AN EMPIRICAL STUDY FOR THE RELIABILITY OF UTILITIES IN UNIX VARIANTS

Mengxiao Zhang
Department of Computer Sciences
University of Wisconsin-Madison
mzhang464@wisc.edu

Emma He
Department of Computer Sciences
University of Wisconsin-Madison
he57@wisc.edu

January 6, 2020

ABSTRACT

We have tested the reliability of utilities on four Unix variants platforms using random input strings, including Linux, MacOS, FreeBSD, and Xv6. We inherited and modified the original fuzz tools to generate test cases and then automatically test and detect failures. We documented and analyzed the testing results to further understand the bugs. We categorized the causes of failures in comparison with previous studies. After nearly three decades from the original study, the testing result indicates that fuzz tools are still sharp in crashing and hanging utilities. The failure rate on all platforms we tested is 24%. Specifically, we discovered 17 crashes/hangs out of 73 utilities on Linux, 10 out of 79 utilities on FreeBSD, and 9 out of 81 on MacOS. The failure rate of Linux utilities is largely increased compiled to the 1995 study, while MacOS and FreeBSD have significantly higher reliability than Linux.

1 INTRODUCTION

Fuzz testing is a powerful automated software testing technique that tests the reliability of softwares and systems. In fuzz testing, a huge amount of random strings are generated and fed into utilities to test whether they are able to tolerate unexpected or invalid data, which indicates the reliability of the utilities. This technique is inexpensive and effective in identifying flaws and increasing system reliability, which is an essential and helpful complement to formal verification techniques [**?**].

In the original fuzz study in 1990, in which 88 utility programs were tested on seven Unix-based systems, 25-33% of them were hung or crashed by certain fuzz input, indicating that fuzz testing is an effective tool in detecting bugs in operating systems [**?**]. A later study in 1995 revisited fuzz testing in nine Unix-based systems, including Linux, and found noticeable improvement of reliability in all systems tested compared to 1990, but still showed significant rates of failure [**?**]. Although more recent fuzz testing has been done in Windows NT and MacOS [**?**] [**?**], primarily focusing on GUI-based applications, a systematic evaluation of Unix-based systems by fuzz is lacking since 1995. Therefore,

it will be extremely valuable to test the reliability of utility applications by fuzz in a variety of current, widely-used, Unix-based systems.

In this study, we applied fuzz testing to the following Unix-based systems: Linux, FreeBSD, MacOS, and Xv6. The reasons why we chose the above systems are as follows:

Linux is the most widely used free Unix-like operating system, the utilities of which were tested by fuzz tools in 1995. After more than 20 years of development and updates, the reliability of utilities on modern Linux system might differ significantly from those detected in the previous study. Therefore, revisiting the utilities in Linux will reveal if the previously-identified bugs have been fixed and if any new flaws have emerged. Similarly, we reapplied fuzz testing to MacOS, which is one of the most prevalent commercial operating systems for laptops and personal computers and has been updated substantially since the last fuzz testing in 2006[**?**]. FreeBSD is an open-source operating system descended from BSD, which is a Unix-like system developed by UCB. Outstanding compatibility has led to its widespread use on web servers. Since previous studies succeeded in crashing Linux utilities, we wondered if fuzz testing will detect bugs in FreeBSD. Xv6 is a lightweight UNIX based operating system designed for educational purposes by MIT, which is prevalent in universities and colleges [**?**]. Fuzzing tests on Xv6 will benefit the students and instructors who use Xv6 in operating system courses.

Our study has 4 parts:

1. Update the source code of fuzz tools, so that they can be compiled successfully and run on modern Unix systems.

2. Provide fuzz tools on Xv6 platform for other developers and researchers.

3. Generate abundant random test data and apply fuzz testing on utilities of Unix variants.

4. Collect the results of fuzz testing, analyze the underlying bugs, compare our record with previous research, and evaluate the reliability of utilities on modern Unix systems.

Section 2 describes how we updated fuzz tools and testing scripts for Linux, MacOS, FreeBSD, and Xv6. Section 3 presents the results from fuzz testing and the analysis for the causes of utility failure. In Section 4, we discussed several challenges in analyzing test results and debugging using debuggers and source codes, as well as some limits of fuzz testing. Section 5 discusses future work and Section 6 provides our conclusions.

