29th Annual International Symposium on Fault-Tolerant Computing, 15-18 June 1999, Madison, Wisconsin.

Comparing the Robustness of POSIX Operating Systems

Philip Koopman & John DeVale
Department of Electrical and Computer Engineering &
Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
koopman@cmu.edu jdevale@ece.cmu.edu

Abstract

Critical system designers are turning to off-the-shelf operating system (OS) software to reduce costs and time-to-market. Unfortunately, general-purpose OSes do not always respond to exceptional conditions robustly, either accepting exceptional values without complaint, or suffering abnormal task termination. Even though direct measurement is impractical, this paper uses a multi-version comparison technique to reveal a 6% to 19% normalized rate at which exceptional parameter values cause no error report in commercial POSIX OS implementations. Additionally, 168 functions across 13 OSes are compared to reveal common mode robustness failures. While the best single OS has a 12.6% robustness failure rate for system calls, 3.8% of failures are common across all 13 OSes examined. However, combining C library calls with system calls increases these rates to 29.5% for the best single OS and 17.0% for common mode failures. These results suggest that OS implementations are not completely diverse, and that C library functions are both less diverse and less robust than system calls.

1. Introduction

The robustness of the operating system (OS) Application Programming Interface (API) is becoming increasingly important. Cost and time-to-market constraints are pressuring even critical-system developers to use off-the-shelf commercial OS software. While many of these OSes are generally considered robust, the Ballista testing system has produced results showing that commercial OSes have a significant robustness failure rate for single procedure calls with exceptional parameter values at the task level, and even a few readily reproducible catastrophic failure modes [13].

This paper compares the results of multiple OS implementations to increase the understanding of their robustness failure modes and, in particular, instances in which they fail to detect exceptional input parameter values. Additionally, a large set of test data on multiple implementions of the same API offers a unique chance to explore the issue of quantifying large-scale software diversity, albeit only of exception handling characteristics.

First, a multi-version comparison of Ballista testing results is used to identify so-called Silent robustness failures in fifteen OSes. A Silent failure is one in which a function call produces no indication of an error when fed exceptional parameter values when, in fact, such an indication should be produced to implement robust behavior. These failures cannot otherwise be measured without a behavioral specification. Additional results presented include using a similar technique to "distill" non-exceptional test cases from the Ballista combinational testing approach, and a listing of the data types most often correlated with robustness failures.

Next, N-way comparisons of different OS robustness testing results are used to measure software diversity from the point of view of robust exception handling. While actually implementing a system with an N-version OS seems impractical, the results of this comparison give the first large-scale experimental results for understanding the level of software diversity that is likely to be found in commercial products (again, limited to exception handling characteristics). The results of comparing thirteen OS implementations (two OSes had to be eliminated to maximize usable comparison data) show that OS implementations are not entirely diverse, and that an assumption of failure independence seems less unreasonable for OS system calls than for C library functions.

2. Background

This paper takes concepts from the field of N-version software fault tolerance and combines them with results from a scalable robustness testing methodology. It then assesses OS robustness failures and the inherent diversity of multiple version techniques on one type of large software system.

2.1. N-version software

N-version software involves using *N* different versions of programs implementing the same specification for building robust, fault-tolerant software [1][2]. Generally these systems use an idea similar to N-version modular hardware redundancy, basing actions upon a majority vote taken of several software versions to determine which output values are correct. Numerous experiments have been performed to study N-version software effectiveness in several mission critical areas, most notably in the aerospace industry [3][5][15]. Variations on this theme have been successfully implemented in safety critical industrial systems such as the Airbus flight control system, NASA's Space Shuttle, and railway control systems.

As an example, a large research effort involving fault tolerant software controls for a NASA avionics system involved five independent teams of developers chosen from various universities [8]. They carefully constructed a specification and performed several levels of testing on the developing software, which upon completion averaged approximately 2500 lines of Pascal code. Detailed analysis was performed on the collected data in a later study, and found that less than 20% of the faults detected could be classified as similar[17]. Further, the study concluded that after certification testing this number dropped significantly, and had eliminated all specification faults.

