Measuring Operating System Robustness

John DeVale
Department of Electrical and Computer Engineering &
Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, PA

Abstract

Robustness is becoming more important as critical software increasingly affects our daily lives. Success in building robust software requires understanding and improving the robustness of the operating system API, but to date there has been no accurate, reproducible way to measure robustness. This paper presents the first full-scale, quantitative measurements of operating system robustness. Each of 15 different operating system's robustness is measured by automatically testing up to 233 POSIX functions and system calls with exceptional parameter values. The work identifies repeatable ways to crash operating systems with a single call, ways to cause task hangs within OS code, ways to cause task core dumps within OS code, failures to implement defined POSIX functionality for unusual conditions, and false indications of successful completion in response to exceptional input parameter values.

Overall, only 55% to 76% of tests performed were handled robustly, depending on the operating system being tested. Approximately 6% to 19% of tests failed to generate any indication of error in the presence of exceptional inputs. Approximately 1% to 3% of calls tested failed to implement defined POSIX functionality for unusual, but specified, conditions. Between 18% and 33% of calls tested dumped core from within a POSIX function or system call, and five operating systems were completely crashed by individual user mode system calls with exceptional parameter values. The most prevalent sources of robustness failures were illegal pointer values, numeric overflows, and end-of-file overruns. The results indicate that there is significant opportunity for increasing robustness within current operating systems. 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.

Acknowledgments: This research was sponsored by DARPA contract DABT63-96-C-0064.

1. Introduction

Computers are becoming essential to everyday life in modern society, but are not necessarily as dependable as one would like. Expensive, specialized computers such as those used to control jet aircraft engines are quite dependable. However, most other computers used in routine business operations, transportation, consumer electronics, and other widespread applications are too cost-sensitive to employ traditional fault tolerance techniques, and crash on a regular basis. While losing a few minutes work on a spreadsheet due to a computer crash may be annoying, the disruption caused by a communications blackout or the failure of a server in a business can be substantial.

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 majority of software failures in practice seem to be due to problems with robustness [Cristian95]. Thirty years ago, the Apollo 11 mission experienced three 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 [Jones96]. 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 [Lions96]. Now that our society relies upon computer systems for everyday tasks, exceptional conditions routinely cause system failures in telecommunication systems, desktop computers, and elsewhere. Since even expensive systems designed with robustness specifically in mind suffer robustness failures, it seems likely that systems designed without robustness as an explicit goal would suffer from a higher incidence of robustness failures.

The robustness of a computer system can depend in large part on the robustness of its operating system (OS). After all, it is the OS that is entrusted with preventing ill-behaved tasks from crashing the rest of the system, and providing each task with appropriate access to system resources. Furthermore, even within a single task it is important that the OS provide robust system calls and functions. As difficult as it is to produce a robust software application, the task becomes especially difficult if the underlying operating system upon which it is built is non-robust, or conforms to an application programming interface (API) which provides insufficient support for robustness.

Even as the importance of using a robust OS increases, cost pressures are forcing the use of Commercial Off-The-Shelf (COTS) software, including operating systems. The rationale for using a COTS OS is that it will cost less because development and support costs are spread over a larger installed base, and that it will likely be of higher

quality because of more extensive field experience. Although COTS components may benefit from a more extensive testing budget than an in-house system, the issue of robustness is not often completely addressed. The ability to deterministically assure sufficient product quality with respect to robustness becomes a critical issue when depending on an external supplier for an operating system. But, until now there has been no way to directly measure OS robustness with a high degree of precision.

Ideally there should be a direct, repeatable, quantitative way to evaluate the latest competing OS releases for robustness (in other words, a way based on measuring properties of the current software that can be verified by prospective customers, rather than an examination of historical data for previous releases, or value judgements about software development methodologies used by the vendors). This would support an educated "make/buy" decision as to whether a COTS OS might in fact be more robust than an existing proprietary OS, and also would enable system designers to make informed comparison shopping decisions when selecting an OS. Equally important, such an evaluation technique would give the developers feedback about a new OS release before it ships. If an OS vendor should desire to improve robustness, it is difficult to measure success at doing so if robustness itself can't be measured with certainty.

This paper is the first to present detailed quantitative results from a full-scale, repeatable, portable comparison of OS robustness measurements. It describes the work of the Ballista project in measuring and comparing the robustness of different OS implementations with respect to responses when encountering exceptional parameter values. Automated testing was performed on fifteen POSIX [IEEE93] 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 resulted in robustness failures, ranging in severity from complete system crashes to false indication of successful operation in unspecified circumstances (the issue of what robustness means in unspecified situations is discussed later).

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 the module detected and notified the calling program of an error, and whether the task or even system suffered a crash or hang.

Beyond the robustness failure rates measured, analysis of testing 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 way to provide portable support for robustness at run time. Alternately, some operating systems emphasize the generation of a signal (typically resulting in a "core dump") when exceptional parameter values are encountered in order to facilitate debugging. However, no OS comes close to implementing complete detection and reporting of exceptional situations by either strategy. While it is recognized that the POSIX standard does not *require* robustness, it seems likely that a growing number of applications will require a robust OS. Evaluating current operating systems with respect to robustness is an important first step in understanding whether change is needed, and what directions it might take.

The balance of the paper describes 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 in this paper 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. 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 fairly traditional "black box," or functional testing techniques [Bezier95] in which only functionality is of concern, not the actual structure of the source code. However, Ballista is not concerned with validating functionality for ordinary operating conditions, but rather determining whether or not a software module is robust.

