SWEN90006 Group Assignment Report

Group 15

1. Team Members and Responsibilities

Team Member	Responsibilities
Zijian Ju	Vulnerability design, implementation and improvement, discussion on Fuzzer
Ruofan Zhang	Designed, implemented and improved the fuzzer
Yuhan Jiang	Evaluation Design, Experiments implementation and Discussion
Lihua Wang	Evaluation Results Analysis, Libfuzzer Experiments Implementation

2. Vulnerabilities

2.1 Vulnerability 1

Line	**475**
Modified	// Original: // if (**stack_size() < 2**) if (**stack_size() == 0**){ debug_printf("Add from insufficient stack\n"); return -1; }

Table 1. Vulnerability 1

Vulnerability 1 changes the condition for checking insufficient stack for ADD command. It will trigger a Read memory access error on AddressSanitizer by accessing -1 index of the stack array in the following two stack_pop() methods when there is only one element on the stack.

It is a reasonable vulnerability as any time the user input an ADD command with only one element in stack will be able to trigger it. This kind of vulnerability could happen when programmers forget to implement the defensive programming code or misunderstand potential invalid cases, which is only thinking of preventing empty stack in this case.

2.2 Vulnerability 2

Line	**431**
Modified	static int execute(void){ char * toks[4]; /* these are pointers to start of different tokens */ //Original: const unsigned int numToks = tokenise(**Inst,toks,4**); const unsigned int numToks = tokenise**(inst,toks,5);**

Table 2. Vulnerability 2

Vulnerability 2 will trigger a stack buffer overflow error when the number of input arguments is over 4. The *tokenise* function will keep splitting the input string by whitespaces and write it into tokens buffer until there is no argument remaining or the number of tokens has reached the upper bound. Rewriting the number of upper bound of *tokenise* function to a larger one will cause buffer overflow by writing tokens array at index 4.

It is a reasonable vulnerability since any input contains more than 4 arguments will trigger it. The reason why I put this is that both the size of *toks* array and the input of the *tokenise* function are magic numbers. They

have to be identical to prevent buffer overflow mentioned above. This may cause a common mistake that programmers may forget to update one of them when the behavior of the system changes.

2.3 Vulnerability 3

Line	**623**
Modified	if (!(inst[MAX_LINE_LENGTH] == '\n' && inst[MAX_LINE_LENGTH+1] == '\0')){fprintf(stderr,"Line %d exceeds maximum length (%d)\n",instructionCount+1,MAX_LINE_LENGTH); debug_printf("(Expected at array index %d to find NUL but found '%c' (%d))\n",MAX_LINE_LENGTH,inst[MAX_LINE_LENGTH], inst[MAX_LINE_LENGTH]); return -1; } **/ Insert** **else { inst[MAX_LINE_LENGTH*10] = '1'; }**

Table 3: Vulnerability 3

Vulnerability 3 will trigger an unknown address WRITE memory access on AddressSanitizer. If the programmer wants to do something when the length of input string is exactly equal to MAX_LINE_LENGTH + 2 (including '\n' and '\0') and there is a memory error in the newly added code. It is reasonable as any instruction whose number of characters is equal to 1022 will trigger. This is non-negligible as the attacker could find it and break the program by keeping expanding the length of input instruction.

2.4 Vulnerability 4

Line	**144**
Modified	static void destroy(node_t *p){ // Original // while (p != NULL){ // node_t * left = p->left; // node_t * const right = p->right; // left = node_insert(left,right); // node_free(p); // p = left; // } //Modified do{ node_t * left = p->left; node_t * const right = p->right; left = node_insert(left,right); node_free(p); p = left; }while (p != NULL); }

Table 4: Vulnerability 4

Vulnerability 4 changes the while loop in *destroy* function to do-while loop.This type of mistake is very likely to be made by people who are just beginning with programming with less understanding of the syntax of programming language.

It can be triggered in a reasonable probability since if the input file ends without storing any value into map, p which is the input of the function should be null. Dereferencing p->left will cause memory error by referencing a NULL pointer.

