Chapter 4

Simulation

This chapter presents the details of our implementation of Waffle.
We implement and evaluate the performance of Waffle on an Ubuntu 18.4.5 LTS operating system. The computer consists of 16GB of RAM and Intel® Core™ i7-3770 CPU.
In the following sections, we will explain the implementation and the execution of Waffle in more detail.

4.1 Implementation

We have implemented Waffle on Memlock, which is based on AFL fuzzer. To develop Waffle, we added and modified approximately 400 lines of code in the C programming language. We also implemented the instrumentations under the LLVM framework.

4.2 Instrumentation

To inject code into the binary of the target program, we modify the two files *waffle-llvm-rt.o.c* and *waffle-llvm-pass.so.cc*. The first file is responsible for the initial setup for the SHM and fork server. The later file injects the basic-block level instrumentation, which contains our feature collection procedure.

In the file *waffle-llvm-rt.o.c* first, we specify an array of 64KB, in addition to the shared memory implemented in Memlock. This array collects the instruction counters and monitors the execution of the program through each basic-block.

```
    u32  __wafl_icnt_initial[ICNT_SIZE];
    u32* __wafl_icnt_ptr = __wafl_icnt_initial;
```

Listing 4.1: waffle-llvm-rt.o.c

Here $ICNT\_SIZE$ is equal to 2^{14} words, which makes the size of 64KB.

In the file *waffle-llvm-pass.so.cc* we implement the LLVM pass, which helps us with injecting basic-block level instructions. For our purposes, we are using the instruction visitor, which we explained in Chapter 2.

After the above modifications on Memlock, we can *make* the LLVM project within Waffle. Our improvements will be applied in the *waffle-clang*, which we can use for compiling the target program.

To compile a source code to an executable, we specify the path to *waffle-clang* as the compiler for building executable target programs. For instance, if the program contains a single source file, we can use the following command:

```
./waffle-clang -i <sourcecode-path> -o <executable-path>
```

Listing 4.2: Compile a single file using waffle-clang

Choosing the right compiler here is necessary as *waffle-clang/waffle-clang++* injects the proposed instrumentation. Generally, we can define the path to the compiler by exporting it to the environment variables:

```
export CC=waffle-clang
export CXX=waffle-clang++
```

Listing 4.3: Compile using waffle-clang

Waffle uses the resulting executables for its fuzzing procedure.

4.3 Start fuzzing

Waffle takes the current entry from the queue, fuzzes it for a while, and when finished, turns back to the queue for another entry.

To use the instrumentation features, Waffle participates in sharing memory with the target program. The instantiated shared memory will be passed to the target program every time the fuzzer executes the program.

After the execution of the target program, collected data are stored in three arrays, *trace_bits*, *perf_bits*, and *icnt_bits*. These arrays contain the information for coverage, stack memory consumption, and the counters for each basic block's instructions. The total size of the shared memory is 128KB.

Memlock calculates a *stallness* for measuring the performance of the fuzzer. If the behavior of execution of input is too stall, Memlock skips fuzzing the current input entry and continues with the next entry of the queue. In Waffle, we do not intend to measure stallness, and we switch between fuzzing approaches, as mentioned before.

AFL trims the test-cases to increase the performance of coverage-finding techniques. As trimming the results may affect the target program's resource consumption, Memlock and Waffle disable this method in the fuzzing stage.

Next, Waffle assesses its interest in considering the input as a favorable input.

```c
total_counts = 0;
for (i = 0; i < ICNT_SIZE; i++) {
  if (icnt_bits[i]) {
    total_counts += icnt_bits[i];
    if (top_rated[i]) {
      if (icnt_bits[i] < max_counts[i]) continue;
    }
    /* Insert ourselves as the new winner. */
    top_rated[i] = q;

    /* if we get here, we know that icnt_bits[i]==max_counts[i] */
    score_changed = 1;
  }
}
if(total_counts >= max_total_counts){
  top_rated[i] = q;
  score_changed = 1;
  }
```

Listing 4.4: Update bitmap scores

Here $ICNT\_SIZE$ is equal to the size of the bitmap for collecting the instruction counters. In case we find a new max for the number of instructions in a basic-block, we select the current input as the winner - favorable. [listing 4.4]

We are also selecting inputs with a total number of instructions more than any other input that was executed before.

```c
if (top_rated[i]) {
  /* if top rated for any i, will be favored */
  u8 was_favored_already = top_rated[i]->favored;

  top_rated[i]->favored = 1;

  /* increments counts only if not also favored for another i */
  if (!was_favored_already){
    queued_favored++;
    if (!top_rated[i]->was_fuzzed) pending_favored++;
  }
}
```

