When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. To make a systematic statement about the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

An Empirical

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing: On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities; (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix™, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a 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 crashs or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons: First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes to the Internet worm (the "gets finger" bug) [2,3] We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carelessly typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and simplify the generation of cases.

Unix is a trademark of AT&T Bell Laboratories.

Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of UNIX Utilities

The following section describes the tools we built to test the utilities. These tools include the fuzz (random character) generator, ptyjig (to test interactive utilities), and scripts to automate the testing process. Next, we will describe the tests we performed, giving the types of input we presented to the utilities. Results from the tests will follow along with an analysis of the results, including identification and classification of the program bugs that caused the crashes. The final section presents concluding remarks, including suggestions for avoiding the types of problems detected by our study and some commentary on the bugs we found. We include an Appendix with the user manual pages for fuzz and ptyjig.

The Tools

We developed two basic programs to test the utilities. The first program, called *fuzz*, generates a stream of random characters to be consumed by a target program. There are various options to fuzz to control the testing activity. A second program, *ptyjig*, was also written to test interactive utility programs. Interactive utilities, such as a screen editor, expect their standard input file to have the characteristics of a terminal device. In addition to these two programs, we used scripts to automate the testing of a large number of utilities.

Fuzz: Generating Random Input Strings. The program fuzz is basically a generator of random characters. It produces a continuous string of characters on its standard output file (see Figure 1). 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 NULL (zero) character. You 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 you 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. For a complete description of fuzz, see the manual page in the Appendix.

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

```
fuzz 100000 -o outfile | deqn
```

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

Ptyjig: Testing Interactive Utilities. There are utility programs whose input (and output) files must have the characteristics of a terminal device, (e.g., the vi editor and the mail program). The standard output from fuzz sent through a pipe is not sufficient to test these programs.

Ptyjig is a program that allows us to test interactive utilities. It first allocates a pseudo-terminal file. This is a two-part device file that, on one side looks like a standard terminal device file (with a name of the form "/dev/ttyp?") and, on the other side can be used to send or receive characters on the terminal file ("/dev/ptyp?", see Figure 2). After creating the pseudo-terminal file, ptyjig then starts the specified utility program. Ptyjig passes characters that are sent to its input through the pseudo-terminal to be read by the utility.

The following is an example of fuzz and ptyjig being used to test vi, a terminal-based screen editor:

```
fuzz 100000 -o outfile | ptyjig vi
```

The output stream of fuzz will be at most 100,000 characters in length and the stream will be recorded in file "output." For a complete description of ptyjig, see the Appendix.

The Scripts: Automating the Tests. A command (shell) script file was written for each type of test. Each script executes all the utilities for a given set of input characteristics. The script checks for the existence of a *core* file after each utility terminates, indicating the crash of that utility. The core file and the offending input data file are saved for later analysis.

The Tests

After building the software tools, we used them to test a large collection of utilities running on several versions of the Unix operating system. Each utility on each system was executed with several different types of input streams. A test of a utility program can produce one of three results: **crash**—the program terminates abnormally producing a core file; **hang**—the program appears to loop indefinitely; or **succeed**—the program terminates normally. Note that in the last case, we do not specify the correctness of the output.

To date, we have tested utilities on seven versions of Unix[1]. These versions are summarized in Table I. Most of these versions are derived from some form of 4.2BSD or 4.3BSD Berkeley Unix. Some versions, like the SunOS release, have undergone substantial revision (especially at the kernel level). The SCO Xenix version is based on the System V standard from AT&T. The IBM AIX 1.1 Unix is a released, tested product, supporting mostly the basic System V utilities. It is also important that the tests covered several hardware architectures, as well as several systems. A program statement with an error might be tolerated on one machine and cause the program to crash on another. Referencing through a null-value pointer is an example of this type of problem.

Our testing covered a total of 88 utility programs on the seven versions of Unix. Most utilities were tested on each system. Table II lists

[1]Only the csh utility was tested on the IBM RT/PC. More complete testing is in progress.

TABLE I.

Versions of Unix TEST			
Identifying Letter	Machine Vendor	Processor	Kernel
[redacted]			
s	Sun 4/110	SPARC	SunOS 3.2 & SunOS 4.0 with NFS
[redacted]			
x	Citrus 80386	i386	SCO Xenix System V Rel. 2.3.1
[redacted]			
a	IBM PS/2-80	i386	AIX 1.1 Unix
[redacted]			

the names of the utilities that were tested, along with the type of each system on which that utility was tested. For a detailed description of each of these utilities, we refer readers to the user manual for appropriate systems. The list of utilities covers a substantial part of those that are commonly used, such as the mail program, screen editors, compilers, and document-formatting packages. The list also includes less commonly used utilities, such as cb, the C language pretty-printer.