2 FUZZ TOOLS

To apply fuzz testing to the Unix systems we selected, we need to update the fuzz tools originated from the 1990 fuzz study. Section 2.1 describes how we updated the fuzz and ptyjig program and testing script for Linux, MacOS and FreeBSD. Section 2.2 describes how we developed fuzz tools for Xv6.

2.1 Fuzz Tools for Linux, MacOS and FreeBSD

2.1.1 Fuzz

The fuzz program, first developed in the 1990 fuzz study, is a generator of random characters. It produces a continuous string of characters on its standard output file. We can generate different types of outputs by specifying the options given to fuzz. These options allow fuzz to produce both printable and control characters, only printable characters, or either of these groups along with the NULL (zero) character. We can also specify a delay between each character. This option can account for the delay in characters passing through a pipe and help the user locate the characters that caused a utility to crash. Another option allows us to specify the seed for the random number generator

for future reproduction. Fuzz can record its output stream in a file, in addition to printing to standard output. This file can be examined later. There are options to randomly insert NEWLINE characters in the output stream, and to limit the length of the output stream.

To apply fuzz tests to the current Unix platforms, we first modified the syntax of the fuzz program to accommodate for the current coding standards. Instead of feeding the utilities with output directly from the fuzz generator, we generated four groups of test cases with varying lengths of input files and then fed the utilities these test files using the python script described in Section 2.1.3. The four groups of test cases were generated as follows using different combinations of options:

1. 1200 small files with 0-1e3 characters or 1-100 lines with 0-50 characters per line.

2. 900 normal files with 0-1e5 lines with 1-255 characters per line, including the 12 test cases from the 1995 fuzz study.

3. 180 big files with 1e6-1e7 characters or 1e4-1e5 lines with 0-1e5 characters per line.

4. 60 huge files with 1e7-1e8 characters or 1e5-1e6 lines with 0-1e5 characters per line.

2.1.2 Ptyjig

The original ptyjig from the 1990 fuzz study was designed for testing interactive utilities, such as vim and htop. ptyjig first creates a child process using fork(). The child process opens an available tty device, copies this tty to its stdin and stdout and then runs the interactive utility by calling exec(). Parent process opens the corresponding pseudo-tty device (pty), creates another child process to write the test data to pty, and keeps reading utility output from pty. With the help of ptyjig, we can apply the fuzzing test to interactive utilities.

However, due to some obsolete macros and functions, the original ptyjig can not be compiled on current systems. Therefore, we made the following updates:

1. We updated the arguments of wait3(). The first argument of wait3() serves as a record of the termination status of child process. We changed its data type from union* to int*.

2. We updated the way to open pty and tty. In the earlier version of Unix systems, there were pseudo-tty devices in /dev/ named "pty%c%x", with %c ranging from p to s and %x ranging from 0 to 15. Ptyjig will go through all pty devices until it finds an available pty device. On the contrary, in today's systems, these devices are not displayed in /dev/. Instead, users call `posix_openpt()` to dynamically allocate a slave (pty), and then call `ptsname()` to find the filename of the corresponding master (tty) and open it.

3. We updated the return value of ptyjig. The original ptyjig checks the termination status of the interactive utility, but does not return it to the operating system. In the updated ptyjig, if the interactive utility crashes, ptyjig will return 139 to operating system, which can be captured by the testing script.

4. We made it compatible with multiple platforms. We determined the current operating system by checking predefined macros.

2.1.3 Testing Script

To test the utilities using the test cases generated by the fuzz program, we wrote a set of python scripts that automated testing of all the utilities. We chose to write python scripts because python works cross-platform, is easy to write and understand, and provides packages that allow executing sub-processes, parsing input strings, and selecting random options.

First, we added the utilities to be tested to a run.master file using the following syntax:

```
[stdin|file|cp|two_files] cmd [options pool]
```

`stdin`, `file`, `cp`, or `two_files` were chosen for each utility based on the type of input it takes. The script will test each utility against test cases in the four groups generated by fuzz. In addition, we added options that do not require arguments to the option pool for each command. For each test case, the python script will randomly choose options from the option pool with a probability of 0.5 for each option.

For example:

```
file as [-a -D -L -R -v -W -Z -w -x]
stdin bc [-l -w -s -q]
cp t.c cc [-c -S -E]
two_files diff [-s -e -p -T]
```

In the above example, a test case is fed into utility `as` as a file; utility `bc` gets input from standard input; a test case is copied to `t.c`, which is then fed info utility `cc` because utility `cc` only recognizes files with extension `".c"`; two test cases are fed into utility `diff`.

