**CSCI 5271 Research Project**

**Implementing Machine Learning Techniques Into a Modern Fuzzer(AFL)**

**Ubuntu WSL Setup Tutorial**

**and AFL Information**

***Part One: Getting started with AFL(Ubuntu WSL)***

Here’s a step-by-step guide to get AFL (American Fuzzy Lop) up and running on your WSL Ubuntu machine.

#### **Step 1: Install Required Dependencies**

Before installing AFL, you need to install a few dependencies:

```bash

sudo apt update

sudo apt install build-essential clang llvm-dev libglib2.0-dev libpixman-1-dev

```

#### **Step 2: Download and Install AFL**

You can download AFL from its official repository:

```bash

git clone https://github.com/google/AFL.git

cd AFL

make

```

After compiling AFL, install it on your system:

```bash

sudo make install

```

You can check if AFL has been installed correctly by running:

```bash

afl-fuzz

```

This should display the usage message for AFL.

#### **Step 3: Compile Your Target with AFL Support**

AFL works by instrumenting programs to collect execution paths. You need to compile your target program with the AFL compiler afl-gcc or afl-clang-fast (recommended for better performance):

If you have a C/C++ program that you want to fuzz, compile it as follows:

```bash

afl-clang-fast -o target\_program target\_program.c

```

Alternatively, if you're dealing with an ELF binary, you can recompile it with AFL support (if you have the source code), or use a precompiled .elf for fuzzing.

#### **Step 4: Create Input and Output Directories**

AFL requires an input directory containing initial test cases (even a single input file will suffice). Create the input and output directories:

```bash

mkdir input\_dir output\_dir

echo "fuzz\_me" > input\_dir/test\_case

```

#### **Step 5: Start Fuzzing**

To start fuzzing, run the following command:

```bash

afl-fuzz -i input\_dir -o output\_dir -- ./target\_program @@

```

Here:

* -i input\_dir: The directory where AFL will find initial input files.
* -o output\_dir: The directory where AFL will save the fuzzing results.
* ./target\_program @@: The program to fuzz. The @@ gets replaced by AFL with the input file generated during fuzzing.

### **Additional Tips**

* **Crashes and Coverage**: AFL will report crashes or hangs during the fuzzing process. These can be found in the output directory (output\_dir/crashes or output\_dir/hangs).

**Minimizing Test Cases**: You can use afl-tmin to minimize the size of input files that trigger crashes:  
  
```bash  
afl-tmin -i crash\_file -o minimized\_file -- ./target\_program @@

```

**AFL GUI**: To visualize the fuzzing process, AFL comes with a simple UI:  
  
```bash  
AFL\_NO\_UI=0 afl-fuzz -i input\_dir -o output\_dir -- ./target\_program @@

```

By following these steps, you should be able to fuzz a program using AFL on your WSL Ubuntu environment.

***Part Two: Fuzzing an Open-Source Library(OpenSSL)***

### **1. Preparing Binaries for Fuzzing**

AFL fuzzes input by observing how inputs influence the execution paths of a target program. Therefore, the binary should be prepared and compiled in a way that AFL can instrument it. Here are a few key steps:

#### **a. Compile the Binary with AFL Instrumentation**

To fuzz a program using AFL, you must recompile the binary using AFL’s compiler wrappers (afl-gcc or afl-clang-fast). This ensures that the binary is instrumented and provides feedback to AFL during fuzzing. The instrumentation allows AFL to observe how different inputs affect the execution paths in your binary.

For example:

```bash

afl-clang-fast -o target\_program target\_program.c

```

This will produce an instrumented binary that AFL can effectively fuzz.

#### **b. Target Functions for Fuzzing**

AFL generally fuzzes the entire program, but to make it more focused, the program should expose a function that takes external input (from a file or standard input) and uses that input in a meaningful way (e.g., parsing, calculations, etc.).

A good target for fuzzing is any function that processes user-provided data. For example, if you are fuzzing a file parser, ensure that the entry function for parsing the file (e.g., parse\_file() or process\_input()) is reached by AFL.