Each utility program we tested was subjected to several different types of input streams. The different types of inputs were intended to test for a variety of errors that might be triggered in the utilities being tested. The major variations in test data were including nonprintable (control) characters, including the NULL (zero) byte, and maximum length of the input stream. These tests are summarized in Table IIIa.

The input streams for interactive utilities have slightly different characteristics. To avoid overflowing the input buffers on the terminal device, the input was split into random length lines (i.e., terminated by a NEWLINE character) with a mean length of 128 characters. The input length parameter is described by the number of lines, and is therefore scaled down by a factor of 100.

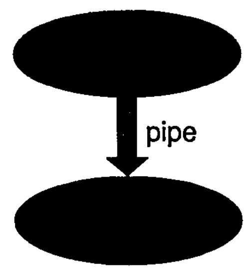

FIGURE 1. Output of Fuzz Piped to a Utility.

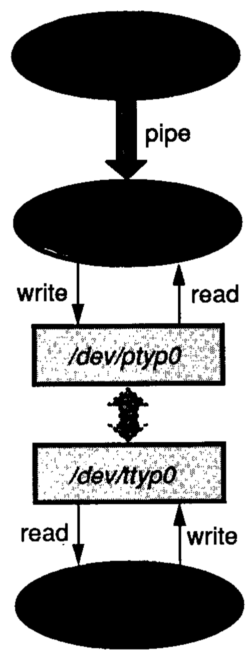

FIGURE 2. Fuzz with ptyjig to Test an Interactive Utility.

The Results And Analysis

Our tests of the Unix utilities produced a surprising number of programs that would crash or hang. In this section, we summarize these results, group the results by the common programming errors that caused the crashes, and show the programming practices that caused the errors. As a side comment, we noticed during our tests that many of the programs that did not crash would terminate with no error message or with a message that was difficult to interpret.

The basic test results are summarized in Table II. The first result to notice is that we were able to crash or hang a significant number of utility programs on each system (from 24%–33%). Included in the list of programs are several commonly used utilities: vi and emacs, the most popular screen editors; csh, the c-shell; and various programs for document formatting. We detected two types of error results—crashing and hanging. A program was considered crashed if it terminated, producing a core (state dump) file, and was considered hung if it continued executing, producing no output while having available input. A program was also considered hung if it continued to produce output after its input had stopped. Hung programs were typically allowed to execute for an additional five minutes after the hung state was detected. Programs that were blocked waiting for input were not considered hung.

Table IV summarizes the list of utility programs that we were able to crash or hang, categorized by the cause of the crash, and showing on which systems we were able to crash the programs. Notice that a utility might crash on one system but not on another. There are several reasons for this: One reason is differences in the processor architecture. For example, while the VAX will (incorrectly) tolerate references through null pointers, many other architectures will not (e.g., the Sun 4). A second reason is that the different systems had differences in the versions of the utilities. Local changes might improve or degrade a utility's reliability. Both internal structure as well as external specification of the utilities change from system to system. It is interesting to note that the commercially tested AIX 1.1 Unix is as susceptible as other versions of Unix to the type of errors for which we tested.

We grouped the causes of the crashes into the following categories: pointer/array errors, not checking return codes, input functions, sub-processes, interaction effects, bad error handler, signed characters, race conditions, and currently undetermined. For each of these categories, we discuss the error, show code fragments as examples of the error, present implications of the error, and suggest fixes for the problem.

Note that, except for one example (noted in the text), all of the crashes or hangs were discovered through automatic testing.

1

Pointer/Arrays

The first class of pointer and array errors is the case where a program might sequentially access the cells of an array with a pointer or array subscript, while not checking for exceeding the range of the array. This was one of the most common programming errors found in our tests. An example (taken from cb) shows this error using character input:

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

This example could be easily fixed to check for a maximum array length. Often the terseness of the C programming style is carried to extremes; form is emphasized over correct function. The ability to overflow an input buffer is also a potential security hole, as shown by the recent Internet worm.

