The Relevance of Classic Fuzz Testing: Have We Solved This One?

Barton P. Miller, *Senior Member, IEEE,* Mengxiao Zhang, and Elisa R. Heymann

Abstract—As fuzz testing has passed its 30th anniversary, and in the face of the incredible progress in fuzz testing techniques and tools, the question arises if the classic, basic fuzz technique is still useful and applicable? In that tradition, we have updated the basic fuzz tools and testing scripts and applied them to a large collection of Unix utilities on Linux, FreeBSD, and MacOS. As before, our failure criteria was whether the program crashed or hung. We found that 9 crash or hang out of 74 utilities on Linux, 15 out of 78 utilities on FreeBSD, and 12 out of 76 utilities on MacOS. A total of 24 different utilities failed across the three platforms. We note that these failure rates are somewhat higher than our in previous 1995, 2000, and 2006 studies of the reliability of command line utilities.
In the basic fuzz tradition, we debugged each failed utility and categorized the causes the failures. Classic categories of failures, such as pointer and array errors and not checking return codes, were still broadly present in the current results. In addition, we found a couple of new categories of failures appearing. We present examples of these failures to illustrate the programming practices that allowed them to happen.
As a side note, we tested the limited number of utilities available in a modern programming language (Rust) and found them to be of no better reliability than the standard ones.

Index Terms—Testing and Debugging; Testing tools

1 Introduction

We conducted the first fuzz random testing study in the fall of 1988 [19] and published the first report in 1990 [21]. This early work provided a clear demonstration of the value of simple black-box random testing by finding real bugs in widely-used software, showing that we could crash 25-33% of the utility programs that we tested on each platform. The simplicity of this technique meant that it was inexpensive and easy to apply. However, the real contribution lay in the transparent and methodical approach taken in that study:

1) All the tools, test cases, and results were public artifacts that any software developer or user could download and reproduce the results for themselves or apply to new software or systems.
2) We analyzed the results, identifying the source of each crash or hang and organized these results into categories, trying to come to some understanding of underlying causes of these bugs in the code.
3) We socialized and proselytized the techniques and results at all the major Unix vendors, of the day hoping to entice them into using these techniques or, at least using the bug reports that we produced to improve their software. As part of those discussions, we surveyed our colleagues at the companies and labs that produced versions of Unix to get insights into their thoughts on these bugs.

Interestingly, the original fuzz tests were occurring at the same time as the worm written by Robert Morris Jr. that brought the then-nascent Internet to its knees [29]. Our 1990 paper observed that the single largest failure category of the day was accessing outside the bounds of a buffer and suggested that fuzz testing might be used "...to help find security holes." This kind of bug is exactly what Morris exploited in his worm. And this is still the kind of error that is continuing to cause massive disruptions in the Internet as evidenced by the Heart Bleed vulnerability in OpenSSL [24] in 2014 and the more recent vulnerability in the classic sudo command in 2018 [23]. Compounding this problem is that even the best static analysis tools have had limited success (if any) in detecting such errors in the code [13].

Our initial fuzz testing results were derided and the original paper received a hostile response from the software engineering conferences and journals of the day; the reviews were not just negative but outright hostile and insulting. It took persistence getting that paper published; there was definitely not an immediately groundswell of adoption of these techniques.

However, we persisted. Five years later, our anecdotal experience suggested that little had changed in software reliability so we repeated and extended these tests [22]. And an important new player had arrived on the scene: the open source system. Even though there were improvements in the reliability of the software from the commercial Unix vendors, the GNU and Linux utilities showed significantly lower crash rates than any of these commercial systems. This was the first concrete evidence that open source software was not just the product a bunch of amateurs pretending to

- *B.P. Miller is with the Computer Sciences Department, University of Wisconsin-Madison. E-mail: bart@cs.wisc.edu*
- *M. Zhang is with the Computer Sciences Department, University of Wisconsin-Madison. E-mail: mzhang464@wisc.edu*
- *E. Heymann is with the Computer Sciences Department, University of Wisconsin-Madison. E-mail: elisa@cs.wisc.edu*

Manuscript received August 13, 2020; revised November 16, 2020.

be software developers; they were actually doing as well or better than the so-called "professionals".

The early studies were focused on Unix systems and that led to snide comments by colleagues from industry that a "real" operating system like Microsoft Windows or DEC VMS would fare better. Our fuzz study in 2000 of the Windows operating system [6] and in 2006 of the MacOS operating system [20] showed that these systems were no more reliable than the Unix variants and, in fact, noticeably less reliable by our fuzz testing measure.

The 1995 study also started to explore more structured fuzz input. X Window System applications communicate with X server via a well-defined message protocol [28]. To test X-Window applications, we first sent unstructured random input over the connection. The result was there were some crashes and some cases where the input was rejected, however the testing reached relatively shallowly into the control flow structure of the program. Most of the unstructured input tests stayed in the X Window Protocol processing code. As a result, we introduced random input that increasingly conformed to random-but-valid user input. That mean that all fields of the X Protocol messages had valid values and all mouse events took place in the bounds of a valid window and all key press events were matched with a corresponding key release event. As we increased the valid structure on the data, we reached deeper into the control flow of the utility being tested and found more bugs.

It is important to note that our work on fuzz testing took a simple, perhaps even simplistic approach to testing. While it was (and still is) an effective technique in testing and security assessment, researchers and developers have since made great advances, producing tools that are both more powerful and more varied in their capabilities. The past decades have produced a substantial amount of interesting and useful work in fuzz testing and applying it to many new contexts [11] [15] [17] [32]. One recent study [16] reported finding over 170 publications on the topic.

New fuzz tools have taken this type of testing well beyond the simple techniques that we developed, taking a gray-box approach, allowing them to dive deeper into a program's control flow [8] [12] [26] [30] [27]. However, these more advanced techniques often require more advanced specification of the input or, for tools that try to find effective erroneous inputs, often require extremely long execution times as they explore the input and program control-flow space. If you are a software developer, then it is certainly worth the time to set up these tools and develop the input strategies that will best test your program. For studies like ours that survey large bodies of software, our simple fuzz tools have the advantage that they are easy to apply and quick to run.

This led us ask the two questions: First, has operating utility software improved in the way that it handles unexpected input? Second, are our simple black-box techniques still relevant and are they still able to find useful errors? It is now well understood by the software community that reliability is the foundation of security and that fuzz testing is a powerful first means of exploration on the path to finding software vulnerabilities. So we hoped that the kind of flaws found by our original techniques should be less common, even rare.

Unfortunately, that did not prove to be the case.

In this study, we applied fuzz testing to three Unix variants:

Linux: The most widely used free Unix operating system of the day. It is widely used in servers and in the cloud, still appears on desktops and, importantly, is the foundation of the Android operating system that appears in billions of mobile devices. Linux also has the nice characteristic that we studied this system previously [22]. In our new study, 9 of the 74 utilities that we tested crashed or hung (12%).

MacOS: An extremely popular desktop operating system, which, at its heart, is based (in part) on FreeBSD. It is also the foundation of a mobile operating system iOS that appears in billions of mobile devices. Again we have the nice characteristic that we studied this system previously [20]. In our new study, 12 of the 76 utilities that we tested crashed or hung (16%).

