Automatic Non-Conventional Fuzzing

Ryan Xu
McKelvey School of Engineering
Washington University in St. Louis
St. Louis, United States
ryanxu@wustl.edu

Zixuan Li
McKelvey School of Engineering
Washington University in St. Louis
St. Louis, United States
li.z@wustl.edu

***Abstract*—In this report, we introduce DeepFuzzer, a python script that automatically generates ready-to-fuzz binaries given a library. The script also has the capability to allow users to select specific functions. DeepFuzzer was evaluated on a number of real-world libraries with an acceptable success rate. We recognize that there is still a lot of work to be done in automatic non-conventional fuzzing, so a few points of future work will be mentioned.**

I. Introduction

AFL, the de-facto fuzzer, utilizes the command line to run. It is a file based fuzzer which feeds input to program via standard input. It performs extremely well against traditional C binaries, but a problem arises when trying to fuzz non-conventional programs such as libraries or networks, to name a couple. It's hard to fuzz libraries since many functions take multiple arguments, some of which are complicated structures, and many functions have dependencies across the library. In addition, AFL cannot pick a specific function within a library that may be of interest to fuzz individually. Fuzzing each functions individually can achieve overall higher code coverage. A network server or client usually reads input from sockets or other network interfaces. AFL is unable to take a packet as input from the command line [1].

To remedy this, we have designed and written DeepFuzzer. An automatic input wrapper for AFL. DeepFuzzer takes three inputs: the source code file, the header file directory path, and library instrumented by AFL. It first scans the source code looking for functions and check whether functions are compatible with DeepFuzzer. Then it prompts the user for if the user wishes to fuzz a specific function. Next, it generates the harness automatically and try to compile with the instrumented library. We've utilized DeepFuzzer on the following libraries: SELA, an audio-related library, PortAudio, also an audio-related, AESCrypt, an crypto library . As far as we know, the work we are doing, though primitive, is novel.

The code can be found in the following repository: https://github.com/z1Tion/DeepFuzzer.

II. Methodology

We chose to use Python because both members are familiar with the language and thought it would be the easiest to use.

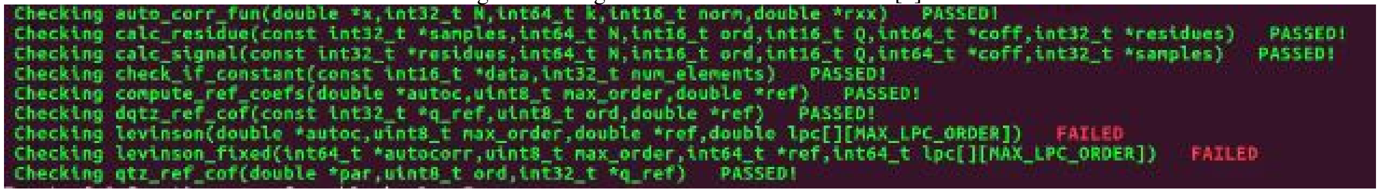
Before DeepFuzzer can be used, the target library must first be compiled with AFL [1]. To run the program, the main script must be given three arguments - the source code path, the compiled binary path, and the include directory path. The workflow of DeepFuzzer can be summarized to the following steps:

1) Generate a list of functions in the source code path
2) Parse functions for parameters and dependencies
3) Analyzing parameters type
4) Assign value to target function parameters
5) Compile with the instrumented library

A. Listing Functions

The majority of code in this step is found in list_function.py. From the given source code, it builds a LibraryInfo object that contains a list of functions, a list of passed functions, and what dependencies are needed. DeepFuzzer is still in its early stages so some complex data structures are not

Fig. 1. Listing of functions of a SELA file [2]



```
Checking auto_corr_fun(double *x,int32_t N,int64_t k,int16_t norm,double *rxx)   PASSED!
Checking calc_residue(const int32_t *samples,int64_t N,int16_t ord,int16_t Q,int64_t *coff,int32_t *residues)   PASSED!
Checking calc_signal(const int32_t *residues,int64_t N,int16_t ord,int16_t Q,int64_t *coff,int32_t *samples)   PASSED!
Checking check_if_constant(const int16_t *data,int32_t num_elements)   PASSED!
Checking compute_ref_coefs(double *autoc,uint8_t max_order,double *ref)   PASSED!
Checking dqtz_ref_cof(const int32_t *q_ref,uint8_t ord,double *ref)   PASSED!
Checking levinson(double *autoc,uint8_t max_order,double *ref,double lpc[][MAX_LPC_ORDER])   FAILED
Checking levinson_fixed(int64_t *autocorr,uint8_t max_order,int64_t *ref,int64_t lpc[][MAX_LPC_ORDER])   FAILED
Checking qtz_ref_cof(double *par,uint8_t ord,int32_t *q_ref)   PASSED!
```

yet able to be handled, so those functions that are not able to be utilized are absent from the list of passed functions.

