Using Fuzz to Test the Reliability of Unix Kernels

Todd M. Bezenek and Derek Wright
CS 736, Advanced Operating Systems
University of Wisconsin–Madison
December 12, 1996

Abstract

This paper presents the results of applying random inputs (fuzz) to the system call interface of several contemporary Unix kernels. Recently published work has shown that applying random inputs to Unix applications and X-Window clients can be surprisingly effective at exposing bugs in those systems. We present a tool that uses the same method to test the Unix kernel, and the results of applying that tool to five kernels. Problems were found in three of the five systems tested. In two of the systems (Solaris 2.5.1 and IRIX 6.2), bugs that are capable of crashing the system were found.

1.0 Introduction

A *fuzz generator* is a program that creates pseudo-random, possibly valid input to a target program or service. By applying random input produced by the fuzz generator to the target program, weaknesses in the robustness of the target program can be exposed.

Fuzz generators have been used to study the reliability of Unix utilities, X-Window servers and clients, and network services [1]. The easily applied technique produced unwanted behavior in 6 to 30% of the Unix utilities in each system tested and more than 25% of the X-Window clients. In almost all cases where the source code was available, the reason for the aberrant behavior was located, producing a significant opportunity to remove bugs. In the case of X-Window servers and network services, no cases of bad performance were found, producing strong support for the high reliability of these systems. This examination of Unix utilities, interfaces, and network services provided information about much of the support system that is available for using the Unix operating system, but it did not directly address the Unix system component upon which everything else relies: the Unix kernel. In this paper, we describe the design of a kernel-specific fuzz tester and present the results of applying it to five Unix kernels.

that demonstrated varying degrees of stability during testing. Several of the Unix kernels tested proved to be entirely stable for all reasonable inputs, with no detectable problems present. Others were stable, but demonstrated undesirable behavior for one or more subsets of all possible inputs. For one kernel, inputs were found that caused it to immediately crash.

The remainder of this paper includes a description of the fuzz tester in Section 2, the procedure that is used to apply the fuzz tester to a kernel in Section 3, and the results of its application to several contemporary kernels in Section 4. A description of a related kernel testing tool is given in Section 5, followed by a summary in Section 6.

2.0 The Fuzz Tester

While creating the kernel fuzz tester, we wanted to produce something that was as automated as possible. But there are issues involved in testing the kernel that require a comprehensive tester to include some manual control. Although the tester we created is not fully automatic, at every point in the design described below where manual control is required, we carefully considered and rejected automatic alternatives.

2.1 The kernel access mechanism

The fuzz tester probes the kernel via the BSD standard C library function `syscall()`. Each probe includes one of the available system functions and a set of random parameters:

```
int syscall(<system function number>, <parameter list>);
```

The `syscall()` interface exists on each of the operating systems we investigated (see Table 3). In Solaris running on a SPARC platform, it is implemented as a wrapper that translates the call to a system trap. The given system function is popped off of the top of the stack and placed in register `g1`, and the remaining parameters are left on the stack where they are available to the system function after the trap instruction completes.

Using the `syscall()` mechanism allows the fuzz tester to call kernel routines directly without using a custom assembly language interface for each platform. In the case that the `syscall()` mechanism is not available, an equivalent wrapper can be hand-coded, and the tool used without modification. This direct mechanism for accessing kernel calls allows automation of the tester across platforms, but it bypasses any parameter checks in the library routines. As it is possible that the library routines may check parameter values, each problematic parameter list should be verified by passing it to the system call via the library interface.

Using the `syscall()` mechanism, the tester repeatedly probes system calls with random arguments. The system function to be used for each probe is randomly chosen from a list contained in a file. The list can be an automatically generated complete list of system calls or a subset of system calls chosen by the user. The parameter list defaults to always contain eight randomly generated parameters. Eight is the largest number of parameters required by all of the system calls on all of the systems that we tested, and is configurable by the user. We chose to fix the size of the parameter list in this way because using too few parameters can cause problems with the tester's stack as discussed in Section 4.1, whereas using too many parameters simply wastes a small amount of stack space.

