# The Exception Handling Effectiveness of POSIX Operating Systems

Philip Koopman, *Senior Member*, *IEEE*, and John DeVale, *Student Member*, *IEEE*

**Abstract**—Operating systems form a foundation for robust application software, making it important to understand how effective they are at handling exceptional conditions. The Ballista testing system was used to characterize the handling of exceptional input parameter values for up to 233 POSIX functions and system calls on each of 15 widely used operating system (OS) implementations. This identified ways to crash systems with a single call, ways to cause task hangs within OS code, ways to cause abnormal task termination within OS and library code, failures to implement defined POSIX functionality, and failures to report unsuccessful operations. Overall, only 55 percent to 76 percent of the exceptional tests performed generated error codes, depending on the operating system being tested. Approximately 6 percent to 19 percent of tests failed to generate any indication of error despite exceptional inputs. Approximately 1 percent to 3 percent of tests revealed failures to implement defined POSIX functionality for unusual, but specified, situations. Between 18 percent and 33 percent of exceptional tests caused the abnormal termination of an OS system call or library function, and five systems were completely crashed by individual system calls with exceptional parameter values. The most prevalent sources of these robustness failures were illegal pointer values, numeric overflows, and end-of-file overruns. There is significant opportunity for improving exception handling within OS calls and especially within C library functions. However, the role of signals vs. error return codes is both controversial and the source of divergent implementation philosophies, forming a potential barrier to writing portable, robust applications.

**Index Terms**—Exception handling, POSIX, operating systems, robustness, testing, Ballista, multiversion comparison.

## 1 Introduction

System crashes are a way of life in any real-world system, no matter how carefully designed. Software is increasingly becoming the source of system failures, and the many software failures seem to be due to problems with exception handling (e.g., [5]). Thirty years ago, the Apollo 11 space flight experienced three mission-threatening computer crashes and reboots during powered descent to lunar landing, caused by exceptional radar configuration settings that resulted in the system running out of memory buffers [17]. Decades later, the maiden flight of the Ariane 5 heavy lifting rocket was lost due to events arising from a floating point-to-integer conversion exception [23]. Now that our society relies upon computer systems for everyday tasks, exceptional conditions are routinely causing system failures in more everyday applications such as telecommunication systems and desktop computing. Since even expensive systems designed with robustness specifically in mind suffer such failures, it seems likely that general purpose systems might be even more vulnerable to incomplete or nongraceful exception handling.

The robustness of a system can depend in large part on the quality of exception handling of its operating system (OS). It is difficult to produce a robust software application, but the task becomes even more difficult if the underlying OS upon which the application is built does not provide extensive exception handling support. This is true not only of desktop computing systems, but also of embedded systems such as telecommunications and transportation applications that are now being built atop commercial operating systems. A trend in new Application Programming Interfaces (APIs) is to require comprehensive exception handling (e.g., CORBA [28] and HLA [10]). Unfortunately, while the POSIX API [16] provides a mechanism for exception reporting in the form of error return codes, implementation of this mechanism is largely optional. This results in uncertainty when adopting a POSIX operating system for use in a critical system, and leads to two important questions: 1) Given the lack of a firm requirement for robustness by the POSIX standard, how robust are actual Commercial Off-The-Shelf (COTS) POSIX implementations? 2) What should application programmers do to minimize the effects of nonrobust OS behavior?

These questions can be answered by creating a direct, repeatable, quantitative assessment of OS exception handling abilities. Such an evaluation technique would give the developers feedback about a new OS version before it is released, and present the opportunity to measure the effectiveness of attempts to improve robustness. Additionally, quantitative assessment would enable system designers to make informed comparison shopping decisions when selecting an OS, and would support an educated "make/buy" decision as to whether a COTS OS might in fact be more robust than an existing proprietary OS. Alternately, knowledge about the exception handling weak spots of an OS would enable application designers to take extra precautions in known problem spots.

• *The authors are with the Department of Electrical and Computer Engineering, Institute for Complex Engineered Systems, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: {koopman, devale}@cmu.edu.*

*Manuscript received 2 Mar. 1999; revised 15 Dec. 1999; accepted 28 Jan. 2000.*
*Recommended for acceptance by A. Romanovsky.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111488.*

The Ballista software robustness testing system automatically tests the exception handling capabilities of APIs [22], [7], [20]. While it can be used for testing APIs beyond operating systems (e.g., a simulation framework [11]), the focus here is on describing POSIX OS test results that may be useful to critical system designers. In brief, the Ballista testing methodology involves automatically generating sets of exceptional parameter values to be used in calling software modules. The results of these calls are examined to determine whether a software module detected and notified the calling program of an error, and whether the task or even system suffered a crash or hang as the result of a call.

POSIX exception handling tests were conducted using Ballista on fifteen POSIX operating system versions from ten different vendors across a variety of hardware platforms. More than one million tests were executed in all, covering up to 233 distinct functions and system calls for each OS. Many of the tests identified instances in which exceptional conditions were handled in a nonrobust manner, ranging in severity from complete system crashes to false indication of success for system calls. Other tests managed to uncover exception-related software defects that apparently were not caught by the POSIX certification process.

Beyond the robustness failure rates measured, analysis of test data and discussion with OS vendors reveals a divergence in approaches to dealing with exceptional parameter values. Some operating systems attempt to use the POSIX-documented error codes to provide portable support for exception reporting at run time. Alternately, some operating systems emphasize the generation of a signal (typically resulting in abnormal process termination) when exceptional parameter values are encountered in order to facilitate debugging. However, there is no way to generalize which OSs handle what situations in a particular manner, and all OSs studied failed to provide either indication of exceptions in a substantial portion of tests conducted. While it is indeed true that the POSIX standard itself does not *require* comprehensive exception reporting, it seems likely that a growing number of applications will need it. Evaluating current operating systems with respect to exception handling is an important first step in understanding whether change is needed, and what directions it might take.

The following sections describe previous work, the testing methodology used, robustness testing results, what these results reveal about current operating systems, and potential directions for future research.

## 2 Previous Work

While the Ballista robustness testing method described here is a form of software testing, its heritage traces back not only to the software testing community, but also to the fault tolerance community as a form of software-based fault injection. Ballista builds upon more than fifteen years of fault injection work at Carnegie Mellon University, including [30], [6], [2], [31], [8], and [9], and makes the contribution of attaining scalability for cost-effective application to a reasonably large API. In software testing terms, Ballista is performing tests for responses to exceptional input conditions (sometimes called "dirty" tests, which involve exceptional situations, as opposed to "clean" tests of correct functionality in normal situations). The test ideas used are based on "black box," or functional testing techniques [3] in which only functionality is of concern, not the actual structure of the source code. However, Ballista is concerned with determining how well a software module handles exceptions rather than with functional correctness.

Some people only use the term robustness to refer to the time between operating system crashes under some usage profile. However, the authoritative definition of *robustness* is "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [15]." This expands the notion of robustness to be more than just catastrophic system crashes and encompasses situations in which small, recoverable failures might occur. While robustness under stressful environmental conditions is indeed an important issue, a desire to attain highly repeatable results has led the Ballista project to consider only robustness issues dealing with invalid inputs for a single invocation of a software module from a single execution thread. It can be conjectured, based on anecdotal evidence, that improving exception handling will reduce stress-related system failures, but substantive results in that area remain a subject of future work.