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* that will be used in the current discussion is "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [IEEE90]". This expands the notion of robustness to be more than catastrophic system crashes, and encompasses situations in which small, recoverable failures might occur. The work presented in this paper concentrates on the portion of robustness dealing with invalid inputs. 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 a single invocation of a software module from a single execution thread.

An early and well known method for automatically testing operating systems for robustness was the development

of the Crashme program[Carrette96]. Crashme operates by writing randomized data values to memory, then spawning large numbers of tasks that attempted to execute those random bytes as 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 it may eventually find some way to crash the system).

More recently, the Fuzz project at the University of Wisconsin used random noise (or "fuzz") injection to discover robustness problems in operating systems. That work documented the source of several problems, and then discovered that the problems were still present in operating systems several years later [Miller98][Miller89]. The Fuzz approach tested specific OS elements and interfaces (compared 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 [Segall90] uses probes placed by the programmer to alter the binary process image in memory during execution. The FERRARI system [Kanawati92] 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 [Tsai95] 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.

There are several commercial products that help in developing robust code, such as Purify and Boundschecker, which instrument software to detect exceptional situations. They work by detecting exceptions that arise during development testing or usage of the software. However, they are not able to find robustness failures that might occur in situations which are not tested (and, even with what would normally be called 100% test coverage, it is unlikely in practice that every exceptional condition which will be encountered in the field is included in the software test suite). The Ballista approach differs from, and complements, these approaches by actively seeking out robustness failures; rather than being an instrumentation tool it actually generates tests for exception handling ability and feeds them directly into software modules. Thus, Ballista is likely to find robustness failures that would otherwise be missed during normal software testing even with available instrumentation tools.

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[Hill71][Goodenough75]. 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 with the POSIX API [IEEE93].

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[Gehani92][Cristian95]. The results presented below consider signal-based termination behavior to be non-robust unless the signals thrown have detailed information about the event causing the problem (so, generating a SIGSEGV signal in POSIX is in general a non-robust response to exceptional inputs, but a signal associated with a data structure having information comparable to POSIX `errno` values might be considered robust)[IEEE93]. It is important to note that saying a particular function (or operating system) is non-robust is not a statement that it is in any way defective (since, in most cases, the POSIX standard does not require robustness). Further, it is recognized that the issue of what exactly should be considered robust behavior is controversial, and the matter is discussed further after experimental data has been presented.

The Xept approach [Vo97] provides an alternate view on the issue error return *vs.* signal-based models. Xept uses software “wrappers” around procedure calls as a way to encapsulate error checking and error handling within the context of a readable program. Although this approach has not at this point been widely adopted, it seems to be a way to overcome objections with respect to using an error return code model in programming. Given that the fault tolerance community has found that transient system failure rates far outnumber manifestations of 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. Given that the current POSIX standard is written such

that error return codes are the only way to write portable robust software [IEEE93 2368-2377], one could envision using the results of Ballista testing to provide information for error checking within an Xept software wrapper.

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 a single time to determine whether it is robust 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 could be described as a tuple: *[MuT_name, test_value1, test_value2, ...]* corresponding to a procedure call of the form: *MuT_name(test_value1, test_value2, ...)*) . Thus, the general approach to Ballista testing is to test the robustness of a single call to a MuT for a single tuple of test values, and then repeat this process for multiple test cases that each have different combinations of valid and invalid test values. A detailed discussion follows.

3.1. Test cases based on data types

The Ballista approach to robustness testing has been implemented for a set of 233 POSIX calls, including real-time extensions for C. Essentially all system calls and functions defined in the IEEE 1003.1b standard [IEEE93] ("POSIX.1b" or "POSIX with real-time extensions" with C language binding) 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`. Although some functions are implemented as macros, they were still tested accurately.

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 how the argument is used was exploited to result in better testing (for example, a file descriptor might be of type `int`, but was described to Ballista as a specific file descriptor data type).

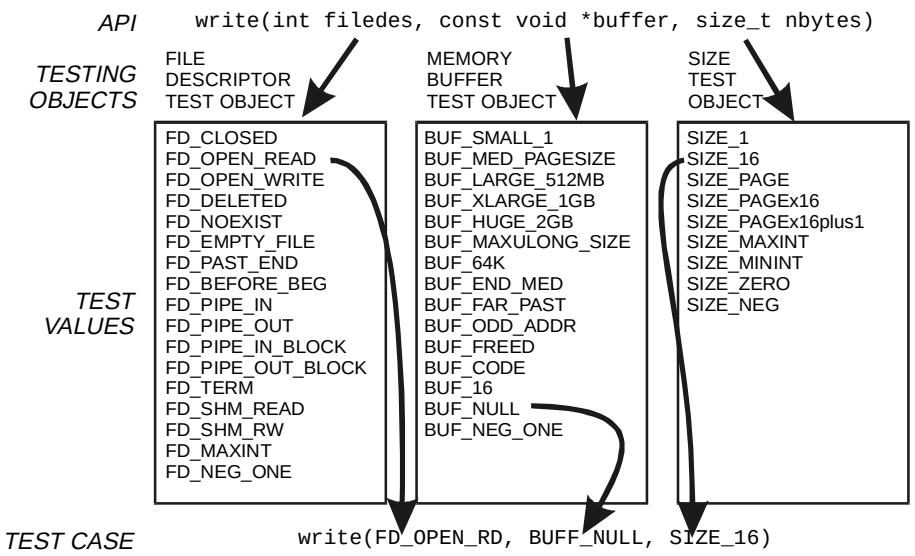

Figure 1. Ballista test case generation for the `write()` function. The arrows show a single test case being generated from three particular test values; in general all combinations of test values are tried in the course of testing.