To get the list of functions, we utilized a tool called cproto, an extraction based on GCC compiler. The function goes through the source code and grabs the name and type of the function as well as the parameters. We store the output into the function_list variable of the LibraryInfo object. Next, for each function cproto finds, it is parsed using standard python string slicing tools and an object called FnInfo is created. The FnInfo object inherits metadata such as the header directory, includes folder, and source directory previously specified on input. In addition, the source code is also scanned for any instances of ”include” and whatever is imported is added to a list. Finally, each function is parsed for its parameter inputs.

B. Analyzing parameter type

After basic parameters inputs been parsed, more complicated analysis is needed to examine whether this function can be fuzzed by DeepFuzzer or not. For a single variable, we need to extract its type information, name, pointer number and length of array if necessary. Then we compare its type information with regular type list. If it is not in the list, we treat it as a structure. Then we utilize LLVMs Clang-Check tool to automatically parse the file and lists out the structure composition, and store related information into the variable information table [3]. For our prototype, a function containing a double pointer as one of the parameters or one with a self-defined nested structure will be treated as unfuzzable function. A check fail information pop up will appear during the process.

C. Picking Functions

After all the functions have been examined, they are listed in the console with a tag that indicates if the function can be used or not.

In Figure 1, the list of functions in a SELA file can be seen. Two functions have the failed tag because their parameters were to complex for DeepFuzzer to handle at this time. However, the rest of the functions passed and are eligible to be used to write fuzzable binaries.

The user will then be prompted for input. The user can either specify a specific function (as long as it passes) or can simply press enter if he or she wishes to fuzz all the functions.

D. Interface Generation

After all the necessary parameters and structure compositions have been extracted, it is time to generate the harness. The harness consume file which contains a byte stream generated by AFL, and our work is based on information we extracted before to build every variable object the target function required and pass them to target functions.

There are some important steps to generate harness. To build a useful harness for testing specific target function, we need first to ensure the harness code is bug free. Since our harness needs to frequently read from byte stream file, this operation may result some bugs. Thus, we want to make sure every time we read a variable from byte stream, the byte stream is still long enough. For example, if we want to read a integer, we want the byte stream at least have 2 bytes remaining (on 32 bit machine). So we need to calculate the size of variable type and compared it with remaining byte stream length before we do the actual reading.

A question arises when we need to read a pointer from byte stream: how many bytes we need for a pointer. For passing a pointer to a function, we need to build the object what this pointer points to. Take a integer pointer as example, we need to build an integer array first. So how to generate an array from a byte stream. Our first idea is to read other

TABLE I
FUNCTIONS UNABLE TO BE USED BY DEEPFUZZER

Function	File	Library
void *aesrandom_open()	aesrandom.c	AESCrypt
int32_t initialize_ao()	ao_output.c	SELA
void PaUtil_TerminateBufferProcessor(PaUtilBufferProcessor* bp)	pa_process.c	portaudio
void PaUtil_DebugPrint(const char *format, ...)	pa_debugprint.c	portaudio
PaError Pa_Terminate(void)	pa_front.c	portaudio
const char *Pa_GetErrorText(PaError errorCode)	pa_front.c	portaudio

types first and whatever left becoming the array. However, this method won't work when we need to construct multiple arrays. Our second idea is to split the remaining byte stream equally. While it might work for other input wrappers, unfortunately it will not work for AFL since it will reduce the input case AFL can generate. The third idea is to use a random number generator where we randomly generate the length of array to avoid this problem. Again, it might work for other input wrapper but still not work for AFL. Because AFL is a coverage guided fuzzer, if given a specific input the program generates different output on each execution, it means the process goes through a different code path, which makes coverage guided meaningless. So we need the a length that is deterministic given a byte stream. Then we realize we don't need a random generator at all since byte stream itself is essentially a random number generator. All we need is read from byte stream one more time to determine the array length.

Then how to read a structure from byte stream? Our approach is to build its field first and then instantiate the structure. In our prototype, we don't deal with nested structure which makes all structure fields are regular type. We just used method above again to read regular type variable and pointer variable from byte stream.