The second class of pointer problems is caused by references through a null pointer. The prolog interpreter, in its main loop, can incorrectly set a pointer value that

TABLE II. *List of Utilities Tested and the Systems on which They Were Tested (part 1)*

*•=utility crashed, ○=utility hung, *=crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. —=utility unavailable on that system. !=utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
adb	• ○	•	•	○	—	—
as	•			•	•	•
awk						
bc				• ○		
bib			—	—	—	—
calendar				—		
cat						
cb	•		•	•	○	•
cc						
/lib/ccom				—	—	•
checkeq				—		
checknr				—	—	
col	• ○	•	•	• ○	•	•
colcrt				—	—	
colrm				—	—	
comm						
compress					—	
/lib/cpp						
csh	• ○	○	○	—	○	○
dbx		*	—	—		
dc				○		
deqn		•	—	—	—	—
deroff	•	•	•		•	•
diction	•	—	•		—	•
diff						
ditroff	• ○	•	—	—	—	
dtbl			—	—	—	—
emacs	•	•	○	—	—	
eqn		•	•	•		
expand					—	
f77	•		—	—	—	—
fmt						
fold					—	
ftp	•	•	•	—	•	•
graph					—	
grep						
grn			—	—	—	—
head					—	
ideal			—	—	—	—
indent	• ○	• ○	•	—	—	•
join		⊕				
latex			—	—	—	—
lex	•	•	•	•	•	•
lint						
lisp		—		—	—	—
look	•	○	•	•	—	•

is then assumed to be valid in the next pass around the loop. A crash caused by this type of error can occur in one of two places. On machines like the VAX™, the reference through the null pointer is valid and reads data at location zero. The data accessed are machine instructions. A field in the (incorrectly) accessed data is then used as a pointer and the crash occurs. On machines like the Sun 4, the reference through the null pointer is an error and the program crashes immediately. If the path from where the pointer was set to where it was used is not an obvious one, extra checking may be needed.

The assembly language debugger (adb) also had a reference through a null pointer. In this case, the pointer was supposed to be a global variable that was set in another module. The external (global) definition was accidentally omitted from the variable declaration in the module that expected to use the pointer. This module then referenced an uninitialized (in Unix, zero) pointer.

Pointer errors do not always appear as bad references. A pointer might contain a bad address that, when used to write a variable, may unintentionally overwrite some other data or code location. It is then unpredictable when the error will manifest itself. In our tests, the crash of lex (scanner generator) and ptx (permuted index generator) were examples of overwriting data, and the crash of ul (underlining text) was an example of overwriting code.

The crash of as (the assembler) originally appeared to be a result of improper use of an input routine. The crash occurred at a call to the standard input library routine ungetc(), which returns a character back to the input buffer (often used for look-ahead processing). The actual cause was that ungetc() was redefined in the program as a macro that performed a similar function. Unfortunately, the new macro had less error checking than the system version of ungetc() and allowed a buffer pointer to be incorrectly set. Since the new macro looks like the original routine, it is easy to forget the differences.

TABLE II. List of Utilities Tested and the Systems on which They Were Tested (part 2)

*•=utility crashed, ○=utility hung, *=crashed on SunOS 3.2 but not on SunOS 4.0, ⊕ = crashed only on SunOS 4.0, not 3.2. —=utility unavailable on that system. !=utility caused the operating system to crash.*

Utility	VAX (v)	Sun (s)	HP (h)	i386 (x)	AIX 1.1 (a)	Sequent (d)
m4				•		
mail						
make			•			
more					—	
nm						
nroff				•		
pc				—	—	—
pic			—	—	—	—
plot	—	○	•	—	—	
pr						—
prolog	• ○	• ○	• ○	—	—	—
psdit				—	—	
ptx	—	•	•	○		○
refer	•	*	•	—	—	! •
rev				—	—	
sed						
sh				—		
soelim					—	
sort						
spell	• ○	•	•	○	•	•
spline					—	
split						
sql		—			—	—
strings					—	
strip						
style	•	—	•		—	•
sum						
tail						
tbl						
tee						
telnet	•	•	•	—	•	○
tex			—	—	—	—
tr						
troff	—	—	—			
tsort	•	*	•	•	•	•
ul	•	•	•	—	—	•
uniq	•	•	•	•	•	•
units	• ○	•	•	•	•	•
vgrind	•		—	—	—	
vi	•		•	—		
wc						
yacc						
# tested	85	83	75	55	49	73
# crashed/hung	25	21	25	16	12	19
%	29.4%	25.3%	33.3%	29.1%	24.5%	26.0%

Not Checking Return Codes

Not checking return codes is a sign of careless programming. It is a favorable comment on the current state of Unix that there are so few examples of this error. During our tests, we were able to crash adb (the assembly language debugger) and col (multi-column output filter ASCII terminals) utilities because of this error. Adb provides an interesting example of a programming practice to avoid. This code

VAX is a trademark of Digital Equipment Corporation.