As an example, Figure 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. The fact that `write()` takes three parameters of three different data types leads Ballista to draw test values from separate test objects established for each of the three data types. In Figure 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 general, all combinations are tested, yielding 17 file descriptor * 15 buffer * 9 size test cases = 2295 tests for `write()`.

Each test value (such as `FD_OPEN_READ` in Figure 1) refers to a pair of code fragments that are kept in a simple database. The first fragment for each test value is a constructor that is called before the test case is executed. 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 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 first calling constructors for instances of all parameters in a selected test case, executing a call to the MuT with those parameter instances, then calling destructors for all the instances. 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.

As a matter of efficiency, a single program for each MuT is compiled which is capable of generating all required test cases. A main program spawns a task which executes a single test case, with the main program using a watchdog timer to detect hangs and monitoring the task status to detect abnormal task terminations. The main program iterates through combinations of test values to execute all desired test cases. While one might think that executing tests in this manner would lead to failures caused by interactions among sequential tests, in practice essentially all failures identified by this test harness have been found to be reproducible in isolation. This independence of tests is attained because the constructor/destructor approach seems to be successful at setting and restoring a fresh copy of relevant system state for each test.

While automatic generation of large variations of test values is a subject of current work, the particular values used at the present time are hand-picked. An average of ten test values were selected by hand for each test object based on experience gained both in general programming and from previous generations of fault injection experiments. For most MuTs, a testing approach is used in which all combinations of test values are used to create test cases. For a half-dozen POSIX calls, the number of parameters is large enough to yield too many test cases for reasonable exhaustive coverage; in these cases a pseudo-random sampling of 5000 test cases is used (based on a comparison to a run with exhaustive searching on one OS, the sampling gives results accurate to within 1 percentage point fore each function at a small fraction of the execution time).

It is important to note that this testing methodology does not generate test cases based on a description of MuT functionality, but rather on the data types of the MuT's arguments. This means that, for example, the set of test cases

Data Type	Test Values
AIO control Block	20
Bit Mask	4
Buffer	15
Directory Pointer	7
File Descriptor	13
File Name	8
File Mode	7
File Open Flags	9
File Pointer	11
Float	9
Integer	16
Message Queue	6
Pointer to INT	11
Process ID	9
Semaphore	8
Signal Set	7
Simple INT	11
Size	9
String	9
Timeout	9

TABLE 1. Only twenty data types are necessary for testing 233 POSIX functions and system calls.

used to test `write()` would be identical to the test cases used to test `read()` because they take identical data types. This testing approach means that customized test scaffolding code does not need to be written for each MuT -- instead the amount of testing software written is proportional to the number of data types. As a result, the Ballista testing method was found to be highly scalable with respect to the amount of effort required per function, needing only 20 data types (Table 1) to test 233 POSIX function calls. An average data type has 10 test cases, each of 10 lines of C code, meaning that the entire test suite required only 2000 lines of C code for test cases (in addition, of course, to the general testing harness code used for all test cases).

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 cases can be used to reproduce robustness failures in isolation for either use by developers or to verify test results.

3.2. Categorizing test results

After each test case is executed, the Ballista test harness categorizes the test results according to the CRASH severity scale: [Kropp98]

- **Catastrophic** failures occur when the OS itself becomes corrupted or the machine crashes or reboots. In other words, this is a complete system crash. These failures are identified manually because of difficulties encountered with loss of newly written, committed, file data across system crashes on several operating systems.
- **Restart** failures occur when a function call to an OS function never returns, resulting in a task that has "hung" and must be terminated using a command such as "kill -9". These failures are identified by a watchdog timer which times out after several seconds of waiting for a test case to complete.
- **Abort** failures tend to be the most prevalent, and result in abnormal termination (a "core dump") of a task caused by a signal generated within an OS function. Abort failures are identified by monitoring the status of the child process executing the test case.
- **Silent** failures occur when an OS returns no indication of error on an exceptional operation 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 [Koopman97], but are not further discussed in this paper due to lack of a way to perform automated identification of these robustness failures.

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 trigger a page fault).

One of the trade-offs made to attain scalability in Ballista testing is that the test harness has no way to tell which test cases are valid or invalid for any particular MuT. Thus, some tests returning no error code are Silent failures, while others are actually a set of valid parameter values which should legitimately return with no error indication. This has the effect of "watering down" the test cases with non-exceptional tests, making the raw failure rates underestimates. Section 5.1 gives an estimated compensation for this effect, but even the raw failure rates are significant enough to make the point that Ballista testing is effective in finding robustness failures.

4. Results

A total of 1,082,541 data points were collected. Operating systems which supported all of the 233 selected POSIX functions and system calls had 92,658 total test cases, but those supporting a subset of the functionality tested had fewer test cases. The final number is dependent on which functions were supported and the number of combinations of tests for those functions which were supported.

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, GNU C version 2.7.2.3 was used to build the test suite.

