Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services

Barton P. Miller
bart@cs.wisc.edu

David Koski
dkoski@cs.wisc.edu

Cjin Pheow Lee
cjin@cs.wisc.edu

Vivekananda Maganty
vivek@cs.wisc.edu

Ravi Murthy
ravim@cs.wisc.edu

Ajitkumar Natarajan
ajitk@cs.wisc.edu

Jeff Steidl
steidl@cae.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706-1685

Abstract

We have tested the reliability of a large collection of basic UNIX utility programs, X-Window applications and servers, and network services. We used a simple testing method of subjecting these programs to a random input stream. Our testing methods and tools are largely automatic and simple to use. We tested programs on nine versions of the UNIX operating system, including seven commercial systems and the freely-available GNU utilities and Linux. We report which programs failed on which systems, and identify and categorize the causes of these failures.

The result of our testing is that we can crash (with core dump) or hang (infinite loop) over 40% (in the worst case) of the basic programs and over 25% of the X-Window applications. We were not able to crash any of the network services that we tested nor any of X-Window servers. This study parallels our 1990 study (that tested only the basic UNIX utilities); all systems that we compared between 1990 and 1995 noticeably improved in reliability, but still had significant rates of failure. The reliability of the basic utilities from GNU and Linux were noticeably better than those of the commercial systems.

We also tested how utility programs checked their return codes from the memory allocation library routines by simulating the unavailability of virtual memory. We could crash almost half of the programs that we tested in this way.

Content Indicators: D.2.5 (Testing and Debugging), D.4.9 (Programs and Utilities), General terms: random testing, reliability, UNIX.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), NSF Grants CCR-9100968 and CDA-9024618, Department of Energy Grant DE-FG02-93ER25176, and Office of Naval Research Grant N00014-89-J-1222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 INTRODUCTION

In 1990, we published the results of a study of the reliability of standard UNIX utility programs[2]. This study showed that by using simple (almost simplistic) random testing techniques, we could crash or hang 25-33% of these utility programs. Five years later, we have repeated and significantly extended this study using the same basic techniques: subjecting programs to random input streams. A distressingly large number of UNIX utilities still crash with our tests.

The essence of our testing is a program called the *fuzz generator* that emits various types of random output streams. These random streams are fed to a wide variety of UNIX utilities. We use a conservative and crude measure of reliability: a program is considered unreliable if it crashes with a core dump or hangs (loops indefinitely). While this type of testing is effective in finding real bugs in real programs, we are not proposing it as a replacement for systematic and formal testing. To quote from the 1990 study:

> There is a rich body of research on program testing and verification. Our approach is not a substitute for formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

Our new study has four parts:

1. Test over 80 utility programs on nine different UNIX platforms, including three platforms tested in the 1990 study. Seven of these platforms come from commercial vendors. These tests are the same type as we conducted in 1990, including the use of the same random streams used in the 1990 study *and* streams newly generated for the current study. As in 1990, we identified and categorized the bugs that caused the failures.
2. Test network services by feeding them random input streams from a fuzz-based client.
3. Test X-window applications and servers by feeding them random input streams.
4. Additional robustness tests of UNIX utility programs to see if they check the return value of system calls. Specifically, we tested calls to the memory allocation C library routines (the malloc() family), simulating the unavailability of additional virtual memory.

The goal of these studies was to find as many bugs as possible using simple, automated techniques. These techniques are intended to expose errors in common programming practices. The major results of this study are:

- ❑ In the last five years, the previously-tested versions of UNIX made noticeable improvements in the reliability of their utilities. But . . .

  . . . the failure rate of these systems is still distressingly high[1].
- ❑ Even worse is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995.
- ❑ The failure rate of utilities on the commercial versions of UNIX that we tested (from Sun, IBM, SGI, DEC, and NEXT) ranged from 15-43%.
- ❑ The failure rate of the utilities on the freely-distributed Linux version of UNIX was second-lowest, at 9%.

1. In this paper, "failure" means a crashing with core dump or hanging (looping indefinitely).

- ❏ The failure rate of the public GNU utilities was the lowest in our study, at only 6%.
- ❏ We could not crash network services on any of the versions of UNIX that we tested.
- ❏ Well more than half of the X-Window applications that we tested crash on random input data streams. More significant is that more than 25% of the applications crash given random, but legal X-event streams.
- ❏ We could not crash the X server on the versions of UNIX that we tested (by sending random data streams to the server).

Section 2 describes the basic tests that we performed on UNIX utilities and the results of those tests. We analyze and categorize the causes of each failure and compare them with the results from three systems that also were tested in 1990. Section 3 reports on our testing of network services and Section 4 reports on testing of X-window applications and servers. Memory allocation library-call tests are described in Section 5. Section 6 presents concluding comments.

2 BASIC TESTS

This section reports on the results of repeating our basic 1990 study. That study tested the reliability of utilities by feeding several variations of random input streams. In the 1990 study, we tested a large number of UNIX utilities on six vendors' platforms (plus a limited amount of testing on a seventh platform). Our current study included the same type of tests on nine UNIX platforms, including seven commercial systems. Three of the systems from the 1990 study are included in our current (1995) study.

These tests were performed on machines available on the University of Wisconsin-Madison campus in the Fall of 1994 and machines belonging to members of the testing team. The tests were repeated and verified in the Winter of 1994-95.