Another study suggests that an assumption of failure independence based on diversity may not be universally possible [4][9][10], but these results are controversial [3][11][17]. More importantly, such studies are inherently limited by the high cost of software development, making it prohibitively expensive to conduct full-scale N-version software studies in general. So, it is reasonable to ask an assumption of multi-version diversity scales to systems having a million lines of code, but in general undertaking a parallel development effort of that magnitude is impractical.

Fortunately, there are a some commercially developed, full-size systems built to standard Application Programming Interfaces (APIs). These systems can be tested to determine the relative independence of failures. The problem of evaluating multiple version effectiveness can thus be reduced to one of finding a test suite which is economical to develop, and yet has not already been used by software vendors for defect removal. This paper uses such results for robustness failures found by the Ballista robustness testing system applied to POSIX [7] OS implementations.

2.2. Ballista testing methodology

In brief, the Ballista testing methodology involves automatically generating combinations of exceptional and valid 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, the task abnormally terminated, the task hung, or whether the entire system crashed. A detailed discussion of test case generation can be found in a previous paper [13], but the general test methodology is summarized below.

Ballista operates at the level of single function calls to create repeatable, simple tests that nonetheless uncover robustness failures. 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 pools 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 is 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 both valid and invalid test values. While actual tests are performed in batches for efficiency, in practice virtually all test results have been found to be reproducible in isolation.

The Ballista test harness categorizes the test results according to the CRASH severity scale [12]:

- **Catastrophic** failures occur when the entire OS becomes corrupted or the machine crashes or reboots. In other words, this is a complete system crash.
- **Restart** failures occur when a function call to an OS function never returns, resulting in a task that has "hung" and must be terminated by force.
- **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 the MuT.
- **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), and which should in fact produce an error report in a robust system. This is not to be confused with the problem of non-diagnosable experiments due to limited observability, because the error reporting mechanism is fully observable, and is observed to falsely indicate "no error" (this is an application-centric rather than OS-centric view).
- **Hindering** failures occur when an incorrect error code is returned from a MuT, which could make it more difficult to execute appropriate error recovery. These failures have been observed in practice, but are beyond the

scope of this paper.

There are two additional possible outcomes of executing a test case. A test case might return 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 desired operation. This happens when the parameters in the test case happen to be all valid (a non-exceptional test case), or when it is unreasonable to expect the OS to detect an exceptional situation (such as pointing to an invalid address in the same memory page as a valid address).

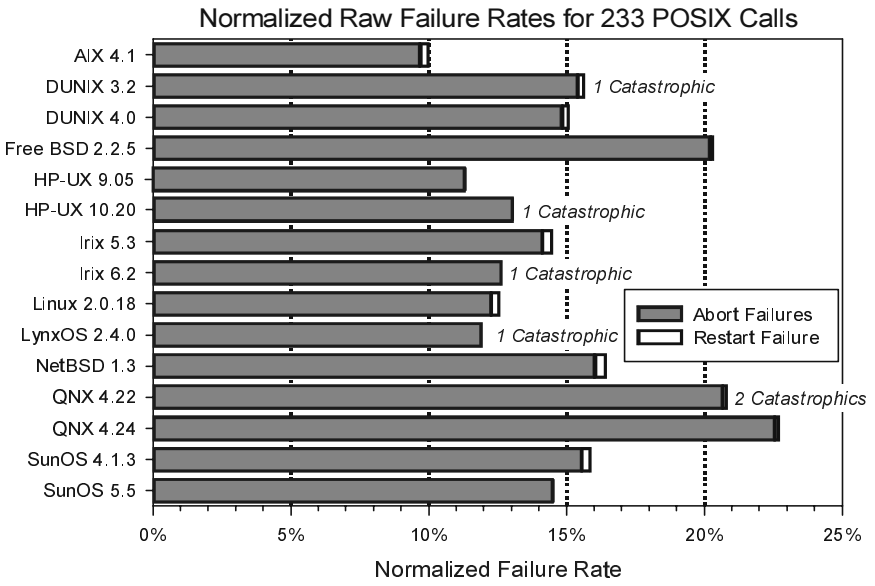

Figure 1. Raw robustness tests on 15 POSIX operating systems reveal a significant Abort failure rate and several Catastrophic failures.