III. EVALUATION

We evaluated DeepFuzzer on a number of real-world libraries. The libraries are the following:

1) SELA - a loss-less audio library [2]
2) AES Crypt - a file-encryption library [4]
3) PortAudio - a real-time audio input and output library [5]

We chose three easily compiled library written in C to evaluate our work. These libraries were chosen to provide a diverse mix of complex and simple library implementations. In addition, these libraries concern real-world systems that could be of interest in the future as cyber-physical system security develops. To test DeepFuzzer on these libraries, each one was compiled using afl-gcc and the include folder and the compiled library binary was located [1]. The results are summarized in Table 2.

It can be seen that for some libraries such as SELA and AES Crypt, DeepFuzzer can be utilized on the majority of functions within the library. However, disappointingly, the same cannot be said for PortAudio. A closer examination of PortAudio reveals that most methods utilize a complex structure which DeepFuzzer cannot handle at this current moment. Some examples of unfuzzable functions can be found in Table 1. Unfuzzable functions in SELA and AES Crypt and the first one in portaudio are functions doesn't take any argument. Second function in portaudio contained a complicated nested structure. Third one takes uncertain numbers of arguments. And last one takes an enumerator type which can not recognized by our prototype. A more in-depth look at the limitations of DeepFuzzer can be found in Section 4: Difficulties and Future Work.

IV. DIFFICULTIES AND FUTURE WORK

A. Header File Absence

In order to feed a target function with a structure argument, the necessary header file should be included in our generated harness. Our approach to deal with this problem is to copy all 'include' statements from target function source code file to our harness. However, when we test our approach with

TABLE II
EVALUATING DEEPFUZZER ON SELECT LIBRARIES

Library	% of Functions That Can Be Used
SELA	97%
AES Crypt	92%
PortAudio	0%

a more complicated library, some of our generated harnesses cannot be successfully compiled with the library and it is the result from some missing header file.

When we look deeper in those failure cases, we found two possible reasons. The first one is we are calling an a function we are not supposed to call. So there is no corresponding header file in include folder. A possible solution is that we can dump the compiled library and checking whether this function is in compiled library or not.

Another reason is when include statement is in the middle of ifdef(ifndef) endif statement. In this situation, unnecessary header file name may be include in our harness. In addition, if ifdef(ifndef) endif statement is using a include guard, it is possible that we include a same header file multiple time which can result compilation failure. For this possible situation, we do not have any solution right now and manual work is necessary.

B. Stream Length Comparison

When dealing with the pointer like integer pointer, we first read one byte from the bitstream and use this byte as the length of array. This methodology insures the length of array is changing with the byte stream and is also deterministic. However, there is also a problem using this technique. The total bytes we need from stream is not fixed every time. Based on our design, we need to check the bitstream size every time before we read certain type of data. When a target function is complicated enough, the total checking time will increase significantly which could result in low efficiency fuzzing. A potential solution is that we preprocess target function - first find out how many bytes we need read to determine the minimum stream size. Then we can calculate the minimum stream size it required and compare it with actual stream size to decide whether it is enough or not.

C. Nested Structure Analysis

Data types in C code are complicated. It includes primary data types like integer, char, double, and derived data types such as structure, union, and pointer. Things can be really complicated when we are dealing with structures. One of the major difficultly we encounter with is the nested structure. We do not deal with nested structure in our current work, but we propose some potential solutions. The nested structure refers to a structure have one or multiple fields being a structure or a pointer points to a structure. When nested structure have field(s) being structure, we can utilizing AST provided by clang parser to instantiate all child structure to eventually instantiate structure consumed by target function. When nested structure have field(s) being a pointer which points to a structure, instantiating all child structure is not feasible, since pointer can be recursive. For example struct A has a pointer which points to struct B and struct B has a pointer which points to struct A. To solve this problem, we can read one byte from byte stream and using its value as maximum instantiate number. If the instantiate times exceed our maximum instantiate number, the last pointer will be assigned to NULL to terminate endless instantiating.

D. Variable Name Collision

During our process to generate argument of specific type from byte stream, we need to declare some intermediate variables. For example, if a function takes an integer pointer as argument, we need to first assign value to an integer array (or integer), and then pass address of this array (or integer) to function. Many variables need to be used to successfully construct a complicated data structure and if they are named poorly, there is a possibility where two variables are using the same name. This problem is referred to as variable name collision.

Variable name collision in simple data structure can be simply solved by prepending some prefix before or appending some suffix after variable name. Using this method we can also separate the variable declaration and usage, since we can deduce variable name based on our naming rules. However, in complicated data structures such as nested structure, the variable name collision problem is very problematic. It is possible that exact same structure need to be instantiated multiple times.