2.5 Vulnerability 5

Line	**449**
Modified	//Original: #define MAX_INSTRUCTIONS 1024 #define MAX_INSTRUCTIONS 2048 --- if (p != NULL){ if (stack_full()){

	//vuln-5 Commented the following two lines **// debug_printf("Trying to load onto full stack\n");** **// return -1;** } stack_push(p->value); }else{ printf("Not found.\n"); }

Table 5: Vulnerability 5

Vulnerability 5 deletes defensive code for preventing loading elements into a full stack. In this way, although the program can still stop the user directly pushing an element to a full stack, the user can escape from this by pushing the stack to full and then storing some of the stack elements to map. After that the user can keep pushing elements until the stack is full again. But now he can cause the stack to overflow by inputting load instructions as the defensive code is missing. This vulnerability is still reasonable as triggering this doesn't rely on any specific input value. However, compared with Vulnerability 2, this one is much harder to detect as *stack*[] is defined as a global variable which is not stored on the stack in memory. The vulnerability will be triggered only if a lot of elements overflow.

The programmer may forget to put the defensive code on all components which access the same resource. This could be utilized by attackers to get past the security check on one resource by accessing it in another way.

3. Fuzzer Design

3.1 Introduction

The final fuzzer essentially utilizes the strategy of generation-based fuzzing. It generates instructions that start with commands in *Instruction.java* followed by legal or illegal arguments according to the serial number of the current *fuzz.txt* being written. The iteration of the fuzzer includes three stages. In the first stage, our fuzzer randomly generated legal instructions that won't cause ERROR while the program is running in the maximum allowed quantity for each output file. It turned out that it could hardly trigger our own vulnerabilities and the source code coverage was low. According to the coverage report, it didn't touch the codes that stopped illegal inputs. Therefore, in the second stage, cases were added to target the illegal instructions such as popping the stack when it is empty. The improvement was obvious as the coverage exceeded 90% and the bugs it detected increased remarkably. Still, it failed to trigger some of our vulnerabilities and a small part of the code remained untouched. To fix these problems, in the third stage, specific cases were added in the fuzzer to target those untouched codes and our vulnerabilities and this became the final fuzzer.

The improvement can be seen in detail in Evaluation Results.

The final fuzzer generates output files in sequence. The first 30 output files contain only illegal instructions such as arithmetic operations when elements in the stack are insufficient or more arguments following a command than required. Then there are two files to target specific cases such as updating the value of a variable by storing a variable twice. The rest of the output files only contains randomly-generated legal instructions that won't cause ERROR defined by program. We don't make the list of specific cases long because how many times the fuzzer will be run is unrevealed and if there are too many cases it's likely that those random instructions won't be generated enough to detect memory errors caused by legal instructions, which contributes the detection of ¼ of the overall bugs. Thus the specific cases we choose are only those we consider more likely to cause a memory error like dereferencing a pointer with NULL pointer check or cases involving iteratively accessing elements in an array.

3.2 Implementation

To run the final fuzzer, three files are needed: *Fuzzer.java*, *Variable.java* and *state.txt*.

The *Fuzzer.java* initially defines some constants and variables, such as the name of the output file, the name of the file that stores information after each time the fuzzer is run, an ArrayList named *stack* that acts as a stack to store numbers that are pushed and another ArrayList named *variables* that acts as the *map* in *dc.c* to store the variables.

The *singleInstructionGenerator* method generates a random legal instruction each time it's called. When the method is called, the fuzzer looks up *stack* and *variables* to decide the current legal command set, which is the variable *instructions*. For example, if the *stack* is empty, the command set won't include POP or STORE.

The *mutationGenerator* method is used to generate illegal instructions. It takes two arguments. One is the case name defined in the enumeration class *Case*, which includes cases like full stack, empty stack, more arguments than required, etc.. Another one is the command name defined in *Instruction.java*. These two tell the method to generate instructions to target specific illegal input cases. For example, if the case name is FULL_STACK and the command name is PUSH, the method will generate PUSH instructions until the *stack* is full. Then the method will generate one more PUSH instruction to simulate the case.