Table 2 reports the directly measured robustness failure rates. Silent error rates are not directly measured because there is no specification available to the testing program as to when error return codes should or should not be

generated (but, Silent errors are measured indirectly as discussed in Section 5.2) There were five functions that resulted in entire operating system crashes (either automatic reboots or system hangs). 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 a core dump 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). A check was made to ensure that Abort failures were not due to corruption of stack values and subsequent corruption/misdirection of calling program operation.

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 4.0	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

TABLE 2. Directly measured robustness failures for fifteen POSIX operating systems.

A MuT that underwent a catastrophic failure could not be completely tested, and resulted in no data on that MuT other than the presence of a catastrophic failure. Since the testing suite is automated, a system crash leaves it in an unrecoverable state with respect to the function in question. Further testing a function which suffered a catastrophic

test would require either hand execution of individual cases, or adding large amounts of waiting and synchronization code into the test benchmark. Hand execution was performed for the Irix 6.2 catastrophic failure in `munmap`, and allowed the identification of this single line of user code which can crash the entire OS, requiring a manual reboot:

```
munmap(malloc((1<<30+1)),MAXINT);
```

While it is tempting to simply use the raw number of tests that fail as a comparative metric, this approach is problematic. Most OS implementations do not support the full set of POSIX real-time extensions, so the raw number of failures cannot be used for comparisons. In addition, the number of tests executed per MuT is determined by the number and types of the arguments. So, a single MuT with a large number of test cases could significantly affect both the number of failures and the ratio of failures to tests executed. Similarly, an OS function with few test cases would have minimal effect on raw failure rates even if it demonstrated a large percentage of failures. Thus, some sort of normalized failure rate metric is called for, and is reported in the last column of Table 2.

4.2. Normalized Failure Rates

We define the normalized failure rate for a particular operating system to be:

$$F = \sum_{i=1}^{N} w_i \frac{f_i}{t_i}$$ with a range of values from 0 to 1 inclusive, where:

N = number of functions tested - Equal to the number of calls a particular OS supports of the 233 in our suite
w_i is a weighting of importance or relative execution frequency of that function where $\sum w_i = 1$
f_i is the number of tests which produced robustness failure for function i
t_i is the number of tests executed for function i

This definition produces a sort of exposure metric, in which the failure rate within each function is weighted and averaged across all functions tested 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. For the results given in Table 2 and the remainder of this paper, an equal weighting is used (*i.e.*, $w_i = 1/N$) to produce a generically applicable result. However, if an OS were being tested with regard to a specific application, weightings should be developed to reflect the dynamic frequency of calling each function to give a more accurate exposure metric.

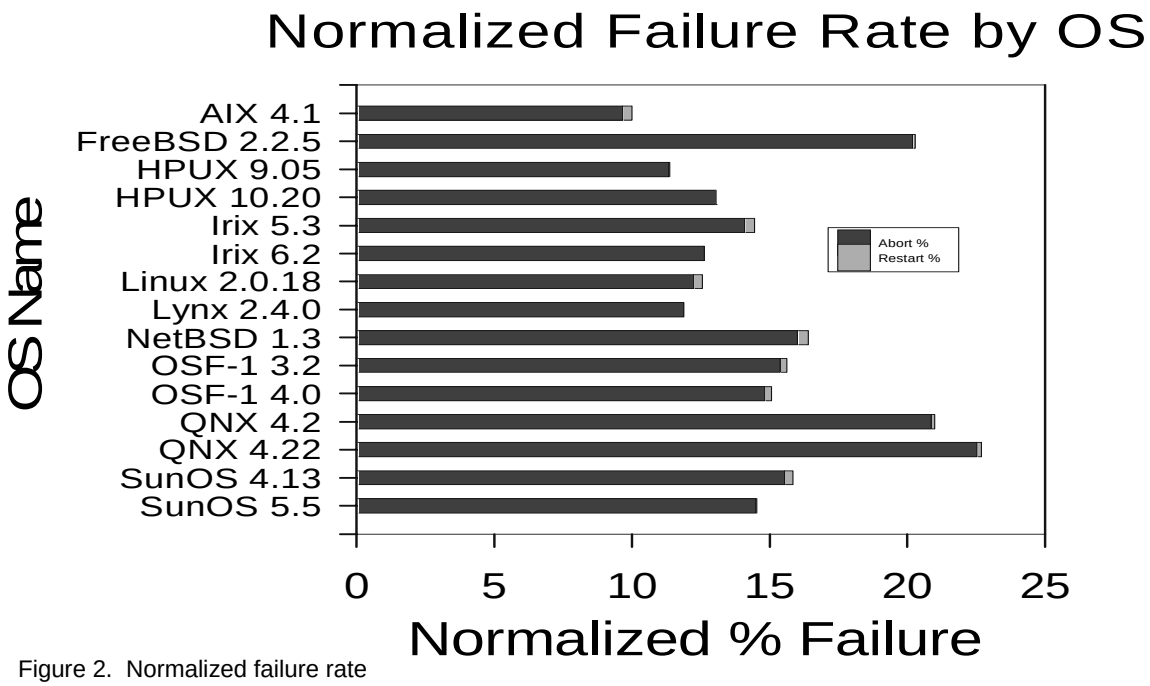

Figure 2. Normalized failure rate

Figure 2 shows the normalized failures rates as calculated for each operating system (this is the same data as the rightmost column in Table 2). Overall failure rates for both Abort and Restart failures range from the low of 9.99% (AIX) to a high of 22.69% (QNX 4.24). The mean failure rate is 15.158% with a standard deviation of 3.678 percentage points. As mentioned previously, these are raw failure rates that include non-exceptional test cases which dilute the failure rates, but do not include Silent failures.

