called "seeds") that will be used to generate other inputs.

The outputs are a list of crashes and hangs, along with the input that caused the crash or hang.

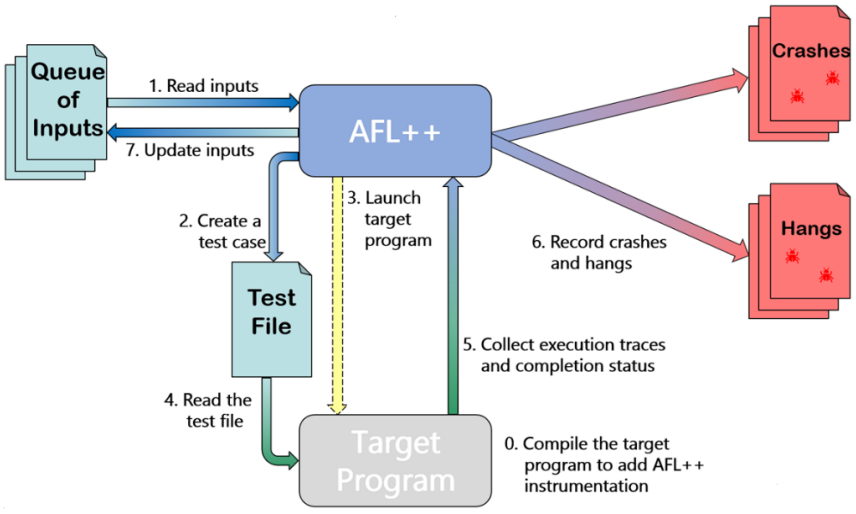

Figure 1: Workflow of the AFL Fuzz Tester

41.2 Steps in the Execution of AFL++

We start with an overview of steps needed in the operation of AFL++, as illustrated in Figure 1.

0. The first step, before you even run AFL++, is to prepare the application program to be run with AFL++ by compiling the program with the `afl-gcc` compiler. This special compiler inserts instrumentation into the program's code to gather basic execution trace data. The details of this step are discussed below in Section 41.3.
1. When we run AFL++, the first thing that it does is to read the input seeds. From these initial seeds, AFL++ forms the queue of inputs.
2. Next, it will mutate one of the inputs in the queue to create the next test input.

3. AFL++ then starts the application program executing, providing the mutated input file from the previous step as input.
4. When the program completes, either normally, with a crash, or interrupted after a time-out (which indicates a hang), AFL++ collects the execution trace and records the completion status.
5. If the completion status was a crash or hang, and this was a new crash or hang, AFL++ records the input that caused it.
6. It then uses information about completion status to update the list set of input seeds.

Now, we discuss steps 2, 5, 6 and 7 in more detail.

41.2.1 Step 2: Create A Test Case

To create a new input, AFL++ first has to select one of the inputs in the input queue. These seeds are kept in a priority order that is roughly based on

input seed size × *how long the application program ran on that input*

A general note about AFL++ and the many other coverage guided fuzzing tools: There are many complicated algorithms used to select the next seed, generate mutations, and clean up the set of inputs. In this chapter, we will be getting only a general idea of how they work as there are just too many tricky details and most of them are not critical to your use of the tool.

AFL takes the selected seed input and generates new inputs based on mutating it. Each input is used in a new execution of the application program.

The mutations used to modify an existing input occur in three stages. The first stage applies simple changes to the input, such a setting a bit or byte to a certain value. These changes are applied to each bit and byte of the input, generating many new inputs (therefore many program executions).

The second stage of mutation is random. Random mutations apply the same type of mutations used in the deterministic stage; however they apply a random number of them to an input in multiple random locations.

The third stage is called the “havoc” stage, where there are random editing changes to one or more inputs to produce a new input.

Simple Mutations

There are a few simple mutations that are used in this stage. The first type of simple mutation is the bit flip. This mutation flips (inverts) from 1 to 4 bits in the selected input. For example, if we were at the 5th bit in the example input and ready to flip 4 bits, we would see the resulting highlighted bits shown in Figure *2* in red. In this example, `0011` flipped to `1100`.

Figure 2: Example of a Bit Flip Mutation: Randomly Inverted 4 Bits

Figure 3: Example of a Byte Flip Mutation: Randomly Inverted a Byte

Figure 4: Example of a Math Mutation: Randomly Added 1 to a Byte

Figure 5: Example of a Set Mutation: Randomly Set a Byte to -1

The second type of mutation is the byte flip. These are similar to bit flips but flip 1 to 4 bytes in the input. In Figure *3*, we see the flipped second byte in our example shown in red.