The *mutate* method is called by the *mutationGenerator* method when the case is MORE_ARGUMENT. The maximum number of arguments is 50, which is defined by the constant MAX_ARGUMENTS. The *mutate* method adds an illegal number of arguments after a command by taking four arguments. The first one is the command name, the second one is the number to be excluded to make the instruction illegal. For example, for the command PUSH, 1 will be excluded. The third one is the type of arguments. Types are defined by the enumeration class *RandomType*. And the last one is the optional suffix of the variable name, which is .txt for the command SAVE.

The *createFullStackInsts*, *createDivideZero*, *createInsufficientStack* and *createEmptyStack* methods are used to create the preconditions for the cases indicated by the method names.

The *randomNumGenerator* and *randomStringGenerator* methods are used to generate random arguments after the commands. They are called by the above methods.

The *Variable* class is used to store variable names and values.

The *state.txt* is used to document information after each time the fuzzer is run. It records the serial number of the current output file being written, elements in the stack and variables in the map after the last time the fuzzer is run. This file is necessary because during the experiments we found that illegal instructions detected more bugs than legal instructions. Since we don't know how many times the fuzzer will be run, we want the fuzzer to generate illegal instructions in the first place to ensure they will be fed to the other people's vulnerabilities and we use the serial number to control this. In addition, we also found that when the output files were fed to the *dc* program in batches, all instructions shared a stack and map that wouldn't be reset even if an illegal instruction in those files triggered an error. In other words, all instructions share a stack and map and the program won't stop until it encounters a bug that makes it abort. Therefore, we store elements in the stack and variables in the map after each time the fuzeer is run and when it is run again, the fuzzer can read the state of the stack and map in the *dc* program since last time.

4. Evaluation Design

Our evaluation is divided into two main parts. The first part focuses on three fuzzer in different development stages. In the first stage, our fuzzer only generated valid instructions; in the second stage, our fuzzer both generated valid instructions and invalid instructions; in the third stage, on top of the second stage, our fuzzer also takes some specific cases into consideration. From 4.1 to 4.3, we mainly focus on comparing these three fuzzers with different metrics to present how our fuzzer's performance improves.

The second part of the evaluation focuses on the comparison with fuzzer in different design principles, i.e. libFuzzer.

In order to evaluate our fuzzer（called it A）reasonably and rigorously, these following indicators need to be taken into account:

- a compelling baseline fuzzer B to compare against
- a sample of target programs—the benchmark suite
- Performance metrics to measure when A and B are run on the benchmark suite
- a meaningful set of configuration parameters

The evaluation also considers random factors brought about by the nature of the fuzzer itself. Therefore, the evaluation uses statistical testing to measure enough trials to sample the overall distribution representing the performance of the fuzzer to determine that the comparative data of A and B is true, not accidental. In 4.4, we conduct comparison experiments to discover the difference between our fuzzer and libFuzzer (Greybox Fuzzing).

4.1 Triggering Vulnerabilities

This experiment is based on five vulnerabilities designed by our team. We count how many vulnerabilities can be triggered by the fuzzer at each stage, and the specific ID of the vulnerabilities that were not triggered. Here we don't design a larger vulnerability sample for there are no guidelines to establish a good vulnerability sample with limitation of time and team scale. It is very hard to design a good vulnerability sample. So, we just use the five vulnerabilities with better tournament results and careful consideration as our vulnerability sample.

4.2 Coverage

This experiment covers the different coverage metrics of the fuzzer at each stage. We collect the region coverage, function coverage and line coverage for these three fuzzer. This also provides a reference for fuzzer performance and we can try to discover if the coverage could impact the capability of vulnerability detection in terms of dc program.

4.3 Tournament Ranking

In order to achieve a more comprehensive test and given that we only have five vulnerabilities, we also design an experiment based on a larger vulnerability sample to evaluate our fuzzer in three stages.

We select a period when the number of vulnerabilities on the tournament exceeded 100 and without any updates. This not only ensures that the vulnerability sample is sufficient and stable during the experiment, but also guarantees that our three fuzzers are based on the same vulnerability sample and reduce the impact of other factors. We upload each fuzzer in three stages to the tournament in turn to collect the ranking data and results based on a larger vulnerability sample.