Moreover, we modified our testing script to automatically detect crash or hang of the utilities. To detect crash, we check the return value of each test. If the return value is 139, the utility being tested is crashed. To test hang, we set a timeout argument at 300 seconds. If the sub-process is not completed before the time expires, a timeout exception will be thrown and recorded by the test script.

The python script can also automate testing for interactive utilities. Similarly, we added the utilities to be tested to a run.master file using the following code:

```
pty cmd [options pool]
```

For example:

```
pty vim [-A -b -d]
```

Unlike basic utilities, every interactive utility needs a specific operation to quit. For example, in vim, we need to type in ESC to go back to normal mode and type in `:wq` or `:q!` to quit vim. Therefore, before testing an interactive utility, the python script will append the corresponding end to the original test data to prevent a hang that is caused by the lack of a quit operation.

2.2 Fuzz Tools for Xv6

Xv6 is a simple Unix-like operating system which lacks system calls used in our original fuzz program. Therefore, we made the following changes to the fuzz program and used it to test utilities in Xv6.

We modified the original fuzz code to generate random character output files in an Xv6 environment. Xv6 does not support the following functions used in fuzz: `rand`, `srand`, `fflush`, `fopen`, `fclose`, `sprintf`, `sscanf` and `getc`. Therefore, we wrote `rand` and `srand` functions, used Xv6-supported `open` and `close` functions, and simply read in command line arguments and then converted them into integer or string to replace the above functions. Hence, in an Xv6 environment, we could run fuzz testing using the following code:

```
fuzz [length] -o [outfile] | [cmd]
```

We modified Xv6 shell source code to run 10 tests for each of the six utilities for one execution. To test utilities reading from standard input, we called fuzz to generate files with random characters and piped the output. For utilities only processing command line arguments, we generated randomized command line arguments.

We made the above changes based on two considerations:

1. The capacity of the current Xv6 file system is 71680 bytes. Though it is possible to expand its capacity to 16MB, it requires changing its inode structure to support doubly-indirect memory blocks by modifying source codes[**?**]. Since the test cases generated for basic tests are close to 20 GB, it would be impossible to apply them directly to Xv6.

2. Only three utilities in Xv6 can be tested by canonical fuzz output, which are `grep`, `cat`, and `wc`. Therefore, we added randomized command line arguments to test three more utilities in Xv6, which are `kill`, `mkdir`, and `echo`.

3 RESULTS

Utility	Linux	MacOS	FreeBSD
as	hang		
awk			
bc	hang	crash	hang
bison	hang		
calendar			
cat			
cc	hang		
checknr	N/A		crash
cmp			
col			
colcrt			
colrm			
comm			
compress	N/A		
cpp			
csh			
ctags			
ctree	N/A	N/A	N/A
cut			
dc	hang		
dd			
diff			
ed			
eqn			

Table 1: List of Utilities Tested and the Systems on which They Were Tested (part 1)

In this section, we recapitulated the fuzz tests reported in our 1990, 1995, and 2006 studies on Unix-based systems using the updated fuzz tools described above. In the previous studies, we tested the reliability of utilities by random input streams on a variety of Unix platforms, but the majority of them are now obsolete. Therefore, our current study

Utility	Linux	MacOS	FreeBSD
ex			
expand			
expr			
f77		N/A	N/A
find	crash		
flex	hang	crash	
fmt	hang		
fold			
ftp		N/A	
gcc			
gdb/lldb	crash	crash	crash
grep			
grn	N/A		
groff	crash	hang	crash
head			
htop			
indent	N/A	crash+hang	
join			
latex		N/A	N/A
lex	hang	crash	
lint	N/A	N/A	N/A
look			
m4			
mail			

Table 1: List of Utilities Tested and the Systems on which They Were Tested (part 2)

focused on widely-used modern Unix variants, including Linux, MacOS, and FreeBSD. Linux was tested on machines in computer science labs at the UW-Madison, with Ubuntu 18.04.3, GCC version 7.4.0 and CPU x86_64. MacOS was tested on personal computers with MacOS Mojave version 10.14.5, Apple LLVM version 10.0.0, and 2.4GHz Intel Core i5. FreeBSD was installed on VirtualBox 6.0.14 platform using FreeBSD-11.3-RELEASE-amd64-disc1.iso file and configured as follows: 4GB memory and 20GB dynamic allocated virtual box disk. Linux and MacOS were tested in the 1995 and 2006 fuzz studies respectively, and the results were compared to our current study. Xv6 was run on QEMU emulator.