FreeBSD: A descendant of the original BSD (Berkeley Software Distribution) that introduced many key advances to Unix, including the Fast File System [18], the TCP/IP socket programming interface, and paged virtual memory. FreeBSD is commonly used in storage servers and appliances. In our new study, 15 of the 78 utilities that we tested crashed or hung (19%).

As we examined the results from our testing, we still found too many failures caused by pointers and arrays being used in loops where the termination condition of the loop is not properly related to the size of the buffer being processed. We also saw a noticeable increase in the number of failures where return values were unchecked or improperly checked. Interestingly, we saw no cases in this new study where signed characters caused failures. However, we are seeing the appearance of failures caused by programmers making their loop conditions and state tracking increasingly (and sometimes incomprehensibly) complex.

Some of the failures were caused by bugs that have been present in the code for years; regular basic fuzz testing could have caught these much earlier. Others have been introduced as recently as 2019, such as for `ftp`.

Of course, we are still dealing with utilities written in C and (increasingly) C++, which are known for their hazards, especially when dealing with arrays and pointers. In recent years, system programming languages like Go [4] and Rust [10] have emerged that, among other things, addresses these pointer and array issues. Some of the "coreutils" (a subset of the Unix utilities) have been implemented in these languages, so it seemed sensible to compare the reliability of these utilities implemented in Go and Rust. We found a current project implementing the coreutils in Rust [3]. From this project, we tested the 14 utilities that take input and do more than simply copy it. We also found two projects that were implementing coreutils in Go, but these are quite incomplete and have not been active for three [14] and five years [9]. Of the 14 coreutils implemented in Rust that we tested, 3 failed (`comm`, `fold`, and `ptx` all hung). For comparison, one of these 14 utilities also failed on our Linux, MacOS and FreeBSD tests: `ptx` hung on Linux. Until more utilities, and more complex ones, are reimplemented in these modern languages, we will have to refrain from making any sweeping statements of comparison.

Section 2 describes how we updated fuzz tools and testing scripts for current Linux, MacOS, and FreeBSD systems. Section 3 presents our experiments and Section 4 presents our results and the analysis of these results including assigning the failures to similar categories that we used in earlier studies. In Section 5, we wax more philosophical about the testing process and the implication of these results. We conclude in Section 6.

2 Fuzz Tools

The fuzz tools consist of three components: (1) fuzz, the random input generator, (2) ptyjig, a tool that can provide input to utilities that use the terminal, and (3) scripts that automate the generation of random input and control the testing of utilities. For this study, we updated fuzz and ptyjig, and redesigned the testing scripts. All source files for the tools and scripts can be found at `github.com/dyninst/fuzz`.

2.1 Fuzz

At its heart, the fuzz program is a generator of random characters. It produces a continuous string of characters on its standard output file, with variation in the types and amount of characters generated controlled by the options given to fuzz. The basic parameter to fuzz specifies the length of generated data in number of character or number of newline-terminated lines. The options to fuzz include:

`-p`: Only the printable ASCII characters.
`-a`: The entire 8-bit character set that includes control characters (it can be important that the high-order bit is set).
`-0`: Include the NULL (zero) character in the random data.
`-l`: Generate random length strings up to a maximum length specified by the value associated with the `-l` option, with each string terminated by the newline character.
`-s`: Specify the random seed used for input generation.
`-m`: Specify a modulus for the random seed, limiting the size of the seed to the value associated with `-m` option minus one.
`-d`: Specify a delay between each character. This option is useful when debugging a program to allow the programmer a chance to observe the utility program's behavior and associate it with the current input.

To run fuzz in its simplest form on utility `as`, you would type:

```
fuzz 1000000 | as
```

Note that we often use fuzz by first storing the random data in a file and later running the utility program on that file:

```
fuzz 1000000 > t1; as t1
```

In our current effort, the fuzz program also was updated to conform to the more recent C90 [1] and gnu11 [2] C standards.

2.2 Ptyjig

The purpose of the ptyjig program is to provide input to utilities that read input from the terminal, a "tty" in Unix parlance or pseudo-tty. Programs such as `vim` or `top` depend on various characteristics of the terminal device, such as backspace and line-delete functionality and ability to move the cursor around the terminal's screen. ptyjig first creates a pseudo-tty device, starts the specified utility, and passes its input through the pseudo-tty to the utility. To run fuzz on such a utility, you would type:

```
fuzz 1000000 | ptyjig vim
```

Our changes to ptyjig included (1) updating the arguments to `wait3`, (2) use the new Posix standard for creating pseudo-tty's, (3) corrected the way that ptyjig reports the tested utility's completion status, and (4) adapted it to work on Linux, MacOS and FreeBSD.

2.3 Testing Scripts

The original fuzz distribution included a shell script for controlling the testing process. In our current effort, we developed a new collection of Python scripts to better generate trace data and to provide more control over how we test the utilities. One important feature in these scripts is that we automate the use of random input files that are named on the command line (versus previously assuming that this random input was only provided through standard input).

Four of the Python scripts (generate_small.py, generate_medium.py, generate_big.py, and generate_huge.py) generate a broad collection of random files, split along three dimensions:

1) Size: The four size categories are: small, medium, large, and huge. Small files are around 1000 characters long or 10 lines that are up to 100 characters long. Medium files are around 100,000 characters long or 1000 lines that are up to 100 characters long. Large files are around 10,000,000 characters long or 100,000 lines that are up to 100 characters long. Huge files are around 100,000,000 characters long or 1,000,000 lines that are up to 100 characters long.
2) Characters: There are three character categories: all ASCII including the zero byte, all ASCII excluding the zero byte, and only the printable characters.
3) Newline: The two categories either include the newline character at random intervals specified by the value associated with `-l` option or do not treat the newline character in any special way.

The fifth Python script, run.py, orchestrates the testing of the utilities. In other words, it actually runs each specified utility program, setting up the test files as input.

```
run.py cnfg_file -itst_dir -orslt_dir
```

Its basic options are:

`config_file`: The file that specifies the list of utilities to be run, the options to pass to that utility, and how the test input will be provided to the utility.
`-i`: The directory that contains the input files that will be fed to each utility program tested.
`-o`: The directory that will contain the results from each test. There will be one file for each run of the fuzz tool.

Each line in the configuration file is of the form:

```
{stdin|file|cp|two_files|pty}
cmd {cmd_options} { [ options_pool ] }
```

where `stdin`, `file`, `cp`, `two_files`, or `pty` control how input is provided to the utility program; `cmd` is the name of the utility to run, `cmd_options` are the options (if any) that will be used for each execution of `cmd`, and `[options_pool]` is a set of options from which to randomly select when running the program. Each option is randomly chosen from the option pool with a probability of 0.5. For example:

```
stdin bc [-l -w -s -q]
```

Run utility `bc`, where the random input is provided to standard input. Randomly select from the pool of options in the square brackets.

```
file as [-a -D -L -R -v -W -Z -w -x]
```

Run utility `as`, where the input is provided as a file.

```
two_files diff [-s -e -p -T]
```

Similar to `file`, except two input files are provided.

```
cp t.c gcc [-c -S -E]
```

Run utility `gcc`, where the random input is first copied to a file named `t.c`. This option is useful for programs like `gcc` that require input file with a particular suffix (in this case `".c"`).