An early method for automatically testing operating systems for robustness was the development of the Crashme program [4]. Crashme operates by writing random data values to memory, then spawning large numbers of tasks that attempt to execute those random bytes as concurrent programs. While many tasks terminate almost immediately due to illegal instruction exceptions, on occasion a single task or a confluence of multiple tasks can cause an operating system to fail. The effectiveness of the Crashme approach relies upon serendipity—in other words, if run long enough Crashme may eventually get lucky and find some way to crash the system.

Similarly, the Fuzz project at the University of Wisconsin has used random noise (or "fuzz") injection to discover robustness problems in operating systems. That work documented the source of several problems [25], and then discovered that the problems were still present in operating systems several years later [26]. The Fuzz approach tested specific OS elements and interfaces (as opposed to the completely random approach of Crashme), although it still relied on random data injection.

Other work in the fault injection area has also tested limited aspects of robustness. The FIAT system [2] uses probes placed by the programmer to alter the binary process image in memory during execution. The FERRARI system [18] is similar in intent to FIAT, but uses software traps in a manner similar to debugger break-points to permit emulation of specific system-level hardware faults (e.g., data address lines, condition codes). The FTAPE system [33] injects faults into a system being exercised with a random workload generator by using a platform-specific device driver to inject the faults. While all of these systems have produced interesting results, none was intended to quantify robustness on the scale of an entire OS API.

While the hardware fault tolerance community has been investigating robustness mechanisms, the software engineering community has been working on ways to implement robust interfaces. As early as the 1970s it was known that there are multiple ways to handle an exception [14], [13]. More recently, the two methods that have become widely used are the signal-based model (also known as the termination model) and the error return code model (also known as the resumption model).

In an error return code model, function calls return an out-of-band value to indicate an exceptional situation has occurred (for example, a NULL pointer might be returned upon failure to create a data structure in the C programming language). This approach is the supported mechanism for creating portable, robust systems in the POSIX API [16, lines 2,368-2,377].

On the other hand, in a signal-based model, the flow of control for a program does not address exceptional situations, but instead describes what will happen in the normal case. Exceptions cause signals to be "thrown" when they occur, and redirect the flow of control to separately written exception handlers. It has been argued that an signal- based approach is superior to an error return code approach based, in part, on performance concerns, and because of ease of programming [12], [5]. However, unlike CORBA and some other interfaces, the POSIX API does not standardize a portable way to use this approach to create robust systems.

The Xept approach [35] provides an generalized framework that illustrates how fine-grain, robust exception handling can be used in critical systems. Xept uses software "wrappers" around procedure calls as a way to encapsulate error checking and error handling within the context of a readable program. In fact, the Xept approach could be used not only to implement graceful recovery from reported exceptions, but also to provide a layer of protection against OS robustness problems by filtering out potentially dangerous parameter values from being sent to the OS. Approaches along these lines seem to be commonly used by industry in designing critical systems, but are typically done in an ad hoc manner and seldom discussed in the literature. Given that the fault tolerance community has found that transient system failures far outnumber permanent system faults/design errors in practice, even as simple a strategy as retrying failed operations from within a software wrapper has the potential to significantly improve system robustness. Thus, there is good reason in many applications to desire robust exception handling from an OS rather than abnormal task termination upon every exception.

## 3 Ballista Testing Methodology

The Ballista robustness testing methodology is based on combinational tests of valid and invalid parameter values for system calls and functions. In each *test case*, a single software Module under Test (or *MuT*) is called once to determine whether it provides robust exception handling when called with a particular set of parameter values. These parameter values, or *test values*, are drawn from a pool of normal and exceptional values based on the data type of each argument passed to the MuT. A test case therefore consists of the name of the MuT and a tuple of test values that are passed as parameters (i.e., a test case would be a procedure call of the form: *MuT_name(test_value1, test_value2, ..., test_valueN)*). Thus, the general Ballista approach is to test the robustness of a single call to a MuT for a single tuple of test values, and then iterate this process for multiple test cases that each have different combinations of valid and invalid test values.

### 3.1 Test Cases Based on Data Types

The Ballista approach to robustness testing has been implemented for a set of 233 POSIX functions and calls defined in the IEEE 1003.1b standard [16] ("POSIX.1b" or "POSIX with real-time extensions with C language binding"). All standard calls and functions were tested except for calls that take no arguments, such as `getpid()`; calls that do not return, such as `exit()`; and calls that intentionally send signals, such as `kill()`.

For each POSIX function tested, an interface description was created with the function name and type information for each argument. In some cases, specific information about argument use was exploited to result in better testing (for example, a file descriptor might be of type `int`, but was described to Ballista as a more specific file descriptor data type).

As an example, Fig. 1 shows the actual test values used to test `write(int filedes, const void *buffer, size_t nbytes)`, which takes parameters specifying a file descriptor, a memory buffer, and a number of bytes to be written. Because `write()` takes three parameters of three different data types, Ballista draws test values from separate test objects established for each of the three data types. In Fig. 1, the arrows indicate that the particular test case being constructed will test a file descriptor for a file which has been opened with only read access, a NULL pointer to the buffer, and a size of 16 bytes. Other combinations of test values are assembled to create other test cases. In the usual case, all combinations of test values are generated to create a combinatorial number of test cases. For a half-dozen POSIX calls, the number of parameters is large enough to yield too many test cases for exhaustive coverage within a reasonable execution time. In these cases a pseudorandom sampling of 5,000 test cases is used. (Based on a comparison to a run with exhaustive searching on one OS, this sampling gives results accurate to within 1 percentage point for each function.)

Each test value (such as `FD_OPEN_READ` in Fig. 1) refers to a pair of code fragments that are kept in a simple database comprised of a specially formatted text file. The first fragment for each test value is a constructor that is called before the test case is executed (it is not literally a C++ constructor, but rather a code fragment identified to the test harness as constructing the instance of a test value). The constructor may simply return a value (such as a NULL), but may also do something more complicated that initializes system state. For example, the constructor for `FD_OPEN_READ` creates a file, puts a predetermined set of bytes into the file, opens the file for read, then returns a file descriptor for that file. The second of the pair of the code fragments for each test value is a destructor that deletes any

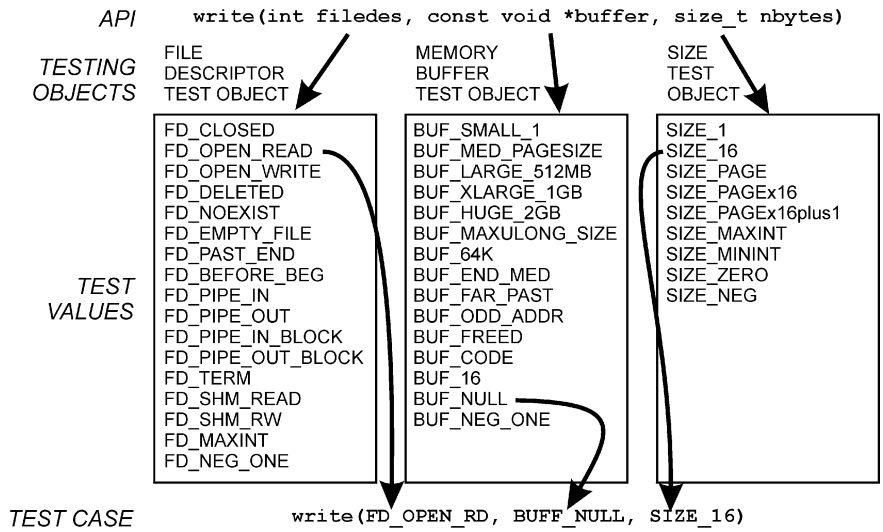



Fig. 1. Ballista test case generation for the `write()` function. The arrows show a single test case being generated from three particular test values; in general, all combinations of test values are tried in the course of testing.

