ically test a target program and for this purpose, instrumentation is required for better performance of the fuzzer.

2.3.1 Stages of fuzzing with AFL

We can start the fuzz testing by running:

```
afl-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
```

Listing 2.1: Execute AFL

Based on 2.1, AFL has the following phases:

1. **Initialization**

2.

3. **Target identification and setup:** The target application for AFL is any executable program. The only limitation of AFL for the target is that the target executable must take a list of at least one input filename as the program's arguments.

   Hence, it is not a necessity for the program to take only one argument in command line, instead, a driver for AFL can add different techniques of getting input for the program.

   Next action before starting the fuzz testing, is instrumentation. If the insturmentation of the target program is not done yet, AFL doesn't have any technique for fuzzing except dumb fuzzing, which mutates the inputs completely randomly, without considering any information from the program's executions. We will discuss about the instrumentation more in this chapter.

4. **Identify and setup inputs:** The seeds for AFL must be placed in a directory, and AFL puts them in it's queue for fuzzing later. Based on what our fuzzing goals are, preparing a corpora of different seeds with different code coverages,

enhances the exploration of AFL to discover more execution paths. AFL can pick a subset of the provided seeds using `afl-cmin`. This tool "tries to find the smallest subset of files in the input directory that still trigger the full range of instrumentation data points seen in the starting corpus". Instrumentation is mandatory for using `afl-cmin`. AFL can also minimize test inputs using `afl-tmin`. "The tool works with crashing and non-crashing test inputs alike. In the crash mode, it will happily accept instrumented and non-instrumented binaries. In the non-crashing mode, the minimizer relies on standard AFL instrumentation to make the file simpler without altering the execution path." [26]

Another option introduced by AFL is using dictionaries for investigating the possible grammars used in input parsing. As it is described in AFL's documentation, "by default, afl-fuzz mutation engine is optimized for compact data formats - say, images, multimedia, compressed data, regular expression syntax, or shell scripts. It is somewhat less suited for languages with particularly verbose and redundant verbiage - notably including HTML, SQL, or JavaScript.

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

After providing required or useful corpora of inputs, AFL can start it's fuzz testing.

5. **Generate fuzzed data:** As discussed before, AFL is a **mutation-based** fuzzer, meaning that it generates new inputs by mutating the (**favored**) inputs.

    - **Favored inputs**: After an input is executed, it's favor factor is calculated and later, if it is still interesting for AFL, it is marked as Favored. AFL

finds favorable path (collected using a **bitmap** after instrumentation) for the purpose of "having a minimal set of paths that trigger all the bits seen in the bitmap so far, and focus on fuzzing them at the expense of the rest." [26]

$$favored\_factor = e.exec\_time \times e.length \tag{2.1}$$

The preference of AFL for the favored inputs, increases the performance of AFL in finding more crashes, but it is against the effort of finding an input with higher time consumption.

- **Mutation strategies**: In order to generate new inputs, AFL takes a queue of inputs and tries mutating and running each one of them. The mutation strategies are in a queue of different strategies that are run on an input to generate more inputs eventually. These strategies include bit-fliping, byte-fliping, simple arithmatics, known integers, stack tweaks, and test case splicing. [27]

6. **Execute fuzzed data:** To simplify, AFL takes an instrumented program, a directory for seeds and a directory for outputs:

```
afl-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
```

Listing 2.2: Execute AFL

Before running the program using the seeds provided, AFL first looks for favored inputs, in the queue of seeds, then picks the favored one and runs the instrumented program using the input. To monitor the execution of the program, AFL creates a shared memory with the program, and during the execution of the program, this shared memory is filled with information from the instrumentation. The only shared information that AFL keeps is a bitmap of the program's basic blocks, that were visited in the execution. After the