### **2. Creating a Fuzz Target Function (For Custom Programs)**

If you’re writing your own program to be fuzzed, you should modify it to include a function that reads external input (from stdin or a file) and passes it to the function you want to fuzz. For example, in a simple C program, you can write something like:

```c

#include <stdio.h>

#include <stdlib.h>

void target\_function(const char \*input) {

// Function you'd like to fuzz

// Perform some operations with input

}

int main(int argc, char \*\*argv) {

FILE \*file = fopen(argv[1], "rb");

if (!file) {

perror("fopen");

return 1;

}

fseek(file, 0, SEEK\_END);

long fsize = ftell(file);

fseek(file, 0, SEEK\_SET);

char \*input = malloc(fsize + 1);

fread(input, 1, fsize, file);

fclose(file);

input[fsize] = 0;

target\_function(input); // Pass the input to the function being fuzzed

free(input);

return 0;

}

```

### **3. Fuzzing Precompiled ELF Binaries**

If you have a precompiled ELF binary and you don't have access to the source code, you can fuzz it using AFL's **QEMU mode**, which allows fuzzing without needing to recompile the binary. However, this method lacks the fine-grained instrumentation that compiling from source provides.

In QEMU mode, AFL hooks into the binary's execution via emulation, so it doesn't need the binary to be recompiled with AFL's instrumentation.

```bash

afl-fuzz -Q -i input\_dir -o output\_dir -- ./your\_binary @@

```

This will use the @@ placeholder to pass fuzzed input files into the program.

### **4. Fuzzing Setup Summary for Binaries**

#### **a. For Source Code:**

* **Use AFL's Instrumented Compiler**: afl-gcc or afl-clang-fast.
* **Target Function**: Ensure the binary reads from stdin or a file and passes the input to the target function you wish to fuzz.

**Compile the Program**: For example:  
```bash  
afl-clang-fast -o target\_program target\_program.c

#### **b. For Precompiled Binaries (ELF):**

**QEMU Mode**: If no source code is available, use QEMU mode for fuzzing without recompiling:  
bash  
```bash  
afl-fuzz -Q -i input\_dir -o output\_dir -- ./your\_binary @@

### **5. Testing Your Instrumented Binary**

You can check whether your program is instrumented correctly using afl-showmap. If the program is instrumented, you'll see feedback on coverage:

```bash

echo "test\_input" | afl-showmap -o map\_output.txt -- ./target\_program

```

This will output a file map\_output.txt, which contains execution trace information. If the file has content, the binary is properly instrumented.

### **6. Optional: Handling Complex Input with LLVM LibFuzzer**

If AFL isn’t suitable for your case (e.g., handling complex structured inputs), you can combine AFL with **LLVM's LibFuzzer** for fuzzing specific functions directly by linking the fuzzer to your target function.

In summary:

* **For fuzzable binaries**: Ensure they are compiled with AFL’s instrumentation, or use QEMU mode for precompiled ELF binaries.
* **Target functions**: Make sure the program or binary processes user input (from stdin or a file).
* **Testing**: Validate the instrumented binary using afl-showmap before starting the fuzzing process.

***Part Three: LibFuzzer vs AFL***

### **1. LibFuzzer (Target Function Required)**

In **LibFuzzer**, you specifically provide a target function—usually LLVMFuzzerTestOneInput(uint8\_t \*Data, size\_t Size)—which receives the fuzzed input directly. This is mandatory in LibFuzzer, as it's structured to test functions directly by passing the input to that function.

For example:

c

```c

extern "C" int LLVMFuzzerTestOneInput(const uint8\_t \*data, size\_t size) {

target\_function(data, size);

return 0;

}

```

LibFuzzer works by repeatedly calling this target function with random fuzz data, thus directly injecting fuzz data into a specific entry point.

### **2. AFL (No Explicit Target Function Required)**

**AFL (American Fuzzy Lop)**, on the other hand, operates differently from LibFuzzer. It doesn't require an explicit target\_function like LLVMFuzzerTestOneInput. Instead, AFL works by fuzzing the **entire program** by providing random input through external interfaces, such as **command-line arguments**, **standard input (stdin)**, or **file input**.

#### **How AFL Injects Random Data**

**File Input**: AFL typically provides random input through files. If your program expects a file as input, AFL will generate test files during fuzzing and pass them to the program as arguments.  
Example:  
bash  
```bash  
afl-fuzz -i input\_dir -o output\_dir -- ./target\_program @@

```

Here, @@ is replaced by AFL with the path of the test file it generates during fuzzing.