data structures or files created by the corresponding constructor (for example, the destructor for `FD_OPEN_READ` closes and deletes the file created by its matching constructor). Tests are executed from within a test harness by having a parent task fork a fresh child process for every test case. The child process first calls constructors for all test values used in a selected test case, then executes a call to the MuT with those test values, then calls destructors for all the test values used. Special care is taken to ensure that any robustness failure is a result of the MuT, and not attributable to the constructors or destructors themselves. Functions implemented as macros are tested using the same technique, and require no special treatment.

The test values used in the experiments were a combination of values suggested in the testing literature (e.g., [24]) and values selected based on personal experience. For example, consider file descriptor test values. File descriptor test values include descriptors to existing files, negative one, the maximum integer number (MAXINT), and zero. Situations that are likely to be exceptional in only some contexts are tested, including file open only for read and file open only for write. File descriptors are also tested for inherently exceptional situations such as file created and opened for read, but then deleted from the file system without the program's knowledge.

The guideline for test value selection for all data types were to include, as appropriate: zero, negative one, maximum/minimum representable values, pointers to nonexistent memory, lengths near virtual memory page size, pointers to heap-allocated memory, files open for combinations of read/write with and without exceptional permission settings, and files/data structures that had been released before the test itself was executed. While creating generically applicable rules for thorough test value selection remains a subject of future work, this experience-driven approach was sufficient to produce useful results.

It is important to note that this testing methodology does not generate tests based on a description of MuT functionality, but rather on the data types of the MuT's arguments. This approach means that per-MuT "scaffolding" code does not need to be written. As a result, the Ballista testing method is highly scalable with respect to the amount of effort required per MuT, needing only 20 data types to test 233 POSIX MuTs. An average data type has 10 test cases, each having 10 lines of C code, meaning that the entire test suite required only 2,000 lines of C code for test cases (in addition, of course, to the general testing harness code used for all test cases and various analysis scripts).

An important benefit derived from the Ballista testing implementation is the ability to automatically generate the source code for any single test case the suite is capable of running. In many cases, only a dozen lines or fewer of executable code in size, these short programs contain the constructors for each parameter, the actual function call, and destructors. These single test case programs have been used to reproduce robustness failures in isolation for use by OS developers and to verify test result data.

### 3.2 Categorizing Test Results

After each test case is executed, the Ballista test harness categorizes the test results according to the first three letters of the "C.R.A.S.H." severity scale [22]:

- **Catastrophic** failures occur when the OS itself becomes corrupted or the machine crashes and reboots.
- **Restart** failures occur when a call to a MuT never returns control to the caller, meaning that in an application program a single task would be "hung," requiring intervention to terminate and restart that task. These failures are identified by a watchdog timer which times out after several seconds of waiting for a test case to complete. (Calls that wait for I/O and other such legitimate "hangs" are not tested.)
- **Abort** failures tend to be the most prevalent, and result in abnormal termination (a "core dump," the POSIX SIGSEGV signal, etc.) of a single task.
- **Silent** failures occur when an OS returns no indication of error on an exceptional operation

TABLE 1
Directly Measured Robustness Failures for Fifteen POSIX Operating Systems

| System | POSIX Fns. Tested | Fns. with Catastro-phic Failures | Fns. with Restart Failures | Fns with Abort Failures | Fns. with No Failures | Number of Tests | Abort Failures | Restart Failures | Normalized Abort + Restart Rate |
|---|---|---|---|---|---|---|---|---|---|
| AIX 4.1 | 186 | 0 | 4 | 77 | 108 | 64009 | 11559 | 13 | 9.99% |
| FreeBSD 2.2.5 | 175 | 0 | 4 | 98 | 77 | 57755 | 14794 | 83 | 20.28 |
| HPUX 9.05 | 186 | 0 | 3 | 87 | 98 | 63913 | 11208 | 13 | 11.39 |
| HPUX 10.20 | 185 | **1** | 2 | 93 | 92 | 54996 | 10717 | 7 | 13.05 |
| IRIX 5.3 | 189 | 0 | 2 | 99 | 90 | 57967 | 10642 | 6 | 14.45 |
| IRIX 6.2 | 225 | **1** | 0 | 94 | 131 | 91470 | 15086 | 0 | 12.62 |
| Linux 2.0.18 | 190 | 0 | 3 | 86 | 104 | 64513 | 11986 | 9 | 12.54 |
| Lynx 2.4.0 | 222 | **1** | 0 | 108 | 114 | 76462 | 14612 | 0 | 11.89 |
| NetBSD 1.3 | 182 | 0 | 4 | 99 | 83 | 60627 | 14904 | 49 | 16.39 |
| OSF1 3.2 | 232 | **1** | 2 | 136 | 96 | 92628 | 18074 | 17 | 15.63 |
| OSF1 4.0B | 233 | 0 | 2 | 124 | 109 | 92658 | 18316 | 17 | 15.07 |
| QNX 4.22 | 203 | **2** | 6 | 125 | 75 | 73488 | 20068 | 505 | 20.99 |
| QNX 4.24 | 206 | 0 | 4 | 127 | 77 | 74893 | 22265 | 655 | 22.69 |
| SunOS 4.13 | 189 | 0 | 2 | 104 | 85 | 64503 | 14227 | 7 | 15.84 |
| SunOS 5.5 | 233 | 0 | 2 | 103 | 129 | 92658 | 15376 | 28 | 14.55 |

which clearly cannot be performed (for example, writing to a read-only file). These failures are not directly measured, but can be inferred as discussed in Section 5.2.

- **Hindering** failures occur when an incorrect error code is returned from a MuT, which could make it more difficult to execute appropriate error recovery. Hindering failures have been observed as fairly common (forming a substantial fraction of cases which returned error codes) in previous work [19], but are not further discussed due to lack of a way to perform automated identification within large experimental data sets.

There are two additional possible outcomes of executing a test case. It is possible that a test case returns with an error code that is appropriate for invalid parameters forming the test case. This is a case in which the test case passes—in other words, generating an error code is the correct response. Additionally, in some tests the MuT legitimately returns no error code and successfully completes the requested operation. This happens when the parameters in the test case happen to be all valid, or when it is unreasonable to expect the OS to detect an exceptional situation (such as pointing to an address past the end of a buffer, but not so far past as to go beyond a virtual memory page or other protection boundary).

## 4 Results

A total of 1,082,541 test cases were executed during data collection. Operating systems which supported all of the 233 selected POSIX functions and system calls each had 92,658 total test cases, but those supporting a subset of the functionality tested had fewer test cases.