3. OS test data

Fifteen OS implementations from ten vendors were tested with Ballista, yielding a total of 1,074,782 data points on up to 233 selected POSIX functions and system calls (most systems did not support all tested calls, and thus not all tests were run on every OS). 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, this meant that GNU C version 2.7.2.3 and the GNU C libraries were used to build the test suite. A summary of robustness failure rates is shown in Figure 1, in which an average failure rate is computed by normalizing the failure rate as a proportion of failed test cases for each function, and then taking a uniformly weighted arithmetic mean across all supported functions. (For a particular application, a weighted mean per an operational profile might be desirable, but the results presented here are generic for the API rather than any particular system.) Test execution took approximately three to eight hours per system.

There were six function/OS pairs that resulted in entire operating system crashes (either automatic reboots or system hangs). As an example of these Catastrophic failures, Irix 6.2 crashes and requires a manual hardware reset when executing the call:

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

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. 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.

A particular point of interest was that previously reported results were subject to criticism by OS vendors because in several cases the latest available OS version had not been tested. However, when a newer version was tested there was not necessarily an improvement in robustness (*e.g.*, for HP-UX and QNX). Additionally, Figure 1 reports results for DUNIX 4.0B with no Catastrophic failures. However, on DUNIX 4.0D a catastrophic failure sprang into existence due to an apparent change in the aio_raw library for the call:

```
mprotect(malloc((1<<29)+1),65537,0);.
```

There are, however, some questions left unanswered by the data in Figure 1. For example, the designers of FreeBSD have told us that they intentionally generate Abort failures as their preferred error reporting mechanism, and suggested it was possible they look bad by comparison because other OS implementations might instead suffer from elevated levels of Silent failures. (The merits of the FreeBSD strategy involve a debate about the desire to make errors highly visible during development *vs.* a desire to perform fine-grain error recovery once an application is fielded, but that is beyond the scope of this discussion.) For example, this might mean that AIX looks better than it really is because address 0 is readable without exception, leading to the possibility of elevated Silent failure rates for NULL pointer dereferencing. Therefore, it was important to determine Silent failure rates even though they could not be directly measured.

4. Data analysis via multi-version comparison

The scalability of the Ballista testing approach hinges on not needing to know the functional specification of a MuT, so that the same combinations of parameter values can be used to test any function taking a given tuple of data types. 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 (in which all parameter values fall within normal, expected ranges) 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 robustness failure rates using a variation of multiple version software voting.

There have been many efforts to define and evaluate various multiple version voting algorithms and schemes (*e.g.*, [6][14][16]). Most of these focus on resolving problems with the existing standard voting techniques. Of particular concern is how separate algorithmic approaches might induce different round-off errors, leading to separate correct, but different, answers.

When applying voting techniques to the domain of software robustness, we benefit from our studied disinterest in the functional correctness of the output. Thus, robustness testing results are only concerned with whether parameter values were in fact exceptional, whether exceptional values were detected, and whether at least one of *N* OS versions responded to exceptional values gracefully, but *not* to whether the function performed as specified (which is obviously important, but is more properly the subject of traditional software testing). Thus, for our purposes, it suffices if *at least one* of *N* versions performs exception detection and returns gracefully, so we use a one-of-N *comparison* strategy rather than requiring an M-of-N majority voting scheme. For example, if any OS version reports an exception condition, it is assumed that all versions should have detected that exception (false alarm rates are discussed shortly).

4.1. Elimination of non-exceptional tests

Ballista uses combinations of exceptional and non-exceptional parameter values to do testing. Partly this is to avoid correct detection of exceptional values for one parameter from masking exceptions that might otherwise go unnoticed on other parameters. But, also, partly this is a side-effect of scalable testing in which all functions are tested with parameter values that may be exceptional for only some functions (for example, a read-only file is exceptional for a write function, but not for a read function tested with the same parameter values).

N-way comparisons were used to identify and prune non-exceptional test cases from the data set. The comparisons 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 2 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, thus suggesting a low rate of false screenings. While there is the possibility that exceptional test cases slipped passed this screening, it seems unlikely that the number involved would materially affect the results.

