Chapter 2

Background

2.1 Introduction

The term **fuzzing** or **fuzz testing** was first introduced in late 80's. OWAPS defines it as a tool for "finding implementation bugs using malformed/semi-malformed data injection in an automated fashion". [2]

There are different implementations of fuzzers for different purposes. Our fuzzer is a coverage-based whitebox fuzzer, and we will explain theis definition in this chapter. In this chapter, we begin by reviewing the previous works that lead to this thesis 2.2. We bring up the features we use from LLVM in section 2.3. In section 2.4 we review the implementation of AFL, and we wrap up this chapter with conclusions.

2.2 Literature Review

Sutton et al. [3] defined the procedure of fuzz testing, as shown in Figure 2.1:

1. Generally, the first stage is **to identify the target**. A target may be a software or a combination of software and hardware [4]. The targeted software is any program that is executed on a machine. For the rest of this article, a target is software.

2. To execute the target program, we need to specify how the program parses the inputs. Inputs are a set of environmental variables, file formats, and any other parameters that affect the program's execution.

   A fuzzer needs a set of seeds for initialization. This set could be empty, and a fuzzer without any sample inputs may find valid fuzzed files out of thin air [5] that the target program considers them as valid.

3. The fuzzing loop starts with generating fuzzed inputs. The fuzzer provides these files for the program to execute and process them.

4. In this stage, the inputs are processed and executed. Depending on the resources needed for the target program's execution, this stage can be the bottleneck for the fuzzing process. The executions' interesting behavior and information are collected for the evolution of the inputs in future iterations.

5. If an exceptional event happens during the program's execution and the program exits unsuccessfully, we say an exception is occurred. These exceptions are the vulnerabilities that may be exploitable and cause security problems. Handling an exception properly and pinpointing the inputs responsible for the

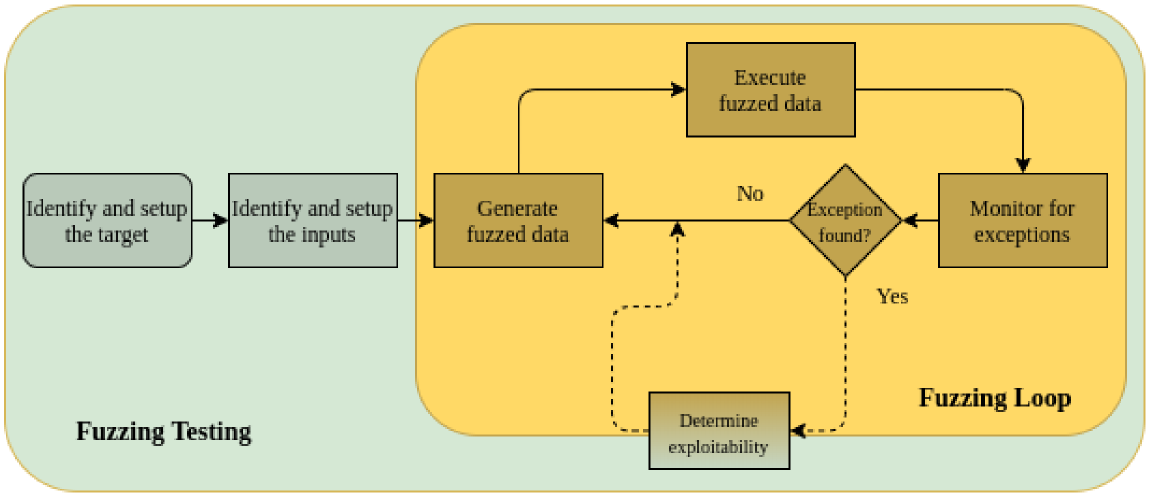


Figure 2.1: Fuzzing phases

exception is the primary purpose of this stage.

6. The last stage of the fuzzing is evaluating the reported vulnerabilities. This evaluation explains whether the vulnerabilities are exploitable or not.

A common feature in most of the fuzzers is the fuzzing loop which is looking for more valuable inputs. The stages may vary depending on the problem and goals of a fuzzer. We will walk over the stages of AFL as our fuzzer is based on AFL, and we will be back on the stages of fuzzing in the following sections.

Sample program

To evaluate the performance of a fuzzer and assess the execution of it, the sample **C** code is implemented. (Listing 2.1) [6]

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void vul_function(char * msg) {
    char *ptr = NULL;
    ptr = malloc(strlen(msg));
    strcpy(ptr, msg);
    printf("%s\n", ptr);
}

int main(int argc, char *argv[])
{
    if(argc<2) {
        printf("At least one input is needed");
        return 1;
    }

    vul_function(argv[1]);

    return 0;
}
```

Listing 2.1: Sample vulnerable program

The sample program has a **Heap buffer overflow vulnerability**. If we compile this code with GCC and debugging flags, the vulnerability stays hidden and the program executes without any errors:

```
$ gcc sample_vul.c -o sample_vul -Wall -Werror -g
$ ./sample_vul hello_world
hello_world
```

Listing 2.2: Compile the sample program

One way to detect the prior vulnerability, is to add AddressSanitizer flag for the compilation [7]. ASan is a fast memory error detector. This tool uses memory poisoning for the detection of a heap buffer overflow. (You can find more features of this tool in the reference) [8]

After we provide the ASan flag for the compilation, we face an error with the same input as the previous example (Listing 2.2):

```
$ gcc sample_vul.c -o sample_vul_asan -Wall -Werror -g -fsanitize=address
$ ./sample_vul_asan hello_world
=================================================================
==304989==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000001b at
     pc 0x7f04ce49215d bp 0x7fff3c893100 sp 0x7fff3c8928a8
WRITE of size 12 at 0x60200000001b thread T0
  ...
  ...
SUMMARY: AddressSanitizer: heap-buffer-overflow (/lib/x86_64-linux-gnu/libasan.so
    .5+0x9c15c)
  ...
  ...
==304989==ABORTING
```

The detection of the vulnerabilities or any other exceptions, signals the operating system for a misbehaviour from the program. A fuzzer would need this signal to evaluate the execution of the target program.

Black/Grey/White-box fuzzing

The *colorful* representation of fuzzers depends on how much information we can collect from any execution of the target. In a blackbox fuzzing, we do not gather any information from the execution, and the fuzzing tries new inputs expecting an error to occur.