4.4 Comparison to libFuzzer

4.4.1 Introduction

We choose the dc.c as our benchmark since our fuzzer is designed for this program particularly. Although our fuzzer may be able to fuzz other programs related to input text files, using dc.c as a benchmark maximizes the possibility for our fuzzer to have a stable performance for each experiment and can help better evaluation for our fuzzer.

For our baseline fuzzer, there are a lot of choices. But to reduce the complexity of the experiment and implement more targeted evaluation, we compare each stage of our fuzzer and compare the final fuzzer with libFuzzer.

We use libFuzzer in two approaches. One is using libFuzzer without any arguments passed in. The other one is using libFuzzer with the following arguments: *-only_ascii*, *-dict* and *-detect_leaks*. As discussed above we choose dc.c as our benchmark. The focus of fuzzing dc.c is the input instructions.

The reasons we choose these three arguments are because the *-only_ascii* and *-dict* arguments can affect the generation and the format of seed file input, which may have the impact on the libFuzzer performance in terms of dc.c. And *-detect_leak* will affect whether to cancel the memory limitation during running time. Adding each of these three arguments can increase the capability of vulnerability detection to some extent.

Another way to use libFuzzer depends on whether to give it a corpus or not. For one reason, as the libFuzzer is coverage-guided and it relies on the corpus of sample inputs for the code under test, this can help discover if different designs of corpus can affect the libFuzzer's efficiency to trigger vulnerabilities. For another reason, since our final fuzzer is completely designed based on the dc program and we analyze its code and control its randomness to a certain extent artificially, the result is somewhat affected by human factors. Therefore, in order to make the results of experiments more convincing and ensure a fair comparison, providing the corpus for libFuzzer allows the operation of libFuzzer to be also affected by human factors. The corpus choices include empty corpus, random corpus with fewer instructions and a well designed corpus with all valid instructions. During the tentative experiments, we found that the random corpus was no better than the empty corpus. Hence, we only focus on comparing our fuzzer against libFuzzer with empty corpus and well designed corpus.

4.4.2 Configuration Parameters

- Experiment Environment: Ubuntu 16.04 in Visual Studio code
- Max Fuzzing Running Time: 300s
- Each test runs 30 times with only 1 job
- No memory usage limitation, default is 2048 Mb
- Arguments:
  -only_ascii=1/=0 <If 1, generate only ASCII (isprint``+``isspace) inputs. Defaults to 0>
  -dict (use dictionaries or not)
  -detect_leaks=1/=0 <enable leak sanitizer/turn off leak sanitizer>
- Corpus: Empty Corpus / Corpus covering all valid instructions

4.4.3 Metrics

- Region Coverage
- Function Coverage
- Line Coverage
- Input Files Generated
- Number of Our Vulnerabilities that are Triggered

4.4.4 Experiment Implementation

We choose two sets of comparison experiments. The first comparison experiment is based on the absence of a corpus. We compare the results of libFuzzer without arguments and each of the three arguments respectively, and compare these four results with our final fuzzer. The second comparison experiment is based on the well designed corpus as the initial seed file. And we conduct the same experiment procedure as the comparison experiment.

The time out limit has little impact on our fuzzer since our fuzzer can trigger vulnerabilities in a short period of time. But considering the efficiency and performance of the libFuzzer may be different under different arguments and corpora, we set the running time as 300s. In this period, the consumption of memory usage will not exceed 500 Mb, which will not bring pressure to our experiment platform and the experiment can be implemented smoothly. In addition to setting the three coverage rates as comparison parameters, we count the test files generated by triggering the vulnerability as our comparison parameter. This can reflect the efficiency of the fuzzer.