There are two possible solutions to this problem. The first one is better naming rules. We can use more factors to determine variable name. For example, we can perform analysis to determine if we need this variable to build a bigger structure or it itself is the structure the target function takes and name it based on the analysis. Another way is that we give all variables a random name and use a global hash map to record it. For example, we can do preprocessing to analysis how many intermediate variables we need to use, when should they be declared and when should they be used.

E. Integration with Clang Front End

Based on experience in this project, writing a parser by ourselves is not feasible. Though we can deal with simple data structure using regexp, we cannot extract information from complicated stricture like nested structure correctly. That is the reason we using cproto, a library based on GCC compiler and clang-check, a tool based on clang compiler to parse our function prototype and structure field. However, integrating with those tools are not easy. Part of the reason is that we are using python as our developing language. Because those parsers have no python interface, though we are using well developed parser, we still use python string operation to extract information from parser output which is still inaccurate. To better integrate with those parser, we should redesign and rewrite our problem in c++ based on clang c++ interface. And most problem and difficulties we have encounter can be eliminated.

V. Detailed Instruction of DeepFuzzer

In this part, we will present how to use DeepFuzzer by fuzzing a real world example, SELA. The following was performed on an Ubuntu 16.04 machine.

```
# update and upgrade apt source
sudo apt-get update
sudo apt-get upgrade
# Install Require Library
sudo apt-get install build-essential git
sudo apt-get install clang
sudo apt-get install clang-format
sudo apt-get install cproto

# Install AFL
wget http://lcamtuf.coredump.cx/afl/
    releases/afl-latest.tgz
tar xvf afl-latest.tgz
rm afl-latest.tgz
cd afl-2.52b/ && make
sudo make install
cd ..

# Get Our Target SELA from GitHub
git clone https://github.com/sahaRatul/
    sela.git

# Compile it with afl-gcc(or afl-clang)
cd sela
export CC=afl-gcc
make all -e
export CC=gcc

# Pack .o file into .a static library
ar rc libsela.a *.o
cd ..

# Get DeepFuzzer
git clone https://github.com/z1Tion/
    DeepFuzzer.git

# Check Prerequisite
cd DeepFuzzer
sh test.sh

# Run DeepFuzzer
python3 main.py ../sela/core/apev2.c ../
    sela/include/ ../sela/libsela.a

# Generated harness source code is in out
    /src and executable is in out/bin

# Lets Fuzz!
cd out/bin
mkdir input output
echo 'fuzz' > input/input.txt
afl-fuzz -i input -o output ./
    init_apev2_fuzz @@
```

The following is the resulting harness for apev2.c from the SELA library:

```
/*
* Generate by Deepfuzzer
* Target Function: init_apev2(apev2_state
    *state)
* Time: Mon May  6 17:46:38 CDT 2019
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include "wavutils.h"
#include "apev2.h"

#include <inttypes.h>

int main(int argc, char **argv) {
    FILE *infile = fopen(argv[1], "rb");

    fseek(infile, 0, SEEK_END);
    int fileSize = (int)ftell(infile);
    int minSize = 0;
    rewind(infile);

    int pointer_size_state = 1;
    if (fileSize < minSize + sizeof(
        uint16_t) * pointer_size_state) {
        fclose(infile);
        return 0;
    }
    uint16_t d1_state;
    fread(&d1_state, sizeof(uint16_t), 1,
         infile);
    if (fileSize < minSize + sizeof(
        uint16_t) * pointer_size_state +
                        sizeof(int) *
                            d1_state) {
        fclose(infile);
        return 0;
    }
    int reference_state[d1_state];
    for (long int i = 0; i < d1_state; ++
        i) {
        int tmp_state;
        fread(&tmp_state, sizeof(int), 1,
             infile);
        reference_state[i] = tmp_state;
    }
    int *state = reference_state;

    init_apev2(state);

    printf("Test Passed!\n");

    return 0;
}
```

VI. Conclusion

In this paper, we design and implemented DeepFuzzer, an automatic harness writer to generate ready to fuzz executable. We test and evaluate our work on three real world target. The result demonstrate our tool have good performance on no nested-structure targets, but poor performance on complicated function prototype. Then we analyze the drawback of our work and propose some potential solutions.

References

[1] AFL https://github.com/jdbirdwell/afl
[2] Sela https://github.com/sahaRatul/sela
[3] Clang-Check https://clang.llvm.org/docs/
[4] PortAudio http://www.portaudio.com/
[5] AES Crypt https://github.com/paulej/AESCrypt