Section 2.1 describes the tools used for the basic tests and Section 2.2 describes the tests themselves and the UNIX platforms on which they were run. The platforms include 1990-versions of three systems and the 1995-versions of all nine systems. Section 2.3 presents the results of our testing and describes the causes of the failures detected by our tests.

2.1 Basic Fuzz Tools

The *fuzz* program is basically a generator of random characters. It produces a continuous string of characters on its standard output file. We can perform different types of tests depending on the options given to fuzz. Fuzz is capable of producing both printable and control characters, only printable characters, or either of these groups along with the NUL (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, to provide for repeatable tests.

Fuzz can record its output stream in a file, in addition to printing to its 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.

The following is an example of fuzz being used to test "eqn", the equation processor.

```
fuzz 100000 -o outfile | eqn
```

The output stream will be at most 100,000 characters in length and the stream will be recorded in file "outfile".

The fuzz tools also include a set of shell scripts that automate much of the testing process. If a crash is detected (by the presence of a "core" file), a "crash" is recorded in a log file. If the test is interrupted by the person performing the tests, a "hang" is recorded in the log file. Each crash and hang was subsequently examined to ensure that they were valid. For example, core files can also be generated when the program received a SIGABRT signal (typically generated by calling the abort() library routine); such cases were not considered crashes.

2.2 Basic Tests

We tested UNIX utilities on nine operating system platforms. Seven of these systems (SunOS, HP-UX, AIX, Solaris, IRIX, Ultrix, and NEXTSTEP) were the most recent commercial software distributions that we had available to us at the time of the tests. Three of these commercial systems (SunOS, HP-UX, and AIX) were also tested in the 1990 study. Results from this earlier

Identifying Letter	Study	Vendor	Architecture	Operating System
s	1990	Sun Microsystems	Sun 4/110	SunOS 3.2 and 4.0
S	1995		SPARCstation 10/40	SunOS 4.1.3
h	1990	Hewlett Packard	HP 9000/330	4.3 BSD + NFS + System V
H	1995		HP 9000/705	HP-UX 9.01
a	1990	IBM	PS/2-80	AIX 1.1
A	1995		RS6000	AIX 3.2
O	1995	Sun Microsystems	SPARCstation 10/40	Solaris 2.3
I	1995	Silicon Graphics	Indy	IRIX 5.1.1.2
U	1995	DEC	DECstation 3100	Ultrix v4.3a rev 146
N	1995	NEXT	Colorstation (MC68040)	NEXTSTEP 3.2
G	1995	GNU, Free Software Foundation		SunOS 4.1.3 & NEXTSTEP 3.2
L	1995	Linux	Cyrix i486	Slackware 2.1.0

Table 1: List of Systems Tested

study [2] are included for comparison. Two of the systems tested are free software distributions. The GNU tools come from the Free Software Foundation and are written by a variety of authors world-wide. Linux is a freely distributed version of the UNIX operating system originally written by Linus Torvalds; the system has since been extensively changed and extended by authors world-wide. The systems tested are listed in Table 1.

Each utility was tested with several random input streams. These streams were varied by combinations of the parameters described in Section 2.1. We tested the utilities with streams generated by the same random seeds as were used in the 1990 study and by several new random seeds.

2.3 Basic Test Results

In our current study, we tested more than 80 UNIX utilities. Each utility is available on at least three (and typically more) of the systems that we tested. The list includes commonly used utilities, such as shells, C compilers, and document formatters. The list also includes more obscure utilities, such as “units”, the unit conversion utility. Similar utilities are grouped under the same name in Table 3 and Table 4. When the name of a utility is different from the name given in these tables, the specific name is listed in Table 2.

Section 2.3.1 summarizes the results of the basic testing. Section 2.3.2 examines the cause of the failures, and Section 2.3.3 compares our current results to those of the 1990 study. The basic tests do *not* include X window-based utilities. Tests on X window-based application programs are described in Section 4.

2.3.1 Quantitative Results

The results of our tests are given in Table 3. Two immediate observations are possible from exam-

Generic Name(s)	Irix	Ultrix	NEXT	GNU	Linux
as			gas	gas	
awk				gawk	
bib/bibtex					
cc			gcc	gcc	
ccom	cfe	cfe	cc1obj	cc1	
compress				gzip	
dbx			gdb	gdb	gdb
ditroff/troff			ptroff		
eqn/deqn		neqn	neqn	geqn	
ex/vi					
lex				flex	flex
more				less	
plot			psplot		
sh				bash	
soelim				gsoelim	
tbl/dtbl				gtbl	
yacc				bison	

Table 2: Similar Utilities.
The utilities listed on the same line of the table have similar function, but vary by name in the systems listed. They are not meant to be identical, but rather are listed for comparison purposes.

ining this table. First, there is a noticeable improvement in reliability from the 1990 study: the failure rate for SunOS went from 29% to 23%, HP-UX went from 33% to 18%, and AIX went from 24% to 20%. Second, the 1995 failure rate is still distressingly high, especially given the ease of the fuzz testing and the public availability of the fuzz tools.

It is also interesting to compare results of testing the commercial systems to the results from testing "freeware" GNU and Linux. The seven commercial systems in the 1995 study have an average failure rate of 23%, while Linux has a failure rate of 9% and the GNU utilities have a failure rate of only 6%. It is reasonable to ask why a globally scattered group of programmers, with no formal testing support or software engineering standards can produce code that is more reliable (at least, by our measure) than commercially produced code. Even if you consider only the utilities that were available from GNU or Linux, the failure rates for these two systems are better than the other systems.