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

Run utility `vim`, where the random input is fed through a pseudo-tty. In addition, interactive utilities often need a specific operation to quit. For example, in vim, we need to type ESC followed by `:q!` to quit. Therefore, before testing an interactive utility, the script will append the corresponding end character

sequence to the original test data to prevent a hang that is caused by the lack of an appropriate quit operation.

The testing script detects when a crash or hang occurs. To detect a crash, we check the return status of the program. To detect a hang, we set a timeout argument at 300 seconds. For each hang result, we verify manually that the program was not just waiting for more input.

3 Experimental Set-up

Our testing followed the same basic model as the previous 1990 [21], 1995s [22], and 2006 [20] studies on Unix-based systems using the updated fuzz tools and scripts described in Section 2[1]. We evaluated the reliability of up to 80 utility programs on Linux, MacOS and FreeBSD. The utilities selected needed to take input from standard input or a file, so a utility like `ls` is not appropriate. In addition, the utility needed to do more than trivial processing of the data, so we did not test utilities like `cp`. We also sought utilities based on their availability on more than one platform.

As in our previous studies, we used the primitive criteria that a program was considered to have failed if it crashed or hung (stopped responding even though input continued to be available).

Each program was tested on each available platform using the input files generated by the generate_small.py, generate_medium.py, generate_big.py, and generate_huge.py scripts. The random input files generated for this study can be found at `ftp://ftp.cs.wisc.edu/paradyn/fuzz/fuzz-2020/`.

In our previous studies, we tested the utilities on a variety of Unix platforms, but the majority of them (SunOS, HP-UX, Irix, AIX, Ultrix, NeXT) are now obsolete. Our current study focused on three widely-used modern Unix variants: [1] Linux (Ubuntu 18.04.3 on an Intel 3.20GHz Core i5 using GCC version 7.5.0), [2] MacOS (Mojave 10.14.5 on an Intel 2.4GHz Core i5 using Apple LLVM version 10.0.0.), and [3] FreeBSD (installed on VirtualBox 6.0.14 using the FreeBSD-11.3-RELEASE-amd64-disc1.iso image and configured with 4GB RAM and 20GB dynamic allocated virtual box disk).

4 Results

We tested 84 utilities across three platforms (Linux, MacOS, and FreeBSD), where 68 of the utilities were available on all three platforms. In this section, we discuss the quantitative results and then look deeper into the cause of the crashes we found.

Since Linux and MacOS were tested in our 1995 [22] and 2006 [20] studies, respectively, we include a comparison of the current results to those earlier studies.

4.1 Quantitative Test Results

As in our previous studies, the experiments described in Section 3 produced a noticeable number of failures, where a failure is a crash or hang. Table 1 lists all utilities tested across three platforms, and Table 2 lists the statistics for the three platforms. There were 9 failures out of 74 utilities test on Linux (12%), 12 out of 76 on MacOS (16%), and 15 out of 78 on FreeBSD (19%). We caused failures in a total of 24 different utilities, 10 of which failed on more than one platform, and three failed on all three platforms (gdb and troff). 3 of the 14 coreutils written in Rust that we tested hung (`comm`, `fold`, and `ptx`), compared to the one that hung on Linux, MacOS, or FreeBSD (`ptx`).

Some interesting and well-known utilities appear in the list of failures, including `as` (the assembler); `bash` and `dash` (shells); `ctags` (C language cross reference generator); `flex` (compiler lexical analyzer generator); `ftp` (file transfer utility); `gdb` (debugger); `pdftex/latex` and `tex` (word processor)[2]; `make` (build system); and `zsh` (shell). This collection of key utilities should be enough to get our attention.

When we reported the results in our earlier studies, skeptics were quick to point out that some of the failed utilities were obscure and of little interest, such as `ul`, a program that adds underscores to the text. However, other utilities that appear archaic are still in common use. For example, the classic `nroff` word processing utility (and its GNU successor based on `groff`) has been around since the earliest releases of Unix, but have long been obsoleted by programs such as LaTex in the 1980's and more modern word processors like Word. However, manual pages displayed from the `man` utility are still processed by `nroff`. Some care is needed when disregarding these more obscure utilities because they may still be in active use in system and production scripts.

Even though the list of utilities in the current study is not the same as those used in the 1995 study, we can still note the overall crash rates. For Linux, we have 12% now compared to 9% in 1995. Linux was in its infancy then so had fewer utilities available. For MacOS, we have 16% now compared to 7% in 2006. While it is difficult to over generalize based on studies done so far apart in time, we can at least say that things have not gotten better.

Lateral studies across such a large period of time are difficult due to utilities falling out of favor, such as `dbx` being replaced by `gdb` and `lldb`; `yacc` being replaced by `bison`; and `vi` being reimplemented as `vim`. In Table 3, we see a list of the utilities that were tested in all on all of Linux/GNU in 1995, MacOS in 2006, and in current study on Linux or MacOS, and that *failed in at least one study*. As we would hope, a couple of the utilities that failed previously, did not fail in the current study (`ex/vim` and `ul`). There were also a couple of utilities that previously passed but failed in the current study, `latex/pdftex` and `tex`. Five of the utilities that failed in previous studies still fail in the current study, though none for the same reason.

In the current study, we used test cases that were much larger than in our earlier studies. In the earlier studies, the largest test input that we used was 100KB. We note that nine of the utilities we tested required more than that amount to crash: `as` hung with 8.8 MB on Linux and FreeBSD, `calendar` crashed with 4.8 MB on MacOS, `checknr` crashed with 4.8 MB on FreeBSD, `col` crashed with 1 MB on FreeBSD, `dc` hung with 1.2 MB on Linux and MacOS, `flex` crashed with 4.8 MB on FreeBSD, `gdb` crashed with 9.6 MB on Linux and MacOS, `groff` hung with 288 KB on MacOS and FreeBSD, and `vgrind` hung with 594 KB on FreeBSD. The remaining utilities crashed with substantially smaller inputs.

4.2 Examples of Crashes and Hangs

While the failure statistics are of interest, it is perhaps more important to understand why these utilities crashed. For each failed utility, we obtained its source code, debugged the failure, and then categorized the cause of each failure. Below is a sample of these results.

4.2.1 Pointers and Arrays

From the earliest days, accesses beyond the bound of a buffer in C and C++ have been a problem. These were the major cause of failure in the first fuzz study and, sadly, are still a major contributor today. A loop termination condition should always have an explicit check based on the size of the buffer and not on

[1]We also ran fuzz tests in 2000 on the Windows operating systems [6], but since Windows uses a mostly different set of utilities, we are not comparing those results in this paper.

[2]`pdftex` and `latex` are aliases to the same executable, while `tex` is an independent and slightly different program. The C code for these programs was generated by web2c from the original code written in Web (an annotated version of Pascal).

