Chapter 3

Proposed Fuzzer

3.1 Introduction

In this chapter a new fuzzer is introduced, that is capable of finding the vulnerabilities related to (theoritically) 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

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 these 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 shares 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} elements, each one as a byte; the second array keeps 2^{14} elements of double words - 32bits.

As mentioned before, Memlock has two different fuzzing methods, one for collecting stack information and the other one for collecting heap information.

After a function is called, the injected instructions increase the counter corresponding to the current basic block by 1; every time a `return` instruction is called, this value is decreased by 1.

On the other hand, the fuzzer considering the heap information, must look for opcodes that affect the heap, e.g. the instructions inserted in **allocation** and **deallocation** functions, such as `malloc` or `free` functions.

After these instrumentations are applied on the target, Memlock can start fuzzing the generated binary.

3.3.1.2 Instruction counters

We are looking for features that can help the fuzzer find inputs causing a timeout, or memory exhaustion. We collect these information in runtime using our instrumentation.

The intuition behind this thesis comes from the fact that all the programming competitions announce two main resources that are limited for the execution of the submitted program. This means that a program must be run without any compile-time or run-time errors and generate the correct output; and the whole execution is constrained with a specific time limit and memory limit. [29] In some competitions, the judge system lets the competitors read each others' submitted program. If they can find a vulnerability in the program that is exploitable and causes any expected result, except the correct answer, the recieve the score for their successful **hacking** attempt. [30]

A fuzzer can automate the process of finding vulnerabilities; and targeting the algorithmic problems requires a resource-aware approach for investigating a larger range of problematic inputs.

Collecting the memory-related features are done by Memlock and Waffle needs to be aware of the time-consuming instructions that may lead to a timeout. To gather these information, Waffle **counts the number of instructions in a basic block and increases a counter by that value**. Waffle gathers the information with the help of **instruction visitors** defined by LLVM.

From the documentations of LLVM, "instruction visitors are used when you want to perform different actions for different kinds of instructions without having to use lots of casts and a big switch statement (in your code, that is)." For our purpose, we only take the visitor class `visitInstruction`, and only count how many instructions are located in the current basic block. This number is later added to the total value whenever the basic block is run in the execution.

The visitor is created and is called at the beginning of the basic block and it visits all the instructions inside the basic block. The visitor counts how many instructions are currently existing in the basic block, before we inject any other instruction. In 3.1 a section of the source code for Waffle is provided. In lines **1** to **8** the definition of the visitor class is shown. The only function that is overriden is `visitInstruction` function, which increments the `Count` variable by 1, for each instruction. In the AFL pass from line **10** to line **25**, the visitor is called within the instruction. After the visitor visited all the instructions (line **14**), the result is injected to collect the number of instructions in the run-time. These values are calculated and injected in the compile time.

```
struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
    unsigned Count;
    CountAllVisitor() : Count(0) {}

    void visitInstruction(Instruction &I) {
        ++Count;
    }
};

bool AFLCoverage::runOnModule(Module &M) {
  for (auto &F : M) {
    for (auto &BB : F) {
      CountAllVisitor CAV;
      CAV.visit(BB);
      LoadInst *IcntTotalCounter = IRB.CreateLoad(IcntPtr);
      IcntTotalCounter->setMetadata(M.getMDKindID("nosanitize"),
   MDNode::get(C, None));
      ConstantInt *CNT = ConstantInt::get(Int32Ty, CAV.Count);
      Value *IcntTotalIncr = IRB.CreateAdd(IcntTotalCounter, CNT);

      Value *IcntTotalIncrCasted = IRB.CreateZExt(IcntTotalIncr, IRB
   .getInt32Ty());
      IRB.CreateStore(IcntTotalIncrCasted, IcntEPtr)
        ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
   None));
    }
  }
}
```

Listing 3.1: Waffle's instruction counting

After the instrumentation is complemented and the target program is generated, Waffle can now start fuzzing the targeted binary.

3.4 Fuzzing the target

3.5 Concluding remarks

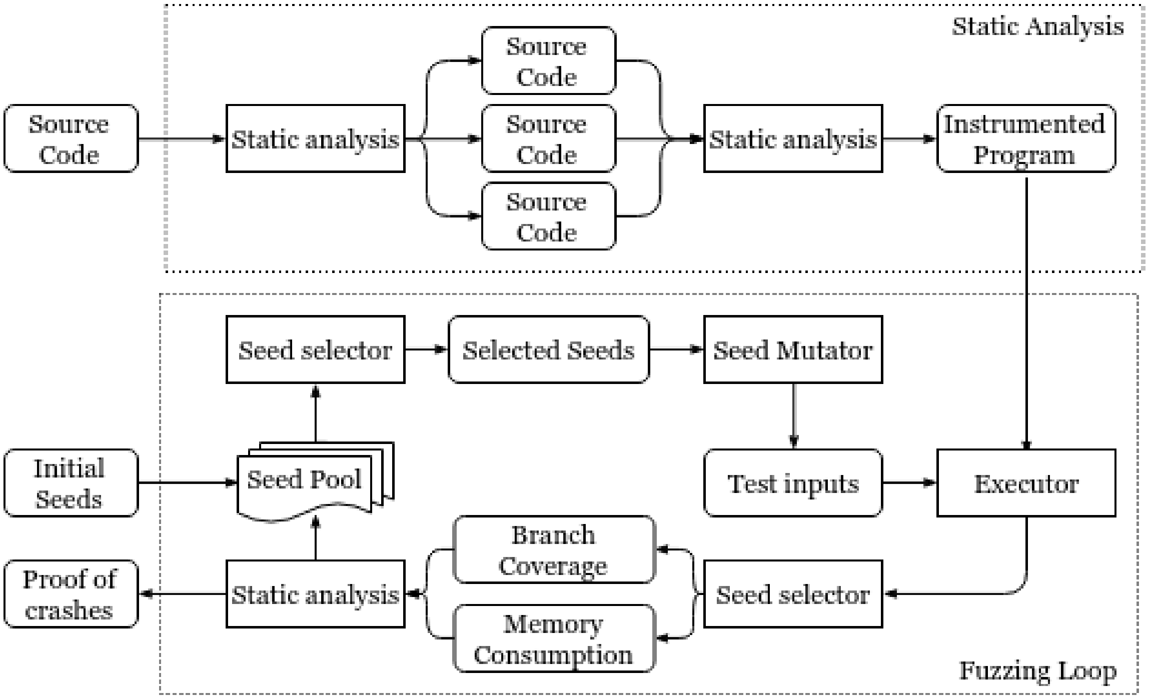

Figure 3.1: Memlock's approach