2

fragment represents a loop in adb and a procedure called from that loop.

```
format.c (line 276):
. . .
while (lastc != '\n') {
      rdc ();
}
. . .

input.c (line 27):

rdc ()
{     do { readchar (); }
      while (lastc == ' ' ||
                 lastc == '\t');
      return (lastc);
}
```

The initial loop reads characters, one by one, terminating when the end of a line has been seen. The rdc() routine calls readchar(), which places the new character into a global variable named "lastc." Rdc() will skip over tab and space characters. Readchar() uses the Unix file read kernel call to read the characters. If readchar() detects the end of the input file, it will set the value of lastc to zero. Neither rdc() nor the initial loop check for the end of file. If the end of file is detected during the middle of a line, this program hangs.

We can speculate as to why there was no end of file check on the initial loop. It may be because the program author thought it unlikely that the end of file would occur in this situation. It might also be that it was awkward to handle the end of file in this location. While this is not difficult to program, it requires extra tests and flags, more complex loop conditions, or possibly the use of a goto statement.

This problem was made more complex to diagnose because of the extensive use of macros (the code fragment above has the macros expanded). These macros may have made it easier to overlook the need for the extra test for the end of file.

3

Input Functions

We have already seen cases where character input routines within a loop can cause a program to store into locations past the end of an array. Input routines that read entire strings are also vulnerable. One of the main holes through which the Internet worm entered was the gets() routine. The gets() routine takes a single parameter that is a pointer to a character string. There is no possible means of bounds checking. Our tests crashed the ftp and telnet utilities through use of gets().

The scanf() routine is also vulnerable. In the input specification, it is possible to specify an unbounded string field. An example of this comes from the topological sort (tsort) utility.

```
x = fscanf(input, "%s%s",
      precedes, follows);
```

The input format field specifies two unbounded strings. In the program, "precedes" and "follows" are declared with the relatively small lengths of 50 characters. It is possible to place a bound on the string field specification, solving this problem.

TABLE IIIA. Variations of Input Data Streams for Testing Utilities
(these were used for the noninteractive utility programs)

Input Streams for Noninteractive Utilities

#	Character Types	NULL character	Input stream size (no. of bytes)
1	printable+nonprintable	YES	1000
2	printable+nonprintable	YES	10000
3	printable+nonprintable	YES	100000
4	printable	YES	1000
5	printable	YES	10000
6	printable	YES	100000
7	printable+nonprintable		1000
8	printable+nonprintable		10000
9	printable+nonprintable		100000
10	printable		1000
11	printable		10000
12	printable		100000

TABLE IIIB. Variations of Input Data Streams for Testing Utilities
(these were used for the interactive utility programs)

Input Streams for Interactive Utilities

#	Character Types	NULL character	Input stream size (no. of strings)
1	printable+nonprintable	YES	10
2	printable+nonprintable	YES	100
3	printable+nonprintable	YES	1000
4	printable	YES	10
5	printable	YES	100
6	printable	YES	1000
7	printable+nonprintable		10
8	printable+nonprintable		100
9	printable+nonprintable		1000
10	printable		10
11	printable		100
12	printable		1000

4 *Sub-Processes*

The code might be carefully designed and written, with the programming following all the good rules for program writing. But this might not be enough if another program is used as part of this one. Several of the Unix utilities execute other utilities as part of doing their work. For example, the diction and style utilities call deroff, vi calls csh, and vgrind calls troff. When these sub-processes are called, they are often given direct access to the raw input data stream, so they are vulnerable to erroneous input. Access to sub-processes should be carefully controlled or insurance provided that the program input to the subprocess is first checked. Alternatively, the utility should be programmed to tolerate the failure of a subprocess (though this can be difficult).

Interaction Effects

Perhaps one of the most interesting errors that we discovered was a result of an unusual interaction of two parts of csh, along with a little careless programming. The following string will cause the VAX version of csh to crash

```
!o%8f
```

and the following string

```
!o%888888888f
```

will hang (continuous output of 5
space characters) most versions of csh. The first example, which triggers the csh's command history mechanism, says "repeat the last

TABLE IV. *List of Utilities that Crashed, Categorized by Cause*

The letters indicate the system on which the crash occurred (see Table I).