When inputs to the system functions cause a system lock-up or crash, information about the state of the fuzz generator that can aid in the discovery of system bugs is lost. Although it is trivial to create reproducible results by fixing the random number generator's seed,[1] once the system has failed, it is not possible to find out which call executed last. This problem is addressed by having two execution modes—normal and logging—that are controlled by a command-line parameter. The normal mode executes quickly, without saving information about each call that could be used to determine the reason for unusual behavior. The logging mode logs each system call—including parameters—to a file, and executes `flush()` and `fsync()` before executing the call, guaranteeing that the information will be available if the call fails. The relative speeds of these operating modes are shown in Table 1.

TABLE 1. System call rates for Solaris on a SPARC 20 workstation.

Mode	Time per call (msec)	Calls per second	Calls per hour
With logging	90	11	~40,000
Without logging (a mix[a] of system calls)	2.03	490	~1,750,000
Without logging (fastest system calls)	0.73	1360	~5,000,000

a. Some system calls execute very fast (e.g. `rename()`), while others spend some time in the kernel (e.g. `brk()`).

1. We assume that an idle kernel represents a single base state and therefore a consistent starting point for repeated tests.

2.2 Handling special case parameters

The variability of system call parameter lists, and the fact that no definitive information is known about each call on a given system[1] makes system calls a hard target for an automated tester. If the tester knew more about what each system call expected for each parameter, it could test each call more deeply. For instance, if a system call has several parameters, where one is a pointer that is checked for validity, any invocation of this call with a bad pointer can return with a SIGSEGV error, regardless of the values of the other parameters. Therefore, the potential for testing the other parameters is masked by the error that is always returned as a result of the bad pointer parameter.

The validity constraint on pointer parameters is satisfied by using the tester in *aggressive mode*, essentially bypassing the first layer of defense inside the kernel. Aggressive mode creates a buffer of random data in user space for each possible system call parameter. Each system call is then made with all parameters being pointers to unique buffers in user memory, guaranteeing that the system call cannot be rejected because of an invalid pointer.

There are other cases of special parameters that can be identified as not valid. For instance, a file number is expected to be a small positive integer. After finding that the aggressive mode did not yield any interesting results (see Section 3.0), we decided to attempt to cover the possible input combinations as completely as possible by including the normal (non-logging) mode. Although it admittedly results in a very sparse coverage of the space of all possible parameters, it allows the largest possible number of calls to be attempted in a given amount of time. The hope is that random brute-force attack will cover as many special cases as possible—that is the essence of the fuzz testing method.

2.3 Handling poorly-behaved system calls

Many system calls exhibit behavior that interferes with the operation of the tester. For instance, the `exit()` system call will cause the tester to exit, and the `pause()` system call will cause the tester to sleep until it receives a signal (which will never happen). These special calls can be tested by using the tool *`runmany`*. `Runmany` serially executes multiple instances of the tester, creating a new child process for each. It can be told on the command line how many cases to run, i.e. instances of the fuzz tester to start, and for each one, how many tests should be performed. `Runmany` monitors the behavior of the tester process, recognizing if it exits prematurely or hangs. If the tester com-

1. A creative tester implementation might parse manual pages to find information about system call parameters.

pletes successfully, it sends the `SIGUSR1` signal to `runmany`. The signal is needed because random calls to `exit()` made by the fuzz tester can produce random return values. `Runmany` creates a summary file. For every case, it records the random number seed used and the result of the fuzz tester's execution. In the event of a crash, the last case can be restarted with logging turned on to find the specific problem. `Runmany` also allows the automated testing of millions of calls by specifying large numbers of cases and tests.