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 pass with no indication of error -- they could either be non-exceptional test cases or Silent failures, depending on the actual functionality of the MuT. However, the availability of a number of operating systems with a standardized API permits estimating and refining failure rates using a variation of N-version software voting.

In N-version software voting, [Avizienis85] multiple independently developed versions of software are used to implement identical functionality. Generally these versions are used in a voting scheme to detect, and potentially

correct, software defects. Once given a set of inputs, if all versions of software agree, they are declared to all have computed a correct result. If they do not agree, then a software defect has been encountered and some corrective action must be taken.

In the case of Ballista testing results for operating systems, software voting can be used to classify test results that do not generate an indication of error (*i.e.*, successfully return to the calling program with no error code -- either an error code or an Abort would be indication of an error). Because all test cases are identical for each MuT regardless of the OS being tested, the set of results for all operating systems for each test case can be used as an N-version software voting set, potentially revealing Silent failures.

5.1. Elimination of non-exceptional tests.

The Ballista test cases carefully include some test values which are not exceptional in any way. This was intentionally done to prevent tests for exceptional inputs for one argument from masking out robustness failures having to do with some other argument. 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, the fact that the second parameter is a NULL pointer might never generate a robustness failures unless the file descriptor parameter and length fields were tested with non-exceptional values (in other words, exceptional values that are correctly handled for one argument might mask non-robust 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.

If the only test values in the database were exceptional, and a function correctly handled all of the exceptional conditions for at least one argument, exception handling for the other arguments would be untested. Additionally, what is exceptional for one MuT may be non-exceptional for another (such as a write-only file for `read()` vs. `write()` testing). However, including valid parameter values in testing leads to generating a significant number of non-exceptional test cases (such as `write(FD_OPEN_WRITE, BUF_MED_PAGESIZE, 16)`) While this increased the coverage of exceptional conditions, the combinatorial testing creates test cases with no exceptional inputs.

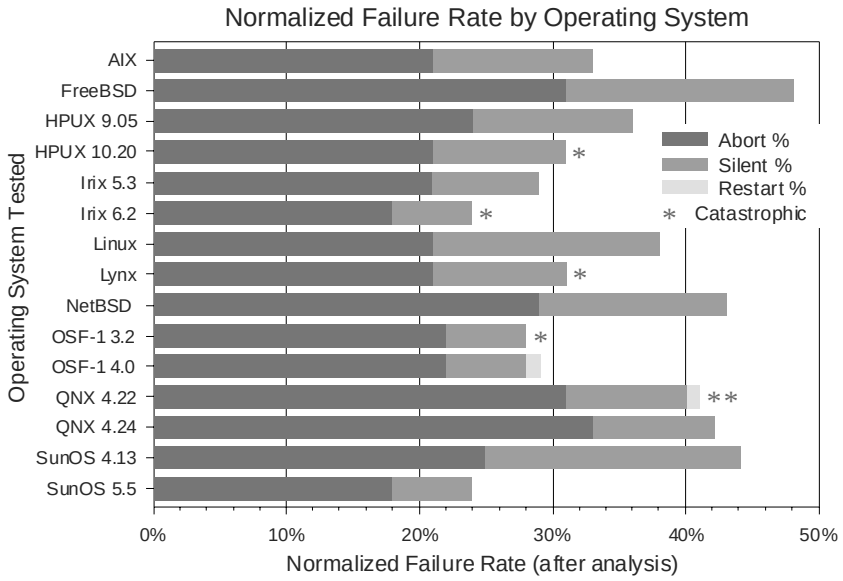

Figure 3. Adjusted, normalized robustness failure rates after software voting techniques. Note that rounding error results in several totals not equal to 100%.

Software voting was used to identify and prune non-exceptional test cases from the data set. The voting assumed that any test case in which all operating systems returned with no indication of error were in fact non-exceptional tests (or, were exceptional tests which could not reasonably be expected to be detected on current computer systems). In all, 129,731 non-exceptional tests were removed across all 15 operating systems. Figure 3 shows the adjusted failure rates after removing non-exceptional tests. Hand sampling of several dozen removed test cases indicated that all of them were indeed non-exceptional. While there is the possibility that some exceptional test cases slipped passed this screening, it seems unlikely that the number involved would materially affect of 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 be designed to avoid generating Abort failures at the expense of error detection. For example, AIX permits reads (but not writes) to the memory page mapped to address zero, meaning that reads from a NULL pointer would not generate Abort failures. (And, in fact, the data in Table 3 show that AIX does have a moderately high Silent failure rate, although by no means the highest rate.)

Once the non-exceptional tests were remove, a variation on software voting 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 failed to return an error code, whereas other operating systems tested did report an error code. This indicated that AIX, HPUX-10, and QNX had suffered Silent robustness failures.

Of course, the heuristic of detection based on a single OS reporting an error code is not a completely accurate mechanism. Manual random sampling of the results indicate that approximately 80% of detected test cases were actually Silent failures (the numbers in Table 3 reflect this, with only 80% of cases reported as Silent failures). Of the 20% of test cases that were misclassified:

- 28% 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% 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 for `ldexp(e,33)`
- 9% were due to a filesystem bug in QNX 4.22, which incorrectly returned errors for filenames having embedded spaces.
- The remaining 42% 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 garbage 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.