4.2. An estimation of Silent failure rates

Once the non-exceptional tests were removed, a different variation on multi-version software comparison was 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

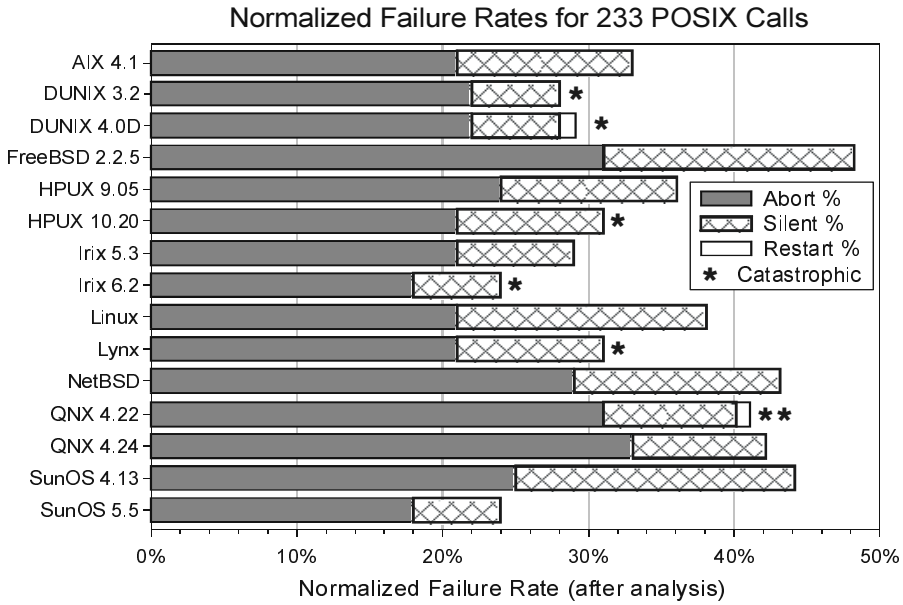

Figure 2. Multi-version comparisons eliminate the effects from non-exceptional tests and permit estimating Silent failure rates.

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 several dozen results indicated that approximately 80% of detected test cases were actually Silent failures. Of the 20% of test cases that were false alarms:

- 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 4.1.3) apparently checked the data for validity, while others did not.

Classifying the Silent failures sampled revealed some additional software defects that manifested in unusual, but specified, situations. For instance, POSIX requires `int fdatasynch(int filedes)` to return the `EBADF` error if `filedes` is not valid, and if the file is not open for write [7]. 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 [7]; however only FreeBSD, Linux, and SunOS 4.1.3 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 caused by 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 in functions for which no error codes were returned. To our surprise this was only somewhat effective at identifying Silent failures, but did turn out to be a fruitful way to reveal software defects. A relatively small number (37,434) of test cases generated an Abort failure for some operating systems, but successfully completed for all other operating systems. 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. For example, writing zero bytes from a NULL pointer memory location might Abort if a byte is fetched from memory before checking remaining length to transfer, or return successfully if length is checked before touching memory.

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)`.

Figure 2 shows the aggregate results of Silent failures from multiple version comparisons with error codes (weighted at 80%) plus Silent failures from multiple version comparisons with only Abort failures (weighted at 50%). These results are obviously approximated, but do indicate that Silent errors can be prevalent. With respect to the earlier discussion of potential AIX Silent errors, it was found that error checking in other areas apparently made up for lack of error detection on NULL pointer reads, giving it a Silent failure rate comparable to several other systems, including FreeBSD.

5. Multi-version comparison of OS diversity

An additional use for multi-version comparison techniques is in attempting to quantify the diversity among OS implementations with respect to exception handling robustness. Analysis in this section was performed on a subset of the robustness testing data: 40,619 tests cases on each of 13 operating systems, for a total of 528,047 test results in all. Two OSes (QNX 4.22 and Irix 5.3) were eliminated because they did not support many of the functions supported by other OSes, and similarly many functions were eliminated because they were supported by few OSes. The selected OSes and functions maximize the number of usable test cases constrained by a requirement of every test case being supported by all OSes used in the comparisons.