3.0 Using the Fuzz Tester

Installation of the kernel tester is done by unpacking the archive and compiling all components with one application of make. For all systems that we tested, the installation required no user intervention. After installation, the test procedure requires four main steps to locate problematic system calls, as diagrammed in Figure 1.

Step ① creates an input file for the tester that maps system call numbers to names, and specifies which calls should be tested. In all cases but IRIX, this is accomplished by parsing `/usr/include/sys/syscall.h` with a simple script. IRIX did a number of things differently and required using a modified script to parse `/usr/include/sys.s`. The output of the script is a file with two columns: the system call number and the system call name. The script removes most special-case system calls that need to be taken out for proper testing on all systems. All systems have additional calls that are easily identified and removed manually by iterating through steps ② and ③ below.

In step ②, run the tester with logging turned on, typically set to do 1000 calls. In our experience, both the system that had kernel bugs and systems that had special-case calls that needed to be removed displayed them on the first run. After the tester exits, hangs, or crashes, the last entry in the log file contains the problematic system call (system calls are logged right before they are executed). In step ③, comment this system call out of the input file. For calls that triggered kernel crashes or hangs, the log file for that run should be renamed and saved for later investigation. Steps ② and ③ are repeated. After a few iterations, you reach a point of stability with the tester completing one thousand cases and not exiting or crashing. At this point, all of the problematic system calls have probably been removed.

Step ④ uses the `runmany` program to make millions of calls with logging turned off. This allows the random inputs to cover a wide range in the space of all possible inputs, hopefully triggering bugs that require more special-