	Cause									
Utility	**array/ pointer**	**NCRC**	**input functions**	**sub-processes**	**interaction effects**	**bad error handler**	**signed characters**	**race condition**	**no source code**	**unknown**
adb	vshx	v								
as	vxad									
bc									x	
cb	vhxad									
/lib/ccom	d									
col		vshxad								
csh					vshra			vshra		
dc									x	
deqn							s			
deroff	vshad									
diction				vhd						
ditroff	vs									
emacs								vsh		
eqn							shx			
f77										v
ftp			vshad							
indent	vshd									
join									s	
lex	vshxad									
look	vshxd									
m4									x	
make							h			
nroff									x	
plot										sh
prolog	vsh									
ptx	shxd									
refer	vshd									
spell							vshxad			
style				vhd						
telnet			vshad							
tsort			vshxad							
ul	vshd									
uniq	vshxad									
units	vshxad					v				
vgrind				v						
vi				vh						

“While our testing strategy sounds somewhat naive, its

command that began with ‘o%8f.’” Since it does not find such a command, csh forms an error message string of the form: “o%8f: Event not found.” This string is passed to the error-printing routine, which uses the string as the first parameter to the printf() function. The first parameter to printf() can include format items, denoted by a “%.” The “%8f” describes a floating point value printed in a field that is 8 characters wide. Each format item expects an additional parameter to printf(), but in the csh error, none is supplied (or expected). This string was generated during the normal random testing.

The second example string follows the same path, but causes csh to try to print the floating point value in a field that is 888,888,888 characters wide. The seemingly infinite loop is the printf() routine's attempt to pad the output field with sufficient leading space characters. This second string was one that we generated by hand after discovering the first string.

Both of these errors could be prevented by substituting the printf() call with a simple string printing routine (such as puts()). The printf() was used for historical reasons related to space efficiency. The error-printing routine assumed that it would always be passed strings that were safe to print.

6

Bad Error Handler

Sometimes the best intentions do not reach completion. The units program detects and traps floating point arithmetic errors. Unfortunately, the error recovery routine only increments a count of the number of errors detected. When control is returned to the faulty code, the error recurs, resulting in an infinite loop.

7

Signed Characters

The ASCII character code is designed so that codes normally fall in the range that can be represented in seven bits. The equation processor (eqn) depends on this assumption. Characters are read into an array of signed 8-bit integers (the default of signed vs. unsigned characters in C varies from compiler to compiler). These characters are then used to compute a hash function. If an 8-bit character value is read, it will appear as a negative number and result in an erroneous hash value. The index to the hash table will then be out of range. This problem can be easily fixed by using unsigned values for the character buffer. In a more sophisticated language than C, characters and strings would be identified as a specific type not related to integers.

This error does not crash all versions of adb. The consequence of the error depends on where in the address space is accessed by the bad hash value. (This error could be considered a subcase of the pointer/array errors.)

8

Race Conditions

Unix provides a signal mechanism to allow a program to asynchronously respond to unusual events. These events include keyboard-selected functions to kill the program (usually control-C), kill the program with a core dump (usually control-\), and suspend the program (usually control-Z). There are some programs that do not want to allow themselves to be interrupted or suspended; they want to process these control characters directly, perhaps taking some intermediate action before terminating or suspending themselves. Programs that make use of the cursor motions features of a terminal are examples of programs that directly process these special characters. When these programs start executing, they place the terminal device in a state that overrides processing of the special characters. When these programs exit, it is important that they restore the device to its original state.

So, when a program, such as emacs, receives the suspend character, it appears as an ordinary control-Z character (not triggering the suspend signal). Emacs will, on reading a control-Z, do the following: (1) reset the terminal to its original state (and will now respond to suspend or terminate signals), (2) clean up its internal data structures, and (3) generate a suspend signal to let the kernel actually stop the program.

If a control-\ character is received on input between steps (1) and (3), then the program will terminate, generating a core dump. This race condition is inherent in the Unix signal mechanism since a process cannot reset the terminal and exit in one atomic operation. Other programs, such as vi and more, are also subject to the same problem. The problem is less likely in these other programs because they do less processing between steps (1) and (3), providing a smaller window of vulnerability.

9

Undetermined Errors

The last two columns of Table IV list the programs where the source code was currently not available to us or where we have not yet determined the cause of the crash.