In Figure 3, the horizontal axis indicates the value of N for N-way comparisons, from N=1 (data points for 13 possibilities, one for each single OS) through intermediate values such as 6 (depicting all combinations of 13 OSes taken 6 at a time), to a single data point for 13-way OS

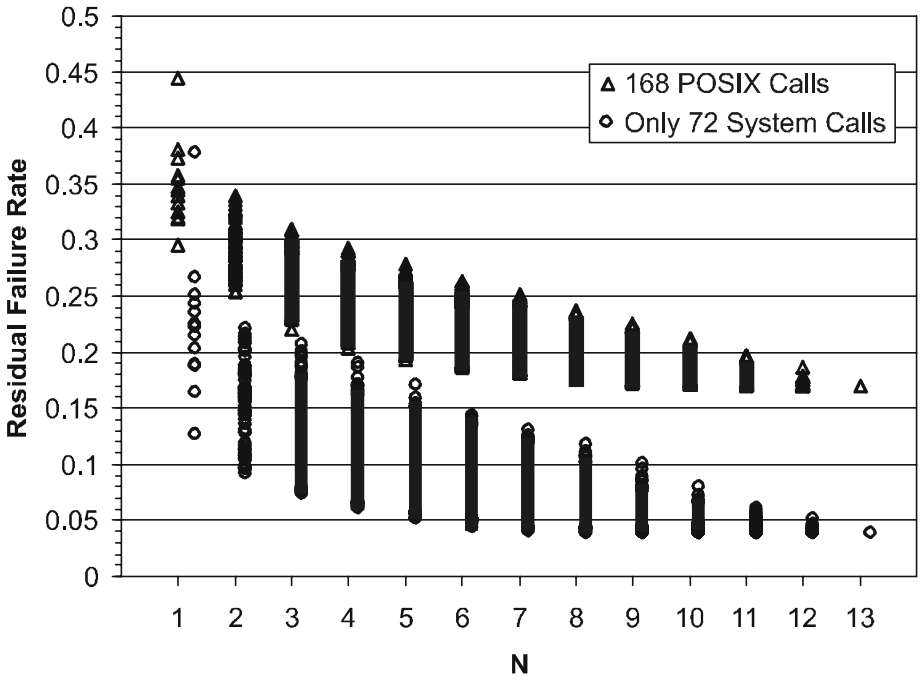

Figure 3. Multi-version comparisons of all combinations of *N* OS implementations reveal wide variations in robustness failure rates and exception handling diversity.

comparisons. For each value of N there are two sets of data: triangles for all 168 calls, and circles for the subset of 72 system calls (excluding C library calls). The data points for most values of N blend together, but the result is one that shows the span of effectiveness for various combinations, with the lowest point being the lowest possible robustness failure rate for the best possible *hypothetical* combination of N OS versions executed back-to-back (maximum available diversity), and the highest point being the worst one could do by hypothetically picking the set of N OS versions with the highest degree of robustness failure overlap (minimum available diversity).

Thus, in Figure 3 the height of any particular data point measures the robustness failure rate not detected by *any* of the N OS implementations for that point. The span of points for a particular *N* value indicates the difference in diversity among combinations (least diverse being the highest). The general difference between triangles and circles indicates differences between aggregate POSIX calls and a system-call-only subset of the POSIX API. Finally, the downward trend in failure rates as N increases indicates the additional diversity added by each additional OS considered in a pool of N OS implementations (*i.e.*, removing additional common mode failures for a hypothetical composite exception detection system).

From Figure 3, one can see that system calls are generally far more robust than C library calls (except for the N=1 point for QNX 4.24 at 37.8% robustness failure rate), especially taking into account that the triangle points are for aggregate behavior of 169 functions that also includes the data for the 72 system calls.

For both C library functions and system calls there is a significant span of robustness failure rates up through moderately high values of N, as well as a noticeable decrease in the lowest possible residual failure rate for increasing N. These trends indicate that an increasing amount of diversity in exception handling robustness is available for increasing N. Or, put another way, Figure 3 suggests that careful selection of *N* OS implementations reduces the common mode robustness failures observed as *N* increases. Furthermore, which *N* OS implementations one selects dramatically affects the degree of common mode failure observed.

To a degree, one might expect a wide spread of residual failure rates due to the fact that BSD and System V Unix implementations have significantly diverged, and that Linux was specifically developed from scratch to avoid licensing entanglements. In fact, the data for system calls suggest that a half-dozen implementations have evolved significant diversity from each other with respect to system calls (the bottom-most circle decreases in residual failure rate significantly through about *N*=6). Similarly, the fact that the GNU C libraries and BSD-based libraries were independently developed helped provide diversity.