One explanation may be that the scale of software that must be supported by a large computer company is more extensive than that of the free software groups. Companies have many more customers and a commitment to support software on many platforms, configurations, and versions of an operating system (especially on older versions of hardware and system software).

Utility	SunOS		HP-UX		AIX		Solaris	Irix	Ultrix	NEXT	GNU	Linux
	90	95	90	95	90	95	95	95	95	95	95	95
adb	●	●	●	●❍	×	×	●❍	×	×	×	×	×
as					●					●		
awk												
bc										●		×
bib		●	×		×	×	×	×			×	
calendar											×	×
cat												
cb			●		❍	❍			●	●	×	×
cc										●		
ccom					×	×	❍			●		×
checkeq				×		●		×			×	×
checknr					×			×			×	×
col	●	❍	●		●	●	●	●	❍	●	×	
colcrt				×	×	●	×	×			×	
colrm				×	×		×	×			×	
comm												
compress					×							
cpp												
csh	❍		❍		❍						×	
ctags	×		×		×	×	●			●		❍
ctree	×	×	×	×	×		×	×			×	×
dbx	●		×							×		●
dc								●		●	●	×
deroff	●	●	●		●		●		●	●	×	×
diction	×	●	●	●	×		×	×	●	●	×	×
diff												
ditroff	●	●	×	●		●	●		●	●		
eqn	●	●	●	●		❍		●	●	●		×
ex			●								×	

Table 3: List of Utilities Tested and Results of Those Tests
● = crashed, ❍ = hung, × = not available

Utility	SunOS 90	SunOS 95	HP-UX 90	HP-UX 95	AIX 90	AIX 95	Solaris 95	Irix 95	Ultrix 95	NEXT 95	GNU 95	Linux 95
expand					✕			✕				
f77			✕		✕			✕		✕	✕	✕
fmt										●	✕	
fold					✕							
ftp	●	●	●		●	●	●	●	●	●	✕	
graph				✕	✕			✕		✕		✕
grep												
head					✕							
indent	●❍	❍	●	❍	✕	●	❍			●	❍	●
join	●						●		●	●		
latex			✕		✕	✕	✕	✕			✕	
lex	●	●	●	●	●	●			●	●	●❍	●
lint										✕	✕	✕
look	❍		●	❍	✕		❍	✕	●	●	✕	
m4				●					●	●		
mail											✕	
Mail	✕		✕		✕			✕		●	✕	✕
make			●									
more					✕							
nm												
nroff		●		●		●❍	●	●	●	●		
pc					✕			✕		✕	✕	✕
plot	❍	✕	●	✕	✕	✕	●❍	✕		❍	✕	✕
pr												
prolog	●❍		●❍	✕	✕	✕	✕	✕	✕	✕	✕	✕
psdit					✕		✕				✕	✕
ptx	●	●	●	●		●	✕					✕
refer	●		●	❍	✕	●❍		✕	●	●		
rev					✕		✕	✕			✕	

Table 3: List of Utilities Tested and Results of Those Tests
● = crashed, ❍ = hung, ✕ = not available

Utility	SunOS 90	SunOS 95	HP-UX 90	HP-UX 95	AIX 90	AIX 95	Solaris 95	Irix 95	Ultrix 95	NEXT 95	GNU 95	Linux 95
sed												
sh												
soelim					✕							✕
sort												
spell	●	●	●		●		●		●	●	✕	✕
spline				✕	✕			✕		●		✕
split												
strings					✕		●					
strip												
style	✕	●	●	●	✕		✕	✕		●	✕	✕
sum												
tail												
tbl												
tee												
telnet	●	●	●		●	●		●	●	●	✕	
tex			✕		✕	✕	✕	✕			✕	
tr												
tsort	●		●		●					●	✕	
ul	●	●	●	●	✕	●	●	●	●	●	✕	●
uniq	●	●	●		●			●	●	●❍		
units	●	●	●		●	●	●	●		●	✕	✕
vgrind			✕		✕					●	✕	✕
wc												
yacc												
# tested	77	80	72	74	49	74	70	60	80	75	47	55
# crash/hang	22	18	24	13	12	15	16	9	17	32	3	5
%	29%	23%	33%	18%	24%	20%	23%	15%	21%	43%	6%	9%

Table 3: List of Utilities Tested and Results of Those Tests
● = crashed, ❍ = hung, ✕ = not available

The elitist explanation is that there is no substitute for excellent programmers. The people who write and support the GNU utilities are typically skilled at their profession and are in it for the fun and intellectual challenge (that is not to say that such people do not exist in corporate software groups!). The users of the GNU utilities are, in many cases, of a like mind; if a bug exists, they will often find and fix it themselves. Having ubiquitous source code is certainly an advantage in this situation.

The free software also has a personal touch that improves communication between the authors and the users. Users have an incentive for reporting bugs. Usually the GNU utilities have an individual's name associated with them. If you have a bug report, you send it to a person (who typically answers you and shows a personal interest in your report). Large companies usually require you to submit a bug report to an anonymous address, something like `OSbugs@BigCompany.com.` Responses to such mail are slow and the user may never see or hear about the resolution of the bug. The structure of the corporate software development, testing, and release process is such that it may be a year before a repair is made available. The long delays can also be discouraging to the programmer. By the time that a bug-fix is deployed, the programmer has long forgotten the problem; there is no gratification in solving a particular person's problem.