Conclusions

This project started as a simple experiment to try to better understand an observed phenomenon—that of programs crashing when we used a noisy dial-up line. As a result of testing a comprehensive list of utility programs on several versions of Unix, it appears that this is not an isolated problem. We offer two tangible products as a result of this project. First, we provide a list of bug reports to fix the utilities that we were able to crash. This should qualitatively improve the reliability of Unix utilities. Second, we provide a simple-to-use, yet surprisingly effective test method (and tools).

We do not claim that our tests are exhaustive; formal verification is

ability to discover fatal program bugs is impressive."

required to make such strong claims. We cannot even estimate how many bugs remain to be found in a given program. But our simple testing technique has discovered a wealth of errors and is likely to be more commonly used (at least in the near term) than more formal procedures. Our tests appear to discover errors that are not easily found by traditional testing practices. This conclusion is based on the results from testing AIX 1.1 Unix.

Comments on the Results

Our examination of the results of the tests have exposed several common mistakes made by programmers. Most of these mistakes involve areas already known to experienced programmers, but an occasional reminder is sometimes helpful. From our inspection of the errors found, we suggest the following guidelines:

(1) Check all array references for valid bounds. This is an argument for using range checking full-time. Even (especially!) pointer-based array references in C should be checked. This spoils the terse and elegant style often used by experienced C programmers, but correct programs are more elegant than incorrect ones.

(2) Be sure that all input fields are bounded—this is just an extension of guideline (1). In Unix, using "%s" without a length specification in an input format is a bad idea.

(3) Check all system call return values; do this checking even when an error result is unlikely and the response to an error result is awkward.

(4) Check pointer values often before using them. If all the paths to a reference are not obvious, an extra sanity check can help catch unexpected problems.

(5) Judiciously extend trust to others; not all programmers exercise the same standards of carefulness. If using someone else's program is necessary, make sure that the data its fed has been checked. This is sometimes called "defensive programming."

(6) In redefining something to look too much like something else, a programmer may eventually forget about the redefinition. He or she then becomes subject to problems that occur because of the hidden differences. This may be an argument against excessive use of procedure overloading in languages such as Ada or C++.

(7) Error handlers should handle errors. These routines should be thoroughly tested so that they do not introduce new errors or obfuscate old ones.

(8) Goto statements are generally a bad idea. Dijkstra observed this many years ago [1], but it is difficult to convince some programmers. Our search for the cause of a bad pointer in the prolog interpreter's main loop was complicated by the interesting weaving of control flow caused by the goto statements.

Comments on Lurking Bugs

An interesting question is: why are there so many buggy programs in Unix? This section contains commentary and speculation; it should be considered more editorial than factual. It is our experience that we often encounter bugs in programs, but ignore them; we do so, not because they are not serious (they often cause crashes). There are, however, two reasons for ignoring bugs: First, it is often difficult to isolate exactly what activity caused the program to crash. Second, it's quicker to try a slightly different method to get the current job done than it is to find and report a bug.

As part of an informal survey of the Unix user community in our department (comprising researchers, staff, and students on several hundred Unix workstations), we asked if they had encountered bugs that they had not reported to anyone. We also asked about the severity of the bugs and why they had not reported them. Many users responded to the survey and all (but one) reported finding bugs that they did not report; about two-thirds of these bugs were serious ones. The commentary of the various users speaks for itself. Following are quotes from the responses of several users:

> "Because ⟨*name of research tool*⟩ was involved, I figured it is too complicated. Besides, by changing a few parameters, I would get a core image that dbx would **not** crash on, thus preventing me from really having to deal with the problem."

> "My experience is that it is largely useless to report bugs unless I can supply algorithms to reproduce them."

> "I haven't reported this because recovery from this error is usually fast and easy. . . That is, the time and effort wasted due to a single occurrence of the bug is usually smaller than the time needed to report it."

> "I don't generally report problems because I have gotten the impression over the years that unless it's a security hole in mail or something, either no-one will look at it, they will chalk it up to a one-time event or user mistake, or it will take forever to fix."

Some users are easy to please. We received one response from our survey that stated:

> "I have not encountered any bugs in Unix software."

The number of bugs in Unix might also be explained by its evolution. Unix has suffered from a "features are more important than testing" mentality. In its early years, it was a research-only tool. The commercial effort required to do complete testing was not part of the environment in which it was used. Later, the Berkeley Unix v. System V ("tastes great" v. "less filling") competition forced a race for features, power, and performance. Absent from that debate was a serious discussion of reliability. There were some claims that the industry version (System V) had support when compared to that of a university product. Support for Unix seems to be more concerned with user complaints than with releasing a significantly more reliable product.