The third type is a collection of math operations, which means adding or subtracting values between 1 and 35 to 1, 2 or 4 bytes values in the input. In the example in Figure *4*, we added 1 to the first byte of the input generating the change shown in red.

The last type is the set operations that set 1, 2, or 4 bytes to specific interesting values as determined by the AFL designers. Figure 5 shows setting a byte to -1.

Here is a recent list of interesting values for the 1, 2, and 4 byte cases:

1 byte : -128, -1, 0, 1, 16, 32, 64, 100, 127

2 byte: -32,767, -129, 128, 255, 256, 512, 1,000, 1,024, 4,096, 32,767

4 byte: -2,147,483,648, -100,663,046, -32,769, 32,768, 65,535, 65,536, 100,663,045, 2,147,483,647

Random Mutations

The random mutations use the same basic operations that we just presented for the simple mutations, but apply them a random number of times in multiple locations of a given input.

Havoc Mutations

The havoc mutations use random editing operations. These editing operations include duplicating bytes at a random location in the input,

remove bytes from a random location, and randomly selecting another input to combine with the current one.

Note that the mutation types and interesting values are ones thought to be interesting by the AFL++ developers. Interesting means that the designers thought that such operations or values are somehow likely to trigger bugs in the application program. The topic of what are appropriate and effect mutations is a huge area of active research in the fuzz testing community. Some of the choices made with current tools are based on careful study and experiment and other choices are based on ad hoc decisions made by the developers.

41.2.2 Step 5: Generating an Execution Trace

As the test program runs, the instrumentation inserted into the program during the compile stage (step 0) generates an execution trace about what parts of the program were executed. This trace information is what makes AFL++ a gray box tester.

AFL++ decides that an input is interesting if it causes the program being tested to execute its code differently. Differently, in this context, means that a different collection of basic blocks were executed. This execution information can be tracked with an *execution trace.*

An execution trace is the sequence of basic blocks in the program that executed. We will start with a description of what are basic blocks and then discuss how AFL++ tracks which basic blocks in a program have executed.

A *basic block* is a sequence of code that always enters at the beginning (no branches into the middle) and ends at a branch or start of another basic block.

For example, in Figure 6 we see a simple C function on the left and the control flow graph (CFG, as we described in Chapter 36) on the left. Each block shown in the CFG corresponds to one basic block.

If we labeled each block in a program with a letter, and recorded that letter each time the block executed, we would see a trace something like this:

$$A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

As with any execution trace, its length increases as the program runs, which leads to no bound on the size of the trace. To avoid the space issues associated with a full execution trace, AFL++ uses a fixed size data structure, called a *basic block transition table,* that approximates the information in a basic block trace.

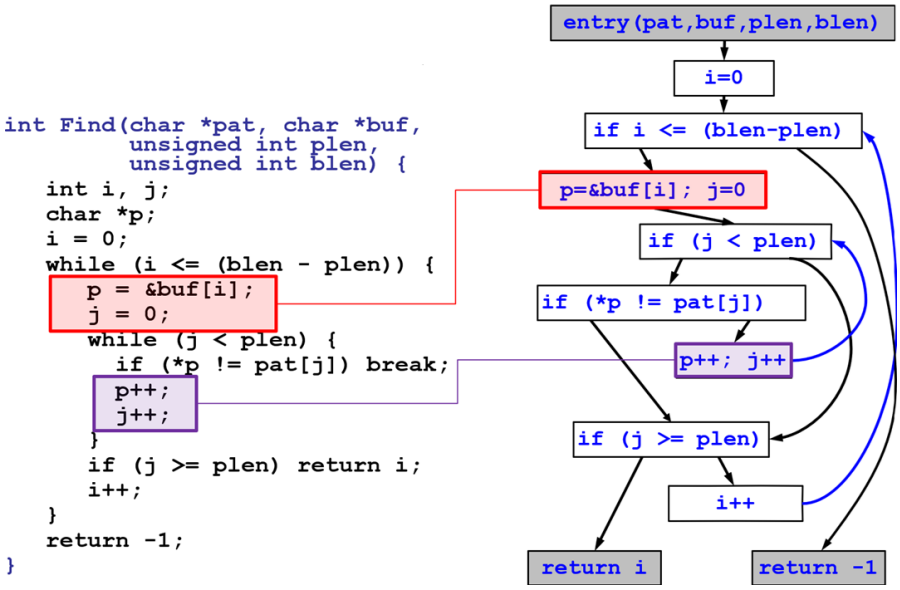

Figure 6: A Simple C Function and Resulting Basic Blocks

The transition table is a list of tuples that contain the basic block transitions, how many times they occurred, and the test input that generated them. So, in the basic block trace above, we see that there are three occurrences of the transition from block B to C, resulting in a tuple in the transition table with the contents (B → C, 3). The table for the entire sample trace would be:

(A→B, 3) (B→C, 3) (C→A, 2) (C→D, 1) (D→E, 1)

Note that the size of the transition table is proportional to the size of the program (so a fixed value) rather than proportional to how long it ran. No matter how long the program runs, the transition table is basically the same size for all runs.

