Chapter 3

Proposed Fuzzer

3.1 Introduction

In this chapter a new fuzzer is introduced, that is capable of finding the vulnerabilities related to (theoretically) any resource's exhaustion. The first section explains a motivating example leading to our proposed fuzzer. The fuzzer is based on AFL and uses the implementation of Memlock for memory usage assessments. For monitoring the resources, we use compile-time instrumentation of the target program using LLVM's APIs; we take advantage of **visiting** APIs that let us keep track of any type of instructions defined for LLVM. As a result, the instructions related to any resource are counted and this information is later used in the fuzzing stage. The vulnerabilities found by our fuzzer are then tested for exploitability. The short-comings and performance expectations of our fuzzer are investigated before we conclude this chapter.

We will call our proposed fuzzer **Waffle**, which is derived from **What An Amazing AFL** - WAAAFL! The summary of our contributions are as follows:

- A new instrumentation for collecting runtime information about resource usages, i.e. memory and time.

- A new fuzz testing algorithm for collectively considering the former features of AFL and Memlock, as well as the features we introduce in Waffle.

3.2 Motivating example

A fuzzer can be considered as a machine for solving a specific set of problems: finding vulnerabilities in a program. In this thesis, we are extending the domain of our problems, so that while we are solving the main problem of examining the vulnerabilities, we are also solving the newly defined problems. For instance, consider:

- **Problem statement:** We have two functions $f1$ and $f2$, we want to find the result of the function $f1 \times f2$. **??**

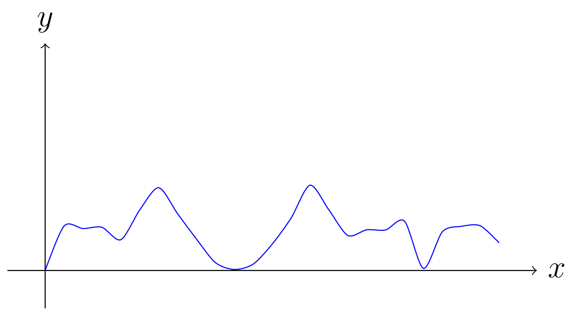
To solve this problem using Waffle, we need a dummy piece of code to mimic the resource consumptions. For now, doing nothing for this purpose would suffice. We must keep in mind that the implementation of this function should not be ignored and we need this function to consume more resources. As we can see in 3.1, the *solver* function simply mimics the multiplication of the two functions $f1$ and $f2$:

```
int solver(float x1, float x2) {
    long long y1, y2;
    y1 = f1(x1);
    y2 = f2(x2);

    // Execution time in Theta(f1*f2)
    for(long long i = 0; i<y1; i++){
        for(long long j = 0; j<y2; j++){
            do_nothing();
        }
    }

    return 0;
}
```

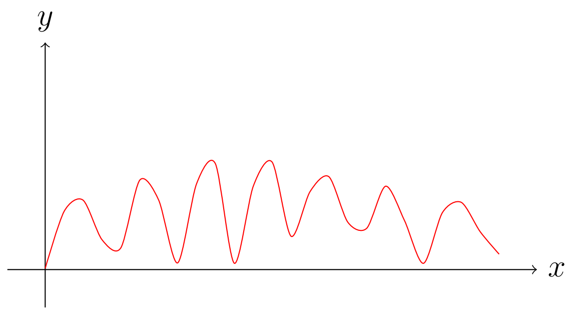
Listing 3.1: Motivating example

Waffle monitors the instructions that were visited during the execution of the targeted program. While Waffle fuzzes the program and considers the visited instructions to count the total number of instructions, Waffle also maximizes this total number to achieve the worst-case scenarios. The instructions that are important for us would be a time-consuming instruction, such as an incrementation of a variable. If we have no preference on the resource-consuming instructions, we count all instructions. We expect a higher execution-time when the number of instructions executed is increased.

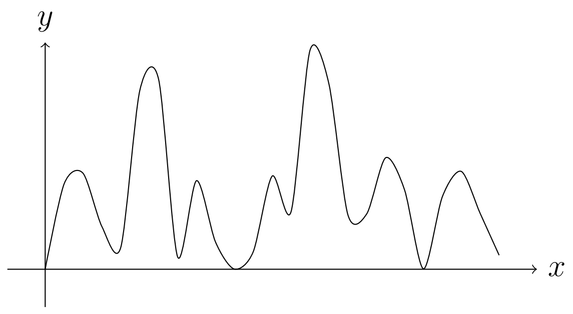
In the following plots we can see



(a) $f1$



(b) $f2$



(c) $f1 \times f2$

3.3 Instrumentation

As mentioned in Chapter 2, AFL uses the instrumentation for increasing the code coverage and Memlock uses the memory features, by calculating the maximum heap/stack size of the memory, used during the runtime. To add more features to our fuzzing, we first need to monitor and collect more runtime features, which are added to the target binary using our enhanced instrumentation.

3.3.1 Features

In addition to the features implemented and used in AFL, Waffle leverages 2 other features for guiding the fuzzing execution.

3.3.1.1 Memory consumption

Our memory consumption features are derived from the features used in Memlock [22]. To collect this information, Memlock monitors the heap or the stack's usage during the runtime, and depending on the allocation or deallocation instructions, the counter is increased or decreased accordingly. In appendix A we can see a section of the source code for Memlock that is responsible for injecting the new instructions. These instructions are added in compile-time and do not change the efficiency of the binary in execution speed. In addition, this job is a one time job that is done before the fuzzing is started.

Before AFL/Memlock starts fuzzing the program, it first sets up a shared memory with the target program. When the fuzzer runs the program, the instrumented binary is capable of filling the **shared memory** according to any strategy we choose and LLVM supports.

Memlock has two arrays for collecting the runtime information. These arrays are `__afl_area_initial` which is implemented in AFL, and `__afl_perf_initial` for collecting the memory consuming features. Currently, the first array can keep 2^{16}