```
$ ./blackbox-fuzzer ./sample_vul_asan
  [Blackbox fuzzing: sample_vul_asan]
```

The introduced fuzzer by Miller [9] was of the very first naive blackbox fuzzers targeting a collection of Unix utilities on different Unix versions. It runs the fuzzing for different lengths of inputs for each utility (of the total 88 utilities) and expects a **crash**, **hang**, or a **succeed** after the execution of the program. The generation of inputs is by the mutation of the inputs' content, and the target program is a blackbox for the fuzzer. As a result, this fuzzer is a mutation-based blackbox fuzzer. One of the **downsides** of the blackbox fuzzing is that the program may face some branches with **magic values**, constraining the variable to a specific set of values; the fuzzer has to apply the exact magic value, which may have a very low probability - it is almost impossible to generate a specific 1 KB string of bytes randomly.. In [10] and [11] a set of network protocols are fuzzed in a blackbox manner, but as the target is specified, the performance is enhanced drastically. Any application on the web may be considered a blackboxed program as well, so as [12] and [13] have targeted web applications and found ways to attack some the websites, looking for different vulnerabilities, such as XSS.

Whitebox fuzzing works with the source code of the target. In this technique, an **instrumentation** is applied before the compilation of the program. Instrumeting a program with debugging instructions, is the procedure of injecting new instructions into the resulting binary, without affecting the logic of the program. We will analyze the instrumentations more in the LLVM section.

The following snippet of code shows an example of such execution 2.2. *whitebox-instr-c* compiles the program with the required instrumentations and *whitebox-fuzzer* fuzzes the program appropriately.

```
$ ./whitebox-instr-c sample_vul.c --output sample_wht_inst
$ ./whitebox-fuzzer sample_wht_inst
  [Whitebox fuzzing: sample_wht_inst]
```

SAGE, a whitebox fuzzer, was developed as an alternative to blackbox fuzzing to cover the lacks of blackbox fuzzers [14]. With the benefit of having the source code and internal knowledge for fuzzing, a whitebox fuzzer can leverage symbolic constraints for symbolic analysis to solve the constraints (such as magic values) in the program [15]. It can also use dynamic and concolic execution [16] and use taint analysis to locate the regions of seed files influencing values used by program [17]. In addition, a whitebox fuzzer can find the grammar for parsing the input files without any prior knowledge [18]. It is a noticeable performance enhancement as we have the source code.

Greybox fuzzing resides between whitebox and blackbox fuzzing, as we only have partial knowledge about the internals of the target program. We do not have the source code, but we have some knowledge about the program (for instance, we have the binary file), and as a result, we have the instructions of the program (2.2):

```
$ ./greybox-instr-c ./sample_vul_asan --output sample_gry_inst
$ ./greybox-fuzzer sample_gry_inst
  [Greybox fuzzing: sample_gry_inst]
```

AFL “allows you to build a standalone feature that leverages the QEMU ”user emulation” mode and allows callers to obtain instrumentation output for black-box, closed-source binaries”, working as a greybox fuzzer [19]. The instrumentation using **QEMU** on a binary has an average performance cost of 2-5x, which is better than other tools such as **DynamoRIO** and **PIN**.

Coverage-based fuzzing

Coverage-based fuzzing is technique for fuzz testing that instruments the target without analyzing the logic of the program. In a greybox and whitebox coverage-based fuzzing, the instrumentation detects the different paths of the executions [20]. AFL instruments the program with only the coverage information (section AFL). The instrumentation can collect execution's data such as data coverage, statement coverage, block coverage, decision coverage, and path coverage [21]. Bohme et al. [22] introduced a coverage-based greybox fuzzer that extends AFL and benefits from the Markov Chain model. The fuzzer calculates the **energy** of the inputs based on the potency of a path for the discovery of new paths.

In another article by Bohme et al. [23], they introduce their directed fuzzing by the idea of checking the code-coverage for providing inputs that guide the program execution toward some targeted locations. Some of the applications of such a fuzzing approach are patch testing and crash reproduction, which has different use cases compared to a non-directed coverage-based fuzzers.

Performance fuzzing

The **types** of vulnerabilities that a fuzzer is involved with may be different from other fuzzers. For example, AFL looks for crashes or hangs by selecting and mutating the inputs, and at the same time, it considers the code coverage, size of the inputs, and each execution time of the target program. PerfFuzz [24] is a greybox fuzzer based on AFL, which aims to generate inputs for executions with higher **execution time** while using most of the features of AFL in code exploration. PerfFuzz counts how many times each of the edges of the control flow graph (CFG) is executed. Using SlowFuzz [25], we can consider any type of resource as a feature to detect the worst-case scenarios (inputs) for a given program. In another project based on AFL, Memlock [26] investigates memory exhaustion by calculating the maximum runtime memory required during executions. A disadvantage in previous works in performance is that the development of the fuzzer for considering different instructions is cumbersome.

Waffle (What An Amazing AFL - WAAAFL) is a coverage-based whitebox fuzzer that is based on AFL's base code. Waffle leverages **visitors** to collect the stats of different instructions during the execution. To learn more about Waffle, we study the features we benefit from the LLVM, as well as the current features of AFL that help us in reaching the goals of this thesis.

2.3 LLVM

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. The LLVM project has multiple components. The core of the project is itself called "LLVM." This project contains all of the tools, libraries, and header files needed to process intermediate representations and converts them into object files. Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer." [27, 28]

LLVM can be used as a compiler framework, separated into "front-end" and "back-end." The front-end contains the lexers and parsers, and it accepts the source code to a program and returns the **intermediate representation (IR)** of the program. The back-end converts the IR into machine language.

For instrumentation, we insert the logging instructions into each basic block of the program in the front-end. **Clang** is part of the LLVM toolchain for compiling C/C++ source code. By definition, "**clang** is a C, C++, and Objective-C compiler that encompasses preprocessing, parsing, optimization, code generation, assembly, and linking."[29] We extend the phases of compilation so that we are injecting the instructions in compilation.

LLVM converts an **IR** of a program into machine language instructions. The structure of the LLVM project is shown in Figure 2.2:

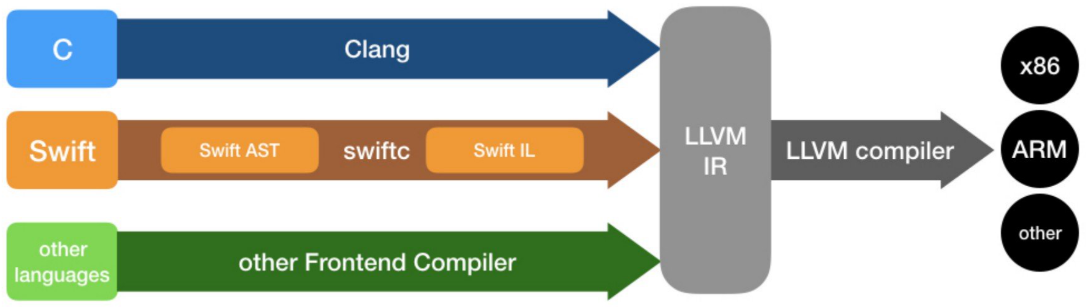


Figure 2.2: LLVM architecture: A front-end compiler generates the LLVM IR, and then it is converted into machine code [1]

The instrumentation is applied before the IR generation, and the LLVM IR is fed into the LLVM compiler to generate the machine-specific instructions. As our instrumentation does not affect the LLVM IR compilation, we will not investigate the generated IR.

2.3.1 Instrumentation and coverage measurements

Waffle is based on AFL and we are extending the AFL's instrumentation in our work. The goal of using instrumentation for AFL is to differentiate code coverages. There are two techniques for instrumentation in AFL:

1. *llvm_mode*: AFL takes the source code and an instrumentation recipe and generates the instrumented binary of the target program.

2. *qemu_mode*: AFL leverages the QEMU mode to obtain instrumentation output for closed-source binaries. We don't use this mode in this thesis.

In the LLVM recipe, we instantiate the bitmap and assign it to the shared memory for modifications. The remaining instructions for the recipe will be applied on the basic blocks in **AFLCoverage** module. This module takes effect in compilation of the program before the generation of IR. We can see some of the implementation of this **pass** in Listing 2.3:

```
// LLVM-mode instrumentation pass
bool AFLCoverage::runOnModule(Module &M) {

  /* Instrument all the things! */
  for (auto &F : M)
    for (auto &BB : F) {
      BasicBlock::iterator IP = BB.getFirstInsertionPt();
      IRBuilder<> IRB(&(*IP));

      if (AFL_R(100) >= inst_ratio) continue;

      /* Make up cur_loc */
      unsigned int cur_loc = AFL_R(MAP_SIZE);
      ConstantInt *CurLoc = ConstantInt::get(Int32Ty, cur_loc);

      /* Load prev_loc */
      LoadInst *PrevLoc = IRB.CreateLoad(AFLPrevLoc);
      PrevLoc->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
    C, None));
      Value *PrevLocCasted = IRB.CreateZExt(PrevLoc, IRB.getInt32Ty
    ());

      /* Load SHM pointer */
      LoadInst *MapPtr = IRB.CreateLoad(AFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C
    , None));
      Value *MapPtrIdx =
          IRB.CreateGEP(MapPtr, IRB.CreateXor(PrevLocCasted, CurLoc)
    );

      /* Update bitmap */
      LoadInst *Counter = IRB.CreateLoad(MapPtrIdx);
      Counter->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(
    C, None));
      Value *Incr = IRB.CreateAdd(Counter, ConstantInt::get(Int8Ty,
    1));
      IRB.CreateStore(Incr, MapPtrIdx)
          ->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
    None));

      /* Set prev_loc to cur_loc >> 1 */
      StoreInst *Store =
          IRB.CreateStore(ConstantInt::get(Int32Ty, cur_loc >> 1),
    AFLPrevLoc);
      Store->setMetadata(M.getMDKindID("nosanitize"), MDNode::get(C,
     None));

      inst_blocks++;
    }

  return true;
}
```
Listing 2.3: AFLCoverage module

The recipe for instrumentation fills out the coverage bitmap with the hash values of the paths executed. The instructions are as followed:

```
  cur_location = <COMPILE_TIME_RANDOM>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
```
Listing 2.4: Select element and update in shared_mem

AFL instruments by adding these instructions into basic blocks. First, a random value is assigned to *curr_location*. Next, it is XORed with the previous location's value, *prev_location*, and the resulting value is the location on *shared_mem*, the *coverage bitmap*, which is incremented by one. The third and final instruction is reseting the *prev_location* to a new value.

When AFL runs the instrumented program, every time an instrumented basic block is executed, a dedicated location of *shared_mem* in the bitmap is incremented. This algorithm recognizes the different paths that AFL runs through. For instance, in figure 2.3, suppose that we have an instrumented program with the random values which is set in compile time. An execution that walks over basic blocks $1 \rightarrow 2 \rightarrow 5$ will increase the value of the corresponding locations by 1; for instance, an increment on $shared\_mem[14287 \oplus 23765]$ is applied for the transition of $1 \rightarrow 2$ and $shared\_mem[7143 \oplus 21689]$ for $2 \rightarrow 5$. We can see that the paths $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 5$ (which contains a loop), set different locations on bitmap.

AFL uses this coverage feature for discovering new inputs with new code coverages.

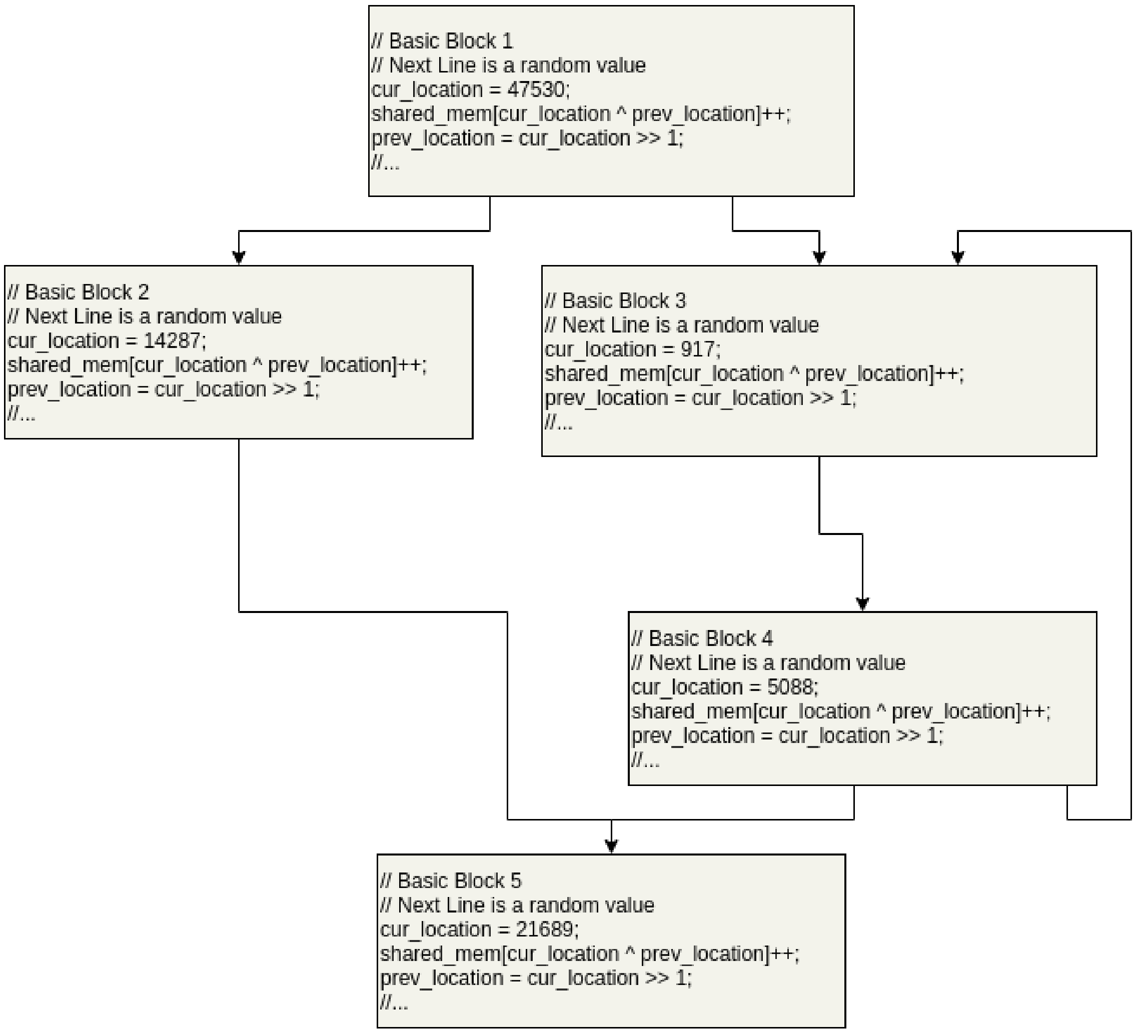

// Basic Block 1
// Next Line is a random value
cur_location = 47530;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
// Basic Block 2
// Next Line is a random value
cur_location = 14287;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
// Basic Block 3
// Next Line is a random value
cur_location = 917;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
// Basic Block 4
// Next Line is a random value
cur_location = 5088;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...
// Basic Block 5
// Next Line is a random value
cur_location = 21689;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
//...

Figure 2.3: Example for instrumented basic blocks

Visitor functions

"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). [30]"