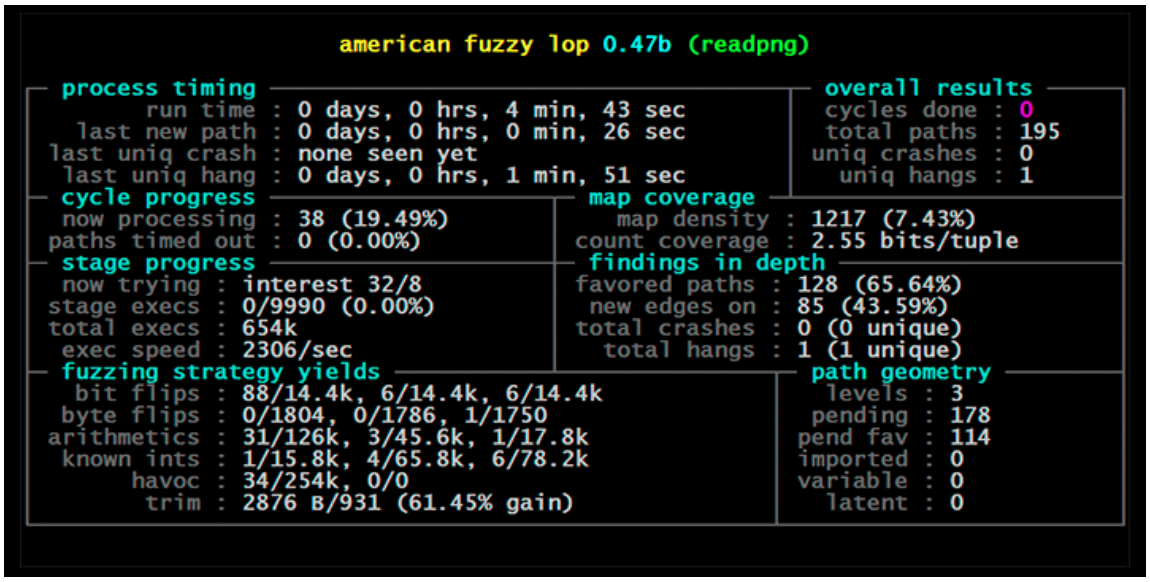

Figure 2.2: AFL status screen

execution is finished, AFL processes the bitmap and decides on what the next stage would be. If the program is crashed or hanged, it is kept for exploitability purposes and if the error is unique, it is stored in the output directory. This loop is then repeated for other favored inputs.

7. **Monitor for exceptions:** Beside reporting the crashes and hangs, AFL also logs the information about it's fuzzing stages and stats. In general, AFL provides a *status screen* which has eight different sections:

   (a) **Process timing**: This section tells about how long the process of fuzzing is running.

   (b) **Overall results**: A simplified information about the progress of AFL in finding paths, hangs and crashes.

   (c) **Cycle progress**: As mentioned before, AFL takes one input and repeats mutating it for a while. This section shows the information about the current cycle that the fuzzer is working on.

(d) **Map coverage**: "The section provides some trivia about the coverage observed by the instrumentation embedded in the target binary. The first line in the box tells you how many branch we have already hit, in proportion to how much the bitmap can hold. The number on the left describes the current input; the one on the right is the value for the entire input corpus. The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every branch, the needle will start to move toward "8.00" (every bit in the 8-bit map hit), but will probably never reach that extreme.

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

(e) **Stage progress**: The information about the current mutation stage is briefly provided here.

(f) **Findings in depth**: The crashes and hangs and any other findings (here we just have the other information about the coverage) are presented in this section.

(g) **Fuzzing strategy yields**: To illusterate more stats about the strategies used since the beginning of fuzzing, and for comparison of those strategies, AFL keeps track of how many paths were explored, in proportion to the number of executions attempted, for each of the fuzzing strategies.

(h) **Path geometry**: The information about the inputs and their depths, which says how many generations of different paths were produced in the process. For instance, we call the seeds we provided for fuzzing, the "level 1" inputs. Next, a new set of inputs is generated as the "level 2", the inputs derived from "level 2" are "level 3" and so on.

8. **Exploitability:** Figuring out whether the crashes and hangs are exploitable or not, is an important stage for exposing and exploiting the vulnerabilities. AFL provides another automated tool for checking the inputs responsible for the faults and mutating those inputs in order to find a more problematic input. To use this tool, we have to provide **-C** option for afl-fuzz:

```
afl-fuzz -C -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
```

Listing 2.3: AFL Crash Triage

If we define the input directory as the directory of the crashes, then AFL starts it's crash exploration, by looking for other inputs with different paths, but the same state.