Two executions of the program that generate the same transitions are considered to be equivalent.

41.2.3 Step 6: Recording Crashes and Hangs

After the program completes, AFL will update the test results. If the program crashed or hung, then there is a potential new result to record. AFL will record the result if this result has not been seen before. “Seen before” means that this run of the application program did notcause a new set of transitions to occur.

41.2.4 Step 7: Updating the Seeds

Once AFL completes a test run of the program, it potentially will update the queue of inputs. After each execution of the program, AFL++ will do a bit of housekeeping on the collection of inputs. This housekeeping involves two kinds of updates, additions and removals.

AFL will add the currently used input to the queue if it caused a crash or hang and generated a basic block transition table that was different from any previously seen transition table.

AFL will also try to clean up the inputs to keep the set a reasonable size. This clean-up step is called "culling". The goal is to remove the inputs that are less likely to produce useful test results. Results that generate high coverage of the program's basic blocks are, in general, preferred over those that generate less coverage. The actual algorithm for culling is intricate and complicated. Strategies for culling remain an active area of research within the fuzz-testing community.

41.3 How to Use AFL++

We now go through the steps that you go through to compile and run a program with AFL++. First, you have to compile the program specially for AFL++, then run the program under AFL++'s control. When AFL++ is running, it displays a dashboard of its progress. We will discuss how to understand the results.

41.3.1 A Sample Program

We will start with a simple test C program called "report". Under normal circumstances (i.e., without AFL++) you might compile it as you see here.

```
gcc -g report.c spread_subrs.c subrs.c search.c countchar.c
-o report
```

To run the program from the command line (click), you simply type its name and a list of input files for it to process.

```
report log1
```

where `log1` is an input file to the `report` program.

You then interact with the report program with commands typed to standard input.

41.3.2 Compiling a Program to Run with AFL++

When you want to test this `report` program with AFL++, you first have to compile it with a special AFL++ compiler. As we mentioned previously, this

compiler will automatically insert instrumentation into the code to collect basic block transition information as the program runs.

The command line looks just like a regular `gcc` command with a different compiler name.

```
afl-gcc -g report.c spread_subrs.c subrs.c search.c
countchar.c -o report-afl
```

Note that we also changed the name of the executable file to `report-afl`, just so that we know that this version of the program is instrumented for AFL++. You would probably have a separate AFL++ build rule in your makefile to do this.

The next step is to actually run AFL++ with the `report` program.

To run AFL++, you use the `afl-fuzz` command. On the command line, we include the application program, `report-afl`, along with its command line parameters.

```
afl-fuzz -i stdin_seeds -o stdin_out report-afl log1
```

The afl-fuzz command also includes a few more parameters to specify, including:

- `-i` specifies the directory containing the input seeds.
- `-o` specifies the directory where AFL++ writes the test results and where it stores the input queue.

The results directory stores some important information including the inputs that caused crashes and the inputs that caused hang.

41.3.3 Run AFL++

When AFL++ starts to run, it displays the dashboard shown in Figure 7, which contains quite a lot of information. Some of this information is pretty obvious and some is quite arcane, based on understanding the internal workings of AFL++. Note that you do not need to understand every item to use AFL affectively!

AFL++ will update the dashboard as the tests run to let you know how it is progressing.

Note that AFL++ will just keep running until you stop it. It is not very subtle or sophisticated, but basically you run AFL++ until you do not want to want any longer, then hit control-C to stop it.