TABLE 1
Utilities Tested and the Testing Results

Utility	Linux		MacOS		FreeBSD		Utility	Linux		MacOS		FreeBSD	
	version	fail	version	fail	version	fail		version	fail	version	fail	version	fail
as	2.30	○	11.0.0		2.17.50	○	look	*		1.18.10		8.2	○
awk	4.1.4		20070501		20121220		m4	1.4.18		1.4.6		1.4.18_1	
bash	4.4.20		3.2.57	●	5.0.16		mail	*		8.2		8.2	
bc	1.07.1		1.07.1		1.1		make	4.1		3.8.1		8.3	●
bison	3.0.4	○	3.3	○	3.4.2		md5/md5su	8.28		1.34		*	
calendar	*		1.19	●	8.3		mig	—		116		—	
cat	8.28		1.32		8.2		more	2.31.1		—		—	
checknr	—		1.9		8.1	●	neqn	1.22.3		1.19.2		1.19.2	
clang	8.0.0		11.0.0		8.0.0		nm	2.30		11.0.0		3504	
cmp	3.6		2.8.1		8.3		pdftex	6.2.3		6.2.3	●	6.2.1	
col	*		1.19		8.5	●	pic	1.22.3		1.19.2		1.19.2	
colcrt	*		1.18		8.1		pr	8.28		1.18		8.2	
colrm	*		1.12		8.2		ptx	8.28	○	—		—	
comm	8.28		1.21		8.4		refer	—		1.19.2		1.19.2	
compress	—		1.23		8.2		rev	2.31.1		1.12		8.3	
csh	20110502-5		—		—		sdiff	3.6		2.8.1		1.36	
ctags	25.2		5.8_1	●	8.4	●	sed	4.4		1.39		8.2	
cut	8.28		1.30		8.3		sh	—		—		8.6	●
dash	0.5.10.2-6		*		0.5.10.2		soelim	1.22.3		1.19.2		*	
dc	1.4.1	○	1.3	● ○	1.3		sort	8.28		2.3		2.3	
dd	8.28		1.36		8.5		spell	1.0	○	—		—	
diff	3.6		2.8.1		2.8.7		split	8.28		1.17		8.2	
ed	1.10		*		1.5		strings	2.30		*		r3614M	
eqn	1.22.3		1.19.2		1.19.2		strip	2.30		*		r3614M	
ex/vim	8.0		8.1		8.1		sum	8.28		1.17		*	
expand	8.28		1.15		8.1		tail	8.28		101.40.1		8.1	
flex	2.6.4		2.5.35		2.5.37	●	tbl	1.22.3		1.19.2		1.19.2	
fmt	8.28		1.22		8.1		tcsh	—		6.21.00		6.20.00	
fold	8.28		1.13		8.1		tee	1.22.3		1.6		8.1	
ftp	0.17-34.1		—		8.6	●	telnet	1.14		1.16		8.4	
gcc	7.4.0		—		9.2.0		tex	6.2.3	●	6.2.3		6.2.1	
gdb	8.1.0	●	8.3.1	●	6.1.1	●	top	3.3.12		125		3.5beta12	
gfortran	7.4.0		—		—		tr	8.28		1.24		8.2	
grep	3.1		2.5.1		2.5.1		troff	1.22.3	●	1.19.2	●	1.19.2	●
grn	—		1.19.2		1.19.2		tsort	8.28		1.13		8.3	
groff	1.22.3		1.19.2	○	1.19.2	○	ul	*		101.40.1		8.1	
head	8.28		1.20		8.2		uniq	8.28		101.40.1		8.3	
htop	2.1.0		2.2.0		2.2.0		units	—		*		*	
indent	—		5.17	●	5.17	●	wc	8.28		1.21		8.1	
join	8.28		1.2		8.6		xargs	4.7.0		1.57		8.1	
less	551	●	487	●	530		zic	2.27		8.22		8.22	
lldb	—		9.0.1	●	8.0.0	●	zsh	5.4.2		5.7.1		5.7.1	●

87 utilities were tested on Unix, MacOS, and freeBSD, 67 of which were tested on all three systems.
● = crashed, ○ = hung, — = unavailable on that system, * = version information unavailable.

TABLE 2
Test Statistics for the 85 Total Utilities Tested

Platform	Linux	MacOS	FreeBSD
# tested	74	76	78
# failed	9	12	15
% failed	12%	16%	19%

TABLE 3
Comparison of Current and Previous Results

Utilities	Linux 1995	MacOS 2006	Linux 2020	MacOS 2020
as		●	○	
ctags	○			●
ex/vim		▼		
indent	● ○	●	—	●
latex/pdftex				▲
nroff		●		○
tex			▲	
troff		●	●	●
ul	▼	▼		

Comparation of utilities that were tested on Linux or GNU in 1995 [22], MacOS in 2006 [20], and in this current study, where each utility failed in at least one of the studies. Utilities that failed in previous studies but not in current study are highlighted in green (▼); utilities that did not fail in previous studies but failed in the current study are highlighted in red (▲).
●,▼,▲ = utility crashed, ○ = utility hung, — = utility unavailable on that system.

TABLE 4
List of Utilities that Failed, Categorized by Cause, Labeled by Platform

	Return Values	Pointers and Arrays	Error Handling	Sub-Process	Complex State	Other
as						LF
bash	M					
bison					LM	
calendar	M					
checknr		F				
col		F				
ctags		MF				
dc		M				LM
flex			F			
ftp	F					
gdb	LM		F			
groff				MF		
indent		MF				
less	M	L				
lldb	MF					
look					F	
make		F				
pdftex			M			
ptx					L	
sh	F					
spell				L		
tex			L			
troff	LMF					
zsh					F	

L = Linux, M = MacOS, F = FreeBSD.

its contents. Of course, it would help to have a language with proper string types and run-time checking of bounds.

Note that bugs that we describe in this section are distressingly basic. We will see more complex bugs in the subsequent sections.

Any unchecked assumptions about the structure of input can be dangerous, such as we see in make on FreeBSD. In function Parse_DoVar in parse.c, there is a for-loop that terminates when an "=" character is seen and when the number of right parentheses (or curly brackets) is greater than or equal to the number of left parentheses (or curly brackets).

```
for (depth = 0, cp = line + 1; depth > 0
     || *cp != '='; cp++) {
```

Our test random input had more left parentheses than right, so the loop did terminate before it went off the end of the buffer. This bug was introduced somewhat recently, in 2016.

The crash of ctags is another simple buffer overflow caused by a loop whose termination condition does not contain a check on the size of the buffer[3].

```
if (xflag)
  for (cp = lbuf; GETC(!=, EOF) && c != '\n';
       *cp++ = c) continue;
else for (cnt = 0, cp = lbuf; GETC(!=, EOF)
          && cnt < ENDLINE; ++cnt) {
```

Interestingly, this first for-loop lacks the proper check, while the for-loop that appears in the else-clause a few lines later does have the proper check. This bug appears to have been introduced in 1994.