### 4.1 Raw Testing Results

The compilers and libraries used to generate the test suite were those provided by the OS vendor. In the case of FreeBSD, NetBSD, Linux, and LynxOS, the GNU C compiler version 2.7.2.3 and associated C libraries were used to build the test suite.

Table 1 reports the robustness failure rates as measured by Ballista. In all, there were five MuTs across the tested systems that resulted in Catastrophic failures. Restart failures were relatively scarce, but present in all but two operating systems. Abort failures were common, indicating that in all operating systems it is relatively straightforward to elicit an abnormal task termination from an instruction within a function or system call (Abort failures do not have to do with subsequent use of an exceptional value returned from a system call—they happen in response to an instruction within the vendor-provided software itself).

Any MuT that suffered Catastrophic failures could not be completely tested due to a lack of time for multiple reboots on borrowed equipment, and thus is excluded from failure rate calculations beyond simply reporting the number of MuTs with such failures. A representative test case causing a Catastrophic failure on Irix 6.2 is:

$$\texttt{munmap}(\texttt{malloc}((1 << 30 + 1)), \texttt{MAXINT});$$

Similarly, the following call crashes the entire OS on Digital Unix (OSF 1) version 4.0D:

$$\texttt{mprotect}(\texttt{malloc}((1 << 29) + 1), 65537, 0);$$

Other calls causing Catastrophic failures were: `munmap()` on QNX 4.22, `mprotect()` on QNX 4.22, `mmap()` on HPUX 10, `setpgid()` on LynxOS, and `mq_receive()` on Digital Unix/OSF 3.2. Note that the tables all report Digital Unix version 4.0B, which did not have the Catastrophic failure found in 4.0D, but is otherwise quite similar in behavior.

### 4.2 Normalized Failure Rate Results

Comparing OS implementations simply on the basis of the number of tests that fail is problematic because, while identical tests were attempted on each OS, different OS implementations supported differing subsets of POSIX functionality. Furthermore, MuTs having many parameters

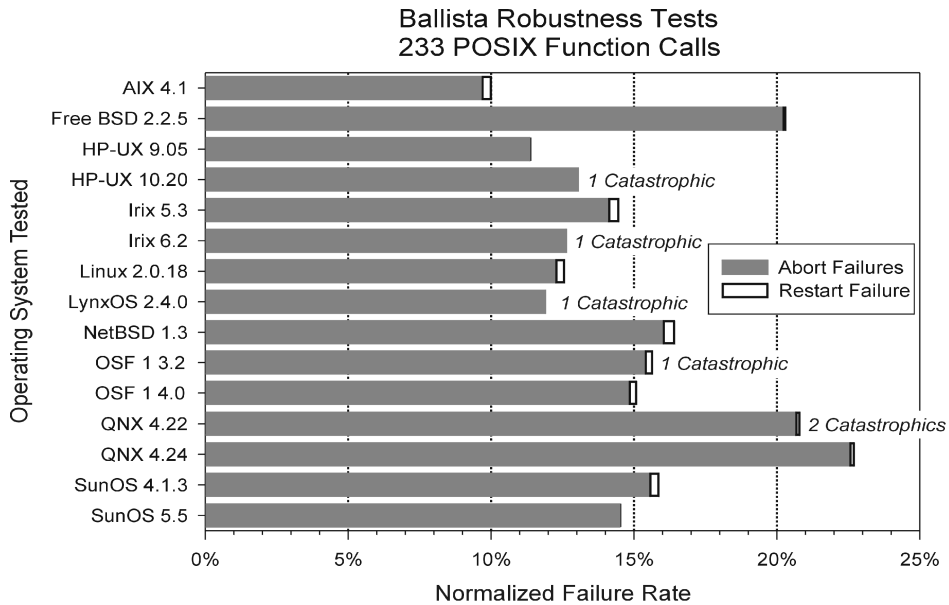



Fig. 2. Normalized failure rates.

execute a large number of test cases, potentially skewing results.

Rather than use raw test results, comparisons should be made based on normalized failure rates. Table 1 shows a normalized failure rate computed by the following process. A ratio of robustness failures to total tests is computed for each MuT within each OS (e.g., a ratio of 0.6 means that 60 percent of the tests failed). Then, the mean ratio across all MuTs for an OS is computed using a simple arithmetic average. This definition produces an exposure metric, which gives the probability that exceptional parameter values of the types tested will cause a robustness failure for a particular OS. This metric has the advantage of removing the effects of differing number of tests per function, and also permits comparing OS implementations with differing numbers of functions implemented according to a single normalized metric.

Overall failure rates considering both Abort and Restart failures range from the low of 9.99 percent (AIX) to a high of 22.69 percent (QNX 4.24). As shown in Fig. 2, the bulk of the failures found are Abort failures. OS implementations having Catastrophic failures are annotated with the number of MuTs capable of causing a system crash.

The first set of experimental data gathered included several relatively old OS versions, representing machines that were in service under a conservative campus-wide software upgrade policy. At the insistence of vendors that newer versions would be dramatically better, tests were run on several borrowed machines configured with the newest available OS releases. The results showed that even major version upgrades did not necessarily improve exception handling capabilities. Failure rates were reduced from Irix 5.3 to Irix 6.2, from OSF 3.2 to OSF 4.0, and from SunOS 4 to SunOS 5, although in all cases the improvement was not overwhelming. However, the failure rates actually increased from HPUX 9 to HPUX 10 (including addition of a Catastrophic failure mode), increased from QNX 4.22 to QNX 4.24 (although with elimination of both Catastrophic failure modes), and stayed essentially identical from OSF 1 4.0B to OSF 1 4.0D (although 4.0D introduced a new Catastrophic failure mode).

### 4.3 Failure Rates Weighted By Operational Profile

The use of a uniformly weighted average gives a convenient single-number metric for comparison purposes. However, it is important to dig a little deeper into the data to determine what functions are driving the failure rates, and whether they are the functions that are frequently used, or instead whether they are obscure functions that don't matter most of the time.

In some situations, it may be desirable to weight vulnerability to exception handling failures by the relative frequency of invocation of each possible function. In other words, rather than using an equal weighting when averaging the failure rates of each MuT in an OS, the average could instead be weighted by relative execution frequency for a "typical" program or set of programs. This approach corresponds to a simple version of operational profiles as used in traditional software testing [27].

Collecting profiling information at the OS system call and function level turned out to be surprisingly difficult for the POSIX API, because most tools are optimized for instrumenting user-written calls rather than OS calls. However, instrumentation of the IBS benchmark suite [34] and the floating point portions of the SPEC95 benchmark suite were possible using the Atom tool set [32] running on Digital Unix to record the number of calls made at run time. Due to problems with compiler option incompatibility between Atom and the benchmark programs, only the IOZone, compress, ftp, and gnuChess programs from IBS were measured.