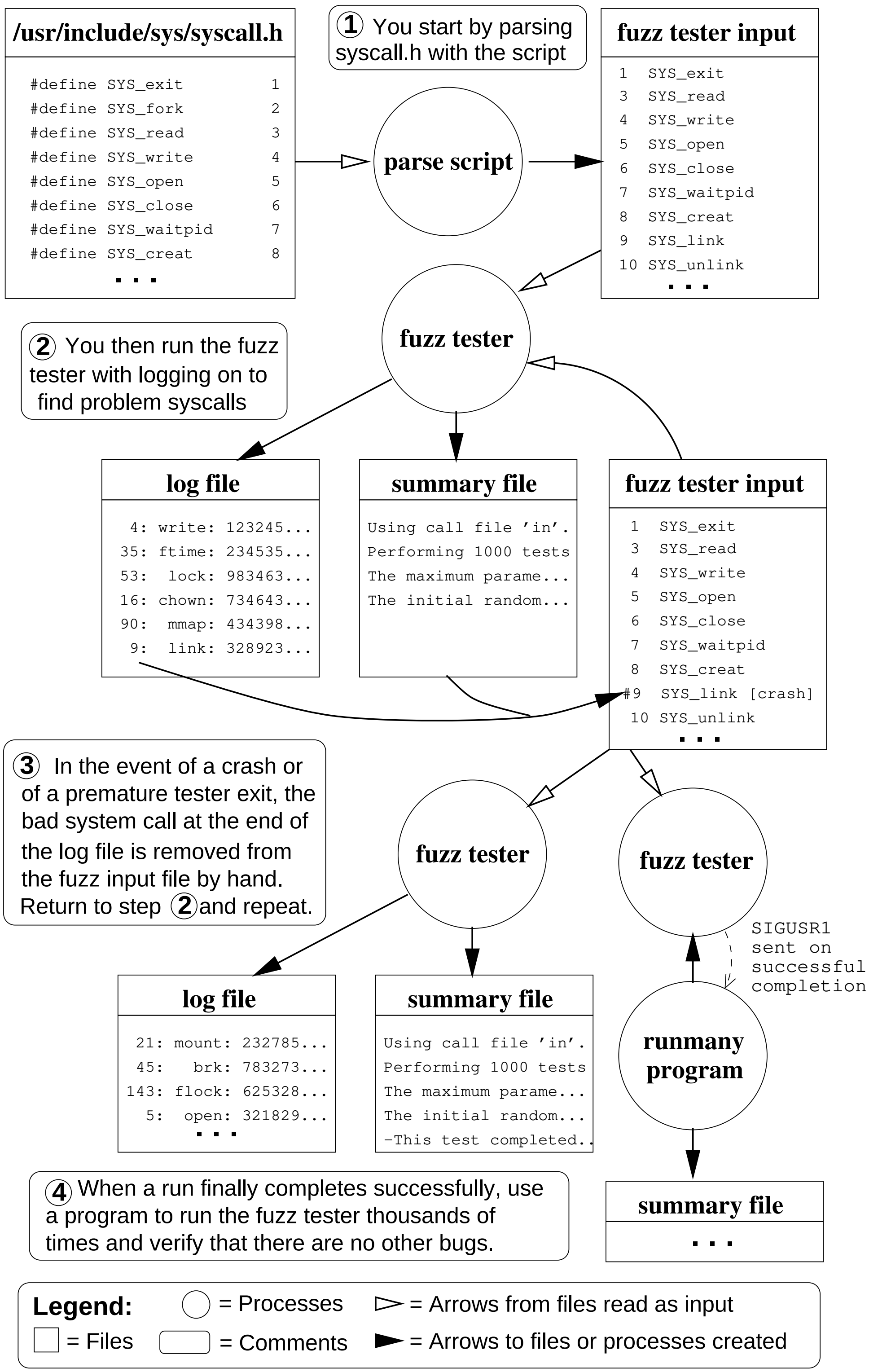

FIGURE 1. Kernel testing methodology as described in the text.

ized inputs. `Runmany` is typically set to perform 1000 cases, of 1000 tests each. In the event of a crash, the last case can be started again in logging mode to find the problem. Since logging mode is so time consuming compared to regular mode (see Table 1), using very large numbers of tests per case can become unreasonable when trying to investigate problems.

All of the systems we tested (with the exception of IRIX) were stable. Once the special-case system calls were isolated, the tester ran for hours at a time, testing hundreds of millions of system calls without any problems.

After the above testing is completed, the tester's aggressive mode (see Section 2.2) can be tried. Note that this method can generate huge amounts of garbage files in the testing directory (due to calls that create files or directories), some of which are difficult to remove. In our experience, aggressive mode yielded no more errors than regular, purely random testing.

The final stage of testing is to look more closely at the problematic calls that were isolated in the previous steps. System calls that caused abnormal behavior can be investigated to find the types of inputs that trigger the bug. We provide several small tools to aid investigation of specific system calls. First, we provide a tool that allows the user to make a single system call exactly as the tester does, providing the arguments to the call on the command line. We also provide the skeleton of a short program that can be modified to see if doing the system call via the library wrapper functions causes the same problems or if the library filters the inputs. Last, we provide a script that reads a list of many calls and creates separate files containing each one individually. This allows the user to easily focus the tester on any single system call, which can be useful in finding specific inputs that cause problems, or in verifying that a given call is behaving normally.

By using the `runmany` program in conjunction with the single system call files, we were able to test the problematic calls that were eliminated in steps ② and ③ (see Section 3.1). In the event of an exit, runmany will start another case immediately. In the event of a hang, runmany will time out, kill the hung instance of the tester, and start another case.

TABLE 2. The list of system calls excluded for each kernel. Underlined calls are automatically removed by the script that builds the input file. Calls in italics are tested using the `runmany` program.