Distributing source code may also be a factor in quality of the GNU and Linux systems. Users are more able and encouraged to be involved in identifying the cause of problems. Computer vendors may be under-estimating the value of widely distributing their source code.

The lessons learned by traditional manufacturing industries, such as steel and automobiles, may need to be learned by the computer industry: flexibility and responsiveness is a major key to long term survival.

2.3.2 Causes of Crashes/Hangs

As in the 1990 study, we examined each program that crashed or hung to identify the cause of the failure. Source code was available to us for utilities on SunOS, Solaris, Ultrix, HP-UX, GNU, and Linux; source code was not available on NEXTSTEP, AIX, or Irix. For each program failure on a system for which we had source code, we categorized the cause; these results are reported in Table 4. The letters in the table entries describe the systems to which the entry applies. The identifying letters are given in the first column of Table 1. For the most common failures, we describe details of the causes.

Pointer/Array

Errors in the use of pointers and array subscripts dominate the results of our tests. These are errors any novice programmer might make, but surprising to find in production code. In all these cases, the programmer made implicit assumptions about the contents of the data being processed; these assumptions caused the programmer to use insufficient checks on their loop termination conditions. The presence of these errors argues (minimally) for garbage-collected languages and full-time array bounds checking to help a sloppy programmer detect these problems.

Most of the pointer errors found in the 1995 study were simple: increment the pointer past the end of an array. The error in “ctags” is representative (file “ctags.c”):

```
char line[4*BUFSIZ];
...
sp = line;
...
do {
      *++sp = c = getc(inf);
} while ((c != '\n') && (c != EOF));
```

Note that the termination condition in the above loop does not include any tests based on the size of array (`line`) being used.

Array subscripting errors were also a common cause of failures in this study. Most of these errors appeared in routines that were using character input or were scanning an input line. An example of an error during input appears in “cb” (file “cb.c”):

```
char string[200];
...
while ((cc = getch()) != c) {
      string[j++] = cc;
      ...
}
```

The termination condition on the above loop checks the input operation, but ignores the size of the buffer into which the data is being read (`string`). Another common type of error happens during string processing, where (again!) the termination condition of the loop does not contain a check on the size of the array. The following example comes from “bibtex” (file “strpascal.c”):

```
void
null_terminate(s)
char *s;
{
      while (*s != ' ') s++;
      ...
}
```

Dangerous Input Functions

The second most common cause of errors was the use of dangerous input functions, such as the notorious gets() function. The problem is that gets() has no parameter to limit the length of the input data. Besides causing reliability problems, use of gets() was also the flaw that permitted a major breach in Internet security[3,4]. By using gets(), the programmer is making implicit assumptions about the structure of the data being processed.

The manual page from the Solaris 2.3 system wisely contains the following warning:

> When using gets(), if the length of an input line exceeds the size of s, indeterminate behavior may result. For this reason, it is strongly recommended that gets() be avoided in favor of fgets().

The fgets() function includes an argument to limit the maximum length of the input line.

The C library input/output routines are not integrated into the language; they appear only as a collection of procedures to be called. Newer languages, like C++, can do a better job by integrating the input operation into the definition of a new type or class (using the “>>” operator). But the definition of the “>>” operator for character strings does ***not*** include information about the length

Utility	Cause						
	Array/ Pointer	Input Functions	Signed Characters	Divide by Zero	EOF Check	Others	No Source Code
adb	sShHO						
as	a						N
bc							N
bib	S						
cb	haU						AN
cc							N
ccom							ON
checkeq							A
col	O				SU	sha	AIN
colcrt							A
csh						sha	
ctags	O				L		N
dbx	L						s
dc						G	IN
deroff	sShaOU						N
diction						ShHU	N
ditroff	s			SHOU			AN
eqn			sShHU				AIN
ex						h	
fmt							N
ftp		sShaOU					AIN
indent	sh				SHOGL		AN
join	OU						sN
lex	sShHaUGL						AN
look	shu				HO		N
m4			HU				N
Mail							N
make	h						

Table 4: List of Utilities that Crashed or Hung, Categorized by Cause
The letters in each entry describe the system on while the failure occurred (see Table 1 for a description of the system letters)

Utility	Cause						
	Array/ Pointer	Input Functions	Signed Characters	Divide by Zero	EOF Check	Others	No Source Code
nroff				SHOU			AIN
plot	O					sh	N
prolog	sh						
ptx	sShH						A
refer	shU						AHN
spell	sha					SOU	N
spline							N
strings	O						
style						ShH	N
telnet		sShaU					AIN
tsort	L	sha					N
ul	sShHOUL						AIN
uniq	sShaU						IN
units	SshaO						AIN
vgrind							N

Table 4: List of Utilities that Crashed or Hung, Categorized by Cause
The letters in each entry describe the system on while the failure occurred (see Table 1 for a description of the system letters)

of the array. Following is a typical example of how “>>” is used:

```
char buff[BUFSIZE];

cin >> buff;
```

The representation of a UNIX character string does not carry information about the size of the array in which it is stored. You can set the maximum input line size using the C++ input/output class (using the cin.width() function), but this requires extra and explicit action by the programmer; the default case has dangerous behavior.