3.1 Quantitative Test Results

We tested 93 utilities across three platforms: Linux, MacOS, and FreeBSD. 65 of them are available in all three platforms. Table 1 lists all utilities tested across three platforms, and the utilities that were crashed or hung by our tests. We detected a total of 22 failed utilities, 17 in Linux, 9 in MacOS, and 10 in FreeBSD. 5 utilities (`troff`, `nroff`,

Utility	Linux	MacOS	FreeBSD
make			crash
md5	N/A		
mdls	N/A		N/A
mig	N/A		N/A
more			
neqn			
nm			
nroff	hang	hang	crash
pc	N/A	N/A	N/A
pic			
pr			
ptags	N/A	N/A	N/A
ptx		N/A	N/A
refer	N/A		
rev			
sch	N/A	N/A	N/A
sdiff			
sed			
sh			
soelim			
sort			
spell	hang	N/A	N/A
split	hang		

Table 1: List of Utilities Tested and the Systems on which They Were Tested (part 3)

`groff`, `bc`, and `gdb/lldb`) were crashed or hung in all three platforms, while the others were failed in one or two systems.

The failure rate of Linux in the current study (17/73, 23%) is apparently higher than in the 1995 study (5/55, 9%). Of the five Linux utilities that were crashed or hung in the 1995 fuzz study, `dbx` and `indent` are no longer available in the current Linux system, `ctags` and `ul` were not crashed or hung in our current studies, and `lex` was hung in our current study rather than crashed in the 1995 study. However, we detected crash or hang in 16 additional Linux utilities. 9 of them were not tested previously, while the other 6 were tested but not failed in the 1995 study. One possible explanation for the additional hangs and crashes detected in the current study is that we generated fuzz input of much greater size (up to 1e8 characters) than any previous studies (up to 1e5 characters), and thus exerted enough stress on the utilities to induce crashes or hangs.

Similarly, we detected a higher rate of failure in MacOS (9/81, 11%) than the 2006 fuzz study (10/135, 7%). One possible explanation is that in the current study, we did not test as many utilities as in the 2006 study, and none of the untested utilities in the current study were crashed or hung in the 2006 study. Of the 10 utilities that were crashed in

Utility	Linux	MacOS	FreeBSD
strings			
strip			
sum			
tail			
tbl			
tee			
telnet			
tex		N/A	N/A
tr			
troff	crash	crash	crash
tsort			
ul			
uniq			
units	N/A		
vim			
vgrind	N/A	N/A	crash
wc			
xargs			
yacc	hang		hang
zic			
zsh		hang	hang
# tested	73	81	79
# crashes/hangs	17	9	10
%	23%	11%	13%

Table 1: List of Utilities Tested and the Systems on which They Were Tested (part 4)

the 2006 study, `expr`, `zic`, `as`, `ul`, and `vim` were not crashed or hung in our current study, `groff`, `zsh`, and `nroff` were hung instead of crashed while, and `indent` and `troff` were still crashed in our current study. We detected four additional failed utilities, all of which were tested but not crashed or hung in the 2006 study.

Comparison between the current study and two previous fuzz studies revealed that some of the bugs that caused utilities to fail in the previous studies were fixed over the years, while some of these bugs still persists. However, we crashed or hung additional utilities in our current study. This could partially result from our updated fuzz test method that included larger size input and random selection of command line arguments.

The failure rate of FreeBSD (10/79, 13%), is slightly higher than MacOS, but significantly lower than Linux. Since FreeBSD was not tested in previous fuzz studies, we do not know how its reliability changed over the years. However, based on the above data, we can conclude that the reliability of utilities in MacOS and FreeBSD is better than that in Linux.

For Xv6, we tested 6 out of 10 utilities in the Xv6 environment, including `echo`, `wc`, `kill`, `grep`, `cat`, and `mkdir`. However, we did not find any crashes or hangs.

3.2 Causes of Crashes and Hangs

To find out the causes of utility failure, we examined each utility that was crashed or hung using the following method.