Unix should not be singled out as a buggy-operating system. Its strengths help make its weaknesses visible—testing programs under Unix was particularly easy because of the mix-and-match modularity provided by pipes and standard I/O. Other systems must undergo similar tests before any conclusion can be made about Unix's reliability compared to other systems.

More to Do

We still have many experiments left to perform. We have tested only the utilities that are directly accessible by the user. Network services should also receive the same attention. It is a simple matter to construct a *portjig* program, analogous to our ptyjig, to allow us to connect to a network service and feed it the output of the fuzz generator. A second area to examine is the processing of command-line parameters to utilities. Again, it would be simple to construct a *parmjig* that would start up utilities with the command-line parameters being generated by the random strings from the fuzz generator. A third area to study is other operating systems. While Unix pipes make it simple to apply our techniques, utility programs can still be tested on other systems. The random strings from fuzz can be placed in a file and the file used as program input. A comparison across different systems would provide a more comprehensive statement on operating-system reliability. A fourth area is using random testing to help find security holes. The testing might involve sending programs random sequences of nonrandom key or command words.

Our next step is to fix the bugs that we have found and reapply our tests. This retesting may discover new program errors that were masked by the errors found in the first study. We believe that a few rounds of testing will be needed before we reach the limits of our tools.

We are making our testing tools generally available and invite others to duplicate and extend our tests. Initial results coming in from other researchers match the experiences in this report.

Acknowledgments.
We are extremely grateful to Jerry Popek, Phil Rush, Jeff Fields, Todd Robertson, and Randy Fishel of Locus Computing Corporation for providing the facilities and support to test the AIX 1.1 Unix system. We are also grateful to Matt Thurmaier of The Computer Classroom (Madison, Wis.) for providing us with technical support and use of the Citrus 386-based XENIX machine. Our thanks to Dave Cohrs for his help in locating the race condition that caused emacs to crash. We thank those people in our Computer Science Department that took the time to respond to our survey. (The suggestion on using random testing to help find security holes is due to one of the anonymous referees.)

References

1. Dijkstra, E. W. GOTO Statement Considered Harmful. *Commun. ACM 11*, 3 (March 1968), 147–8.
2. Rochlis, J. A., and Eichin, M. W. With microscope and tweezers: The Worm from MIT's perspective. *Commun. ACM 32*, 6 (June 1989), 689–698.
3. Spafford, E. H. The Internet Worm: Crisis and aftermath. *Commun. ACM 32*, 6 (June 1989), 678–687.
4. Wood, D. A., Gibson, G. A., and Katz, R. H. Verifying a multiprocessor cache controller using random case generation. Computer Science Tech. Rep. UCB/CSD 89/490, University of California, Berkeley (January 1989).

CR Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: testing and debugging; D.4.9 [**Operating Systems**];Systems Programs and Utilities

General Terms: Reliability

Additional Key Words and Phrases: Unix

About the Authors

BARTON P. MILLER is an associate professor of computer science at the University of Wisconsin-Madison. His research interests include parallel and distributed debugging, network management and naming service, distributed operating systems, and user interfaces. **Author's Present Address:** Computer Science Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706. email bart@cs.wisc.edu.

LARS FREDRIKSEN is a member of the technical staff at AT&T Bell Labs. His research interests include software development dealing with real-time database and operating systems, development tools and application programming. **Author's Present Address:** AT&T Bell Labs, 2000 N. Naperville Rd., Naperville, IL 60566. email L.Fredriksen@att.com.

BRYAN SO is a Ph.D. candidate at the University of Wisconsin-Madison. His research interests include hypertext systems and expert systems. **Author's Present Address:** Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706. email so@cs.wisc.edu.

Research was supported in part by National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant NOOO14-89-1222, and a Digital Equipment Corporation External Research Grant.

APPENDIX: USER COMMANDS

FUZZ(1)

NAME
fuzz—random character generator

SYNOPSIS
fuzz length [option] . . .

DESCRIPTION
The main purpose of *fuzz* is to test the robustness of system utilities. We use *fuzz* to generate random characters. These are then piped to a system utility (using *pty(1)* is necessary). If the utility crashes, the saved input and output streams can then be analyzed to decide what sorts of input cause problems.

Length is taken to be the length of the output stream, usually in bytes, When –l is selected the length is in number of strings.

The following options can be specified.