However, the potential bad news from a diversity point of view is that no *two* OS implementations approach the system call diversity of all 13 OS implementations. Suppose that one were to, in the best case, select the two most diverse OS implementations (AIX and FreeBSD, which also appeared in personal correspondence to have the most extreme views as to what constituted robustness). If one were to make that selection with the intent of avoiding all common-mode OS exception handling failures among multiple machines running the same application software, one would only achieve a 9.7% common mode failure exposure for system calls and a 25.4% exposure for all calls tested, assuming equal weighting among all functions. And, if one were to pick any two other OS implementations, the common mode failure exposure would be higher. Selecting the best four or five or six OS

implementations improves the situation to a degree, but becomes increasingly impractical in a real installation.

An additional conclusion from Figure 3 and other analysis we have performed is that a significant fraction of Abort failures come from C library calls, and tend to be correlated among OS versions. This is perhaps not a surprise considering that these calls, most notably the string handling routines, are notoriously susceptible to memory over-run and pointer value problems, and have an API definition that does not specifically encourage robustness. Similarly it is not unreasonable to conjecture that C library code has been shared and redistributed largely unchanged, since most OS vendors concentrate on porting an OS to their hardware and optimizing performance of system calls rather than the C library functions. Nonetheless, if one were to presume a high degree of implementation diversity simply because C libraries were (presumed to be) independently developed and came from independent vendors, one would be mistaken from an exception-handling perspective.

5.1. Frequent sources of robustness failure

Given that robustness failures are prevalent, what might be fixed to improve them? 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 is impractical with such a large set of experimental data. Therefore, the best that can be presented is a list of data values that are highly correlated with robustness failures (Table 1) and functions that have high robustness failure rates even after 13-way failure detection comparisons are applied (Table 2).

Table 1: Data types most commonly associated with abort robustness failures for 15 operating systems.

Data Value	Percent associated with robustness failures
Invalid file pointers (excluding NULL)	94.0%
NULL file pointers	82.5%
Invalid buffer pointers (excluding NULL)	49.8%
NULL buffer pointers	46.0%
MININT integers	44.3%
MAXINT integers	36.3%

Table 1 shows that NULL and invalid pointer values are the most common causes of Abort failures, which is probably no great surprise. However, minimum and maximum integer values also seem to be correlated with robustness failures, which was not an obvious outcome.

Table 2 shows the sigjmp/longjmp pair high on the robustness failure rate list for both system calls and C library calls. Note that Table 2 deals with residual failure rates after comparisons, and so represents failures that no OS dealt with gracefully. Beyond that, string functions and math functions appear on the C library list, but are not the only culprits.

Table 2: Functions with highest residual failure rates after 13-way failure detection voting.

System Calls		C Library Calls	
siglongjmp	66.7%	longjmp	100%
sigsetjmp	40.0%	strcpy	50%
ctermid	20.0%	atan	44.4%
closedir	14.3%	frexp	40.0%
readdir	14.3%	modf	40.0%
getenv	13.3%	setjmp	40.0%
getgrnam	12.5%	sprintf	39.8%
getpwnam	12.5%	strftime	33.3%
getgrnam	12.5%	strncat	29.2%
getcwd	12.5%	strcat	28.1%
creat	11.1%	printf	25%
execlp	11.1%	strncpy	23.3%
execvp	11.1%	fabs	22.2%
sigaddset	8.2%	tan	22.2%

6. Conclusions

This paper documents the use of multi-version software comparison techniques in analyzing a set of large, mature, commercially available software systems. In terms of robustness assessment, the multi-version approach permitted identifying Silent failures (failure to indicate the occurrance of an exceptional condition when one could have been indicated) with a reasonable degree of accuracy, but without need to create functional specifications for all 233 functions tested. The result was the discovery of a normalized Silent robustness failure rate of 6% to 19% for single-OS tests. Additionally, a multi-version comparison approach permitted screening out the non-exceptional tests generated by Ballista robustness testing. An additional, unexpected, result was finding some bugs (software defects with respect to required POSIX functionality) on POSIX-certified, commercial operating systems.

A second approach to using multi-version software comparisons produced measurements of the diversity of POSIX operating systems. In particular, measurements were made for the residual robustness failure rate that would remain if one were to (hypothetically) combine the

most graceful exception handling abilities of every one of *N* different OS implementations, with the value of *N* ranging from one to thirteen. The results were that the core set of 78 system calls tested were moderately robust and diverse, but not perfectly so. This result suggests that system calls have a reasonable level of software diversity for multiple version purposes, or alternately that a developer combining techniques already in existing OS implementations into a single OS might possibly (barring technical hurdles) improve the robustness that single OS. However, selecting any *pair* of OS implementations did not seem to result in the highest possible degree of implementation diversity.