First, we found out the version of each failed utility and obtained its source code. For Linux, we used GNU official website as our main source. For MacOS, we searched for Apple's official document to locate the source code. For FreeBSD, we obtained the utility source code from github. Then we compiled the source code with desired flags on each platform in our own working directory. We changed the default prefix and the directory to install the utilities to avoid overwriting the existing utilities. Additionally, we used "-O0" flag to avoid compiler optimization and added "-g" flag to allow debugging. Finally, we used debugger to trace bugs and stepped through the program to find out the causes of failure. We used `gdb` and `strace` on Linux, `gdb` on FreeBSD, and `lldb` on MacOS.

Using the above method, we categorized the causes of each utility failure into the following categories.

3.2.1 Failure to Check Return Value

Not checking the return value of a called function seems to be a silly mistake; however, this mistake is still present in modern codes, even codes from GNU free software. Programmers often assume that a call can never fail or it is too much work or inconvenient to handle the case when it does fail. Unfortunately, these assumptions are often wrong. In the following code of `troff`, the function `fontno` will return -1 with invalid input. However, the program does not check the return value; instead, -1 is directly used as an index number for the array font_table[ ], which exceeds the range of the array and causes the program to crash. This mistake can be easily avoided by checking the return value of `fontno`, and throw an error when the return value is -1.

```
special_node()
{
...
    int fontno = env_definite_font(curenv);
    tf = font_table[fontno]->get_tfont(fs, char_height, char_slant, fontno);
...
}
```

Another example is `gdb`, which uses the input name to open a TUI window. When the input name is not found, a NULL pointer is returned. However, the program does not check the return value, instead it tries to access the window through the NULL pointer and caused the crash.

```
parse_scroll_args(char* arg, struct tui_win_info** win_to_scroll)
{
    ...
    *win_to_scroll = tui_partial_win_by_name(uname)
    // *win_to_scroll was 0x0
    ...
    tui_scroll(win_to_scroll);
```

```
    // crashed
    ...
}

struct tui_win_info* tui_partial_win_by_name(char* name)
{
    struct tui_win_info* win_info = NULL;
    ...
    // failed to find the window by name
    ...
    return win_info;
    // NULL was returned
}
```

3.2.2 Pointer/Array

Although pointer/array errors did not dominate the results of our current fuzz test as they did in the 1995 and 2006 studies, such errors still exist in current Unix utilities. The programmer made implicit assumptions about the contents of the data being processed, which caused the programmer to use insufficient checks on their loop termination conditions. Such errors were often simple, for example, the Linux utility `find` assumes there is an argument following the flag "-D". Therefore, it automatically reads the next element without checking the size of the array. When "-D" is not followed by any arguments, argv[i+1] will exceed the boundary of the array and `0x0` will be passed to strtok_r, which leads to crash.

```
process_leading_options()
{
    ...
    if (0 == strcmp ("-D", argv[i]))
        {
            process_debug_options (argv[i+1]);
            ++i;
        }
    ...
}
process_debug_options(char* arg)
{
    ...
    p = strtok_r (arg, delimiters, &token_context);
    ...
}
```

Another severe but pernicious problem caused by pointer/array error is heap memory corruption. It might not directly cause program failure, but undermine the integrity of heap causing fatal error at some later points. For example, the MacOS utility `bc` was programmed to access array memory before checking if the array needed to be resized.

```
lookup(name, namekind)
{
    ...
    id->a_name = next_array++;
    a_names[id->a_name] = name;
    ...
    if (id->a_name >= a_count)
    more_arrays();
    ...
}
```

It first put one pointer to `name` in the array `a_names` accessed by an index number `id->a_name`. It then checked if this index number `id->a_name` was greater or equal to the `a_count`, which was the length of array `a_names` to decide whether to grow the array. Each time, the index number `id->a_name` was incremented by 1. The largest valid index number of one array was 1 less than the length of this array. Hence, it allowed writing on one section of heap memory which did not belong to the allocated array.

In our debugging processes, this heap memory corruption crashed this utility for two reasons:

1. The checksum of free block on heap was contaminated. This crash happened in `malloc` function.

2. The leaf node of the tree-structured symbol table was overwritten. While reading that memory section, the program assumed a memory address on heap. The contaminated memory contained the content of symbol name. When the program tried to treat this content as heap memory address, the segmentation fault happened.

3.2.3 Bad Error Handling

Similar to pointer/array errors, the programmer often assumes that the input should meet certain criteria, thus does not use sufficient checks on their loop termination condition to handle input errors. The utility `make` in FreeBSD is representative: the `for` loop terminates when the number of left parentheses equals to the number of right parentheses. Since the random input did not have equal numbers of "(" and ")" , the `for` loop would never terminate, which made the pointer to a string keep increasing until it reached an invalid address and crashed the program.