**Standard Input (stdin)**: If your program reads from stdin, AFL can directly inject fuzzed data into the program. AFL feeds random input by piping fuzzed data into the program’s stdin:  
Example:  
bash  
```bash  
afl-fuzz -i input\_dir -o output\_dir -- ./target\_program

```

#### **No Mandatory Target Function in AFL**

In AFL, there’s **no need for a specific target function like LLVMFuzzerTestOneInput** because AFL fuzzes the entire program. AFL instruments the program at compile time (with afl-gcc or afl-clang-fast) and observes which parts of the code are executed when fuzzed input is provided. It uses this feedback to guide further mutations in the input.

#### **How AFL "Finds" the Fuzzing Entry Point**

1. **Main Function**: Typically, fuzzed data enters the program through its main() function. For instance, if your program expects a file as an argument or reads from stdin, AFL passes the fuzzed input through those interfaces.
2. **File Reading**: AFL will generate test files and pass their paths to your program via the command line if your program expects file input.
3. **Piping Input**: If your program reads from stdin, AFL injects fuzzed data through stdin as it runs.

### **3. Comparing Fuzzing Approaches: AFL vs. LibFuzzer**

* **LibFuzzer**: Requires explicit target function (e.g., LLVMFuzzerTestOneInput) that receives fuzzed input. It directly fuzzes functions within a program.
* **AFL**: Does not require an explicit target function. Instead, it fuzzes the entire program by providing fuzzed input via files, command-line arguments, or stdin. AFL instruments the program during compilation to observe execution paths and guide the fuzzing process.

### **4. Making Programs Fuzzable by AFL**

If you're developing a program that you'd like to fuzz with AFL, ensure:

* **Input Handling**: Your program should handle input in a way that AFL can influence. For example:
  + Reading input from files.
  + Reading input from stdin.
  + Accepting command-line arguments.

Example of a simple program to fuzz with AFL (file input):

```c

#include <stdio.h>

#include <stdlib.h>

void target\_function(const char \*input) {

// This function processes the fuzzed input

}

int main(int argc, char \*\*argv) {

if (argc < 2) {

fprintf(stderr, "Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE \*file = fopen(argv[1], "rb");

if (!file) {

perror("fopen");

return 1;

}

fseek(file, 0, SEEK\_END);

long fsize = ftell(file);

fseek(file, 0, SEEK\_SET);

char \*input = malloc(fsize + 1);

fread(input, 1, fsize, file);

fclose(file);

input[fsize] = 0;

target\_function(input); // Pass the input to the function being fuzzed

free(input);

return 0;

}

```

* **Instrumentation**: When you compile the program with AFL’s compiler (afl-gcc or afl-clang-fast), the binary will be instrumented to allow AFL to track its execution paths.

Compile it with AFL:

```bash

afl-clang-fast -o target\_program target\_program.c

```

Then, you can run AFL:

```bash

afl-fuzz -i input\_dir -o output\_dir -- ./target\_program @@

```

### **Summary**

* **LibFuzzer** requires a specific target function (LLVMFuzzerTestOneInput) to fuzz individual functions directly.
* **AFL** fuzzes the entire program and does not require an explicit target function. It injects fuzzed data via stdin, file input, or command-line arguments.
* AFL is designed to be more "black-box" and works by observing how different inputs influence the overall execution path of the program.

To fuzz with AFL, simply ensure your program handles input in a way AFL can fuzz (e.g., files or stdin), and AFL will take care of injecting fuzz data and monitoring program execution.

### ***Part Four: Fuzzing an Open-Source Library(OpenSSL)***

### **Steps to Fuzz an Older Version of OpenSSL with AFL**

#### **1. Download the Older Version of OpenSSL**

Choose the version of OpenSSL you want to fuzz. You can find older releases of OpenSSL here. For example:

```bash

wget https://www.openssl.org/source/old/1.0.1/openssl-1.0.1.tar.gz

tar -xvzf openssl-1.0.1.tar.gz

cd openssl-1.0.1

```

#### **2. Prepare OpenSSL for Fuzzing**

You need to compile OpenSSL with AFL’s instrumentation. This means using AFL’s compiler wrappers (afl-gcc or afl-clang-fast) instead of regular gcc or clang. Here's how to do that:

##### **a. Configure OpenSSL with AFL Instrumentation**

Set the AFL compiler wrappers as the CC (C compiler) and CXX (C++ compiler) before configuring OpenSSL:

```bash

export CC=afl-clang-fast

export CXX=afl-clang-fast++

```

Now, configure and build OpenSSL:

```bash

./config

make

```

This will build an instrumented version of OpenSSL that can be fuzzed with AFL.