Signed Characters

The conversion of numbers from one size to another can cause problems; the problem is compounded by using characters in both their symbolic and numeric forms. In C (and C++), the type “char” is a signed, 8-bit integer on most UNIX systems[1]. The presence of a sign bit can be confus-

1. Both the number of bits and presence of a sign bit are system dependent. If the programmer really does not want a sign bit, then the declaration should include “unsigned”.

ing and error prone (with the possibility of sign-extension) when doing arithmetic. The following example comes from "eqn" (file “lookup.c”):

```
register int h;
...
register char *s = name;

for (h = 0; *s != '\0';)
      h += *s++;
h %= TBLSIZE;
```

The value pointed to by `s` is a signed character and `h` is an integer. Characters that are pointed to by `s` may have their high-order bit on, making `h` negative[2] (`h` will subsequently be used as a subscript).

End-of-File Checks

Checking for end-of-file is another case of the programmer making implicit assumptions about the structure of input data. It is a common, but dangerous assumption that end-of-file will occur only after a complete input line; i.e., end-of-file will always immediately follow a newline character. While this assumption can simplify the structure of the application code, it leaves the application vulnerable to crashing or hanging.

2.3.3 Comparison of Results to the 1990 Study

When we compare the results from the 1995 study to those from 1990, it is interesting to go beyond the raw numbers. When we examined the bugs that caused the failures, a distressing phenomenon emerged: many of the bugs discovered (approximately 40%) and reported in 1990 are still present in their exact form in 1995. The 1990 study was widely published in at least two languages. The code was made freely available via anonymous ftp. The exact random data streams used in our testing were freely available via ftp. The identification of failures that we found were also made freely available via ftp; these included code fragments with file and line number for the errant code. According to our records, over 2000 copies of the fuzz tools and bug identifications were fetched from our ftp site.

Several of the bugs found in the 1995 study were likely present in the 1990 study, but were masked by the original bugs. Fixing the original bugs and re-testing should have exposed these new ones.

The techniques used in this study are simple and mostly automatic. It is difficult to understand why a vendor would not partake of a free and easy source of reliability improvements.

2. The standard, printable characters do not have their high-order bit on, but it is not safe to assume that these are the only characters that will be read as input.

3 NETWORK SERVICES

The fuzz testing techniques are effective for finding reliability problems in real programs. A natural question is: in what other domains can these techniques be applied? Our first new application of the fuzz techniques was to test network services.

Internet network services are identified by a host name and port number. Most hosts support a collection of services, such as remote login ("rlogind" and "telnetd"), file transfer ("ftpd"), user information ("fingerd"), time synchronization protocols ("timed"), and remote procedures calls. To test these services, we wrote a simple program (called "portjig") that would attach to a network port and then send random data from the fuzz generator. This testing configuration is illustrated in Figure 1.

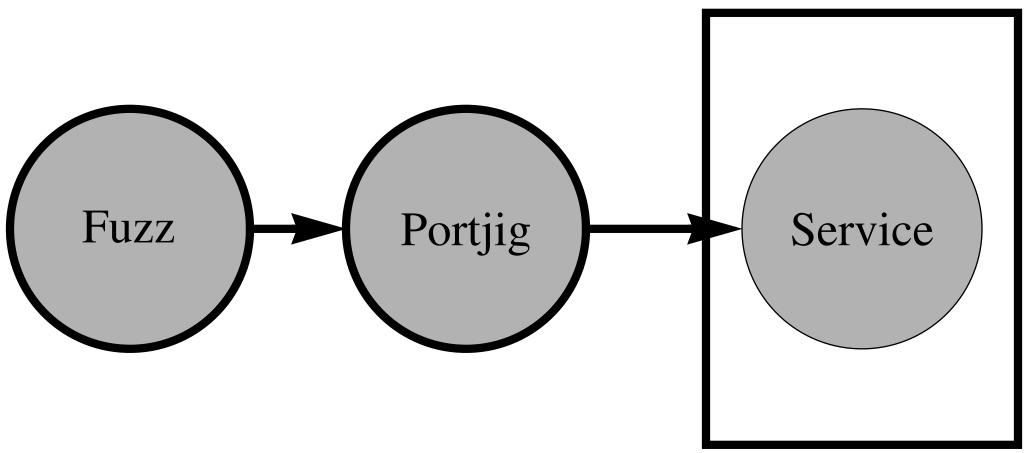

Figure 1: Testing Network Services

Most UNIX services are typically listed in a file called "/etc/services". Our test script selected each service in this file and sent it random data. We tested both TCP (connection-based) and UDP (datagram-based) services.

Several years ago, we informally tested network services on a few UNIX systems and were able to crash only two utilities ("ftpd" and "telnetd"). We ran our current tests on SunOS, NEXTSTEP, and Linux. *In this study, we were not able to crash any of the services that we tested on any UNIX system.* This result bodes well for network reliability and safety. Curiously though, we were able to crash some of the client programs for network services (such as "telnet" and "ftp", see Table 4).

4 X-WINDOW APPLICATIONS AND SERVERS

Our next target for fuzz testing was the window system and its application programs. An increasing number of application programs are based on graphical user interfaces, so X-Window based applications and servers were natural targets for the fuzz testing. Even though most of these applications were written more recently than the basic UNIX utilities, they still had high failure rates under random input tests. The X server proved resistant to crashing. Section 4.1 describes the tools we used to test the window server and applications, Section 4.2 describes the tests that we performed, and Section 4.3 presents the results from applying these tests to applications.