The results were that the weighted failure rates vary dramatically in both magnitude and distribution among operating systems, depending on the workload being executed. For example, IBS weighted failure rates varied from 19 percent to 29 percent depending on the operating

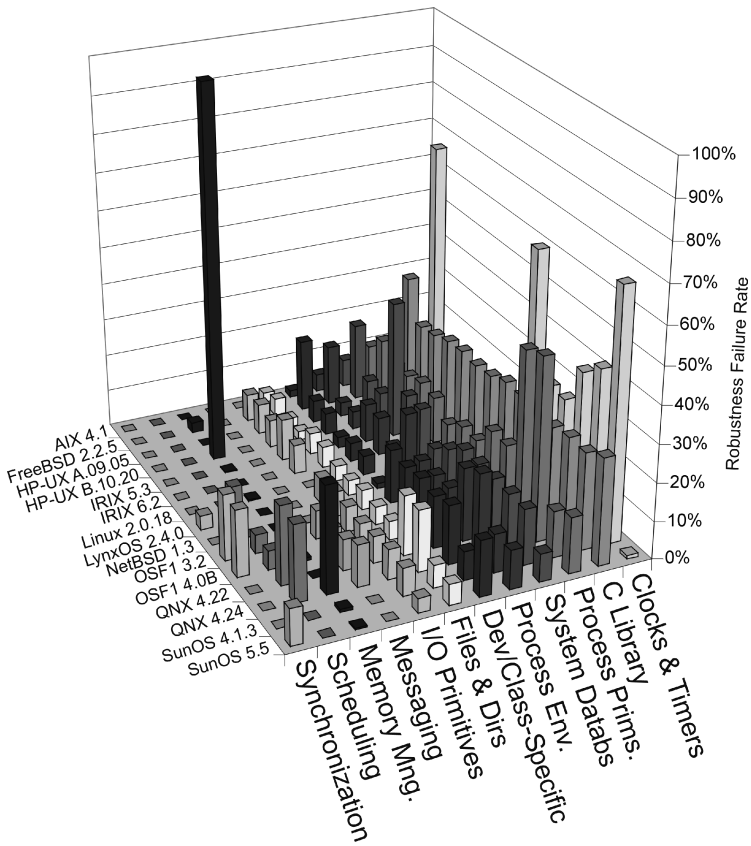



Fig. 3. Normalized failure rates by call/function category, divided per the POSIX document chapters

system. However, for SPEC95 floating point programs, the weighted failure rate was less than 1 percent for all operating systems except FreeBSD. Because FreeBSD intentionally uses a SIGFPE floating point exception signal instead of error return codes in many cases, it happens to have a high percentage of Abort results on functions heavily used by SPEC95.

Specific weighted failure rates are not described because the results of attempting operational profiling point out that there is no single operational profile that makes sense for an interface as versatile as an OS API. The only definite point that can be made is that there are clearly some profiles for which the robustness failure rates could be significant. Beyond that, publishing a weighted average would, at best, be overly simplistic. Instead, readers are invited to obtain the raw OS failure rate data from the authors and apply operational profiles appropriate to their particular application area.

### 4.4 Failure Rates By Call/Function Category

A somewhat different way to view the failure rate data is by breaking up aggregate failure rates into separate failure rates grouped by the type of call or function [29]. This gives some general insight into the portions of the implementations that tend to be robust at handling exceptions without becoming bogged down in the details of individual call/function failure rates.

Fig. 3 shows the failure rate of different categories of calls/functions as grouped within the chapters of the POSIX specification [16]. In this figure, both a general name for the category and an example function from that category are given for convenience. Within each category, failure rates are normalized and averaged with equal per- function weighting factors.

Several categories in Fig. 3 have pronounced failure rates. The clocks and timers category (Section 14 of the Standard) had a bimodal distribution of failure rates: 30 percent to 69 percent for seven of the OS implementations (the visible bars in Fig. 3), and low values for the remaining OSs (the hidden bars are 7 percent for Irix 6.2, 1 percent for SunOS 5.5, and 0 percent for the rest). This set and the memory management set (Section 12 of the Standard, which deals with memory locking/mapping/sharing, but not "malloc"-type C-library operations) are representative of areas in which there is a very noticeable variation among OS implementations with respect to exception handling.

While in many cases failure rates are comparable across OS implementations for the different call categories, there are some bars which show significantly higher failure rates. HPUX 10 has a 100 percent failure rate for memory management functions. Worse, it was one of the memory management calls that produced HPUX 10's Catastrophic system failure, indicating that this area is indeed a robustness vulnerability compared to HPUX 9, which had no such problems. (We have learned that HPUX 10 has a new implementation of these functions, accounting for a potentially higher failure rate.) This and other similar observations indicate that there may be specific areas of reduced exception handling effectiveness within any particular OS.

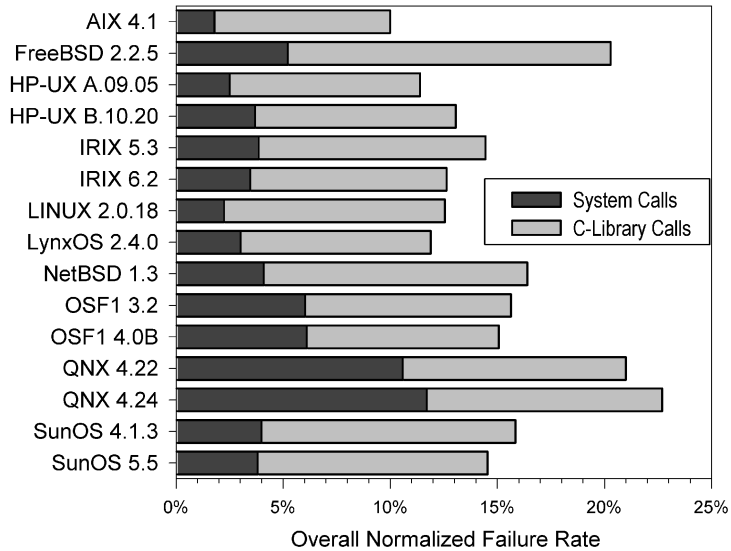



Fig. 4. The C-Library functions contribute a large proportion of the overall raw failure rate.

### 4.5 C-Library Failure Rates

The general failure rate of the C library calls in Fig. 3 is uniformly high across all OS implementations tested. Fig. 4 shows the same data as Fig. 2, but shows the portions of each failure rate that are attributable to the C library functions. Part of the large contribution of C library functions to overall failure rates is that they account for approximately half of the MuTs for each OS (a total of 102 C library functions were tested, out of 175 to 223 total MuTs per OS). The C library functions had a failure rate about the same as system calls on QNX, but had failure rates between 1.8 and 3.8 times higher than system calls on all other OS versions tested. Within the C library, string functions, time functions, and stream I/O tended to have the highest robustness failure rates.

## 5 Data Analysis via N-Version Software Voting

The scalability of the Ballista testing approach hinges on not needing to know the functional specification of a MuT. In the general case, this results in having no way to deal with tests that have no indication of error—they could either be nonexceptional test cases or Silent failures, depending on the actual functionality of the MuT. However, the availability of a number of operating systems that all conform to a standardized API permits estimating and refining failure rates using an idea inspired by multiversion software voting (e.g., [1]). Ballista testing results for multiple implementations of a single API can be compared to identify test cases that are either nonexceptional, or that are likely to be Silent failures. This is, of course, not really multiversion software voting, but rather a similar sort of idea that identifies problems by finding areas in which the various versions disagree as to results for identical tests.