```cpp
#include "llvm/IR/InstVisitor.h"

struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
  unsigned Count;
  CountAllVisitor() : Count(0) {}
  // Any visited instruction is counted in a specified range
  void visitInstruction(Instruction &I) {
    ++Count;
  }
};
```

Listing 2.5: Visitors example

The specified range can be any two iterators, which can be a Module, Function, BasicBlock, Instruction or any other range between two instruction addresses.

2.4 AFL

Michal Zalewski initially developed American Fuzzy Lopper. He introduces this open-source project as "a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test inputs that trigger new internal states of the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool help seed other, more labor- or resource-intensive testing regimes down the road." [31]

AFL is designed to perform **fast** and **reliable**, and at the same time, benefit from the **simplicity** and **chainability** of the fuzzer [32]:

- **Speed:** Avoiding the time-consuming operations and increasing the number of executions over time.

- **Reliability:** AFL takes strategies that are program-agnostic, leveraging only the coverage metrics for more discoveries. This feature helps the fuzzer to perform consistent in finding the vulnerabilities in different programs.
- **Simplicity:** AFL provides different options, helping the users enhance the fuzz testing in a straightforward and meaningful way.
- **Chainability:** AFL can test any binary which is executable, and is not constrained by the target software. A driver for the target program can connect the binary to the fuzzer.

afl-fuzz.c has the instructions for fuzzing the target instrumented-program. The Algorithm 1 illustrates a brief pseudocode of the execution of *afl-fuzz*:

Algorithm 1: afl-fuzz

```
  Input: $in\_dir$, $out\_dir$, $instrumented\ Target$
1 initialize fuzzer;
2 while fuzzing is not terminated do
3     cull queue and update bitmaps;
4     $Entry \leftarrow q.first\_entry()$;
5     $fuzz\_one(Entry)$;
```

After the environmental initializations, the fuzzing loop continues until receiving a termination signal. In every iteration of the loop, AFL first culls the corpus of the generated entries. This method assigns a **favor-factor** (Eq 2.1) to each queue_entry and marks the **favorite entries**, as they execute faster and the size of the files are smaller than the rest of the corpus. AFL finds a favorable path for "having a minimal set of paths that trigger all the bits seen in the bitmap so far, and focus on fuzzing them at the expense of the rest." [33]

$$fav\_factor = e.exec\_time \times e.length \tag{2.1}$$

An entry is selected after culling the queue. AFL evolves the input corpus by generating new entries out of the selected input entry - `fuzz_one()`.

fuzz_one()

Algorithm 2: *fuzz_one*: Fuzz one Entry

```
    Input: $queueEntry$
1   $T \leftarrow Entry.test\_case$;
2   $calibrate(T)$;
3   $trim(T)$;
4   $perf\_score \leftarrow calculate\_score(T)$;
5   if $deterministic\_mode$ then
6       $deterministic\_stages(T)$;
7   $MX \leftarrow calc\_stages(perf\_score, \mathcal{C})$;
8   foreach $element$ in $(0, MX)$ do
9       $random\_havoc(T)$;
```

Fuzzing a single entry requires the calibration of the test-cases; calibration helps AFL in evaluating the stats of the current entry. This evaluation is stored in `perf_score`, and the number of trials for generating new inputs from the **random_havoc** stage is calculated using the `perf_score`. AFL, as a coverage-based fuzzer, assigns higher performance score for the entries that are executed faster and have bigger bitmaps. The evolutionary algorithms for generating new entries are applied in two stages: **deterministic** and **random_havoc** stages. As shown in Algorithm 2, AFL initially tries the basic, deterministic algorithms. These algorithms are executed for a specific number of times, in the same order, and once for each fuzzing trial. . Bit-flipping, byte-flipping, simple arithmetic operations, using known integers and values from dictionaries, are a sequence of mutations that AFL applies on an entry in **deterministic** stage. Each one of the above operations tweak a small portion of the fuzzed input, and does not modify the file in large portions - up to 32 bits changes in each tweak.

The **havoc** stage is a cycle of stacked random tweaks. AFL assesses the current entry to insert into the queue. Each mutation is selected randomly and with a higher `perf_score`, this stage continues more fuzzing over the current entry. The `random_havoc` stage consists techniques such as bit flips, overwriting with random and interesting integers, block deletion, block duplication, and (if supplied) assorted dictionary-related operations [34]. An abstract implementation of the `havoc_stage` can be found in Appendix 5.A.

calculate_score()

This function calculates how much AFL desires to iterate in havoc stage, for the current entry. By default, AFL is interested in fuzzing an input with less execution-time, and simultaneously, showing more coverage, and it's generation depth is higher. The depth of a child entry is one more than the depth of it's parent. For more information, check the following abstraction of the function [Listing 2.6]:

```
/* Calculate case desirability score to adjust the length of havoc
   fuzzing. A helper function for fuzz_one(). Maybe some of these
   constants should go into config.h. */

static u32 calculate_score(struct queue_entry* q) {
  u32 perf_score = 100;

  /* Adjust score based on execution speed of this path, compared to
     the global average. Multiplier ranges from 0.1x to 3x. Fast
    inputs are less expensive to fuzz, so we're giving them more air
     time. */

  if (q->exec_us * 0.1 > avg_exec_us) perf_score = 10;
  else if (q->exec_us * 4 < avg_exec_us) perf_score = 300;
  // Check other conditions in between

  /* Adjust score based on bitmap size. The working theory is that
    better coverage translates to better targets. Multiplier from
    0.25x to 3x. */
  if (q->bitmap_size * 0.3 > avg_bitmap_size) perf_score *= 3;
  else if (q->bitmap_size * 3 < avg_bitmap_size) perf_score *= 0.25;
  // Check other bitmap_sizes in between

  /* Adjust score based on handicap. Handicap is proportional to how
     late in the game we learned about this path. Latecomers are
    allowed to run for a bit longer until they catch up with the
    rest. */

  /* Final adjustment based on input depth, under the assumption
     that fuzzing deeper test cases is more likely to reveal stuff
     that can't be discovered with traditional fuzzers. */

  switch (q->depth) {
    case 0 ... 3:   break;
    case 14 ... 25: perf_score *= 4; break;
    // Check other cases in between
    default:        perf_score *= 5;
  }

  /* Make sure that we don't go over limit. */
  if (perf_score > HAVOC_MAX_MULT * 100) perf_score = HAVOC_MAX_MULT
    * 100;

  return perf_score;
}
```

Listing 2.6: An abstract implementation of calculate_score()

common_fuzz_stuff()

The newly fuzzed (generated) inputs must pass `common_fuzz_stuff()` for validating and instantiating a `queue_entry`. The validation checks the length of the fuzzed file, executes the program and keeps the exit-value of the execution. In the end, AFL calls `save_if_interesting()` to insert the entry into the queue, if it is interesting. The function `has_new_bits()` considers how interesting an entry is.

```
/* Write a modified test case, run program, process results. Handle
   error conditions, returning 1 if it's time to bail out. This is
   a helper function for fuzz_one(). */

EXP_ST u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {
  // Validate the file
  // ...
  // Validate the execution
  fault = run_target(argv, exec_tmout);
  // If the file is "interesting", add it into queue
  queued_discovered += save_if_interesting(argv, out_buf, len, fault
   );
  // ...
  return 0;
}

/* Check if the result of an execve() during routine fuzzing is
   interesting, save or queue the input test case for further
   analysis if so. Returns 1 if entry is saved, 0 otherwise. */

```

```c
static u8 save_if_interesting(char** argv, void* mem, u32 len, u8
    fault) {
  if (fault == crash_mode) {
      hnb = has_new_bits(virgin_bits);
  }
  if(!hnb) return 0;

  queue_top->exec_cksum = hash32(trace_bits, MAP_SIZE, HASH_CONST);

  /* Try to calibrate inline; this also calls update_bitmap_score()
    when successful. */
  res = calibrate_case(argv, queue_top, mem, queue_cycle - 1, 0);

  // Save the file and return
}
```

Listing 2.7: An abstract implementation of common_fuzz_stuff()

has_new_bits()

AFL calls this function after each execution of the program, and the optimization of this code participates an important role.

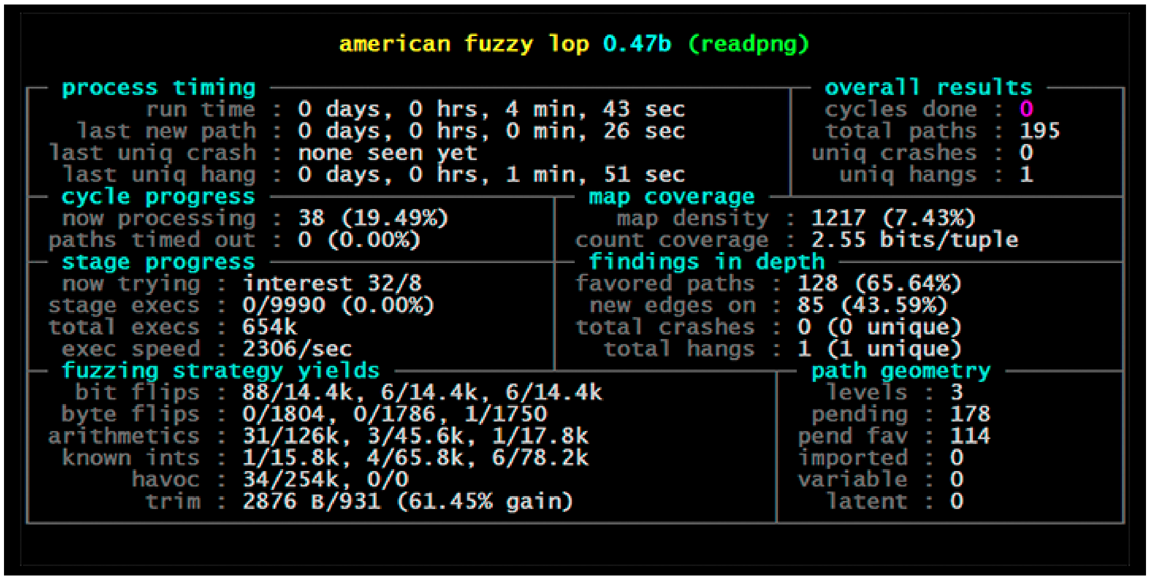
```c
/* Check if the current execution path brings anything new to the
    table. Update virgin bits to reflect the finds. Returns 1 if the
     only change is the hit-count for a particular tuple; 2 if there
     are new tuples seen. Updates the map, so subsequent calls will
    always return 0.

   This function is called after every exec() on a fairly large
    buffer, so it needs to be fast. We do this in 32-bit and 64-bit
    flavors. */

static inline u8 has_new_bits(u8* virgin_map) {
  u64* current = (u64*)trace_bits;
  u64* virgin  = (u64*)virgin_map;
  u32  i = (MAP_SIZE >> 3);
  while (i--) {

  /* Optimize for (*current & *virgin) == 0 - i.e., no bits in
    current bitmap that have not been already cleared from the
    virgin map - since this will almost always be the case. */

    if (unlikely(*current) && unlikely(*current & *virgin)) {
      if (likely(ret < 2)) {
        u8* cur = (u8*)current;
        u8* vir = (u8*)virgin;

        /* Looks like we have not found any new bytes yet; see if
    any non-zero bytes in current[] are pristine in virgin[]. */
        if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff)
     ||
            (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff) ||
            (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff) ||
            (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff)) ret = 2;
        else ret = 1;
      }
      *virgin &= ~*current;
    }
    current++; virgin++;
  }

  if (ret && virgin_map == virgin_bits) bitmap_changed = 1;
  return ret;
}
```

Listing 2.8: The implementation of has_new_bits()

Status screen

The **status screen** is a UI for the status of the fuzzing procedure. As it is shown in Figure 2.4, there are multiple stats provided in real-time updates:

1. **Process timing**: This section tells about how long the fuzzing process is running.