Another example of bad error handling is caused by using unreliable macro, `assert`, in the case of utility `flex` on MacOS. It was programmed to have an array-structured stack to trace the depth of the input. When a "(" was read, `sf_push` was called. When a ")" was read, `sf_pop` was called. The variable `_sf_top_ix` was to track how many elements in the stack, which was also the index number to put a new element. This variable was initially set to 0. While parsing our test case, the `flex` first read a ")" before reading any "(". Hence it called `sf_pop` when `_sf_top_ix` was 0.

```
sf_pop (void)
{
    assert(_sf_top_ix > 0);
```

```
    --_sf_top_ix;
}
```

In `sf_pop`, before `_sf_top_ix` was decremented by 1, it called assert to check if `_sf_top_ix` was larger than 0. However, assert.h file was not included because `HAVE_ASSERT_H` was false. At this condition, `assert(Pred)` was defined but its function body was not declared.

```
#ifdef HAVE_ASSERT_H
#include <assert.h>
#else
#define assert(Pred)
#endif
```

As a result, `_sf_top_ix` was decremented by 1 whose value became the max value of a 8 bytes unsigned integer, 18,446,744,073,709,551,615. Later, when `sf_push` was called, it tried to access the array-structured stack by the index of 18,446,744,073,709,551,615. This caused a page fault hence crashing the program.

3.2.4 Sub-processes

A robust program might crash if it invokes a vulnerable program as a sub-process. In our current study as well as previous fuzz studies, on FreeBSD platform, `nroff`, `groff`, and `vgrind` crashed because they called `troff` as a sub-process. As we described before, troff can be crashed by certain input due to unchecked return value. Thus, the test cases that crashed `troff` would also crash `nroff`, `groff`, and `vgrind`.

3.2.5 Exceed Buffer Size

`spell` is a spell-checking utility which scans a text file for misspelled words. In the source code, it forks to create a child process and lets the child process run `ispell` using `exec()`. In the parent process, the input from `stdin` is written to a pipe. In the child process, the pipe serves as the `stdin` of `ispell`. When parent process writes a long string to the pipe, which exceeds the capacity of the pipe, the writer hangs.

3.2.6 Loop

`yylex()` is a commonly used lexical analyzer function, which recognizes tokens from the input string and returns them to the parser. This function recognizes tokens by building a state machine, and is used in `bc`, `flex`, `bison` and other utilities which need to parse input. However, random strings may lead to an infinite loop in state machine. However, due to the confusing logic that was encoded into hundreds of states, we are still debugging it.

3.2.7 Others

`dc` is a reverse-polish calculator which supports arbitrary-precision arithmetic. Some arithmetic with high precision will take a long time, which cannot be considered as hang. For example, in the fuzz test, one test case triggered a square root operation, requiring 800000 digits after the decimal point. However, it will be user-friendly to show the current step or to estimate remaining time.

Although we did not find any crashes or hangs on Xv6, we detected another unreliability of this platform. What we found out is that Xv6 allows `kill` to terminate the init process, whose pid number is 1. It causes the Xv6 system to report error and fail to respond. In Linux, only signals for which init has explicitly installed signal handlers can be

sent to process ID 1, the init process[**?**]. It protects the unexpected errors and improves the robustness of the system. Similar modifications should be applied to Xv6 to avoid this error.

4 DISCUSSION

Our study demonstrates that fuzz tools are still effective in crashing/hanging utilities in modern Unix Systems. While certain bugs detected in previous studies have been fixed over time, additional bugs were detected by our updated fuzz tools. In the process of debugging, sometimes we found it difficult to identify the source of crashes or hangs due to the following challenges:

1. The source causing the bug was far from or seemingly irrelevant to the location of the bug. While debugging `bc` utility on MacOS, we found out one leaf node became a pointer by checking heap stack of this memory address. We traced through how this AVL tree was built, searched, and rotated and found out nothing. Since this pointer was used to access one deep copy of the token generated by `yylex`, we then traced through paths of where different deep copies go and found out that, this deep copy was added to a global array with index number exceeding the length of this array.

2. The test cases failing utilities tend to be large, e.g. 120MB. If a utility hangs on such a big test file, it takes effort to distinguish between an unfinished run and a real hang before making judgement. For example, `tsort`, a wildly used utility for topological sort, did not finish on one of the largest test cases(119MB) in 300 seconds. However, it was not a hang, for it terminated after 40 minutes.