### 5.1 Elimination of Nonexceptional Tests

The Ballista test cases carefully include some test values which are not exceptional in any way. This is done intentionally to prevent the masking of robustness failures. A correctly handled exceptional condition for one value in a tuple of those passed into a function may cause the system to not even look at other values. The concept is similar to obtaining high branch coverage for nested branches in traditional testing. For instance, in the test case: `write(-1,NULL,0)`, some operating systems test the third parameter, a length field of zero, and legitimately return with success on zero length regardless of other parameter values. Alternately, the file descriptor might be checked and an error code returned. Thus, having a second parameter value of a NULL pointer might never generate a robustness failure caused by a pointer dereference unless the file descriptor parameter and length fields were tested with nonexceptional values. In other words, exceptional values that are correctly handled for one argument might mask nonrobust handling of exceptional values for some other argument. If, on the other hand, the test case `write(FD_OPEN_WRITE, NULL, 16)` were executed, it might lead to an Abort failure when the NULL pointer is dereferenced. Additionally, test cases that are exceptional for some calls may non-exceptional for others (e.g., using read permissions for testing `read()` vs. `write()`). Thus, by including nonexceptional test cases we force the module under test to attempt to handle each value that might be exceptional. While both uses of nonexceptional test values are important, they necessarily lead to test cases that are not, in fact, tests of exceptional conditions (e.g., reading from a read-only file is not exceptional).

Multiversion software comparisons can prune nonexceptional test cases from the results data set. This is done by assuming that any test case in which all operating systems return with no indication of error are in fact nonexceptional tests (or, are exceptional tests which cannot be detected within reason on current computer systems). In all, 129,731 nonexceptional tests were removed across all 15 operating systems. Fig. 5 shows the adjusted abort and restart failure rates after removing nonexceptional tests. Manual verification of 100 randomly selected test cases thus removed indicated that all of them were indeed nonexceptional, but

it was impractical to examine a larger sample using this very labor-intensive process. While it is possible that some test cases were incorrectly removed, based on this sample and intuition gained during the sampling process, it seems unlikely that the number of false removals involved would materially affect the results.

### 5.2 An Estimation of Silent Failure Rates

One of the potential problems with leaving out Silent failures in reporting results is that an OS might conceivably be designed to avoid generating Abort failures at any cost. For example, AIX intentionally permits reads (but not writes) to the memory page mapped to address zero to support legacy code, meaning that dereferences of a NULL pointer would not generate Abort failures. And, in fact, AIX does have a moderately high Silent failure rate because of this implementation decision.

Once the nonexceptional tests were removed, a multiversion software comparison technique was again used to detect Silent Failures. The heuristic used was that if at least one OS returns an error code, then all other operating systems should either return an error code or suffer some form of robustness failure (typically an Abort failure). As an example, when attempting to compute the logarithm of zero, AIX, HPUX-10, and both versions of QNX completed the requested operation without an error code, whereas other OS implementations did return an error code. This indicated that AIX, HPUX-10, and QNX had suffered Silent robustness failures for that test case.

Of course, the heuristic of detection based on a single OS reporting an error code is not perfect. Manual verification of 100 randomly sampled test cases, with each test case compared across all the OS implementations, indicates that approximately 80 percent of cases predicted to be Silent failures by this technique were actually Silent failures. Of the approximately 20 percent of test cases that were misclassified:

- 28 percent were due to POSIX permitting discretion in how to handle an exceptional situation. For example, `mprotect()` is permitted, but not required, to return an error if the address of memory space does not fall on a page boundary.
- 21 percent were due to bugs in C library floating point routines returning false error codes. For example, Irix 5.3 returns an error for `tan(-1.0)` instead of the correct result of -1.557408. Two instances were found that are likely due to overflow of intermediate results—HPUX 9 returns an error code for `fmod(DBL_MAX,PI);` and QNX 4.24 returns an error code for `ldexp(e,33).`
- 9 percent were due to lack of support for required POSIX features in QNX 4.22, which incorrectly returned errors for filenames having embedded spaces.
- The remaining 42 percent were instances in which it was not obvious whether an error code could reasonably be required. This was mainly a concern when passing a pointer to a structure containing meaningless data, where some operating systems (such as SunOS 4.13, which returned an error code for each test case it did not abort on) apparently checked the data for validity and returned an error code.

Examining potential Silent failures manually also revealed some software defects ("bugs") generated by unusual, but specified, situations. For instance, POSIX requires `int fdatasynch(int filedes)` to return the EBADF error if `filedes` is not valid, and the file open is for write [16]. Yet when tested, only one operating system, IRIX 6.2, implemented the specified behavior, with the other OS implementations failing to indicate that an error occurred. The POSIX standard also specifically permits writes to files past EOF, requiring the file length be updated to allow the write [16]; however only FreeBSD, Linux, and SunOS 4.13 returned successfully after an attempt to write data to a file past its EOF, while every other implementation returned EBADF. It is estimated that the failure rates for these problems is quite low (perhaps 1 percent to 3 percent overall depending on the OS), but is definitely present, and is apparently not caught by the process of validating POSIX compliance.

A second approach was attempted for detecting Silent failures based on comparing test cases having no error indication against instances of the same test case suffering an Abort failures on some other OS. With some surprise, this turned out to be as good at revealing software defects as it was at identifying Silent failures. A relatively small number (37,434 total) of test cases generated an Abort failure for some operating systems, but completed with no error indication at all for other operating systems. But, manual verification 100 randomly sampled test cases indicated that this detection mechanism had approximately a 50 percent false alarm rate.

Part of the high false alarm rate for this second approach was due to differing orders for checking arguments among the various operating systems (related to the discussion of fault masking earlier). For example, reading bytes from an empty file to a NULL pointer memory location might abort if end-of-file is checked after attempting to move a byte, or return successfully with zero bytes having been read if end-of-file is checked before moving a byte. The other part of the false alarm rate was apparently due to limitations within floating point libraries. For instance, FreeBSD suffered an Abort failure on both `fabs(DBL_MAX)` and `fabs(-DBL_MAX)`, whereas it should have returned without an error.

Based on these estimated accuracy rates, results reported in Fig. 5 reflect only 80 percent of the silent errors measured and 50 percent of the silent aborts measured, thus compensating for the estimated false alarm rates. With all of the manual examination techniques it was impractical to gather a much larger sample, so these percentages should be considered gross approximations, but are believed to be reasonably accurate based on intuition gained during the sampling process.

### 5.3 Frequent Sources of Robustness Failure

Given that robustness failures are prevalent, what are the common sources? Source code to most of the operating systems tested was not available, and manual examination

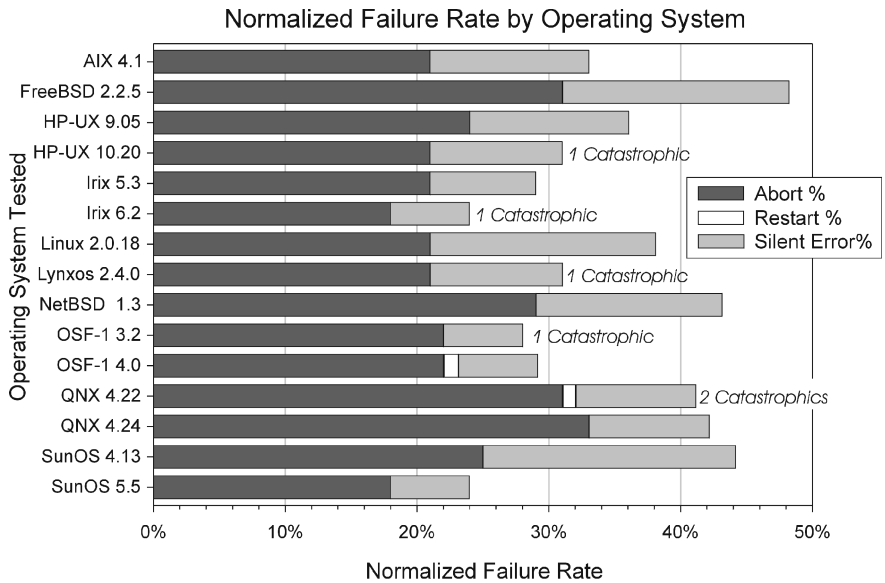



Fig. 5. The adjusted, normalized robustness failure rates after using multiversion software techniques. Results are APPROXIMATE due to the use of heuristics.