4.1 X-Window Fuzz Tools

To send random input to the X-Window server or applications, we interposed our testing tools between the client and the server. The interposed program, called *xwinjig*[1], can generate random input or modify the regular communication stream between the X-Window application and server.

The xwinjig program pretends to be an X server to the applications, and pretends to be a client to the real server. The X-Window system is capable of running on machines with multiple displays, and the X server has a separate TCP/IP port for each of its displays. These ports are numbered $6000 + N$, where N is the display number (display 0 is the standard default display). We change the default display by setting the DISPLAY environment variable to a higher-numbered display. Xwinjig listens on this port, and then connects to the real X server on the standard port.

The X server has an authentication mechanism, and xwinjig must mimic or circumvent it. We provide proof of authentication by reading the user's ~/.Xauthority file and sending the appropriate authorization "cookie" to the server. We circumvent the authentication by disabling access control checking on the server (by executing the command: `xhost +`).

4.2 X-Window Tests

In testing the X-Window server and applications, we used a combination of four different variations of random input streams. The first two techniques are used to test both the server and the applications, whereas the last two are used to test only the applications. Each input type is closer to valid input than the previous one (and therefore potentially tests deeper layers of the input processing code). The last technique listed below is important because it simulates a user randomly using the keyboard and mouse. Each application was tested with either (a) a combination of the Types 1-4 types of random input, or (b) only Type 4 (legal) random input.

1. Completely Random Messages: xwinjig concocts a random series of bytes and ships it off to the server or the client in a message.
2. Garbled Messages: xwinjig randomly inserts, deletes, or modifies parts of the message stream between the application and server.
3. Random Events: xwinjig keeps track of message boundaries defined by the X Protocol Reference Manual[5]. xwinjig randomly inserts events[2] that are of the proper size and have valid

1. *Xwinjig* actually has two implementations (called *xjig* and *winjig*) whose combined features are call xwinjig. This section reports on the combined results of testing with these two tools.
2. An event is a message sent by the server to the client to indicate that something of interest to the client happened at the server side. E.g., an event is sent each time a key is pressed.

opcodes (in the range 2 to 34). The sequence number, time stamp, and payload may be random.

4. Legal Events: These are protocol conformant messages that are logically correct individually and in sequence. These events have valid values for such things as X-Y coordinates within a window. Information such as window geometry, parent/child relationships, event time stamps, and sequence numbers are obtained by monitoring client/server traffic and are used to generate these events.

The xwinjig program has options to control the rate of injection of events, the frequency and method of randomizing the event stream, the direction of operation (client to server or vice versa), and the event types (keyboard and mouse events only, or all events).

4.3 X-Window Test Results

We tested the version 11R5 X-Window server on the SunOS and Ultrix systems; the X-Window applications were tested on SunOS. Testing was done only on these few systems because of the significant time it takes to complete this type of testing. The X-Window application programs include ones distributed by the vendor, locally written, freely distributed, and purchased from third party vendors.

4.3.1 Quantitative Results

The results of our tests on X-Window applications are given in Table 5. The first conclusion to draw is that the fuzz testing techniques are effective for testing programs based on graphical user interfaces. This result should encourage this type of testing on other window-based systems, such as the Macintosh and PC.

The test inputs of Types 1 and 2 represent some failure in the X server or its associate libraries. It may be reasonable to argue that these errors are unlikely, or that if they occur, the application programs cannot do much to counteract them. Good software design practice says that programs should have reasonable error checking on all external interfaces; in case of a crash, at least these checks will help localize the problem (and help convince the programmer that it is not in their own code). These checks can result in better bug reports to the software vendor.

The Type 3 and 4 inputs contain enough valid data that they take us past most of the basic checks in the X library. The Type 4 test input produced what is probably the most condemning result. Given legitimate input event streams, more than 25% of the programs tested crashed. Application programs from all sources failed on this type of input. Errors of this type are common; most users have, at some time, selected large amounts of text from a window and accidentally “pasted” it into the wrong window.

For the X-Window system, the “hang” results may be more serious than the crashes. In many cases, an X-Window application will hang while it is has exclusive control over input (keyboard and mouse). This means that the user cannot select another window to terminate the hung application; the user must use another workstation and remotely kill the application.

4.3.2 Causes of Crashes/Hangs

We identified the cause of the failures of several of the X-Window applications that we tested in this study. In this section, we describe some of the errors that we found. In general, the type of

X Utility	Input Data Stream Types (described in Section 4.2)	
	Combination Input (A Mix of Types 1-4)	Legal Events Only (Type 4)
bitmap	❷❸❹	❹
emacs	❷❹ ③	
ghostview	③	
idraw	③	④
mosaic		④
mxrn	③	
netscape	❸	❹
puzzle	③	
rxvt	③	④
xboard	③	
xcalc		
xclipboard		④
xclock		
xconsole		
xcutsel	③	
xditview	③	
xdvi	②③④	
xedit		
xev		
xfig	②④	
xfontsel		
xgas		
xgc	③	
xmag	③	
xman		
xmh		
xminesweep		
xneko		

Table 5: List of X Applications Tested and Results of Those Tests
❶❷❸❹= crashed on Type 1, 2, 3, or 4 input. ①②③④ = hung on Type 1, 2, 3, or 4 input.
When more than one combination of input types caused a crash, all are listed.
Tests run on SunOS 4.1.3.