Type of Call	Solaris 2.5.1	Digital UNIX 3.2	HP-UX 9.05	Linux	IRIX 6.2
Exit–causes the process to exit	*exit* *sigreturn* *lwp_exit*	*exit* *sigreturn*	*exit* *sigreturn*	*exit* *sigreturn*	*exit* *sigreturn*[b]
Sleep–causes the process to hang	*pause* *select* *sigsuspend* *sigtimedwait* *lwp_sema_wait* *nanosleep*	*wait4* *select* *sigsuspend*	*pause* *select* *sigsuspend*	*pause* *select* *sigsuspend*	*pause* *select* *sigsuspend* *sginap*
Fork–creates unwanted copies of the tester	fork vfork fork1	fork vfork	fork vfork	fork clone	fork sproc sprocsp
Not defined–these calls are not defined because they are not implemented or have been replaced by user-level library routines	syscall ptrace xenix plock syssun evsys evtrapret hrtsys acancel async clocal vtrace	syscall chflags fchflags security	swapoff gethostid sethostid test bigio pipenode rmtprocess dskless_stats bfactl ulconnect ulcontrol ulcreate uldest ulrecv ulrecvcn ulsend ulshutdown getmsg putmsg getpmsg putpmsg strioctl		syscall stat
Miscellaneous[a]		*kmodcall*	*async_daemon*	*vm86*	*async_daemon* getitimer[b] nsproc setcontext[b]

a. These calls caused problems based on some special case. Each is explained in the section of the text that discusses the system in which it appears.

b. These calls caused kernel crashes. Each is discussed in the text.

3.1 The problematic system calls

Problematic calls fall into five main categories on the systems we tested, as shown in Table 2. The first group is calls that cause the tester to exit immediately, such as `exit()` or `sigreturn()`. The second group is calls that can cause the tester to hang, such as `pause()` or `sigsuspend()`. Both of these types of calls were isolated and tested separately to ensure that the testing of the remaining calls would proceed rapidly, without interruption. The third group of calls is `fork()` and its relatives. They create extra copies of the testing process and produce log files that are difficult to interpret. Application of the fuzz tester to `fork()` is uninteresting because it requires no parameters. The fourth group is system calls that exist in `syscall.h`, but are not implemented within the kernel. Attempts to make these calls generate the message "Bad System Call.[1]" Some of these calls are implemented as library functions instead of functions within the kernel.[2] Others seem to be calls that are optionally compiled into the kernel. The final group of calls are miscellaneous calls that varied from system to system. Some caused the tester to hang, others caused it to exit prematurely. Also included in this group are the calls that caused kernel crashes.

4.0 Results

This section describes our experiences when applying the fuzz tester to the kernels of several systems. The systems and the degree to which each failed are summarized in Table 3.

TABLE 3. The systems that were evaluated and a summary of the results for each.

Operating System	Platform	Results
Solaris 2.5.1[a] (also called SunOS 5.5.1)	Sun SPARC 20 workstation	The kernel is stable. We found an AFS problem.
Digital UNIX 3.2	Alpha	The kernel is stable.
HP-UX 9.05	HPPA RISC	The kernel is stable. We found a poorly handled pointer.
Linux[a]	Intel Architecture PC clone	The kernel is stable.
IRIX 6.2	SGI Indigo2	The kernel is not stable.

a. Source code was available for these systems.

1. The message text varies between kernel implementations.
2. If these system calls are library functions and not system calls at all, why are they included in `syscall.h`? We consider this a bug.

4.1 Solaris 2.5.1

Solaris 2.5.1 was tested using a Sun SPARC 20 workstation. The Andrew File System (AFS) [2] is used in the environment that we tested.

The fuzz tester was developed on this platform. This, and the fact that this kernel proved to be stable in all cases resulted in very aggressive testing of this system. Each method of kernel testing that we developed was first applied to this kernel. Many of the methods were created with the intent of discovering problems with it.

In total, more than 100 million random system calls with random data were made to the Solaris kernel. Its behavior was stable in all cases. However, we did find a method of crashing the kernel by exploiting its interaction with AFS.