3. The indeterministic failing behaviors with the same test case: the utilities might crash at this time but not the other time, the locations of crashes might differ from time to time, and how utilities process the random input or files might be different. To debug these utilities, we had to repeatedly run the debugging tools until the same results appeared, which was time-consuming.

Although fuzz tools are efficient in detecting bugs resulting in program crashes or hangs, obviously not all bugs can be detected by fuzz. When using fuzz to test the reliability of programs, we should consider the following limits:

1. A program is detected as faulty only if there is a core dump or hang. Therefore, fuzz testing is less effective in uncovering security threats that do not lead to crashes and hangs, such as some viruses, worms, etc.

2. Fuzzing tests require significant time to implement effectively.

3. Test cases generated by fuzz tools can be extremely difficult to interpret, which makes debugging an arduous task.

4. Mangled invalid inputs might not penetrate very deep into the utility, therefore the crashes and hangs we caught might not be the only ones in a utility.

5 FUTURE WORK

Although we have updated the fuzz tools to make it work cross-platform and included random combination of options during testing, we can further improve it to enable more systematic fuzz testing.

Ironically, we discovered bugs of debugger tools on all three platforms. Especially on Linux, we found out more than one bugs in `gdb`, which still exist in the latest version. Moreover, these bugs are not something hidden deeply. To crash the latest version of `gdb` on Linux, one can just start `gdb` and type in:

```
"clear ,"
```

or `"ed ,"`

or `"faas"`

The above bugs are all caused by invalid arguments after a command in `gdb`. Based on this discovery, we found that test cases generated by fuzz without any other processes are not suitable for tests on interactive command-line utilities, such as `gdb` and `telnet`. If every line of a test case starts with a random valid command that the utility can recognize, which is followed by a random string, the chance of detecting bugs could be greatly enhanced. For example, to fuzz `gdb`, a line of the modified test case can be:

`"breakpoint" + a random string`

or `"clear" + a random string`

or `"next" + a random string`

In addition, to make the test more comprehensive, we would like to test how utilities react to random non-existing options. Furthermore, for options followed by inputs, it will be instructive to know how they react to invalid arguments. By specifying the requirement of each option in man page syntax, we can describe the option space to be fuzzed. For example:

```
stdin find [-P] [-L] [-H] [-D exec|opt|tree] [-Olevel 0|1] [-maxdepth int(0, 20)]
[-name string(5, 10)] [-anewer file(./test/)]
```

Then the testing script will parse the description and randomize the options as follows: randomly choose options in `[]` with a probability of 0.5 for each option; for -D option, randomly select one argument among "exec", "opt" and "tree"; for -Olevel option, randomly select one between "0" and "1"; for -maxdepth option, randomly draw an integer from [0, 20); for -name option, randomly generate a string, with length ranging from 5 to 9; for -anewer option, randomly select a file from "./test" directory.

The above systematic fuzz tool will be of great use to developers and researchers, since the option space is under user's control. No one knows a utility better than its developer, and the developers could do more targeted testing on some vulnerable options using this tool. Moreover, for some utilities, the whole option space can be fuzzed with the help of it.

6 CONCLUSION

Although there are different critiques against fuzz testing, the effectiveness of fuzz testing is still sharp. After over three decades since the first fuzz study, the overall crash/hang rate of common utilities on three Unix variant platform is 24%. We believe with more pushes on existing fuzz tool, systematically randomizing command line arguments, and expanding our tools on X-windows, more crashes/hangs would be uncovered.

Based on analyzing the above results from fuzz testing, we have made three observations in comparison with previous studies. The versions of utilities matter, because bugs are more version specific, such as `flex` and `find`. Based on the versions of utilities we studied, some bugs have been fixed, such as end-of-file and divide by zero. Though other bugs are still prevalent, such as pointer/array, they become less vulnerable and were embedded in the deeper layer. In most cases, they do not directly crash or hang utilities but are responsible for later failing behaviors.

In conclusion, our data shows that the number of failing utilities on Linux has largely increased since 1995, whereas FreeBSD and MacOS have a lower failing rate than Linux. Xv6 could be more robust for educational purposes.

Some of the bugs detected in previous studies still persist and additional bugs have emerged with the development of systems. Hence, we believe that more platforms and utilities will benefit from fuzz testing tools.