of available source code to search for root causes of robustness failures was impractical with such a large set of experimental data. Therefore, the best data available is based on a correlation of input values to robustness failures rather than analysis of causality. The test values most frequently associated with robustness failures are:

- 94.0 percent of invalid file pointers (excluding NULL) were associated with a robustness failure.
- 82.5 percent of NULL file pointers were associated with a robustness failure.
- 49.8 percent of invalid buffer pointers (excluding NULL) were associated with a robustness failure.
- 46.0 percent of NULL buffer pointers were associated with a robustness failure.
- 44.3 percent of MININT integer values were associated with a robustness failure.
- 36.3 percent of MAXINT integer values were associated with a robustness failure.

Perhaps surprisingly, system state changes induced by any particular test did not prove to be a source of robustness failures for other tests. Apparently, the use of a separate process per test case provided sufficient intertest isolation to contain the effects of damage to system state for all tests except Catastrophic failures. This was verified by vendors reproducing tests in isolation with single-test programs, and by verifying that test results remained the same even if tests were run in a different order within the test harness. This is not to say that such problems don't exist, but rather that they are rather more difficult to elicit on these operating systems than one might think.

## 6 Issues in Attaining Improved Robustness

When preliminary testing results were shown to OS vendors, it became very apparent that some developers took a dim view of a SIGSEGV or SIGFPE signal being considered a robustness failure. In fact, in some cases the developers stated that they specifically and purposefully generated signals as an error reporting mechanism, in order to make it more difficult for developers to miss bugs. On the other hand, other developers provide extensive support for a wide variety of error return codes and make attempts to minimize abnormal task terminations from within system calls and library functions. The importance of such comprehensive exception handling was underscored by many conversations with application developers who develop critical systems. There are two parts to this story: the relative strengths and weaknesses of each philosophy, and whether either goal (robust return codes or signaling for all exceptions) was attained in practice.

### 6.1 Signals vs. Error Codes

While discussions with OS developers have proven that exception handling robustness is a controversial, even "religious" subject, the fact remains that there are significant applications in several industries in which developers have stated very clearly that fine-grain error reporting is extremely desirable, and that signals accompanied by task restarts are unacceptable. These applications include telecommunication switches, railroad train controllers, real-time simulations, uninterruptible power supplies, factory automation control, ultra-high availability mainframe computers, and submarine navigation, to name a few real examples encountered during the course of this project. While these may not be the intended application areas for most OS authors, the fact is that COTS OS implementations will be pressed into service for such critical systems to meet cost and time-to-market constraints. Thus, evaluating the robustness of an OS is useful, even though robustness is not required by the POSIX standard.

That having been said, the results reported here suggest that there are issues at hand that go beyond a preference for signals vs. error return codes. One issue is simply that divergence in implementations hinders writing portable, robust applications. A second issue is that no operating

systems examined actually succeeded in attaining a high degree of robustness, even if signals were considered to be a desirable exception reporting mechanism.

### 6.2 Building More Robust Systems

Traditionally, software robustness has been achieved through a variety of techniques such as checking error codes, performing range checks on values, and using testing to flush out problems. However, Ballista robustness testing results have eliminated any slender hopes that these approaches were entirely sufficient for critical systems. Checking error codes might work on one OS, but might not work when porting to another OS (or even to a minor version change of the same OS) which generates a SIGSEGV instead of an error code, or which generates no error code at all in response to an exceptional situation. Similarly, it is clear that POSIX functions often do not perform even a cursory check for NULL pointer values, which could be accomplished with minimal speed impact. Finally, vendor testing of OS implementations has been demonstrated to miss some very simple ways to cause system crashes in both major and minor version changes.

Thus, a useful additional step in building more robust systems is to use API-level fault injection such as that performed by the Ballista testing system. This will, at a minimum, identify certain classes of Catastrophic failures so that manual intervention can be performed via software "wrappers" to screen out exceptional parameters for specific system calls, or to permit application developers to otherwise pay specific attention to eliminating the possibility of such situations.

For C library functions, it may be possible to use alternate libraries that are specifically designed for increased robustness. One example is the Safe Fast I/O library (SFIO) [21] that can replace portions of the C library. For system calls, one can select an existing OS that tends to have low failure rates as shown in Fig. 4, if Abort failures are a primary concern. Or, one might even find it necessary to add extra parameter-checking wrappers around system calls to reduce Silent failure rates.

For any application it is important to realize that abnormal task terminations are to be expected as a matter of course, and provide for automatic recovery from such events. In some applications this is sufficient to attain a reasonable level of robustness. For other applications, this is merely a way to reduce the damage caused by a system failure, but is not a viable substitute for more robust error identification and recovery.

Finally, a potential long-term approach to increasing the robustness of OS implementations is to modify the POSIX standard to include a requirement for comprehensive exception handling, with no exception left undefined. While this might impose a modest performance penalty, it might well be viable as an optional (but well-specified) extended feature set. Further research should be performed to quantify and reduce any associated performance penalties associated with increased exception handling abilities.

## 7 Conclusions

The Ballista testing approach provides repeatable, scalable measurements for robustness with respect to exceptional parameter values. Over one million total tests were automatically generated for up to 233 POSIX function and system calls spanning fifteen operating systems. The most significant result was that no operating system displayed a high level of robustness. The normalized rate for robust handling of exceptional inputs ranged from a low of 52 percent for FreeBSD version 2.2.5 to a high of 76 percent for SunOS version 5.5 and Irix version 6.2. The majority of robustness failures were Abort failures (ranging from 18 percent to 33 percent), in which a signal was sent from the system call or library function itself, causing an abnormal task termination. The next most prevalent failures were Silent failures (ranging from 6 percent to 19 percent), in which exceptional inputs to a Module under Test resulted in erroneous indication of successful completion. Additionally, five operating systems each had at least one situation that caused a system crash in response to executing a single system call. The largest vulnerabilities to robustness failures occurred when processing illegal memory pointer values, illegal file pointer values, and extremely large integer and floating point numbers. In retrospect, it is really no surprise that NULL pointers cause problems when passed to system calls. Regardless, the single most effective way to improve robustness for the operating systems examined would be to add tests for NULL pointer values for relevant calls.