Early versions of the fuzz tester randomly chose the number of parameters to put on the stack for each test. It was therefore possible for the called routine to write values past the end of the stacked parameter list, overwriting the tester's local variables. In some cases, the mangled values caused the tester to core dump. A core dump would normally not cause a problem, but in this case, random calls to `brk()` had increased the data segment of the process to a large size (>200MB). When the kernel tried to write the large core file to the AFS, it overflowed the local file cache, and eventually locked the system.

4.2 Digital UNIX 3.2

Digital Unix proved to be a stable system. The Digital Unix kernel survived hundreds of millions of purely random calls, as well as aggressive testing.

As shown in Table 2, the standard system calls were removed from the list of calls to test. Also removed were several calls that fit the bad system call category. The only unusual case for Digital Unix was `kmodcall()`. There is no mention of this call in any include files (other than `syscall.h`) or man pages. This routine hung the testing process most of the time it was called. It is still unclear what it is trying to do, and therefore if it is a bug that is causing the hanging or if that is the expected behavior. This call was removed to allow the automated scripts to run without interruptions.

4.3 HP-UX 9.05

HP-UX 9.05 was tested on a 700 series HP workstation. The kernel proved to be stable in all cases. However, we did find evidence of an exception case that the kernel handles incorrectly.

The mechanism that the kernel uses to report exceptions is by returning `-1` from the system call. The calling program must then inspect the value of the integer variable `errno` to determine what exception occurred. No error should be reported from within the kernel.

Under HP-UX, if a pointer is passed to and referenced by the kernel, there are three cases to consider:

1. The pointer references memory that is contained in the current user data or stack segment.
2. The pointer references memory that can never be contained in the user or stack segment, as defined by the start of each segment and the maximum size of the segment. The maximum size of the segment is determined when the kernel is built.
3. The pointer references memory that is not defined by the first two cases. That is, the memory is not currently part of the user data or stack segment, but it is at a location that could become part of one of these segments if the segment were to grow.

Case 1 represents a valid memory reference and generates no error. Case 2 is handled in the expected manner: `-1` is returned from the system call and `errno` is set to `SIGSEGV`. Case 3 should be treated the same as case 2, but is not. Instead, case 3 causes the kernel to emit an error message to `stderr`, and return `-1` with `errno` set to `SIGSEGV`.[1] The error message can be suppressed by using the `signal()` mechanism to either ignore segmentation violations or to install a user handler. However, the kernel does not call the user handler, so the only behavior that can be produced is the behavior produced by case 2, which is conveniently the correct behavior.

The only special case call that needed to be removed from the input file was `async_daemon()`, an NFS-related system call that never returns.

4.4 Linux

Linux proved to be a stable system under heavy abuse. The Linux kernel survived hundreds of millions of purely random calls, as well as aggressive testing. The only special-case calls that needed to be removed were `clone()`, which is Linux's more powerful version of `fork()`, and `vm86()`. `vm86()` tries to make the system enter virtual

1. This happens whether the system call is accessed via the `syscall()` mechanism or through the system library wrapper.

8086 mode. This call almost always caused the tester to abort. It was removed from the list so the fully-automated scripts that ran millions of cases could proceed without interruption.

4.5 IRIX 6.2

IRIX 6.2 provided us with obvious kernel bugs. Two system calls were isolated in a matter of minutes, each of which caused the machine we were testing to either kernel panic and crash, or to hang completely (the mouse and keyboard stopped working and the machine no longer responded to pings, telnet requests, or other network operations). A third system call that causes a kernel crash was discovered during the application of the `runmany` program to the list of problematic calls.

Our experience with IRIX was much different than the other systems we tested. IRIX does not store its system call information in `syscall.h`, as the other versions of Unix did. Instead, IRIX stores it in `/usr/include/sys.s`. Also, IRIX offsets all the system call numbers by 1000, so `read()` is syscall number 1003 instead of number 3. This required some extra work in the parsing script.