–0 Include NULL (ASCII 0) characters

–a Include all ASCII characters except NULL (default)

–d *delay*
Specify a delay in seconds between each character.

–e *string*
Send *string* after all the characters. This feature can be used to send termination strings to the test programs. Standard C escape sequences can be used.

–l *[len]*
Generate random length strings. If *len* is specified, it is taken to be the maximum length of each string (default = 225). Strings are terminated with the ASCII new-line character.

–o *file* Store the output stream to *file* as well as sending them to *stdout.*

–p Generate printable ASCII characters only.

–r *file* Replay characters stored in *file.*

–s *seed* Use *seed* as the seed to the random number generator.

–x Print the seed as the first line of stdout.

AUTHORS
Lars Fredriksen, Bryan So.

SEE ALSO
pty(1)

USER COMMANDS

PTYJIG(1)

NAME
ptygig—pseudo-terminal pipe

SYNOPSIS
ptyjig [option] . . . command [args] . . .

DESCRIPTION
Pty executes the Unix command with args as its arguments if supplied. The standard input of ptyjig is piped to command as if typed at a terminal. Ptyjib is expected to be used with *fuzz(1)* to test interactive (terminal-based) programs.

The following options can be specified.

–e Do not send EOF character after *stdin* has exhausted.

–s Do not process interrupt signals, such as SIGINT, SIGQUIT and SIGSTOP.

–x Do not write output from command to *stdout.*

–i *file* Save the input stream sent to command into file.

–o *file* Save the output produced by command into file.

–d *delay*
Wait *delay* seconds after sending each character.

–t *interval*

If input has exhausted but *command* has neither exited nor sent any output, exit after *interval* seconds. Default is 2.0 seconds.

Delay and *interval* can have fractions.

EXAMPLE

ptyjig -o out -d 0.2 -t 10 vi text1 < text2

Runs "vi text1" in background, typing the characters in *text2* into it with a delay of 0.2sec between characters, and save the output to *out*. The program stops when *vi* stops outputing for 10 seconds.

AUTHORS

Lars Fredricksen, Bryan So.

FILES

/dev/tty*
/dev/pty*

SEE ALSO

fuzz(1), sigvec(2), pty(4), tty(4)

BUGS

The trace files specified by -i and -o options may contain more than actual characters sent to and received from *command*. This is due to the fact that after command exits and before ptyjig is signaled, some characters may be sent. This can be prevented by setting -d option to some suitable delay.

If the test program terminates abnormally, the usual core dumped message is not printed.

CP

(continued from page 17)
bipolar as it does in CMOS. Since any processor, especially a CMOS processor, gains greatly in performance by having a large amount of on-chip memory, this advantage could well tip the balance in favor of CMOS.

The advantage that would result from being able to put CMOS transistors and bipolar transistors on the same chip has not gone unnoticed in the industry. Active development is proceeding in this area, under the generic name BiCMOS. BiCMOS is also of interest for analogue integrated circuits.

If the BiCMOS process were optimized for bipolar transistors it would be possible to have a very high-performance bipolar processor with CMOS on-chip memory. If the bipolar transistors were of lower-performance levels they would still be of value for driving off-chip connections and also for driving long-distance connections on the chip itself.

A pure bipolar chip, with a million transistors on it, will dissipate at least 50 watts, probably a good deal more. Removing the heat presents problems, but these are far from being insuperable. More severe problems are encountered in supplying the power to the chip and distributing it without a serious voltage drop or without incurring unwanted coupling. Design tools to help with these problems are lacking. A BiCMOS chip of similar size will dissipate much less power. On the other hand, BiCMOS will undoubtedly bring a spate of problems of its own, particularly as the noise characteristics of CMOS and bipolar circuits are very different.

CMOS, bipolar, and BiCMOS technologies are all in a fluid state of evolution.

CMOS, bipolar, and BiCMOS technologies are all in a fluid state of evolution. It is possible to make projections about what may happen in the short term, but what will happen in the long term can only be a matter of guess work. Moreover, designing a computer is an exercise in system design and the overall performance depends on the statistical properties of programs as much as on the performance of the individual components. It would be a bold person who would attempt any firm predictions.

And then, finally, there is the challenge of gallium arsenide. A colleague, with whom I recently corresponded, put it very well when he described gallium arsenide as the Wankel engine of the semiconductor industry!

Maurice V. Wilkes received the ACM Turing Award in 1967 and is the author of Memoirs of a Computer Pioneer, *MIT Press, 1985.*