```
                        american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec        │  total paths : 195   │
│ last uniq crash : none seen yet                       │ uniq crashes : 0     │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec        │   uniq hangs : 1     │
├─ cycle progress ───────────────────┬─ map coverage ──┴──────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ───────────────────┼─ findings in depth ─────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ──────────┴──────────────┬─ path geometry ─────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k           │    levels : 3           │
│  byte flips : 0/1804, 0/1786, 1/1750               │   pending : 178         │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k            │  pend fav : 114         │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k            │  imported : 0           │
│       havoc : 34/254k, 0/0                         │  variable : 0           │
│        trim : 2876 B/931 (61.45% gain)             │    latent : 0           │
└────────────────────────────────────────────────────┴─────────────────────────┘
```

Figure 2.4: AFL status screen

2. **Overall results**: A simplified information about the progress of AFL in finding paths, hangs, and crashes.

3. **Cycle progress**: As mentioned before, AFL takes one input and repeats mutating it for a while. This section shows the information about the current cycle that the fuzzer is working on.

4. **Map coverage**: "The section provides some trivia about the coverage observed by the instrumentation embedded in the target binary. The first line in the box tells you how many branches we have already hit, in proportion to how much the bitmap can hold. The number on the left describes the current input; the one on the right is the entire input corpus's value. The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every branch, the needle will start to move toward "8.00" (every bit in the 8-bit map hit) but will probably never reach that extreme.

   Together, the values can help compare the coverage of several different fuzzing jobs that rely on the same instrumented binary. "

5. **Stage progress**: The information about the current mutation stage is briefly provided here.

6. **Findings in depth**: The crashes and hangs and any other findings (here we have the other information about the coverage) are presented in this section.

7. **Fuzzing strategy yields**: To illustrate more stats about the strategies used since the beginning of fuzzing, and for comparison of those strategies, AFL keeps track of how many paths were explored, in proportion to the number of executions attempted, for each of the fuzzing strategies.

8. **Path geometry**: The information about the inputs and their depths, which says how many generations of different paths were produced in the process. For instance, we call the seeds we provided for fuzzing the ”level 1” inputs. Next, a new set of inputs is generated as ”level 2”, the inputs derived from ”level 2” are ”level 3,” and so on.

Start Fuzzing

AFL requires the instrumented binary for execution. To start the instrumentation, AFL uses *afl-clang*, which is built with the coverage recipe included. The following command instruments the sample program 2.1:

```
afl-clang sample_vul.c -o sample_vul_i
```

Listing 2.9: Instrument *sample_vul*.c

Now AFL can run this program in *afl-fuzz* with the coverage instrumentations.

```
# afl-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
afl-fuzz -i in_dir -o out_dir -- ./sample_vul_i
```

Listing 2.10: Execute AFL

The fuzzing continues until the fuzz testing is stopped using a termination signal. Pressing *Ctrl+C* is a common command for this purpose. All of the recorded information are accessed through the output directory *out_dir*.

2.5 Concluding remarks

In this chapter, we reviewed the previous works that inspired us for the development of Waffle. We covered these topics:

- A brief description of the previous fuzzers.
- The recognition of whitebox, blackbox and greybox fuzzers.

- Code coverage technique and its applications in fuzz testing were explained.
- We briefly explained the instrumentation with LLVM and visitor functions.
- We dug into the state-of-the-art fuzzer, AFL, and researched its fuzzing procedure.

In the next chapter we will explore more into the modifications we applied on AFL to achieve Waffle.

Chapter 3

Proposed Fuzzer

3.1 Introduction

In this chapter, we introduce **Waffle** in more details. Waffle is a tool capable of finding the vulnerabilities related to (theoretically) any resource exhaustion. The first section explains a motivating example leading to our proposed fuzzer. This fuzzer is based on AFL and extends its implementation. 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 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.
AFL is the state-of-the-art in finding vulnerabilities and as it is amazing to be developed, the name of our fuzzer comes after *WAAAFL*!

In this chapter, we are contributing the following topics:

- An implementation of a fuzzer for finding the worst-case scenario in an algebraic problem.
- We use the **visitor** functions, which are not used in previous works, as we are

aware of.

- A new instrumentation for collecting runtime information about resource usages. In Waffle, we focus on maximising the number of instructions.

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

3.2 Motivating example

The number of effective instructions can affect on the time complexity of a program. To exemplify our problem, we pick a program that has a variety of different execution-times, based on the inputs we provide for the program.

Quicksort [35] is a well-known fast algorithm for sorting a list of numbers. This divide-and-conquer algorithm selects a pivot and finds the position of the pivot on the list. After the selection, the other numbers of the list are swapped, until all the numbers that are less than the pivot are on one side, and the rest are on the other side of the pivot. Then A quicksort is called on each side, and we continue until there is no more unknown position for the numbers in the final sorted list.

This algorithm has a best-case scenario with $\mathcal{O}(n \log n)$ for the time complexity, and the worst-case occurs happens in $\mathcal{O}(n^2)$. The worst scenario is when we select the pivot and all other numbers are not swapped; as a result, we have to try the remaining elements of the list before the selection of the next pivot. The best-case scenario occurs when the pivot splits the list into two partitions that the difference between the length of the partitions is less than or equal to one. The average time complexity is $\mathcal{O}(n \log n)$.

The following code is an implementation of **quicksort** in C language:

```c
#include<stdio.h>

void swap(int* a, int* b) {
  int t = *a; *a = *b; *b = t;
}

int partition (int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1); // Index of smaller element
  for (int j = low; j <= high- 1; j++)
    if (arr[j] <= pivot) {
      i++; // increment index of smaller element
      swap(&arr[i], &arr[j]);
    }

  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

void quickSort(int arr[], int low, int high){
  if (low < high) {
    int pi = partition(arr, low, high);

    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

void printArray(int arr[], int size) {
  int i;
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

// Driver program to test above functions
int main() {
  int arr[] = {10, 7, 8, 9, 1, 5};
  int n = sizeof(arr)/sizeof(arr[0]);
  quickSort(arr, 0, n-1);
  printf("Sorted array: \n");
  // printArray(arr, n);
  return 0;
}
```

Listing 3.1: Quicksort

We will consider testing the above code with Waffle in the next chapter.

3.3 Instrumentation

waffle-llvm-rt.o.c

We initialize the instrumentation with setting up the shared memory for Waffle (Listing 3.2):