#### **3. Create a Target Program for Fuzzing**

OpenSSL has many functions that could be fuzzed (e.g., functions for parsing certificates, handling SSL connections, etc.). You need to create a small "harness" program that calls one of these functions and accepts input (usually through a file or stdin). Here’s an example of a basic harness for fuzzing OpenSSL’s PEM\_read\_bio\_X509() function, which reads and parses X509 certificates from a PEM file:

```c

#include <openssl/pem.h>

#include <openssl/x509.h>

#include <openssl/bio.h>

int main(int argc, char \*argv[]) {

if (argc < 2) {

fprintf(stderr, "Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE \*file = fopen(argv[1], "rb");

if (!file) {

perror("fopen");

return 1;

}

BIO \*bio = BIO\_new\_fp(file, BIO\_NOCLOSE);

if (!bio) {

fprintf(stderr, "Error creating BIO\n");

fclose(file);

return 1;

}

X509 \*x509 = PEM\_read\_bio\_X509(bio, NULL, 0, NULL);

if (x509) {

X509\_free(x509);

}

BIO\_free(bio);

fclose(file);

return 0;

}

```

This program takes a file as input (which AFL will fuzz) and attempts to read it as a PEM-encoded X509 certificate.

##### **b. Compile the Fuzzing Harness with AFL**

You need to compile the harness program with AFL's instrumentation:

```bash

afl-clang-fast -o openssl\_fuzz\_harness openssl\_fuzz\_harness.c -lssl -lcrypto

```

This links it against the instrumented OpenSSL libraries, so AFL can now fuzz the PEM\_read\_bio\_X509() function.

#### **4. Set Up the Input and Output Directories for AFL**

Create directories for the input seeds and AFL's output:

```bash

mkdir input\_dir output\_dir

```

Place a valid PEM-encoded X509 certificate in the input\_dir to serve as a seed file for AFL. This will give AFL a starting point for generating new test cases. If you don’t have one, you can easily generate one using OpenSSL:

```bash

openssl req -new -x509 -keyout key.pem -out cert.pem -days 365 -nodes

mv cert.pem input\_dir/

```

#### **5. Run AFL to Fuzz OpenSSL**

Now that everything is set up, you can start fuzzing with AFL:

```bash

afl-fuzz -i input\_dir -o output\_dir -- ./openssl\_fuzz\_harness @@

```

* -i input\_dir: The directory with the seed files (starting inputs).
* -o output\_dir: The directory where AFL will store its results (crashes, hangs, etc.).
* @@: This is replaced by the fuzzed input file generated by AFL.

AFL will now start fuzzing OpenSSL, trying different inputs to find crashes.

#### **6. Monitor Coverage and Crashes**

##### **a. View Coverage Information**

AFL automatically tracks coverage by observing which paths in the code are hit by different inputs. To see this information, use the following:

```bash

cat output\_dir/fuzzer\_stats

```

This will show AFL’s statistics, including how many unique paths have been discovered.

##### **b. Check for Crashes**

AFL stores all crashing inputs in the output\_dir/crashes/ directory. You can view the crashing files and reproduce the crash by running the harness with those inputs:

```bash

./openssl\_fuzz\_harness output\_dir/crashes/id:000000,sig:11,src:000000,op:havoc,rep:64

```

#### **7. Optional: Use afl-cov for Code Coverage Analysis**

If you want more detailed code coverage information, you can use afl-cov, a tool that provides fine-grained code coverage reports for AFL fuzzing sessions.

To use afl-cov, install it and then run it as follows (assuming you have AFL results):

```bash

afl-cov --coverage-cmd "./openssl\_fuzz\_harness @@"

```

This will show detailed coverage information, including which lines of code are being hit during fuzzing.

### **Summary**

1. **Set up AFL**: Ensure AFL is installed and ready.
2. **Download OpenSSL**: Download and compile the older version of OpenSSL with AFL’s compiler wrappers (afl-clang-fast).
3. **Create a Fuzzing Harness**: Write a small program that reads fuzzed input and calls an OpenSSL function (e.g., PEM\_read\_bio\_X509()).
4. **Set up AFL**: Create input and output directories, and provide AFL with a valid input seed.
5. **Run AFL**: Start fuzzing OpenSSL with AFL using the harness and monitor the fuzzing session.
6. **Monitor Coverage and Crashes**: Check AFL’s output for crash reports and debug them with tools like gdb.

With this setup, you should be able to fuzz an older version of OpenSSL and gather coverage and crash data to explore potential vulnerabilities.