Classifying the Silent failures sampled 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 for write [IEEE93]. Yet when tested, only one operating system, IRIX 6.2 followed the specification correctly. All other operating systems which supported the `fdatasynch` call did not indicate that an error occurred. POSIX also specifically allows writes to files past EOF, requiring the file length to be updated to allow the write [IEEE93]; 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. Manual checking of random samples of operating system calls indicated the failure rates for these problems ranged from 1% to 3% overall.

A second approach was attempted for detecting Silent failures based on voting successful returns against Abort failures, but to our surprise turned out to be good at revealing software defects. A relatively small number (37434) of test cases generated an Abort failure for some operating systems, but successfully completed for other operating systems. At first it was thought that these might be treated as if the Abort failure were a substitute for an error code with N-version software voting. But, a randomly sampled hand analysis indicated that this detection mechanism was incorrect approximately 50% of the time.

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 programming errors in floating point libraries. For instance, FreeBSD suffered an Abort failure on both `fabs(DBL_MAX)` and `fabs(-DBL_MAX)`.

Note that results reported in figure 3 reflect only 80% of the silent errors measured and 50% of the silent aborts measured

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 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 currently available is based on a correlation of input values to robustness failures rather than analysis of causality. Data analysis was performed to determine which exceptional parameter values were most frequently associated with robustness failures, with the following being the leading causes. The numbers indicate the percent of all test cases in which a given test value appears that result in a robustness failure.

- 94.0% of invalid file pointers (excluding NULL) were associated with a robustness failure
- 82.5% of NULL file pointers were associated with a robustness failure

- 49.8% of invalid buffer pointers (excluding NULL) were associated with a robustness failure
- 46.0% of NULL buffer pointers were associated with a robustness failure
- 44.3% of MININT integer values were associated with a robustness failure
- 36.3% of MAXINT integer values were associated with a robustness failure

6. Issues in attaining improved robustness

When preliminary testing results were shown to OS vendors in early 1998, it became very apparent that some developers took a dim view of a core dump 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 core dumps from within system calls and library functions. There are two parts to this story: the relative strengths and weaknesses of each philosophy, and whether the goal was attained in practice.

6.1. Signals vs. error codes

Opinions are divided in the OS developer community as to whether exceptions should be handled with signals or error codes. POSIX standardizes a way to communicate exceptional conditions using error codes, and requires their use in certain situations. Thus, robust software is likely to check at least some error codes. The advantage to going further and indicating all exceptions with error codes is that it permits a uniform, portable treatment of exceptions according to the existing POSIX standard API. An additional advantage is that errors can be checked after operations as desired, supporting the strategies of retrying failed operations, executing lightweight recovery mechanisms, or performing alternate degraded mode actions specific to the error encountered. The disadvantage is that actually writing the potentially large amount of error handling software can be expensive, and might be an excessive burden in applications where fine-grain error handling is not required.

The advantage of generating a signal when encountering an exceptional conditions is that it is more difficult for exceptions caused by software defects to slip through testing -- it is more difficult to ignore a core dump than to ignore an error return code. This approach presumes that software testing will be highly effective so that no core dumps will occur once the software product is shipped. POSIX does not require a task to be restartable once a signal has occurred, and in general it is assumed that data structures may have lost their integrity after a signal. Thus, using