5. Evaluation Results

5.1 Triggering Vulnerabilities

```
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-1/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-2/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-3/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-4/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-5/dc-san tests/*
```

Figure 1. Stage-1

```
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-1/dc-san tests/*
AddressSanitizer:DEADLYSIGNAL
=================================================================
==3847==ERROR: AddressSanitizer: SEGV on unknown address 0x00010026e6bf (pc 0x00000051643a bp 0x7fffd0720490 sp 0x7fffd0720480 T0)
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-2/dc-san tests/*
=================================================================
==3848==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffcc9af200 at pc 0x000000512300 bp 0x7fffcc9af120 sp 0x7fffcc9af118
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-3/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-4/dc-san tests/*
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-5/dc-san tests/*
```

Figure 2. Stage-2

```
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-1/dc-san tests/*
AddressSanitizer:DEADLYSIGNAL
=================================================================
==5582==ERROR: AddressSanitizer: SEGV on unknown address 0x00010026e6bf (pc 0x00000051643a bp 0x7fffe1478790 sp 0x7fffe1478780 T0)
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-2/dc-san tests/*
=================================================================
==5583==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fffc8622720 at pc 0x000000512300 bp 0x7fffc8622640 sp 0x7fffc8622638
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-3/dc-san tests/*
AddressSanitizer:DEADLYSIGNAL
=================================================================
==5584==ERROR: AddressSanitizer: SEGV on unknown address 0x0000013cc138 (pc 0x000000514f3a bp 0x7fffde392170 sp 0x7fffde392020 T0)
ruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-4/dc-san tests/*
^[[Aruofzhang@DESKTOP-SJ58K0S:~/testing_group_assignment/swen90006-a2-2020$ ./bin/vuln-5/dc-san tests/*
AddressSanitizer:DEADLYSIGNAL
=================================================================
==5587==ERROR: AddressSanitizer: SEGV on unknown address 0x0000013b4000 (pc 0x0000005163b4 bp 0x7fffe5d9ccf0 sp 0x7fffe5d9ccd0 T0)
```

Figure 3. Stage-3

The stage-1 fuzzer can not trigger any vulnerabilities we designed. The stage-2 fuzzer can only trigger segment false in vuln-1 and stack-buffer-overflow error in vuln-2. The stage-3 fuzzer can detect 4 vulnerabilities where only vuln-4 is not triggered.

5.2 Coverage

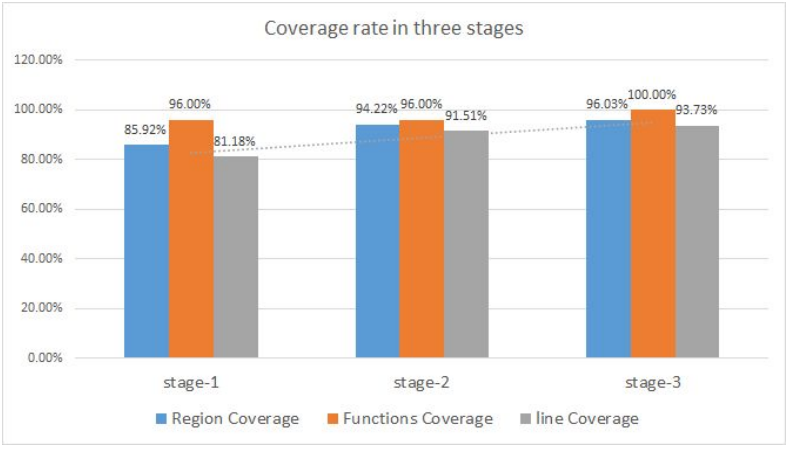

Figure 4. Coverage Rate based on Stage-1, Stage-2, Stage-3 Fuzzers

We compute the coverage of the fuzzers modified in every stage. As we can see, based on random fuzzer (stage 1), which focuses on generating valid instructions, the overall coverage rate has a good performance and is almost covered over 80%. During stage-2, we add some invalid instruction cases and cover some statements which are not triggered in stage-1, and the fuzzer reaches over 90% coverage rate. However, there are still several lines uncovered. For example, from line114 to 118, where implements continuously ‘store’ the same variable to update the variable value. So in stage-3, we design some cases specifically to deal with

uncovered lines based on the results of stage-2, and get a decent coverage rate of about 95%. We can not make line coverage and region coverage be 100% because there are some small areas that cannot be covered.