The indent utility formats a C source code to generate a consistent style of indenting. When indent is processing a line that has a preprocessor command (starts with "#") followed immediately by a comment ("/*"), it first finds the end of the comment and then uses bcopy to copy the characters to its output buffer. The loop that finds the comment operates correctly, but in our test case, finds a comment whose length was longer than that of the output buffer. The code leading up to the bcopy never checked the length, resulting in a buffer overwrite. This bug appear to have been introduced in 1994 and fixed in 2018.

col is a utility that scans text for reverse line feeds caused by the vertical tab character, ^K, and eliminates these characters by reordering the text so that they are not necessary. If ^K is the first character in the input, there is an error condition that allows a null pointer to be dereferenced. The col utility uses a two-pass algorithm to process the text. On the second pass (in function flush_lines in file col.c), when ^K is the first character processed, a pointer that is used to move backwards has not yet been initialized, causing the crash. This bug appears to have been introduced in 2015.

The checknr utility scans documents written for nroff and troff to find unknown comments and mismatched opening and closing delimiters. For example, some macros must come in pairs, such as .TS and .TE (start and end table definition) and .EQ and .EN (start and end equation definition) When checknr finds an opening delimiter, it pushes it into a stack (stk) and then checks for a match when it finds the corresponding closing delimiter. However, the stack is of a fixed size (100, based on defined constant MAXSTK), with no checks to see if an overflow occurs. Our random input caused this stack to overflow, resulting in a write to an element beyond the bounds of the stk array. This bug appears to have been introduced in 1994.

The crash of less on Linux is an old fashioned double free. This crash happens when there is an attempt to overwrite an existing log file and the user elects not to do so. The buffer that holds the file name is deleted first in function use_logfile in edit.c and later in opt_o in optfunc.c. The second free corrupts memory causing a fault on an unrelated memory access (in our case, it was a call to getc). This bug appears to have been introduced in 2018 and quite recently.

[3] GETC(!=,EOF) is a macro that expands to c = getc(inf) != EOF.

4.2.2 Failure to Check Return Value

Not checking return value from system calls or functions is a basic programming mistake. Often times, this will be an indication of some implicit (rarely documented) assumptions about parameter values or the behavior of the function being called.

The troff text formatter is part of a suite of utilities that share common code. In code that selects a font, the random input can confuse the selection of fonts families and styles. This feature, when combined with trying to embed Postscript in the document will cause some table look-ups to unexpectedly fail. The constructor special_node::special_node calls env_definite_font, which has three separate places in which it checks for an error, causing env_definite_font to return -1. However, special_node::special_node does not check to see if env_definite_font returned an error:

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

So, when fontno is -1, it causes an out-of-bounds access in the next statement when it accesses font_table. From the gnu.org git repository, this bug seems to have been introduced in 2002.

lldb is a newer-generation debugger, built as part of the Clang/LLVM project and written in C++. Unfortunately, it has a crash due to an unchecked return value. When trying to evaluate a command expression, in this case `$0`, it first looks up the "Target", the object indicating which process is being debugged. However, for our random input, no valid target was previously defined.

The code in method Target::EvaluateExpression in file Target.cpp does the look up:

```
persistent_var_sp =
   GetScratchTypeSystemForLanguage
     (nullptr, eLanguageTypeC)
     ->GetPersistentExpressionState()
     ->GetVariable(expr);
```

This code is interesting and compact, where the return value from one method is immediately dereferenced to call another method (and this happens twice). The problem with such code is that the compactness does not allow for error checking of the intermediate values. In this case, our random input causes GetScratchTypeSystemForLanguage to fail, returning NULL, causing the subsequent dereference to fail. This bug appears to have been introduced in 1994 and fixed in the repository in 2019.

ftp is a key network utility that uses the popular libedit library to allow line editing of ftp command lines. Unfortunately, this library is not careful about how it handles command line parsing errors. In function tok_line in file tokenizer.c, if there is an unmatched ' (single quote) character, then a specific error value (1) is returned. In this error case, the parameters (confusingly) named argc and argv are not initialized. However, the return value from tok_line is never checked and then uninitialized pointer argv is subsequently dereferenced causing a crash. This bug seemed to have been recently introduced in 2019.

The shell sh uses the same libedit library and will fail under identical input.

4.2.3 Bad Error Handling

The crash of flex is an example of improper error handling (actually, a suppressed error), which is then followed by a loop with an insufficient termination condition. Strangely, this is another case of not dealing properly with matching parentheses. A variable _sf_top_ix tracks how many elements are on relatively small stack called _sf_stk. When a "(" is read, sf_push is called (and _sf_top_ix is incremented) and when a ")" is read, sf_pop is called (and _sf_top_ix is decremented). In our test case, flex reads more right parentheses than left parentheses, calling sf_pop to make _sf_top_ix zero, then decremented one more time. _sf_top_ix is declared as a size_t, which is an unsigned long integer on this platform, so additional decrement of _sf_top_ix results in the maximum positive value.

Interestingly, sf_pop has a check for valid values for_sf_top_ix:

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

There are two major problems with this check. First, the assert macro had a null definition on the failing platform:

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

As a result, _sf_top_ix becomes zero and the program silently continues. Next, _sf_top_ix is decremented again, this time wrapping to the largest positive number.

Second, even if the check was fixed to be effective, an assert rarely provides a reasonable or graceful form of error checking. We note that this bug was introduced in 2011 and was fixed in a later flex release but MacOS is still using the buggy version 2.5.35.

The crash of pdftex/latex is a case of complex processing that never leaves the error handler. Input was being processed in function getnext. Note that the structure of getnext is quite complex, including a switch-statement with 42 cases and cases with up to nine levels of nested if-statements. The random input (to standard input) had an invalid character followed by a sequence of letters. When getnext sees an invalid character, it returns and then expand is called, which then calls error to handle it. error then calls gettoken, which calls getnext, which calls error (again, recursively). error then calls terminput to get another command. This token happens to be "Q", which tells pdftex to go into "quiet" mode where it does not print error messages. To go into quiet mode, pdftex tries to change the selector variable that controls where the output goes by decrementing it. This is an overly clever way of suppressing printing, as a value of 17 for selector means to print to the terminal and 16 means to not print. Simply setting the variable to 16 would have been less fragile. Note that values 0–15 mean that output should to one of the file descriptors stored in the array writefile. While still in error, pdftex calls terminput to read the next command and print it. It does *another* decrement of the selector variable, which changes it from 16 (no printing) to 15 (printing to the 15th file descriptor. Unfortunately, there is no 15th open file, so writefile[15] is NULL, causing a dereference of a NULL pointer. The complexity of the error handling function and general use of huge switch-statements and highly nested if-statements is certainly a concern for this utility. This bug appears to have been introduced in 2017.

In the popular gdb debugger, the crash on FreeBSD provides an example of an *under-checked* return value. gdb uses the TUI (Text User Interface) to create windows within the terminal (shell) window, which is based on the curses library, This code controls which window to select, but not all the calls to look-up functions in this code have their return values checked. In our test case, in parse_scrolling_args in tui-win.c, the window look-up fails, returning NULL.

When the window name is not found, a NULL pointer is returned and assigned to win_to_scroll. However, the error check only prints out a warning message and allows the program to continue:

```
if (*win_to_scroll == ( . . . ) NULL
    || !(*win_to_scroll)->generic.is_visible)
  warning (" . . .  ");
```

Later, in tui_scroll_backward, this pointer is dereferenced causing the crash:

```
num_to_scroll=win_to_scroll->generic.height-3;
```

The bug was introduced with a commit in 1998 and corrected in 2019 in a commit more recent than the version we had tested.