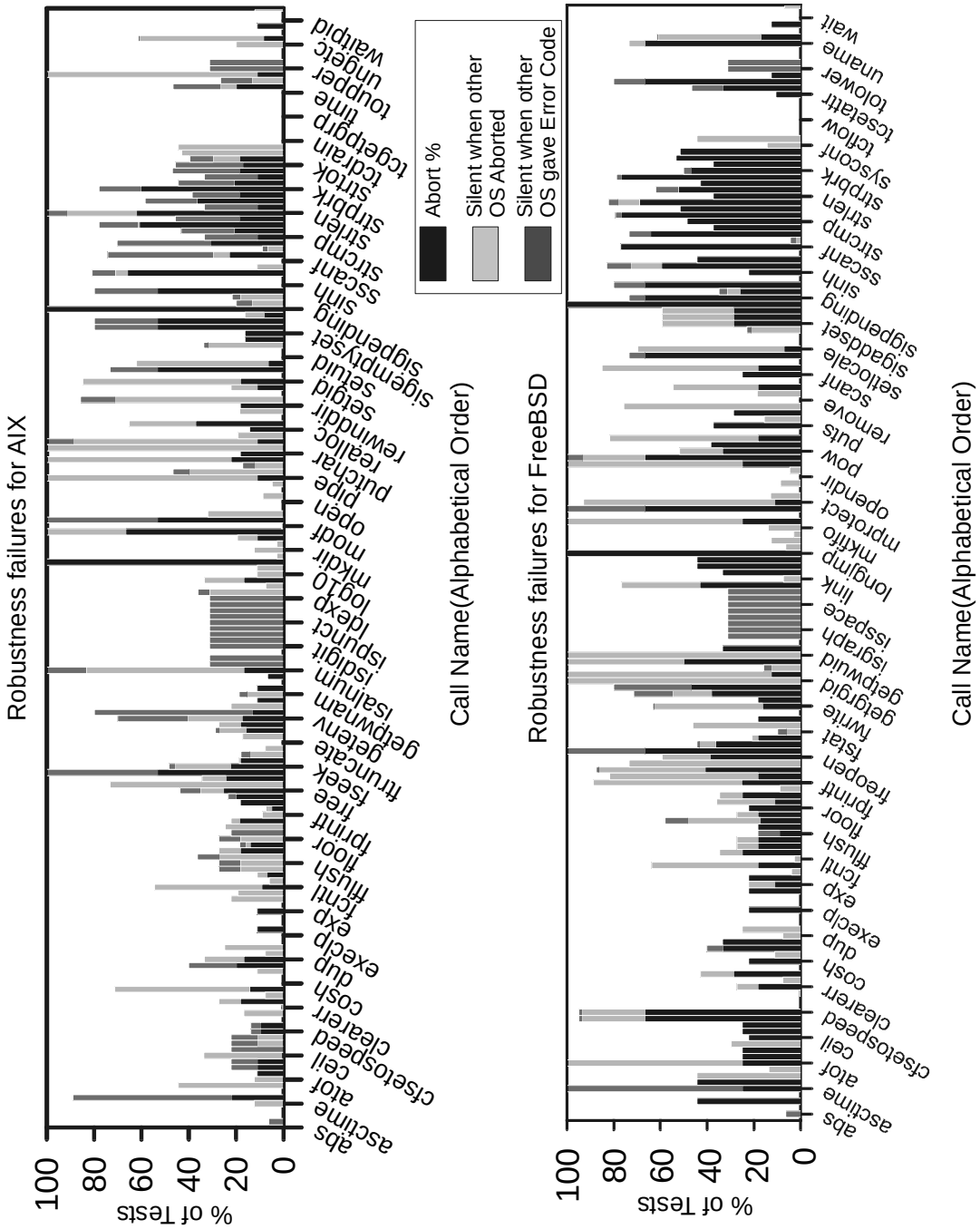

Figure 4. 2-version software voting comparison of AIX and FreeBSD.

a signal-based approach within POSIX means in practice that a task must be re-initialized and restarted after every exception. A large-grain recovery based on task restarting is probably simpler than writing detailed recovery code for a large number of possible exception sources, but may make it more difficult to perform diagnosis and correction of an exceptional situation at run-time should that be necessary. (It should be noted that POSIX provides hooks to associate detailed error information with signals, but does not provide a standard error reporting mechanism for signals of comparable power to the error return code mechanisms.)

Signals are considered Abort "failures" within the context of Ballista testing for two reasons. The first reason is that the POSIX API states that robustness is achievable by using error return codes[IEEE93 681-684], and not by using signals[IEEE93 2371-2374]. (It is no doubt possible to write robust software *systems* in a signal-based implementation, but each individual software task is not considered to be robust.) The second reason is that Ballista testing has deep roots in the fault tolerance community, where it is an article of faith that error codes will be checked, and that system designers should always be given the chance to perform fine-grain recovery. It is understood that the effort required to do so is not necessarily available or even desirable in some contexts; but precluding designers from doing fine-grain error checking is undesirable for a variety of mission- and safety-critical computing applications.

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 the preferred exception reporting mechanism.

6.2. AIX compared to FreeBSD

As a concrete example, consider two operating systems with different design philosophies for exception handling. From our correspondence with developers and the results of data analysis, it is clear AIX is designed to minimize Abort failures and provide extensive error return code support (although there is no stated or implied commitment to do so). Conversely, FreeBSD developers have specifically designed that OS to generate signals in many exceptional situations where error codes are not required by for POSIX compliance. As an example, consider a function call to compute the arc-cosine of an exceptional value (less than -1 or greater than 1). On AIX, `acos(e)` returns an error code of EDOM, while FreeBSD throws a signal of SIGFPE, aborting the calling process. Both of these responses are designed to indicate an error, and can be argued at some level to both be more robust than those operating systems

which suffered a Silent failure for this case. (It should be noted that these two operating systems are being compared because the philosophy of their developers was clearly divergent, and not to make a value judgement with respect to those or other operating systems or development teams.)

So, did either OS attain a high degree of robustness, even admitting the possibility of signals being a desirable response to exceptions? Figure 3 reveals that neither development team attained either no Aborts or all-Aborts for exceptional conditions.

Figure 4 shows 2-version software voting results between AIX and FreeBSD (in other words, Silent failures are located taking into account only mis-matches between results of those two operating systems, and thus are not identical to the 15-version software voting results presented earlier). 2-version voting is used to emphasize instances in which one OS was able to detect exceptions compared to another, rather than comparing them to an aggregate level of performance which might not for some reason be practicably attainable by any single OS. The Silent failure rates are likely to be over-estimates by a factor of approximately 1.25 based on the 20% false alarm rate discussed earlier, but simply applying an across-the-board reduction did not seem wise without an impractical amount of manual analysis to characterize each function involved.

A detailed analysis of the comparative data reveals that AIX still suffers a normalized Abort failure rate of 18% of exceptional conditions. Additionally, a normalized Silent failure rate of 22% occurred in instances that either generated error return codes or signals in FreeBSD (almost half of these were due to NULL pointer references that were permitted to proceed with read operations from address zero). Thus, writing a robust application for AIX would require dealing with a significant number of situations that potentially generate signals instead of error return codes, and a significant number of opportunities for Silent failures.