```c
// snippet of wafl-llvm-rt.o.c

u8  __wafl_icnt_initial[ICNT_SIZE];
u8* __wafl_area_ptr = __wafl_icnt_initial;

static void __afl_map_shm(void) {

  u8 *id_str = getenv(SHM_ENV_VAR);

  if (id_str) {
    u32 shm_id = atoi(id_str);
    __wafl_area_ptr = shmat(shm_id, NULL, 0);

    if (__wafl_area_ptr == (void *)-1) _exit(1);

    memset(__wafl_area_ptr, 0, sizeof __wafl_icnt_ptr);
  }
}
```

Listing 3.2: LLVM instrumentation bootstrap

`__wafl_area_ptr` is the region that is allocated for counting the instructions, and is later shared when the instrumented program is running in fuzz testing.
The size of the bitmap `__wafl_icnt_ptr` is equal to $ICNT\_SIZE = 2^{16}$; the size of the bitmaps are equal in both AFL and Waffle.

wafl-llvm-pass.so.cc

Next, we inject our instrumentation into the program. As mentioned in the previous chapter, this stage requires the LLVM modules to analyze and insert the instructions. First, we define the **visitors**:

```cpp
struct CountAllVisitor : public InstVisitor<CountAllVisitor> {
  unsigned Count;
  CountAllVisitor() : Count(0) {}
  void visitMemCpyInst(MemCpyInst &I) { ++Count;}
};
```

We count the number of memory copies in an execution. The `CountAllVisitor` structure keeps track of the instructions that LLVM considers them as memory copies.

The highest number of instructions is a goal that Waffle follows, so that the new generations of the inputs are executed with more instructions. The hit-counts in the shared array starting from `__wafl_area_ptr`, are the number of times an edge is visited. Each time we visit an edge, we add the counted instructions to the appropriate index. The content of the array tracks the impact of edges in increasing the total number of instructions, and Waffle leverages these values to measures the influence of the changes in each index. [next section: Fuzzing]

The length of the shared memories are constant, but each time Waffle fuzzes a test-case, it needs to aggregate the content of the array and find the total number of instructions. Instead of calculating this total number in the fuzzing procedure, Waffle stores the total number of instructions in another shared memory region, `sys_data` [Listing 3.3]. Whenever the function `instr_AddInsts()` is called, the `MaxInstCount` is increased by the provided value. Later, when the program is finishing its execution, the shared memory containing the `MaxInstCount` is updated.

```
struct sys_data {
  int MaxInstCount;
};

static int MaxInstCount = 0;

void __attribute__((destructor)) traceEnd(void) {
  unsigned char *mem_str = getenv(MEM_ENV_VAR);

  if (mem_str) {
    unsigned int shm_mem_id = atoi(mem_str);
    struct sys_data *da;

    da = shmat(shm_mem_id, NULL, 0);
    if (da == (void *)-1) _exit(1);

    da->MaxInstCount = MaxInstCount;
  }
}

void instr_AddInsts(int cnt) {
  MaxInstCount += cnt;
}
```

Listing 3.3: sys_data in instrumentation

Now we can insert our instructions to the basicblocks:

```
// snippet of wafl-llvm-pass.so.cc
#include <math.h>
// ...
bool WAFLCoverage::runOnModule(Module &M) {
  // ...
  GlobalVariable *WAFLMapPtr =
    new GlobalVariable(M, PointerType::get(Int8Ty, 0), false,
    GlobalValue::ExternalLinkage, 0, "__wafl_area_ptr");

  llvm::FunctionType *icntIncrement =
    llvm::FunctionType::get(builder.getVoidTy(), icnt_args, false);
  llvm::Function *icnt_Increment =
    llvm::Function::Create(icntIncrement,
      llvm::Function::ExternalLinkage, "instr_AddInsts", &M);
  // ...
  for (auto &F : M) {
    for (auto &BB : F) {
      // ...
      LoadInst *IcntPtr = IRB.CreateLoad(WAFLMapPtr);
      MapPtr->setMetadata(M.getMDKindID("nosanitize"),
        MDNode::get(C, None));

      Value* EdgeId = IRB.CreateXor(PrevLocCasted, CurLoc);
      Value *IcntPtrIdx =
          IRB.CreateGEP(IcntPtr, EdgeId);

      /* Count the instructions */
      CountAllVisitor CAV;
      CAV.visit(BB);

      /* Setup the counter for storage */
      u8 log_count = (u8) log2(CAV.Count);
      Value *CNT = IRB.getInt8(log_count);

      LoadInst *IcntLoad = IRB.CreateLoad(IcntPtrIdx);
      Value *IcntIncr = IRB.CreateAdd(IcntLoad, CNT);

      IRB.CreateStore(IcntIncr, IcntPtrIdx)
        ->setMetadata(M.getMDKindID("nosanitize"),
        MDNode::get(C, None));

      IRB.CreateCall(icnt_Increment, ArrayRef<Value*>({ CNT }));

      inst_blocks++;
    }
  }
47 }
```

Listing 3.4: LLVM-mode instrumentation pass

In Line 6, we locate the shared bitmap. Line 19 loads the pointer to the bitmap and configures the meta-data for storage [36].

Same as AFL, Waffle stores the counters in the **hashed value** of the path we explored (Listing 2.4). The usage of the coverage-guided hashed values, helps Waffle in collecting the coverage information, and at the same time, maximizing the number of instructions in executions. Instructions in lines 23 to 25, load the pointer to the appropriate location on the bitmap.

In each basicblock, the visitors look for the **memory copies**. In different executions, we could see that the number of instructions increases rapidly, and to control this number, we calculate its log value (Lines 32-33):

$$CNT = \log_2^{CAV} \tag{3.1}$$

Now that we have calculated `CNT`, we load the pointer on the bitmap and add `CNT` to the content of the pointer and replaces it with the new summation. (Lines 36-39)

The last step in instrumenting a basic-block, is to call the `instr_AddInsts`. The command in line 42 calls the function with the previously set pointer, and sends `CNT` as the argument to this function.

Waffle applies these instrumentations in the compile-time, and when the program is being run, the performance and coverage information are set.

Run the instrumentation

To compile the target program with the adjusted instrumentation, we replace the C/C++ compilers (clang/clang++) with `waffle-clang`. "This program is a drop-in replacement for clang, similar in most respects to ../afl-gcc. It tries to figure out compilation mode, adds a bunch of flags, and then calls the real compiler." [37]

Except refactoring the filenames and the name of the variables, we did not apply any other modifications on `waffle-clang.c`.

Same as 2.2, we can start instrumentation by the command below:

```
./waffle-clang sample_vul.c -o sample_vul_waffle
```

Notice that we can insert compilation flags, such as `-fsanitize=address`, to enhance the performance of Waffle.

3.4 Fuzzing

Waffle extends the instrumentation and the fuzzing procedure of AFL. The instrumented binary from the previous stage is given to the *waffle-fuzz*, and Waffle develops the new features in *waffle-fuzz.c*.

To analyze the implementation of Waffle, we merge the algorithm 1 and algorithm 2:

Algorithm 3: $waffle - fuzz$

```
   Input: $queueEntry$
1  initialize fuzzer;
2  while fuzzing is not terminated do
3      cull_queue();
4      $T \leftarrow q.first\_entry()$;
5      $calibrate(T)$;
6      $trim(T)$;
7      $perf\_score \leftarrow calculate\_score(T)$;
8      if $deterministic\_mode$ then
9          $deterministic\_stages(T)$;
10     $MX \leftarrow calc\_stages(perf\_score, \mathcal{C})$;
11     foreach $element$ in $(0, MX)$ do
12         $random\_havoc(T)$;
```

The lines in red are the instructions that were modified in Waffle.

Calibration

Each entry of the queue is calibrated in order to collect the execution stats. Besides the execution-time and the bitmap-coverage information, Waffle runs the target program and collects the added **instruction counters**. Each instruction counter is monitored in the shared_memory, `icnt_bits[]`, and `top_rated_icnt[]` tracks the changes on `icnt_bits[]` in an `struct` of type `queue_entries`, .

Waffle collects the instrumentation's data after running the binary. The values of `icnt_bits` and `trace_bits` track the performance and the coverage-instrumentation. As described in Section Instrumentation, `sys_data` stores the sum of the instruction counters.

AFL checks the uniqueness of an execution by keeping a hashed value of `trace_bits[]`, representing a unique ID for a path-coverage. Within the discovery of a new coverage, Waffle analyzes both the *coverage* array and *instruction counters* to process the new hit-counts:

- `has_new_bits()`: AFL keeps the hit-counts of `trace_bits[]`, for identifying the new edges appeared in the execution of the current test-case. `trace_bits[]` is a sparse array, and AFL tracks the modified indices for better performance.

- `has_new_icnt()`: In addition to the previous method, Waffle searches for the highest values in `icnt_bits[]`. Each cell in `icnt_bits` contains 4 bytes of information. We saw in Listing 3.4 that in an index, such as `i`, `icnt_bits[i]` is added with `CNT`, while the increments in `trace_bits[]` are always by one. An analysis on `./sample_vul_instr` showed that the average value for `CNT`s is more than 3. As a result, the variance for the values in `icnt_bits[]` is higher than `trace_bits[]`'s. To decrease this variance and enhance the performance of Waffle, it leverages a constant float number, `MAX_CNT_MULT`:

```
static inline u8 has_new_max() {
  // #define MAX_CNT_MULT 1.05
```

```c
  int ret = 0;
  for (int i = 0; i < ICNT_SIZE; i++) {
    if (unlikely(icnt_bits[i]) && unlikely(icnt_bits[i] >
   MAX_CNT_MULT*max_icnts[i])) {
      if(likely(ret<2)) {
        ret = 2;
        max_icnts[i] = icnt_bits[i];
      }
      else ret = 1;
    }
  }
  return ret;
}
```

Listing 3.5: has_new_max()

By default, Waffle sets $MAX\_CNT\_MULT = 1.05$, which means it expects at least 5% increase for `icnt_bits[i]` before updating the content of index i.

The length of both `icnt_bits[]` and `trace_bits[]` is the same, but `icnt_bits` consumes 4x more of the memory:

```c
#define MAP_SIZE_POW2     16
#define MAP_SIZE          (1 << MAP_SIZE_POW2)
#define ICNT_SIZE         MAP_SIZE

EXP_ST u8* trace_bits;    /* SHM with coverage bitmap    */
EXP_ST u32* icnt_bits;    /* SHM with performance bitmap */
```

Listing 3.6: Configurations of the bitmaps

After analyzing the execution information, the calibration stage updates the undefined or old values of the current `queue_entry`.

Calculate performance score

The most favorable entries for Waffle, are the inputs with the highest `total_icnt`. In addition to the calculations of AFL, Waffle also compares the `q->total_icnt` with the average `total_icnt` of all previous entries, `avg_total_icnt`.

```c
static u32 calculate_score(struct queue_entry* q) {
  u32 perf_score = 100;
  u64 avg_total_icnt = total_icnt / total_bitmap_entries;
  // ...
  if (q->total_icnt > avg_total_icnt * 3) perf_score *= 1.4;
  else if(q->total_icnt > avg_total_icnt * 2) perf_score *= 1.2;
  else if(q->total_icnt > avg_total_icnt) perf_score *= 0.85;
  else if(q->total_icnt > avg_total_icnt * 0.75) perf_score *= 0.7;
  else perf_score *= 0.5;
  // ...
}
```

Listing 3.7: The modifications of Waffle in `calc_score()`

With all these modifications, Waffle can start it's fuzzing. The only difference between the execution of Waffle and AFL, is the name of the fuzzer.

```
# waffle-fuzz -i <in_dir> -o <out_dir> [options] -- /path/to/fuzzed/app [params]
waffle-fuzz -i in_dir -o out_dir -- ./sample_vul_waffle
```

Listing 3.8: Execute AFL

3.5 Concluding remarks

This chapter introduced the development of Waffle. Waffle extends AFL in two parts:

1. **Instrumentation**: **llvm_mode** directory is responsible for the instrumentation in Waffle. As we explained in this chapter, `llvm_mode/waffle-llvm-rt.o.c` contains the recipe for instrumenting the program in the compilation.

   AFL only keeps a coverage-bitmap, but in addition to the coverage-finding methodology, Waffle leverages the **visitor functions** in LLVM for assessing the more resource-exhaustive executions. Visitor functions count the targeted instructions, and Waffle saves the result in an extra *shared_memory*.

   Waffle counts the instructions in each basic-block, and reduces the counted values for saving into the memory. The size of the *shared_memory* has increased 4x in Waffle.

2. **Fuzzing**: Waffle uses the coverage bitmap and the instruction-counter bitmap for emphasizing the more beneficial fuzzing entries. The main changes are in the procedures of the functions `calibrate_case` and `calc_score()`. Generally speaking, an interesting test-case runs faster, has more code coverage, and executes more instructions from a specified set of instructions.

We also reviewed some of the modifications in the source code to Waffle's project. The project is located on github, in a public repository [38].