X Utility	Input Data Stream Types (described in Section 4.2)	
	Combination Input (A Mix of Types 1-4)	Legal Events Only (Type 4)
xpaint	❸❹ ③	
xpbiff		
xpostit		④
xsnow	❸	
xspread	③④	
xterm	②	
xtv	③	④
xv	②④ ③	
xweather	❷	❹
xxgdb		④
# tested	38	38
# crash/hang	22	10
%	58%	26%

Table 5: List of X Applications Tested and Results of Those Tests
❶❷❸❹= crashed on Type 1, 2, 3, or 4 input. ①②③④ = hung on Type 1, 2, 3, or 4 input.
When more than one combination of input types caused a crash, all are listed.
Tests run on SunOS 4.1.3.

programming errors that we found were similar to those found in the basic tests.

The "xpaint" application crashes because of a common error with pointers: dereferencing a NULL pointer. During input, an X library function returns a window with zero height. The structure of the subsequent code is awkward. There are many places where the pointer might be used and each of these (except one!) has a check for NULL. The code structure has the appearance of evolving with incremental fixes rather than systematic overhaul.

Another example of not checking for NULL values can be found in "xsnow", an X application that creates snowflakes on the screen and drops them towards the bottom. The bug in this utility is that it does not sufficiently check the return values from X library functions. XCreateRegion() returns a NULL pointer that is subsequently passed to XPointInRegion(). Many of the X library functions (including XPointInRegion) do not check their arguments for validity in the interests of performance. (While the client side X libraries scrupulously check message from the server for errors, they trust the client (of which they are a part) to pass in correct arguments and to check return values.)

5 MEMORY ALLOCATION CALLS

We applied our random testing techniques to the interface between application programs and the system call library. We created a library that looks like the standard UNIX C library, but allows us to simulate error conditions. We limited our experiment to replacing one family of library routines: the dynamic memory allocation routines[1]. malloc() is a C library function that allocates memory on the heap, extending the program's virtual address space if necessary. A zero (NULL) return value from malloc() typically means that no more virtual memory is available to this process (because of insufficient swap space or system imposed resource constraints). It is a common, but dangerous, programming practice to ignore the return values from the memory allocation routines; failure to check for a zero return value can result in dereferencing a NULL pointer.

We extracted the object files for the malloc() family of functions from the standard C library and used a binary rewriting utility to rename the symbols [1]. By creating an object file with the original library call names (called "libjig") and linking against the new library, we were able to intercept calls to malloc() (see Figure 2). Any call to malloc() in the user program or the system library first went through libjig, before calling the real malloc(). The routines in libjig control the average percent of the time that malloc() fails.

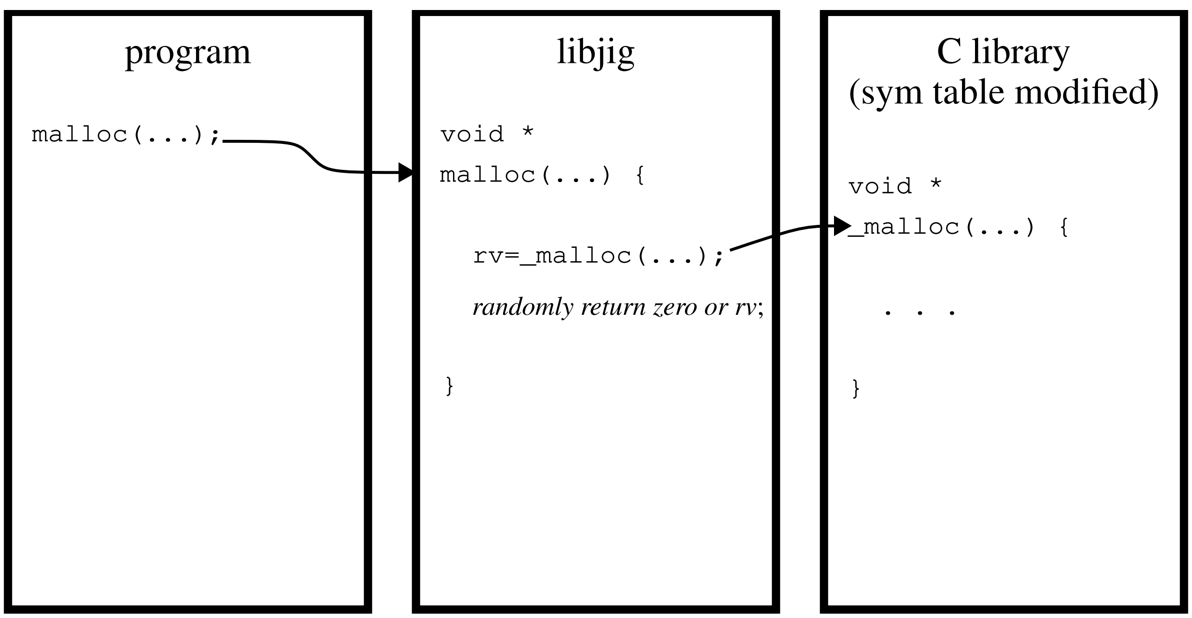

Figure 2: Intercepting calls to the memory allocation routine

We tested the programs in the "/bin" and "/usr/ucb" directories on a system running SunOS 4.1.3. Of the programs that had source code readily available, 53 made use of malloc(), and 25 (47%) crashed with our library. The utilities that crashed are listed in Table 6.