4.2.4 Sub-processes

The text processing utility groff hangs because it calls grops as a sub-process (see Section 4.2.5 below).

Another case of vulnerable subprocesses occurs in spell on Linux. This utility failed on a relatively long line (more than 80,000 characters) without any newline character. In this case, spell creates a subprocess with which it communicates via pipes. A combination of errors and the long input string causes a deadlock on the pipe communication.

Spell creates three pipes for parent-child communication: to send input to the child, to receive output from the child, and to receive error messages from the child. This is a pretty conventional arrangement to provide stdin, stdout, and stderr to the child process.

The parent forks a child process that then calls execl to run ispell. Next, the parent sends input to the child through pipe1 and then waits to read errors or normal output. The input is quite long (longer than the pipe's buffer) and has a lot of erroneous data, so when the child writes to the stderr pipe, it eventually creates a deadlock. The child sees enough errors to fill up the pipe buffer for stderr, so cannot continue reading from the stdin pipe; and the parent is blocked on writing to the stdin pipe, so does not get to the read on the stderr pipe. The input line lengths from the parent and quantity of error messages from ispell are both unexpected behaviors that combine to cause the failure. This bug appears to have been introduced in 1997.

4.2.5 Complex State

In this current study, we caused several utilities to crash or hang, where the underlying cause was the tracking of complex state in a loop, often spread across several functions. Programmers develop complex criteria in their code that are not well documented nor symmetrically coded. As a result, unusual input sequences can cause unexpected patterns of execution.

The zsh shell will crash in its line editing mode given an unexpected sequences of commands. (Note that this is the third utility that we tested that crashed in code implementing command line editing. The other two utilities were another shell, sh, and ftp.)

Our random input triggered the line editing mode with the escape (^[) character. Once in line editing mode, the next character was a d to indicate a delete operation, but this character was not followed by a valid qualifier character to indicate what to delete. Instead, there was a ^P character, indicating that the line editor should move upward to the previous line of input. zsh handled this invalid sequence by moving to the

previous line but not properly updating the line length. Since the previous line was only one character long, the length field now points beyond the end of a valid string. The next character of input is a `j`, which tells the command line editor to move downward. This downward scanning code starts by trying to find the end of the current line in function `findeol` in file `zle_utils.c`. However, this scan starts from a point beyond the end of the string. This bug appears to have been introduced in 1994.

The `look` utility is a program that looks for lines with a specified string as a prefix; it will hang when invoked with a file whose first character is the NULL byte. There are several tests of conditions in the loop in function `compare` in `look.c`, where some of the tests are in four different `if` statements. Here, the input byte (the null character) does not fall into any of the expected categories: alphanumeric character, end of line, or end of file. As a result, the pointer to the buffer does not advance. This bug appears to have been introduced in 2004.

The well-known parser-generator `bison` will hang on extremely simple input, where the only character is a carriage return (`\r`). The hang occurs because of a common programming failing: not correctly handling input lines that do not end with a newline character. While bison is scanning input character-by-character, if it comes across this return character, it attempts to generate an error message, which results in calling down through six levels of error functions, ending up in `location_caret` in `location.c`. In `location_caret`, there is a loop that calls `getc` in an attempt to scan to the end of the line; this loop terminates when it finds a newline character. However, since the last available character has already been read, the `getc` function will always return a value of -1. The bug was introduced with a commit in 2012 and corrected in 2019 in a commit more recent than the version we had tested.

The `ptx` indexing utility has a function with similarly complex state. In function `define_all_fields`, there is a loop that separates an input line into components. There is a complex and inobvious relationship between the conditions that track which characters to parse and the movement of a pointer to the current starting point in the buffer. In this case, one component of the input line, the "reference", is longer than expected, causing a state where a pointer does not advance, so the loop hangs. This bug appears to have been introduced in 1998.

In Rust, `ptx` appeared to hang because of a N^2 algorithm in function `format_roff_line`. This function processes almost every substring on the input line that ends with the last character. Since our input was a single line of 93,290,954 bytes, N^2 is 8.7×10^{15}. We ran this test case for 34 hours and it finished 22% of the input, so we estimate completion time in 155 hours. While technically not a hang, this ran long enough that we considered it one.

There is a similar story for `fold` in Rust, where the apparent hang is caused by an algorithm complexity issue. For an input with M lines and N characters per line, the C version of `fold` has complexity of O($M \times N$), while the Rust version has complexity O($M^2 \times 2$). The result of this complexity issue is that some of our tests of `fold` will continue for more than 30 hours (the longest we ran a test).

Yet another similar hang occurs in `groff` when it sends data to the `grops` utility that formats groff output to PostScript. The code that sets line width ends up in function `get_possibly_integer_args` in file `ptx.c`. This function contains a loop that makes up most of the function. In this loop, state is tracked in a sequence of `if` statements followed by a `switch` statement. The random input included a `D` (draw line) command with the `t` (thickness) qualifier. At this point, the code is looking for an integer thickness, but unexpectedly finds a letter (`n` in our random input) and reaches a state where it can never satisfy the loop termination condition. This bug appears to have been introduced in 2002.

4.2.6 Others

Two programs, `dc` and `as` can be caused to loop for so long that their behavior is practically indistinguishable from an infinite loop (hang).

`dc` is a reverse-polish calculator that supports arbitrary-precision arithmetic; it uses the same arbitrary-length numeric functions as `bc`. Our testing input caused an *apparent* hang, where the `bc_sqrt` function in `numeric.c` was called with a parameter that indicated that the number had 759,375 digits after the decimal point. After subsequent tests, we saw that this case actually finishes after six hours (so it might be worth investigating their implementation of the Newton Raphson Method).

The GNU assembler `as/gas` is another interesting and strange example of a utility looping for an extremely long time. The assembler allows expressions in instructions and directives that have forward references to symbols defined later in the code. To resolve forward references, it uses a multi-pass algorithm that stops when it reaches a fixed point (no symbols that has been used previously changes value). To prevent an infinite loop when there is a cycle in the expressions, they limit the loop to execute N^2 times, where N is the number of "fragments" in the code, where a fragment typically corresponds to a line of assembly code. Each time through this loop, the assembler goes through the entire list of fragments (so this actually has a complexity of N^3).

Our random input happened to contain a cycle that is effectively a sequence like:

```
    .org  a
    .byte 0
  a:
```

Note that our random input actually had sequence:

```
    .=Ñ ... JG ... Ñ:
```

where the `.=` is equivalent to `.org`. The total size of our random input was quite large, so we estimate that it would take almost a year before the N^2 iterations would be complete. An algorithm that directly detects cycles in the expression would be clearer and less prone to unpredictable behaviors.

5 Discussion

In 1990 fuzz testing was unknown, so finding a lot of bugs that caused crashes or hangs was not (in retrospect) surprising. In 1995, well after the results were published and the tool made public, the failure rate was still significant. Even at that point in time, many people in the testing community found the technique to be suspect at best. Fast forward to 2020, where fuzz testing in its various guises is widely used by both the testing and security communities, and where everyday, the tools get smarter and more capable, we are still seeing failure rates from 12% to 19% with the original simple methods. In 2006, we wrote: *Plus ça change, plus c'est la même chose,* and this still appears to be valid.