The results for multi-version assessment of C library calls were less promising. C library calls appear to be significantly less robust on average than OS system calls, and additionally do not appear to be highly diverse in implementation. This finding means that presuming that C library implementations are diverse simply because they come from different vendors or have been independently developed is probably not a good idea.

Although the data presented here do not purport to apply to functional correctness, they do suggest that commercial off-the-shelf software developed to a particular API might possibly lack diversity, at least with respect to exception handling. It seems very likely that some of this lack of diversity is due to the design of the API itself, but it is difficult to believe that this is the only factor at work.

The detailed source data used in preparing this paper are available at http://www.ices.cmu.edu/ballista

7. Acknowledgment

This research was sponsored by DARPA contract DABT63-96-C-0064, the Ballista project.

8. References

[1] Avizienis, A., "The N-version approach to fault-tolerant software", *IEEE Trans. on Software Engineering*, vol.SE-11, no.12, 1985, p. 1491-501.

[2] Avizienis, A., Gunningberg, P., Kelly, J.P.J., Stringini, L., Traverse, P.J., Tso, K.S., Voges, U., "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software", *15th Fault Tolerant Computing Symp.*, 1985, p. 126-134.

[3] Avizienis, A., Lyu, M., Schutz, W., "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," *18th Intl. Symp. on Fault Tolerant Computing*, 1988.

[4] Brilliant, S.S., Knight, J.C., Leveson, N.G., "Analysis of Faults in an N-Version Software Experiment", *IEEE Trans. on Software Engineering*, vol. 16, no. 2, 1990, pp. 238-47.

[5] Chen, L., Avizienis, A., "N-Version Programming : A Fault Tolerance Approach to Reliability of Software Operation," *The Eighth Intl. Symp. on Fault Tolerant Computing*, 1978.

[6] Gersting, J.L., Nist, R.L., Roberts, D.B., Van Valkenburg, R.L., "A Comparison of Voting Algorithms for N-Version Programming," *Proceedings of the twenty-fourth Annual Hawaii Intl. Conference on System Sciences*, vol. 2, 1991, pp. 253-62.

[7] *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.

[8] Kelly, J.P., Eckhardt, D.E., Vouk, M.A., McAllister, D.F., Caglayan, A.K., "A Large scale Second Generation Experiment in Multi-Version Software: Description and Early Results," *Eighteenth Intl. Syposium on Fault Tolerant Computing*, 1988.

[9] Knight J.C., Leveson, N.G., St. Jean L.D., "A large scale experiment in N-version programming", *15th Fault Tolerant Computing Symp.*, 1985, 135-139

[10] Knight Leveson, "An empricial suty of failure probabilities in multi-version software", *16th Intl. Fault Tolerant Computing Symp.*, 1986, 165-170

[11]Knight, J.C., Leveson, N.G., "A reply to the criticisms of the Knight and Leveson experiment", *SIGSOFT Software Engineering Notes*, vol. 15, no.1, 1990, p. 24-35.

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

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

[14] Lorczak, P.R., Caglayan, A.K., Eckhardt, D.E., "A Theoretical Investigation of Generalized Voters for Redundant Systems," *19th Intl. Symp. on Fault Tolerant Computing*, 1989.

[15] Lyu, M.R., "Software Reliability Measurements in N-Version Software Execution Environment," *Proceedings of the Third Intl. Symp. on Software Reliability and Engineering*, 1992.

[16] McAllister, D.F., Sun, C., Vouk, M.A., "Reliability of Voting in Fault-Tolerant Software Systems for Small Output-Spaces," *IEEE Trans. on Reliability*, vol. 39, no. 5, 1990, pp. 524-34.

[17] Vouk, M.A., McAllister, D.F., Caglayan, A.K., Walker, J.L., Eckhardt, D.E., "Analysis of Faults Detected in a Large-Scale Multi-Version Software Development Experiment," *Proceedings of the Ninth Digital Avionics Systems Conference*, 1990.