Another technique for assessing the exploitability manually, is to use a debugger and investigate the causes of the crashes/hangs.

2.4 LLVM

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. The LLVM project has multiple components. The core of the project is itself called "LLVM". This contains all of the tools, libraries, and header files needed to process intermediate representations and converts it into object files. Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer." [28, 29] LLVM takes an **intermediate representation (IR)** of a program, and translates it to machine language.

"**clang** is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking."[30] It parses the source code, using the language-specific syntax, and converts it into a language-agnostic IR. We can use this feature for our instrumentation as it is followed.

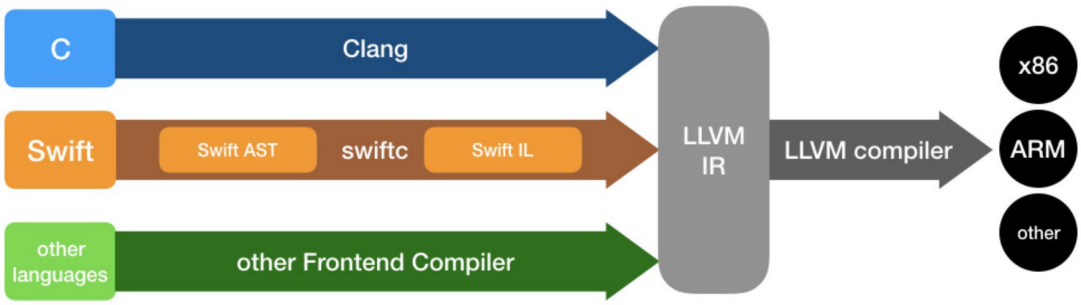

Figure 2.3: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]

2.4.1 Instrumentation and coverage measurements

The goal of using instrumentation for AFL, is to differentiate code coverages. AFL takes the source code and an instrumentation recipe, and generates the instrumented binary of the target program.

The recipe for instrumentation, fills out the coverage bitmap with the hash values of the path executed. The instructions are as followed:

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

AFL instruments by adding these instructions into basic blocks. First, a random value is assigned to *curr_location*. Next, it is XORed with the previous location's value, *prev_location*, and the resulting value is the location on *shared_mem*, the *coverage bitmap*, which is incremented by one. The third and final instruction is reseting the *prev_location* to a new value.

When AFL runs the instrumented program, everytime an instrumented basic block is executed, a location of *shared_mem* in bitmap is increased. The reason for this algorithm is for differentiating different paths that go over the same basic blocks. For instance, in figure 2.4, suppose that we have an instrumented program, with the random values set in compile time. A simple execution would be walking over basic

blocks $1 \rightarrow 2 \rightarrow 5$ will increase these locations by 1: $shared\_mem[2362223]$ for the transition of $1 \rightarrow 2$ and $shared\_mem[368221416]$ for $2 \rightarrow 5$. We can see that the paths $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5$ (which contains a loop), set different values on bitmap.