The memory allocation routines return zero typically when a user or system resource limit is reached. A common example is when the swap space is full. Note that many of the utilities listed in Table 6 are ones that a programmer might want to use when this situation occurs: "finger", "w", or "users" (to see who is logged in), "login" (to log in as super-user to try to fix the situation), and

1. The memory allocation routines that we considered were calloc(), malloc(), and realloc().

"df" (to check the amount of disk space used).

Utilities that Crashed				
bar	df	login	rup	tsort
cc	finger	ls	ruptime	users
checknr	graph	man	rusers	vplot
ctags	iostat	mkstr	sdiff	w
deroff	last	rsh	symorder	xsend

Table 6: Utilities that Crashed when malloc() Returns Zero Tested on SunOS 4.1.3.

In all but one case that we investigated, the programs simply dereference the address returned by malloc() without any checking. Some of the programs checked the return values in one place, and not another, while other programs did not check at all. The one case that was different was "df" (a program that shows the amount of disk space available); it checked all its calls to malloc(). However "df" calls another C library routine (getmntent()), which then calls malloc() without checking its return code. This is an example of a program being as strong as its weakest link.

This testing technique of modifying the return value of a call to a library can be easily applied to any other library routine. A common cause of programming error is not checking the return value on file operations (open, read, or write). Libjig could be used to find potential bugs in the use of these calls.

6 CONCLUSIONS

We revisited our original testing study with the expectation that it would be difficult to find many bugs in the new versions of the utilities that we tested. Our 1995 study surprised us in several ways.

First, the continued prevalence of bugs in the basic UNIX utilities seems a bit disturbing. The simplicity of performing random testing and its demonstrated effectiveness would seem to be irresistible to corporate testing groups. The basic utilities may simply fall between the cracks. Most of these are not major flashy components, such as a kernel or compiler. There is little glory or marketing impact associated with them and different companies assign these utilities to different groups.

Second, the reliability of network services and X-Window servers is good news. These basic system components are getting enough attention within the computer companies that they have been honed to a (relatively) high level of reliability.

Third, X-Window applications are no less prone to failure (and seem to be more so) than the basic utilities. These applications are generally newer than the basic utilities so, we hope, would be designed with better engineering techniques. Perhaps the large additional complexity of constructing a visual interface is too much of a burden. Hanging an X-Window application can cause the server to ignore all other input until the hanging application is terminated (which must be done remotely).

Fourth, the reliability of the freely-distributed GNU and Linux software was surprisingly good, and noticeably better than the commercially produced software. It is difficult to tell how much of this is a result of programmer quality, the culture of the programming environment, or the general burden supported by the software developers. Large companies will need to make some concrete changes in their software development environments and culture if they hope to produce higher quality software.

Modern compilers, languages, and development tools should be helping us develop more reliable programs. However it is clear that these new tools can be as easily abused as more primitive tools (old assembly language programmers are proud to point this out). A good example was shown in Section 2.3.2, where the problem of the missing input field-width specification of the C-library gets() can still be found in the C++ “>>” operator.

New versions of software are being released all the time. Our results represent a snapshot of versions that we had available to us at the time of testing.

There are certainly many things left to do in a study such as this one. The random testing can be applied to kernel calls and to randomly generated command-line parameters to utilities. Checking return codes from library routines should be extended to input/output and other types of calls. Any procedure call interface can be checked using libjig. Other areas of system software are amenable to this type of testing. The system call interface is an ideal candidate; calling these routines with random parameter values is likely to produce interesting results.

Other operating systems should be given the same scrutiny as we gave to UNIX. The problems that we found in the systems that we tested should not be interpreted to say that UNIX is any worse (or better) than other systems. Certainly the popular Apple Macintosh and IBM PC systems should receive the same level of testing.

SOURCE CODE AND RELATED PAPERS

Note that the source and binary code for the fuzz tools (for UNIX and Windows NT) is available from our Web page at: `ftp://grilled.cs.wisc.edu/fuzz`.

A more recent paper, applying fuzz testing techniques to applications running on Windows NT can be found at `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-nt.pdf`.

ACKNOWLEDGMENTS

We gratefully thank Mike Powell for his observations about the software development process.

REFERENCES

[1] Cargille, J., and Miller, B.P., Binary Wrapping: A Technique for Instrumenting Object Code. *SIGPLAN Notices 27*, 6 (June 1992), 17-18.

[2] Miller, B.P., Fredrikson, L., and So, B., An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM 33*, 12 (December 1990), 32-44. Also appears in German translation as Fatale Fehlerträchtigkeit: Eine Eimpirische Studie zur Zuverlassigkeit von UNIX-Utilties, *iX* (March 1991). `ftp://grilled.cs.wisc.edu/technical_papers/fuzz.pdf`.

[3] Rochlis, J.A., and Eichin, M.W., With Microscope and Tweezers: The Worm from MIT's Perspective. *Communications of the ACM 32*, 6 (June 1989), 689-698.

[4] Spafford, E.H., The Internet Worm: Crisis and Aftermath. *Communications of the ACM 32*, 6 (June 1989), 678-687.

[5] X Consortium, **X Protocol Reference Manual**. O'Reilly and Associates, Inc., 1992.