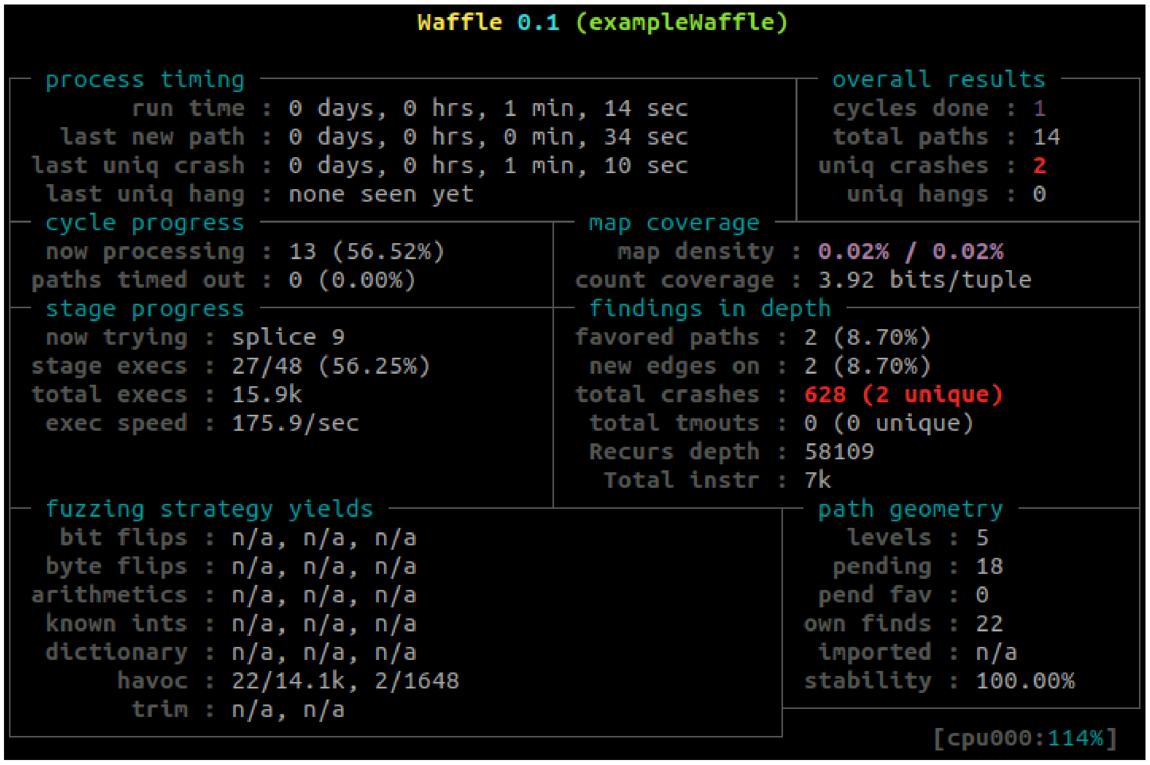
Listing 4.5: Cull queue

After collecting the execution features for an input, Waffle continues culling the queue and selects unique favorite inputs for the next generations. [listing 4.5]

Notice that AFL and Memlock do the same method for selecting favorite inputs, except that they do not consider an overall execution feature, which Waffle knows as max_total_counts.

4.4 Monitoring fuzzing procedure

To measure the performance of AFL, we can use the live status screen.



```
                        Waffle 0.1 (exampleWaffle)

┌─ process timing ─────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 1 min, 14 sec       │  cycles done : 1   │
│   last new path : 0 days, 0 hrs, 0 min, 34 sec       │  total paths : 14  │
│ last uniq crash : 0 days, 0 hrs, 1 min, 10 sec       │ uniq crashes : 2   │
│  last uniq hang : none seen yet                      │   uniq hangs : 0   │
├─ cycle progress ──────────────────┬─ map coverage ───┴────────────────────┤
│  now processing : 13 (56.52%)     │    map density : 0.02% / 0.02%        │
│ paths timed out : 0 (0.00%)       │ count coverage : 3.92 bits/tuple      │
├─ stage progress ──────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 9            │ favored paths : 2 (8.70%)             │
│ stage execs : 27/48 (56.25%)      │  new edges on : 2 (8.70%)             │
│ total execs : 15.9k               │ total crashes : 628 (2 unique)        │
│  exec speed : 175.9/sec           │  total tmouts : 0 (0 unique)          │
│                                   │  Recurs depth : 58109                 │
│                                   │   Total instr : 7k                    │
├─ fuzzing strategy yields ─────────┴─────────────┬─ path geometry ─────────┤
│   bit flips : n/a, n/a, n/a                     │    levels : 5           │
│  byte flips : n/a, n/a, n/a                     │   pending : 18          │
│ arithmetics : n/a, n/a, n/a                     │  pend fav : 0           │
│  known ints : n/a, n/a, n/a                     │ own finds : 22          │
│  dictionary : n/a, n/a, n/a                     │  imported : n/a         │
│       havoc : 22/14.1k, 2/1648                  │ stability : 100.00%     │
│        trim : n/a, n/a                          ├─────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000:114%]
```

Figure 4.1: Status screen

In the above screen, the total number of instructions executed and the maximum depths for the stack is presented. Eventually, as the

4.5 Summary

In this chapter, we covered the following topics:

- The implementation of the instrumentation
- The modifications in the fuzzing procedure

- We also improved the status screen to illustrate the details of the fuzzing procedure and for future assessments