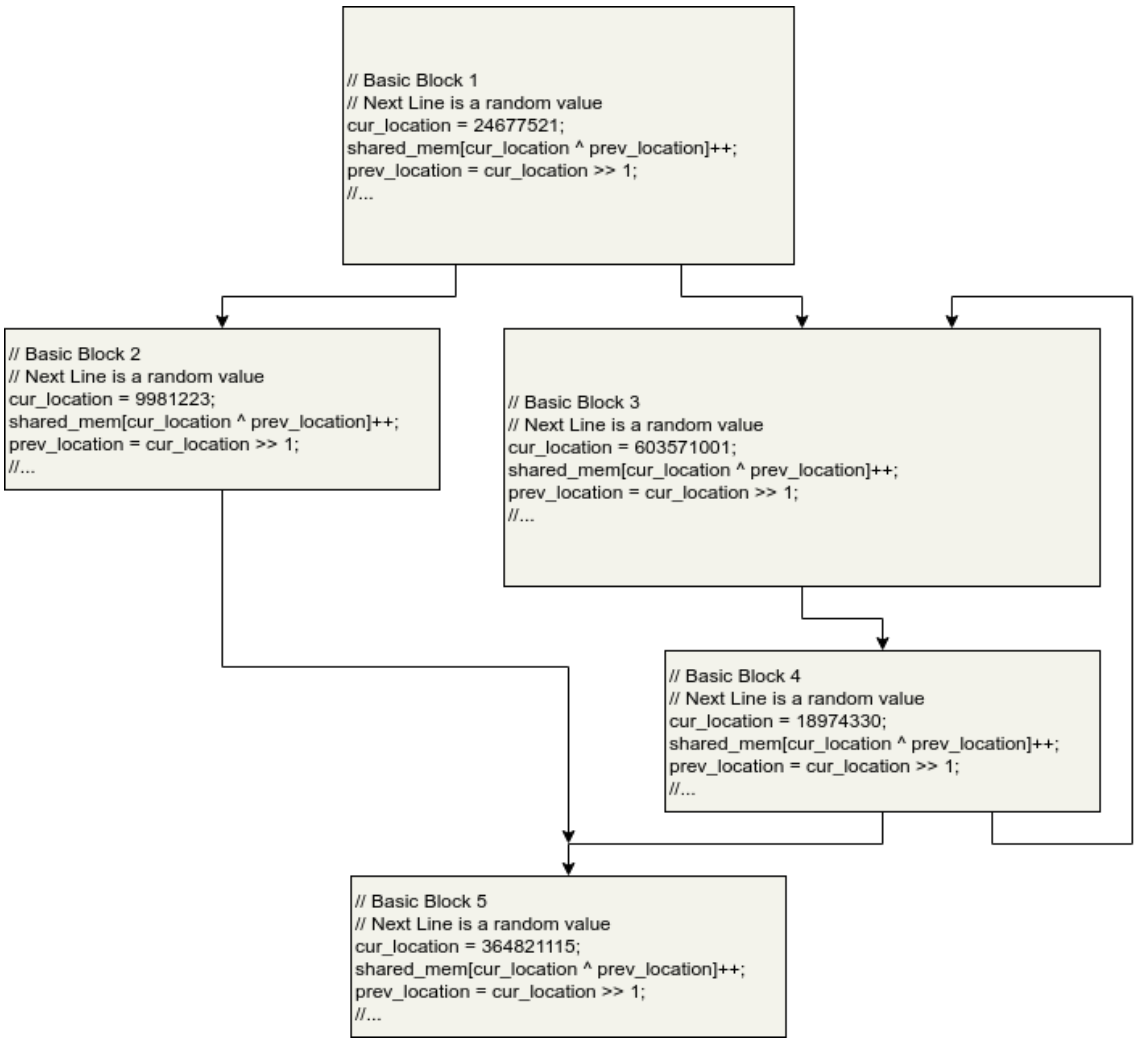

Figure 2.4: Example for instrumented basic blocks

AFL has the recipe for inserting the above instructions to the basic blocks using LLVM. It first compiles the program using this recipe for clang and after the execution of the program, we have a coverage bitmap of the locations we have visited during the execution. As a result, AFL uses this coverage feature for announcing

new inputs with new coverages.

2.5 Concluding remarks

In this chapter we described some basics of our work, that inspired us and lets us implement the current version of our thesis, Waffle. The topics covered in this chapter are:

- A brief description of the previous inspiring works for our fuzzer, as well as the stages of a fuzz testing procedure.
- We split different types of fuzzers into three categories, whitebox, blackbox, and greybox. These fuzzers have different access to the program resources and as a result, the fuzzing approaches were different.
- Code coverage and it's applications in fuzz testing is explained.
- We dig into the state-of-the-art fuzzer, AFL, and walked over it's stages that are important for our fuzzer.
- LLVM and it's applications in AFL, as well as the instrumentations done by AFL are described.

Altogether, we come up with a new approach for fuzzing, as we are aware of. These knowledge led us to introduce Waffle, which is explained in more details in the next chapters.