5.3 Tournament Ranking

13	Sun	2020-10-22 19:00:06	[59]	[19, 19, 11, 25, 10]
14	Jiao	2020-10-21 22:36:37	[60]	[19, 13, 15, 19, 15]
15	LIHUWANG	2020-10-22 19:01:45	[36]	[16, 18, 19, 27, 23]

Stage-1 ranking result

2	JiahaoXu	2020-10-22 01:17:13	[117]	[25, 20, 18, 18, 24]
3	LIHUWANG	2020-10-22 19:04:31	[95]	[16, 18, 18, 27, 23]
4	shubhamp	2020-10-22 16:06:59	[110]	[12, 14, 18, 18, 12]

Stage-2 ranking result

1	z	2020-10-22 05:05:43	[111]	[24, 26, 19, 18, 23]
2	JiahaoXu	2020-10-22 01:17:13	[117]	[25, 19, 18, 18, 24]
3	LIHUWANG	2020-10-22 19:06:37	[108]	[16, 18, 18, 27, 23]

Stage-3 ranking result

Figure 5. Tournament Ranking Results

As Figure 5 shown above, the ranking in the tournament has improved significantly step by step after completing three stages modified.

5.3 Comparison to libFuzzer

The following matrices show the performance for applying LLVM libFuzzer in different configurations to trigger out 5 vulnerabilities we design, representing the evaluation of the coverage results of libFuzzer as well.

1) Define libFuzzer without Corpus

   a) Results of Generating libFuzzer without Arguments

The Results of Configurations 1 { -max_total_time=300}							
	CORP	COV	RSS(MB)	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	78.67	71.00	419.00	23	56.56%	52.00%	45.51%
vuln-2	24.67	26.33	69.00	22	33.45%	24.00%	23.92%
vuln-3	138.00	86.00	334.67	9	62.70%	61.33%	68.30%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	200.33	104.67	457.33	2	62.82%	69.33%	54.31%

Table 6. The libFuzzer Performance after Applying Configuration 1

   b) Results of Generating libFuzzer with Arguments

The Results of Configurations 2 {-max_total_time=300, -dict=DICTIONARY_FILE}							
	CORP	COV	rss	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	68.33	70.67	302.00	25	59.93%	55.61%	47.61%
vuln-2	24.00	35.67	64.00	26	42.96%	33.33%	31.06%
vuln-3	60.00	56.33	86.00	10	54.51%	48.00%	42.37%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	304.67	135.67	569.33	5	78.58%	85.33%	73.62%
The Results of Configurations 3 { -max_total_time=300, -only_ascii=1}							
	CORP	COV	rss	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	84.33	76.33	353.33	23	60.56%	52.00%	56.34%
vuln-2	11.67	17.00	38.00	22	33.45%	24.00%	20.48%
vuln-3	174.33	111.33	396.33	9	63.70%	81.33%	80.46%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	235.33	139.00	452.00	4	63.82%	80.33%	79.04%
The Results of Configurations 4 {-max_total_time=300, -detect_leak=0}							
	CORP	COV	RSS(MB)	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	79.67	69.33	416.33	23	58.60%	53.33%	47.29%
vuln-2	27.33	32.67	174.00	24	37.42%	29.33%	28.35%
vuln-3	192.00	117.00	452.67	11	77.74%	85.33%	72.82%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	161.67	90.00	431.00	3	64.26%	69.33%	55.53%

Table 7. The libFuzzer Performance after Applying Configuration 2, 3, 4

Given Table 6 and 7, on condition of no corpus, the arguments used to limit generating inputs by libFuzzer have made slight optimization during fuzzing. And argument *'-dict'* has a better performance among the three arguments. However, the coverage rate raises slightly and can be varying in an unstable way. Compared with Figure 4, these libFuzzers also cannot perform as well as fuzzers we designed because of their random mechanism without any guidance of generating valid and invalid inputs. Besides, due to the time limitation, vuln-3 and vuln-5 can not be triggered, which means the corpora generated by these libFuzzer are not good.

