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.

4.1 Implementation

We have implemented Waffle on Memlock which is based on AFL fuzzer. To develop Memlock, we added and modified approximately 400 lines of code in C programming language. We also implemented the instrumentations under 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 forkserver and etc. and 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, that makes the size of to 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 modifications, we can *make* the LLVM project within Waffle and our changes 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 it's 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 features specified in the instrumentation stage, we tell Waffle to participate 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 the instructions within each basic block. 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 an 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 the results of trimming may affect the resource consumption of the target program, Memlock and Waffle disable this method in the fuzzing stage.

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

```
for (i = 0; i < ICNT_SIZE; i++) {
  if (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;
  }
}
```

Listing 4.4: Update bitmap scores

```
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

4.4 Monitoring for exceptions

4.5 Report vulnerabilities