Application of the Ballista testing approach to measuring OS robustness has made several contributions. It provides repeatable, quantified measurements of the effectiveness of off-the-shelf operating systems at handling exceptional input parameter values. Such a metric is important to those application developers designing the growing number of critical systems which require high levels of robustness. Beyond this, it documents a divergence of exception handling strategies between using error return codes and throwing signals in current operating systems, which may make it difficult to write portable robust applications. Finally, all operating systems examined had a large number of instances in which exceptional input parameter values resulted in an erroneous indication of success from a system call or library function, which would seem to further complicate creating robust applications.

There are many factors which should properly be taken into account when evaluating an operating system. Exception handling is only one factor, and may range from extremely important to irrelevant depending on the application. However, for those applications where it matters, there is now a way to quantify and compare OS exception handling effectiveness.

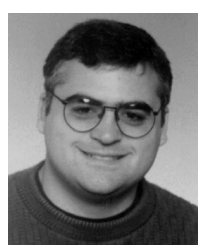
## ACKNOWLEDGMENTS

The contributions of the Ballista team have been crucial in performing this research, especially data collection work by Nathan Kropp and Jiantao Pan. We also appreciate the conversations with various OS developers that helped us better understand the issues. This research was sponsored by DARPA contract DABT63-96-C-0064. Additionally, equipment donations from Intel and significant equipment subsidies from Digital Equipment Corporation (now Compaq) were instrumental in carrying out this research.

## REFERENCES

[1] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1,491-1,501, 1985.
[2] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 575-82, 1990.
[3] B. Beizer, *Black Box Testing.* New York: Wiley 1995.
[4] G. Carrette, "CRASHME: Random Input Testing," http://people.delphi.com/gjc/crashme.html, accessed 6 July 1998.
[5] F. Cristian, "Exception Handling and Tolerance of Software Faults," *Software Fault Tolerance,* Michael R. Lyu, ed., chapter 4, pp. 81-107, Chichester: Wiley, 1995.
[6] E. Czeck, F. Feather, A. Grizzaffi, G. Finelli, Z. Segall, and D. Siewiorek, "Fault-Free Performance Validation of Avionic Multiprocessors," *Proc. IEEE/AIAA Seventh Digital Avionics Systems Conf.,* pp. 670-677, Oct. 1986.
[7] J. DeVale, P. Koopman, D. Guttendorf, "The Ballista Software Robustness Testing Service," *Proc. 16th Int'l Conf. Testing Computer Software,* pp. 33–42, 1999.
[8] C. Dingman, "Measuring Robustness of a Fault Tolerant Aerospace System," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* pp. 522-527, June 1995.
[9] C. Dingman, "Portable Robustness Benchmarks," Ph.D. thesis, Dept. of Electrical and Computer Eng., Carnegie Mellon Univ., Pittsburgh, Penn., May 1997.
[10] U.S. Department of Defense, "High Level Architecture Run Time Infrastructure Programmer's Guide, RTI 1.3 Version 5," Dec. 16, 1998.
[11] K. Fernsler and P. Koopman, "Robustness Testing of a Distributed Simulation Backplane," *Proc.10th Int'l Symp. Software Reliability Eng.,* Nov. 1-4, 1999.
[12] N. Gehani, "Exceptional C or C with Exceptions," *Software–Practice and Experience,* vol. 22, no. 10, pp. 827-48, 1992.
[13] J. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Comm. ACM,* vol. 18, no. 12, pp. 683–696, Dec. 1975.
[14] I. Hill, "Faults in Functions, in ALGOL and FORTRAN," *The Computer J.,* vol. 14, no. 3, pp. 315–316, Aug. 1971.
[15] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12–1990, IEEE CS, Dec. 10, 1990.
[16] *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment 1: Realtime Extension [C Language].* IEEE Std 1003.1b–1993, IEEE CS, 1994.
[17] E. Jones, ed., *The Apollo Lunar Surface J., Apollo 11 Lunar Landing,* Entries 102:38:30, 102:42:22, and 102:42:41, National Aeronautics and Space Administration, Washington, DC, 1996.
[18] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 1992 IEEE Workshop Fault-Tolerant Parallel and Distributed Systems,* pp. 336–344, July 1992.
[19] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing Operating Systems Using Robustness Benchmarks," *Proc. Symp. Reliable and Distributed Systems,* pp. 72–79, 1997.
[20] P. Koopman, J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proc. 28th Fault-Tolerant Computing Symp.,* pp. 30-37, 1999.
[21] D. Korn and K.-P. Vo, "SFIO: Safe/Fast String/File IO," *Proc. Summer 1991 USENIX Conf.,* pp. 235-256, 1991.
[22] N. Kropp, P. Koopman, D. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components," *Proc. 28th Fault-Tolerant Computing Symp.,* pp. 230–239, 1998.
[23] J.L. Lions, chairman,, "Ariane 5 Flight 501 Failure: Report by the Inquiry Board," European Space Agency, July 19, 1996.
[24] B. Marick, *The Craft of Software Testing.* Prentice Hall, 1995.
[25] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of Operating System Utilities," *Comm. ACM,* vol. 33, pp. 32–44, Dec. 1990.
[26] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services" Computer Science Technical Report 1268, Univ. of Wisconsin-Madison, May 1998.
[27] J. Musa, G. Fuoco, N. Irving, and D. Kropfl, B. Juhlin, "The Operational Profile," *Handbook of Software Reliability Eng.* Los Alamitos, Calif.: IEEE CS Press, pp. 167–216, 1996.
[28] Object Management Group, "The Common Object Request Broker: Architecture and Specification, Revision 2.0," July 1995.
[29] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM,* vol. 31, no. 6, pp. 676-686, 1987.
[30] M. Schuette, J. Shen, D. Siewiorek, and Y. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Digest of Papers; Proc. 16th Ann. Int'l Symp. Fault-Tolerant Computing Systems,* pp. 138-43, 1986.
[31] D. Siewiorek, J. Hudak, B. Suh, and Z. Segal, "Development of a Benchmark to Measure System Robustness," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing,* pp. 88-97, 1993.
[32] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," Research Report WRL–94/2, Digital Western Research Lab, Palo Alto, Calif., 1994.
[33] T. Tsai and R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," *Proc. Eighth Int'l Conf. Modeling Techniques and Tools for Computer Performance Evaluation,* pp. 26-40, 1995.
[34] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer, "Instruction Fetching: Coping with Code Bloat," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 345–356, 1995.
[35] K-P. Vo, Y.-M. Wang, P. Chung, and Y. Huang, "Xept: A Software Instrumentation Method for Exception Handling," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 60–69, 1997.

**Philip J. Koopman** received the BS and MEng degrees in computer engineering from Rensselaer Polytechnic Institute in 1982. After serving as a US Navy submarine officer, he returned to graduate school and received the PhD degree from Carnegie Mellon University in 1989. During several years in the industry, he was a CPU designer for Harris Semiconductor and an embedded systems researcher at United Technologies Research Center. He has been at Carnegie Mellon University since 1996, where he is currently an assistant professor. His research interests include distributed embedded systems, dependability, and system architecture. He is an associate editor of *Design Automation for Embedded Systems: An International Journal*. He is also a member of the ACM, and a senior member of the IEEE.



**John P. DeVale** served six years in the US Navy prior to obtaining the BS degree in computer engineering from Old Dominion University in 1997. He subsequently received the MS degree in computer engineering from Carnegie Mellon University, where he is currently enrolled as a PhD student. His research interests include computer architecture and dependability. He is a student member of the IEEE.