- 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 trying to solve the new defined problems. For instance, consider the following problem:

- **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 set the target program for fuzzing as in 3.1:

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

    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. The idea is that Waffle fuzzes the program and adds visiting instructions to count the number of instructions, and maximizing this number results in the worst-case scenario when the number of a specific set of instructions is considered as important. Here, we focus on any instruction and as a result, we are counting

every instruction and the more instructions we execute, the more execution-time we expect.
To go into more details, let us examine the binary of the instrumented program...

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