In this section, we first discuss some issues that came up during testing and then present a commentary on the results.

5.1 Challenges in Bulk Testing

The goal of our testing process was to proceed in the most automated way possible. That means making it easy to set up the tests for a large number of utilities and running the tests with minimum human intervention. There were a few issues that made this goal more challenging.

5.1.1 Avoiding Fatal Side Effects

Utility programs can sometimes have side effects on the system or other processes. Some sides, like creating or removing a file,

are easy to deal with, while others are more difficult. For example, in our early fuzz studies, we learned the obvious-when-you-think-about-it lesson that you cannot test the `shutdown` utility. If you run as a normal user, it does nothing; if you run as root, it takes down the entire system. More subtly, utilities like `top` and `htop` can terminate other processes. During random testing, we had these programs kill the fuzz tools or important other processes like the current login shell. Perhaps testing in a container could help to isolate these affects, something that we plan to do in the future.

5.1.2 Hanging or Just Waiting?

Some utilities start other programs, and if the utility waits for the new program to complete, it is difficult to distinguish a hang from a program waiting to be told to terminate. Of course shells do this, but there are more subtle cases such as `less`, which can start an editor on the file being inspected.

5.1.3 What You See Is Not Always What You Get

Another issue that came up is determining exactly what we were testing. Things are not always what they appear. For example, the command `cc` on our Linux platform is a symbolic link to `gcc` but on MacOS is a symbolic link to `clang`. On MacOS, `gcc` is an actual binary for `clang`. However, if you run `gcc` on MacOS, you get `clang` version 11.0.0 while if you explicitly run `clang`, you get version 9.0.1. For the `sh` command on Linux, you get a symbolic link to `dash`, while the `sh` command on MacOS is the actual `bash` binary and on FreeBSD it is the actual `sh` binary. On MacOS, there is also a binary named `bash`, which is the same 3.2.57(1)-release version as `sh`. On MacOS, the `latex` binary is actually `pdftex` (a related but not identical program), And on MacOS and FreeBSD, the `more` binary is actually `less` (again, a related but not identical program). The moral is that on any given platform, you need to check carefully as to what program will run for each command name.

Determining the source of the code and the version number can sometimes be challenging. For many utilities, you can simply run the program with the `-v` or `--version` option. For others, you have to look at the repository or into the executable file, perhaps using the `strings` utility to look for a version string. Other utilities, such as `calendar` and `col` on Linux, simply defeated our ability to find the version.

5.1.4 Testing Pseudo-TTY Programs

A couple of additional issues came up when testing programs meant to run in terminal windows. First, because an end-of-file indication for the random input does not propagate through the pseudo-tty, it is difficult to distinguish a program waiting for more input after the end of random input from a program that is hung. This has been an issue since the first fuzz testing study. To avoid this case, for each program tested with ptyjig, we specify a string to append to the random input to attempt to terminate the utility. For example, when testing `vim`, we append the sequence "ESC `:q!`".

Second, if our test input contains certain control sequences such as `^C`, the testing process can inadvertently kill the utility being tested. We avoid this case by eliminating such characters from our random input files.

5.2 Commentary on Results

The utilities that we tested are a dynamic and changing body of code. So, we expect to see the introduction of new bugs; no software release is perfect and bug-free. However, the difference between now and the early fuzz studies is that fuzz testing is now a widely known and used technique. And, while there is a definite benefit and attraction from using the newest in fuzz testing technology, there appear to be some advantages to not discarding the quick and dirty approach. In 1990, we wrote:

> *Our approach is not a substitute for a formal verification or testing procedure, but rather an inexpensive mechanism to identify bugs and increase overall system reliability.*

This still appears to be true, both considering basic fuzz testing to be an easy and effective mechanism and an addition – not a replacement – for other forms of testing.

While many programmers still find C to be a fun and satisfying language to use, it is well known to be hazardous and has been widely decried by both the software engineering and security communities. The continued presence of failures due to misuse of pointers and arrays (Section 4.2.1) is evidence that programmers today are just as vulnerable to these issues as they were 30 years ago. We need to be using better programming languages.

Some of the world is skeptical. In an online forum [31], we found this strident but pointed commentary:

> *Oh sure, let's rewrite well tested utilities written in C in randomly popular language of the month. You'd spend quite a bit of time rewriting them, then significantly more time testing them, and then more time each time you use them because they'd more than likely run slower. And of course, as they wouldn't be nearly as mature, you'd regularly encounter bugs and/or exploits (and yes, languages that aren't C are still susceptible to those). So after all that is done, you get a grep that is slower, more likely to crash and overall less likely to work as intended, but you can at least fill the readme with buzzwords.*

Our test results for Rust were not as doom-filled as this poster suggested, but they were not as good as we hoped. However, the Rust versions were as fast (or faster) than the standard utilities. It is interesting to note that all the Rust failures were hangs and not crashes. These results are limited because we could not find Rust versions of the more complex utilities. As a community, we truly need to move away from our legacy C code base, and give the newly rewritten versions time to mature. And, we need to continue to test these new versions as they are updated.

The lack of uniform application of well-understood good programming practices is still problematic. We still too frequently find such things as not checking for return values (Section 4.2.2) and careless error handling (Section 4.2.3). These are exactly the kinds of errors that we have found in our in-depth software assessment activities [5] and that we teach about in our Introduction to Software Security course [7]. Computer science curriculums need to emphasize these practices at every level and in every area, not just in security or software engineering classes.

Two new categories of errors appeared in our current study. The first new category, which we called "Complex State", showed programmers packing more and more complex conditions and state into a loop. These loops are often many hundreds of lines long, crossing function boundaries, with embedded switch-statements and many-nested if-statements. Some of this complexity is caused by the incremental accretion of features, without any effort to clean up, unify, or refactor the code. Some are just the product of programmers who cannot see the usefulness of simplicity in their code structure.

The second new category was caused by programs adding line-editing and history functionality into their utilities. This occurred in `ftp`, `sh`, and `zsh`. The authors of the Plan 9 operating system [25], which was considered the ultimate successor to Unix, factored out this functionality into their window system. The result was a single system implementation of the functionality, where individual programs would not be tempted to reimplement it.

The good news is that several categories of errors that appeared in previous studies did not appear in our current

one. These are crashes related to end-of-file handling, divide-by-zero, signed characters, and dangerous input functions. So, the word has gotten out on some of these problematic programming practices.

We also note that the Unix world is blessed by too much of a good thing. As you look at Table 1, you can see that we often found a different version of a utility on each platform that we tested. And utilities that are maintained on one version of Unix are often independent from those on other versions. It would be helpful if there was a common code base for each utility, however that is complicated by differences in operating system library and system call interfaces. It is also complicated by changes in language version that may be adopted at different times by different operating systems (or even different distributions of the same operating system).