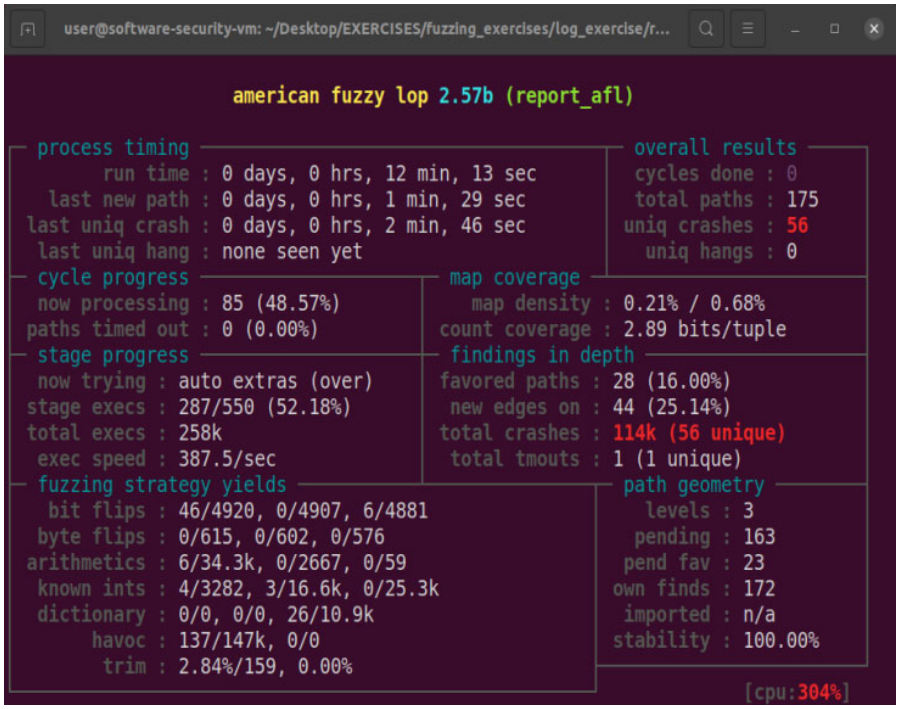

Figure 7: The AFL++Dashboard (Running the `report` Program)

Deciding how long to let AFL++ run is a bit tricky. For a simple program, you might get good results in a few minutes or an hour. For more complex programs, it could take quite a while. This process is slow because the program executes somewhat slower due to instrumentation. More importantly, it is slow because, AFL++ is going to run the program many times, each time with a new mutation of the input.

The AFL++ documentation advises us that for more complex programs, it might take days or weeks – or even more – to get satisfactory results from AFL++. However, remember that this is a fully automated tool, so you can just start AFL running and get on other aspects of your life.

Now, we will examine the dashboard and discuss some of the more important fields. The good news is that the most interesting information is highlighted in red.

The first thing to look at is `overall results` section, where we can see the number of unique crashes and hangs. These numbers show you how much productive progress is being made. We can also see the total paths explored in the tests.

In the `findings in depth` section, we can also see the total number of crashes found. This number is typically bigger than the unique crashes, because many similar inputs can trigger the same failure.

Under the `process timing` section, we can see how long AFL++ has been running and when it found the last crash or hang.

Back in the `overall results` pane, we can see the number of cycles completed. A cycle is one iteration through the input queue. AFL++ tries to give you some feedback about how much progress you are making on each iteration.

The cycle count is shown in magenta (as it is in Figure *7*) on the first pass, then yellow when each subsequent pass finds a new crash or hang, then blue when none are being found, and then green when it is a "longer while" since any were found. It is best to let AFL++ complete at least one full cycle.

The dashboard also provides a measure of your code coverage, relative to the number of transitions that were instrumented. This is labeled "map density" in the `map coverage` section of the dashboard. The term "map" is used because AFL++ uses a bitmap to track which transitions occurred in an execution On the left side, we see the percentage coverage in the current execution and on the right, we see the total coverage so far from all inputs.

41.3.4 How to Use the Results from AFL++

Once AFL has found crashes or hangs, what do you do next? In general, your goal is to now find the cause of those failures, essentially debugging the failure.

You start by looking in the directory that your specified with the `-o` option. In this directory, you will see a few subdirectories, including one that contains files for the inputs that caused the crashes and one that contains inputs that caused hangs.

If you re-run the application program with any of these input files, you should be able to preproduce the failure. With these input files, there are some immediate obvious next things to do.

The most basic thing you can do is use a conventional debugger, like `gdb`, to work through the program's execution until you find the cause of the failure. If the application program is your code, then this is just another bug to find. If the application was not written by you, this becomes a more onerous task as you have to familiarize yourself with a large piece of unknown code.

There are also other tools that you can use to help you find the bug. If the crash was caused by a memory error, such as a buffer overrun, then a memory checking tool, such as we described in Chapter 43 can be really helpful.

41.4 Summary

In this chapter, we have learned about the popular and useful AFL++ coverage guided fuzz tester, including a bit of its history. In the process of learning about AFL++, we learned how it tracks which parts of the program were executed, how to compile a program to run with AFL++, and how to understand the results presented by AFL++.

41.5 Exercises

1. Take a program that you have written and run it under AFL++. It is best to start with a small simple program to get some initial experience. If that goes well, move on to a more substantial and complicated program.
2. Using the program(s) that you tested for Question 1, try different input seed sets. Your goal is to experiment with the number and choice of input seeds to see how that affect the execution efficiency of AFL++. Does this choice affect the speed at which AFL++ finds crashes?