### ***Part Five: Changing AFL’s Input Mutation Logic:***

### **Key Components of AFL for Mutation and Feedback**

1. **Mutation Logic**
   * AFL’s input mutation logic is primarily located in the file **afl-fuzz.c**.
   * The function you want to focus on is **fuzz\_one()**, which drives the core fuzzing loop for a single input (i.e., the iteration loop for fuzzing a seed).
   * The actual mutations are handled within **fuzz\_one()** using helper functions such as:
     + **common\_fuzz\_stuff()**: Passes mutated input to the target program for execution.
     + **queue\_testcase()**: Prepares and schedules the next input for mutation.
     + **calibrate\_case()**: Tracks coverage of the new test cases.
2. **Mutation Operations**
   * AFL applies mutations using various **mutation strategies** defined in the functions under afl-fuzz.c.
   * These mutations include operations like bit flips, arithmetic changes, dictionary-based substitutions, etc. You’ll find the mutation strategies inside **perform\_dry\_run()** and **fuzz\_one()**.
   * Functions for specific mutations:
     + **flip\_bit()**: Mutates a single bit in the input.
     + **arith\_add()**: Adds a number to a byte in the input.
     + **havoc()**: The general-purpose random mutation strategy that applies a mix of mutation techniques.
3. **Coverage Feedback**
   * AFL uses **edge coverage** to guide the fuzzing process. The coverage information is tracked and used to determine whether an input is interesting enough to be added to the queue.
   * Look at the function **has\_new\_bits()** in afl-fuzz.c, which checks whether the execution of a mutated input leads to new code coverage.
   * **update\_bitmap\_score()** tracks new coverage and determines whether a particular input has found new execution paths, making it eligible for re-fuzzing.
4. **Queue Management (Seeds)**
   * AFL maintains a queue of inputs (seeds) to fuzz, and this is managed in the function **add\_to\_queue()** in afl-fuzz.c. This is where you can modify how new seeds are added based on your custom logic.
   * Your custom logic for input gain or seed evaluation can be integrated here, so inputs with higher coverage or meeting specific criteria (e.g., crashes) can be appended to the queue.
5. **Execution and Results**
   * AFL executes the target program on fuzzed inputs using the function **run\_target()**, also found in afl-fuzz.c. This is where AFL checks for crashes, timeouts, or hangs.
   * You may also want to review **save\_if\_interesting()**, which determines if the input caused a crash or triggered new coverage and should be saved.

### **Steps to Implement Custom Mutation Logic (Machine Learning Integration)**

#### **1. Custom Mutation Strategy**

You can modify the **havoc()** mutation strategy (or write your own) to integrate machine learning-based mutations. For example, you can replace random mutations with more guided mutations based on ML models or heuristics.

You’ll find havoc() around the line 2072 of afl-fuzz.c:

```c

static void havoc\_stage(...) {

// Here, you can integrate ML-based mutations or modify random byte selections

for (i = 0; i < havoc\_max\_mult; i++) {

switch (random\_action) {

case 0: flip\_bit(...); break;

case 1: arith\_add(...); break;

case 2: ml\_guided\_mutation(...); break;

// Add your custom logic here

}

}

}

```

#### **2. Modify Seed Queue and Coverage Feedback**

Incorporate your custom coverage gain logic by modifying **has\_new\_bits()** and **add\_to\_queue()** to implement your own conditions for adding new seeds based on machine learning predictions or heuristics.

#### **3. Tracking Crashes**

Customize **save\_if\_interesting()** to handle the logic when a crash occurs and append the crashing input to your custom malicious inputs list.

### **Functions and Files to Focus On**

* **fuzz\_one()** (afl-fuzz.c): The main fuzzing loop.
* **havoc()** (afl-fuzz.c): AFL’s primary mutation stage.
* **flip\_bit(), arith\_add()** (afl-fuzz.c): Specific mutation techniques.
* **has\_new\_bits()** (afl-fuzz.c): Coverage feedback logic.
* **add\_to\_queue()** (afl-fuzz.c): Logic for appending inputs to the seed queue.
* **run\_target()** (afl-fuzz.c): Executes the target binary on a fuzzed input.
* **save\_if\_interesting()** (afl-fuzz.c): Determines if an input found new behavior or crashes.

***End of Tutorial***

ChatGPT was used in the creation of this document

Link to conversation:

<https://chatgpt.com/share/6716e794-d8f8-800f-aa0f-49092328b266>