The first kernel bugs we found were in the IRIX implementation of `setcontext()` and `getitimer()`. `setcontext()` expects just one argument: a pointer to a context structure. If this pointer is invalid, instead of returning `-1` and setting `errno` to `EFAULT` (like other system calls that are given bad pointers), the kernel panics and the system crashes in nearly all cases. Calling `setcontext()` either directly with `syscall()` or via the library function yields the same result.

The bug in `getitimer()` results in either the machine hanging or a kernel panic, depending on the parameters used. `getitimer()` expects two parameters: an integer and a pointer. `getitimer()` properly checks for invalid pointers and returns `-1`, setting `errno` to `EFAULT` as you would expect. However, when the integer is a negative value below `-851`, the machine hangs. For certain ranges of very large negative numbers, the call to `getitimer()` results in a bus error and kernel panic, instead of a hang. Like the bug in `setcontext()`, this bug can be triggered via either the library wrapper or a direct call.

Other special case calls that needed to be removed from the input file were `async_daemon()`, an NFS-related system call that never returns, and `nsproc()`, which usually hung the tester.

`Runmany` was used to test the previously excluded calls and another kernel bug was found in the implementation of `sigreturn()`. For a small range of values of `sigreturn()` 's parameter,[1] the kernel panics and dumps core. This call has a comment in the `sys.s` file that reads:

```
/* This system call is internal for mips signal handling code */
```

There is no library function for `sigreturn()` and no man pages. It seems as though the designers of IRIX never intended the call to be made from outside of the kernel. If this were the case, it would be easy for them to simply return -1 and set errno to `EPERM` ("operation not permitted") as is done in similar cases. However, the call is allowed to proceed, and for a small range of values, the call produces a kernel panic and crash.

5.0 Other Tools

Another program for testing a kernel that is based on random input is called *crashme* and was written by George J. Carrette.[2] Crashme works by generating a random string of bytes, then executing that string as if it were a function. This program has been used for years by the developers of the Linux kernel to test for bugs. Crashme's success can be easily seen by the number of references to it in bug reports to the Linux kernel developer's mailing list and newsgroups. The use of crashme during development of the Linux kernel may have contributed to Linux's invulnerability to our tool.

6.0 Summary

Using random data proved to be a simple and effective way to test the robustness of many aspects of operating systems. Bugs that can be the source of unwanted crashes, security holes, or other undesirable effects have been found inside standard utilities, X-Window clients, and the kernel itself. The tool described here can be used for easy testing of any modern Unix kernel. The developers of these kernels would benefit from using this tool as a way to improve the reliability and security of their systems. To their credit, most of the systems we tested survived our tool's abuses, which indicates a high degree of fault tolerance within the kernel. However, the failure of the most current release of IRIX, a major operating system used by engineers to design things like the Boeing 777, speaks to the con-

1. In this case, we were unable to determine any meaningful pattern to the values of the parameter which caused the crash.
2. George Carrette can be contacted at `GJC@BU-IT.BU.EDU`.

tinued need for this form of testing. Every new release of an operating system should be subjected to a few days of kernel fuzz testing. In the case of IRIX, its bugs would have been discovered in the first half-hour, and could have been fixed.

Acknowledgments

We would like to thank Todd Tannenbaum and Jeff Ballard of the Model Advanced Facility at the UW College of Engineering for the use of an SGI Indigo2 workstation that suffered many hard crashes as a result of our testing. We would also like to thank Bart Miller, Oscar Naim, and the Paradyn project at the UW Computer Sciences Department for the use of an HP series 700 workstation. It did not suffer as badly as the SGI, but the potential was there, and we appreciate the sacrifice made by its users. We would also like to thank Sun Microsystems for making the Solaris source code available to us.

References

[1] Barton P. Miller, et al. “Fuzz Revisited: A Re-examination of the Reliability of Unix Utilities and Services,” Preliminary copy, November 9, 1995, 23 pages.

[2] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West, “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems*, **6** 1, pp. 51–81. February 1988.