Some of the errors that we found have been present in the code for many years, as far back as 1994 for `checknr`, `ctags`, `dc`, and `indent`; 1997 for `spell`; and 1998 for `gdb` and `ptx`. More frequent application of the basic fuzz tests could help to avoid this situation. And a few errors had been fixed in newer versions of the software than we tested. Putting together a operating system release is a complex process, and there can be delays in bringing the latest version of utility into a release. As with the multiplicity of versions, delays can be influenced by changes in language versions and changes in operating system library and system call interfaces.

6 Conclusion

After more than thirty years, it appears that there is still a place for this type of basic fuzz testing. Standard operating system utilities are still crashing at a noticeable rate and not getting better over time. And errors seem to persist in the code bases for too long. If this testing was integrated to the release process of these operating systems, most of these failures could have been avoided. Such testing should be paired with careful functional testing and modern fuzz tools.

The prevalence of errors based on pointers and arrays is not surprising. As long as we use languages like C with inherently unsafe constructs, it will be difficult to eliminate these errors. However, as we have found in our security studies, and we can see in this current fuzz study, there are other categories of errors that can happen in any language. Good design, good education, ongoing training, testing integrated into the development cycle, and (perhaps the most important) a culture that promotes and rewards reliability are all essential to making real progress here.

Acknowledgments

The pioneering fuzz studies done at the University of Wisconsin-Madison were all conducted as semester projects in Miller's graduate Advanced Operating Systems (CS736) course. The work of the students who conducted those studies went well beyond the reaches of their classroom and continues to influence software testing and security practices to this day. These students were: Lars Fredriksen and Bryan So (1990); David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Nataranjan, and Jeff Steidl (1995); Justin Forrester (2000); and Gregory Cooksey and Frederick Moore (2006).

In this study, we would like to thank Yi (Emma) He for her efforts in helping to revise the fuzz tools and run the initial experiments. We also thank James Kupsch and Josef (Bolo) Burger for helping to find an obscure bug in ptyjig.

References

[1] ANSI C. https://en.wikipedia.org/wiki/ANSI_C.
[2] Language Standards Supported by GCC. https://gcc.gnu.org/onlinedocs/gcc/Standards.html.
[3] Rust coreutils Repository. https://github.com/uutils/coreutils.
[4] A. Donovan and B. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.
[5] J. Eichenhofer, E. Heymann, B. Miller, and K. A. An In-Depth Security Assessment of Maritime Container Terminal Software Systems. *IEEE Access*, 8, July 2020.
[6] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications using Random Testing. In *4th USENIX Windows System Symposium*, Seattle, September 2000.
[7] E. Heymann, B. Miller, and L. Kohnfelder. Introduction to Software Security, 2020. https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/.
[8] Icamtuf. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.
[9] A. Isola. Go coreutils Repository. https://github.com/aisola/go-coreutils.
[10] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2019.
[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, October 2018.
[12] Kplybon. ClusterFuzz. https://google.github.io/clusterfuzz/.
[13] J. Kupsch and B. P. Miller. Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed? Technical Report WP003, Software Assurance Marketplace (SWAMP), University of Wisconsin-Madison, April 2014. https://www.swampinabox.org/doc/SWAMP-WP003-Heartbleed.pdf.
[14] E. Lagergren. Go coreutils Repository. https://github.com/ericlagergren/go-coreutils.
[15] J. Li, B. Zhao, and C. Zhang. Fuzzing: A Survey. *Cybersecurity*, 1, June 2018.
[16] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, September 2018.
[17] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, October 2019.
[18] M. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Trans. on Computer Systems*, 2(3):181–197, 1984.
[19] B. Miller. Foreward to the 1st edition. In A. Takanen, J. DeMott, C. Miller, and A. Kettunen, editors, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 1 edition, 2008.
[20] B. P. Miller, G. Cooksey, and F. Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *1st international workshop on Random testing*, Portland, Maine, July 2006.
[21] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.
[22] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report 1268, Computer Sciences Department, University of Wisconsin-Madison, 1995.
[23] N. I. of Standards and Technology. CVE-2019-18634, January 2020. https://nvd.nist.gov/vuln/detail/CVE-2019-18634.
[24] I. OpenSSL Foundation. OpenSSL. https://www.openssl.org/.
[25] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computer Systems*, 8(3):221–254, Summer 1995.
[26] L. C. I. Project. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[27] R. Rymon, A. Cohen, and R. Sass. Whitesource bolt. https://bolt.whitesourcesoftware.com/.
[28] R. Scheifler. X Window System Protocol. Technical Report X Version 11, Release 7.7, X Consortium Inc., 2004. https://www.x.org/releases/current/doc/xproto/x11protocol.html.
[29] E. Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):672–676, 1989.
[30] R. Swiecki. honggfuzz. https://honggfuzz.dev/.
[31] R. Velting. Would rewriting Unix utilities in modern programming languages, e.g. Go, Rust, Scala, Kotlin, Swift, Erlang, and Elixir, provide any good benefits, July 2017. https://www.quora.com/Would-rewriting-Unix-utilities-in-modern-programming-

languages-e-g-Go-Rust-Scala-Kotlin-Swift-Erlang-and-Elixir-provide-any-good-benefits.
[32] M. Vimpari. An Evaluation of Free Fuzzing Tools, 2016. http://jultika.oulu.fi/files/nbnfioulu-201505211594.pdf.

Elisa R. Heymann is a Senior Scientist at the NSF Cybersecurity Center of Excellence at the University of Wisconsin-Madison and an Associate Professor at the Autonomous University of Barcelona. She was also in charge of the Grid/Cloud security group at the UAB and participated in major Grid European Projects. Dr. Heymann's research interests include software security and resource management for Grid and Cloud environments. Her research is supported by the NSF, Spanish government, the European Commission, and NATO.

Barton P. Miller is the Vilas Distinguished Achievement Professor and Amar & Belinder Sohi Professor of Computer Sciences at the University of Wisconsin-Madison. From 2013-2020, he was Chief Scientist for the DHS Software Assurance Marketplace research facility and is Software Assurance Lead on the NSF Cybersecurity Center of Excellence. In addition, he co-directs the MIST software vulnerability assessment project in collaboration with his colleagues at the Autonomous University of Barcelona. He also leads the Paradyn Parallel Performance Tool project, which is investigating performance and instrumentation technologies for parallel and distributed applications and systems. His research interests include systems security, binary and malicious code analysis and instrumentation of extreme scale systems, and parallel and distributed program measurement and debugging. Miller's research is supported by the U.S. Department of Energy, National Science Foundation, NATO, and various corporations. In 1988, Miller founded the field of fuzz random software testing, which is the foundation of many security and software engineering disciplines. In 1992, Miller (working with his then-student, Prof. Jeffrey Hollingsworth), founded the field of dynamic binary code instrumentation and coined the term "dynamic instrumentation". Dynamic instrumentation forms the basis for his current efforts in malware analysis and instrumentation. He is a Fellow of the ACM.

Mengxiao Zhang is a masters degree student in Computer Sciences at the University of Wisconsin-Madison. He earned his bachelor's degree in Computer Science and Engineering from the Huazhong University of Science and Technology in 2019. During the masters study, he assisted in research related to software vulnerability assessment. He is now a developer of ImageJ, an open platform for scientific image analysis.