In comparison, FreeBSD reports 89% of exceptional conditions with either error return codes or signals (the FreeBSD developers do not consider signals to be a failure, but rather an intended design feature). However, 10% of exceptional conditions which generate error return codes in AIX are not detected by FreeBSD, resulting in Silent failures. Furthermore, 1% of exceptional conditions which generate a signal in AIX do not generate a signal in FreeBSD, also resulting in Silent failures. One might have speculated that AIX had a significantly reduced Abort

failure rate compared to FreeBSD simply because it ignores problems (resulting in elevated levels of Silent failures). But, in fact, FreeBSD did not have significantly fewer Silent failures than AIX. Of FreeBSD's silent failures, a bit more than half involve exceptional file pointer values (NULL, pointers to closed files, deleted files, and incompatible file mode versus access permission).

In part, the graphs in Figure 3 show that despite being POSIX compliant, there are significant implementation differences that lead to differing robustness failure profiles (this is true of other operating systems tested as well). Some specific differences to note are that the character conversion routines (*e.g.*, `isspace`, `isdigit`) seem to miss approximately the same number of chances to generate error codes, and string functions (*e.g.*, strlen) generate far more Silent failures on AIX than on FreeBSD, due to permitting reads to address zero.

The purpose of this comparison is not to take either AIX or FreeBSD to task, but rather to illustrate two key points:

- There is a wide variation of how exceptional parameter values are handled by different operating systems. Although exception handling is not required to be POSIX compliant, writing a portable and robust application is made more difficult by such variation
- No OS examined could be considered to be highly robust, even setting aside the issue of whether error codes or signals are the "right" mechanism to use for exception reporting.

7. Conclusions

The Ballista testing methodology 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% for FreeBSD version 2.2.5 to a high of 76% for SunOS version 5.5 and Irix version 6.2. The majority of robustness failures were Abort failures (ranging from 18% to 33%), in which a signal was sent from the system call or library function itself, causing an abnormal task end. The next most prevalent failures were Silent failures (ranging from 6% to 19%), in which exceptional inputs to a Module under Test resulted in erroneous indication of successful completion. Additionally five operating systems had at least one system call which caused a catastrophic system failure in response to a single function 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 core dumps 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.

This work makes several contributions. It provides the first repeatable, quantified measurements of operating system robustness, although the measurements are limited to exceptional conditions. Such a metric is important to those application developers designing the growing numbers of critical applications that 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. Robustness 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 robustness.

8. References

[Avizienis85] Avizienis, A., "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol.SE-11, no.12, p. 1491-501.

[Beizer95] Beizer, B., *Black Box Testing*, New York: Wiley, 1995.

[Carrette96] Carrette, G., "CRASHME: Random input testing," (no formal publication available) http://people.delphi.com/gjc/crashme.html accessed July 6, 1998.

[Cristian95] Cristian, Flaviu, "Exception Handling and Tolerance of Software Faults," In Software Fault Tolerance, Michael R. Lyu (Ed.). Chichester: Wiley, 1995. pp. 81-107, Ch. 4.

[Gehani92] Gehani, N., "Exceptional C or C with Exceptions," Software - Practice and Experience, 22(10): 827-48.

[Goodenough75] Goodenough, J., "Exception handling: issues and a proposed notation," Communications of the ACM, 18(12): 683-696, December 1975.

[Hill71] Hill, I., "Faults in functions, in ALGOL and FORTRAN," The Computer Journal, 14(3): 315-316, August 1971.

[IEEE90] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, IEEE Computer Soc., Dec. 10, 1990.

[IEEE93] 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 Computer Society, 1994.

[Jones96] Jones, E., (ed.) The Apollo Lunar Surface Journal, Apollo 11 lunar landing, entries at times 102:38:30, 102:42:22, and 102:42:41, National Aeronautics and Space Administration, Washington, DC, 1996.

[Kanawati92] Kanawati, G., Kanawati, N. & Abraham, J., "FERRARI: a tool for the validation of system dependability properties," 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems. Amherst, MA, USA, July 1992, pp. 336-344.

[Koopman97] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. & Marz, T., "Comparing Operating Systems Using Robustness Benchmarks," Proceedings Symposium on Reliable and Distributed Systems, Durham, NC, Oct. 22-24 1997, pp. 72-79.

[Kropp98] Kropp, N., Koopman, P. & Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," 28th Fault Tolerant Computing Symposium, in press, June 23-25, 1998.

[Lions96] Lions, J.L. (chairman) Ariane 5 Flight 501 Failure: report by the inquiry board, European Space Agency, Paris, July 19, 1996.

[Miller89]Miller, B., Fredriksen, L., So, B. , An empirical study of the reliability of operating system utilities, Computer Science Technical Report 830, Univ. of Wisconsin-Madison, March 1989.

[Miller98] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A. & Steidl, J., Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, Computer Science Technical Report 1268, Univ. of Wisconsin-Madison, May 1998.

[Segall90] Barton, J., Czeck, E., Segall, Z., Siewiorek, D., "Fault injection experiments using FIAT," IEEE Transactions on Computers, 39(4): 575-82.

[Tsai95] Tsai, T., & R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," Proceedings Eighth International Conference. on Modeling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, Sept. 20-22 1995, Springer-Verlag, pp. 26-40.

[Vo97] Vo, K-P., Wang, Y-M., Chung, P. & Huang, Y., "Xept: a software instrumentation method for exception handling," The Eighth International Symposium on Software Reliability Engineering, Albuquerque, NM, USA; 2-5 Nov. 1997, pp. 60-69.