2) Define libFuzzer with Corpus

a) Results of Generating libFuzzer without Arguments

The Results of Configurations 5 {corpus, -max_total_time=300}							
	CORP	COV	rss	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	50.67	109.00	53.67	26	86.04%	96.00%	81.98%
vuln-2	100.67	121.67	447.67	26	90.13%	98.67%	86.90%
vuln-3	84.33	88.67	325.33	19	70.64%	73.33%	66.79%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	116.33	125.00	465.33	7	94.95%	100.00%	92.25%

Table 8. The libFuzzer Performance after Applying Configuration 5

b) Results of Generating libFuzzer with Arguments

The Results of Configurations 6 {corpus, -max_total_time=300, -dict=DICTIONARY_FILE }							
	CORP	COV	rss(MB)	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	64.67	118.00	176.33	28	88.33%	98.67%	85.37%
vuln-2	101.00	128.00	436.67	27	90.61%	98.67%	87.40%
vuln-3	109.00	130.67	455.33	23	92.42%	100.00%	90.22%
vuln-4	N/A	N/A	N/A	30	84.48%	96.08%	79.89%
vuln-5	145.00	130.00	459.33	5	95.18%	100.00%	92.96%
The Results of Configurations 7 {corpus, -max_total_time=300, -only_ascii=1}							
	CORP	COV	rss	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	52.33	112.33	64.33	27	87.24%	96.00%	82.72%
vuln-2	71.33	119.67	89.00	26	87.37%	67.33%	83.15%
vuln-3	110.67	130.67	448.00	20	93.50%	100.00%	91.64%
vuln-4	N/A	N/A	N/A	30	84.48%	96.00%	79.89%
vuln-5	115.67	128.33	453.00	5	93.98%	98.67%	91.27%
The Results of Configurations 8 {corpus, -max_total_time=300, -detect_leaks=0}							
	CORP	COV	rss	Trigger times (per 30)	Region Coverage	Functions Coverage	line Coverage
vuln-1	58.33	110.67	58.00	26	86.52%	32.64%	27.92%
vuln-2	100.33	122.67	440.33	27	90.62%	98.67%	87.39%
vuln-3	102.33	125.00	348.33	23	92.18%	98.67%	89.55%
vuln-4	N/A	N/A	N/A	30	N/A	N/A	N/A
vuln-5	116.00	125.00	466.33	6	94.34%	96.00%	90.26%

Table 9. The libFuzzer Performance after Applying Configuration 6, 7, 8

After adding some valid inputs as default corpus, the libFuzzer can train itself by learning from these presuppose inputs and generate more valid or invalid corpus which can efficiently trigger the vulnerabilities as well as increasing the coverage rates. As the output shown above, comparing the Table 9 and 7, it is significantly found that the coverage rate has grown rapidly after using a default corpus. However, the vuln-3 and vuln-5 are still seldom to be triggered due to their specification designed, that is, can only be detected by particular input.

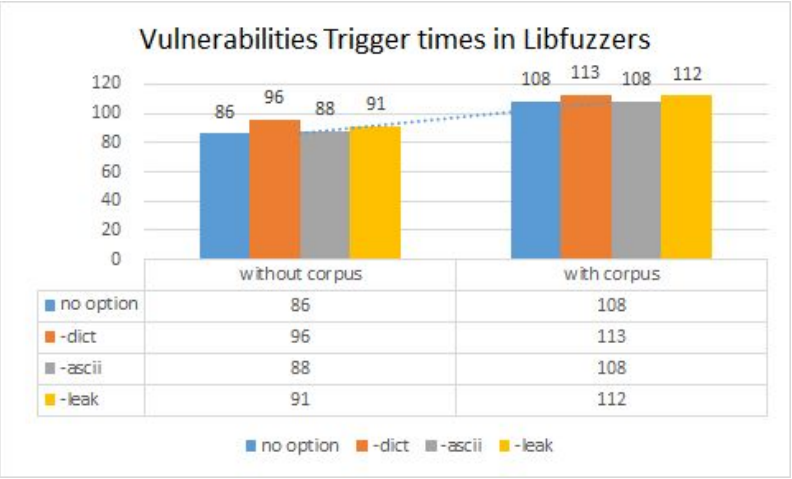

Figure 6. Vulnerabilities Tigger Times in libFuzzers

To better compare the performance of 8 different libFuzzers, we calculate the trigger times of each vulnerability by using these libFuzzers. As the Figure 6 shown above, we can easily find that the fuzzers with default corpus can trigger more vulnerabilities, whereas the *'-dict'* option also plays a significant role in generating valid inputs.

6. Discussion

In general, our fuzzer is mainly based on generation-based fuzzing technique which randomly generates legal commands as much as possible expecting to achieve good coverage to detect memory error. The reason is that the format of input in the original program is greatly restricted. Either a non-exist command which is not in the dictionary or an incorrect format following an existing command will cause the program to terminate by its defensive code. Compared with random fuzzing and mutation fuzzing whose generated input may cause the program to finish executing after running a few lines of code in this case, our fuzzer (stage-1) is much more efficient as ideally all the inputs can be executed by the program without triggering an ERROR. As a result, our fuzzer achieved great coverage.

However, there is a significant tradeoff which is that our fuzzer assumes all the defensive code is implemented correctly. Thus it cannot detect errors like dereferencing a NULL pointer or buffer overflow which is caused by absence or incorrect defensive code. Although, we got it partially fixed by manually inserting that kind of cases to test and achieved nearly 100% coverage core, it doesn't guarantee all those cases are covered. And even that block of code is covered, the new added cases are still insufficient due to our bias of the understanding of code. Otherwise, compared with libFuzzer, our fuzzer does not keep tracking the execution state and create input based on that. All the legal commands are generated randomly. Thus for errors triggered by only some specific value like the boundary value or the upper bound of an array, our fuzzer can hardly detect this kind of error by purely generating random inputs.

As for the evaluation on libFuzzer. Firstly, the libFuzzer without a given corpus has lower coverage than our fuzzer at each stage. But it triggers more vulnerabilities than our first two stage fuzzer. Secondly, without a given corpus, the libFuzzer with *-detect_leaks* has more higher coverage on the vuln-3 but it still cannot trigger vuln-3. In sum, code coverage can be the objective metric to evaluate our fuzzer, but the number of vulnerabilities detected should be considered first.

Moreover, it can be seen that the initial corpus covers more distinct input and the libFuzzer efficiency is higher. A libFuzzer with a well designed corpus can trigger more vulnerabilities in a shorter time (less test file generated in total and less memory consumption) than a libFuzzer without a corpus. Meanwhile, with a given comprehensive corpus, the impact of each option on libFuzzer to fuzz the *dc* program is reduced. They all trigger four vulnerabilities quickly. The reason is related to libFuzzer design principles. The performance of libFuzzer is affected by the corpus design as initial seed as it is coverage guided and relies on. This Fuzzing automatically detects programs by repeatedly executing the generated input. These inputs are designed to find vulnerabilities in the target program by running as many code paths as possible under different conditions.

Overall, black-box fuzzing techniques (e.g. fuzzer based on random, like our first stage fuzzer) use the smallest amount of program information to quickly generate input, and the result is that they produce the smallest depth of code coverage. White-box fuzzing(e.g. our second stage fuzzer to cover invalid instructions) uses a constraint solver to analyze the source code in detail, revealing the deep path of program branches. Although the white-box fuzzer can achieve high code coverage, each generated input requires the largest computational cost. Grey-box fuzzer(e.g, libFuzzer) finds a balance between these two extremes, emphasizing a simple input generation strategy that can effectively monitor program status with minimal overhead. A key difference between the three main categories of fuzzer black-box, grey-box and white-box is the degree of program analysis that occurs during testing. During compilation, Black-box fuzzing completely ignores states to achieve maximum test throughput, while white box technology focuses on weight analysis to have a greater impact on each attempt. The Greybox fuzzer instrument program can effectively monitor the progress during testing, enabling them to use the amount of coverage achieved so far to dynamically guide the input generation for future executions.

Reference

Klees, George, et al. "Evaluating fuzz testing." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018.