Comparing Operating Systems Using Robustness Benchmarks

Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, Ted Marz
Carnegie Mellon University
Pittsburgh, Pennsylvania

Abstract

When creating mission-critical distributed systems using off-the-shelf components, it is important to assess the dependability of not only the hardware, but the software as well. This paper proposes a way to test operating system dependability. The concept of response regions is presented as a way to visualize erroneous system behavior and gain insight into failure mechanisms. A 5-point ***"CRASH"*** *scale is defined for grading the severity of robustness vulnerabilities encountered. Test results from five operating systems are analyzed for robustness vulnerabilities, and exhibit a range of dependability. Robustness benchmarking comparisons of this type may provide important information to both users and designers of off-the-shelf software for dependable systems.*

1. Introduction

Two forces are combining to promote increased use of Commercial Off-The-Shelf (COTS) software components in mission-critical distributed systems. First, inexpensive standardized computers are assuming mission-critical roles in an increasing number of embedded application areas as they displace custom-designed electromechanical, digital, and analog components. Second, a need for cost reduction is pushing designers to use inexpensive, general-purpose COTS software rather than custom-designed software for mission-critical systems.

Most general-purpose applications seem to have little demand for graceful and robust error recovery. In fact, it is common for personal computer software to crash or "hang", requiring task restarts and even machine reboots. While users may want more dependable general-purpose software, they have not yet created a marketplace in which vendors are compelled to provide it.

On the other hand, it is possible that widely deployed COTS software has fewer bugs overall than custom software due to corrective actions in response to bug reports from the field. But, it seems likely that when a bug is encountered the failure response is less graceful than in software specifically designed to be dependable. A significant challenge is to be able to use COTS software in order to reduce cost while still maintaining system dependability.

1.1. Is off-the-shelf software dependable?

An important issue when using COTS software is whether its reliability and failure responses will be adequate for a mission-critical application. One way to approach this area is to focus on the operating system (OS), which is the foundation upon which application software rests. Because they serve a general-purpose and widely employed function, OSs are obvious candidates for off-the-shelf component acquisition. Furthermore, because of the widely installed base and varied workloads of many OSs, it seems reasonable to expect that they should have evolved to be fairly robust. In particular, they have been subjected to a large amount of ad-hoc testing by application developers who have both the interest and technical expertise to submit bug reports. So, one reason to look at OSs is that they may well represent a fairly optimistic case of what one can expect in terms of robust COTS software.

The work presented here employs a portable robustness benchmarking methodology to assess the dependability of COTS OSs. The focus is on *robustness gaps*, which are situations in which the OS fails to properly detect or contain an exceptional condition. Because real-world application software is seldom, if ever, bug-free, the emphasis here is upon testing the robustness of an OS when application code provides invalid inputs to it.

1.2. Portability permits comparisons

Comparisons are most readily performed when the benchmark being used has a portable, high-level fault injection mechanism. In order to attain this goal, it is important to avoid use of special-purpose hardware or platform-dependent software.

Previous work in Software-Implemented Fault Injection (SWIFI) reduced or eliminated the need for special-purpose hardware in order to perform fault injection (*e.g.*, FIAT [1],

and FERRARI [2]). Unfortunately, these approaches were still platform-specific, and thus did not provide an easy way to compare the relative robustness of different systems.

The work presented in this paper achieves portability among platforms. It does so by using the naturally provided entry point into the system software — operating system calls — rather than injecting faults into arbitrarily accessible locations in the system. While this approach may be less general than arbitrary fault injection, it has the advantage of portability. Moreover, it exercises the same mechanism for introducing exceptional inputs that real application software uses — the OS call interface.

1.3. Overview

The remaining sections in this paper describe the implementation and results of a set of portable robustness benchmarks. Section 2 discusses the approach to portable robustness benchmarks. Section 3 defines the five-point ***CRASH*** robustness gap severity scale. Section 4 presents the experimental results for five OSs: Mach, HP-UX, QNX, LynxOS, and Stratus FTX. Section 5 introduces the concept of response regions as a way to describe and visualize robustness gaps.

2. Portable benchmarks

The robustness testing methodology used has its origins in the work of Dingman *et al.* [3], which performed repeatable robustness testing on a special-purpose, fault tolerant aerospace computer. In this methodology, operating system calls were made with various combinations of valid and invalid parameters. The resultant stress on the OS revealed erroneous system responses, including the ability to crash the entire system from within user code.

This previous generation of robustness testing was accomplished on a single system, and established the viability of invalid value parameter testing as a way of uncovering OS robustness gaps. The work presented here extends that methodology to a set of portable robustness benchmarks with metrics available to compare the performance of different OSs.

2.1. Test case generation

The focus of the robustness testing presented here is on use of operating system calls rather than user code. In order to generate an exemplary workload, the most-used software tools on a graduate student's Unix workstation were instrumented to count the dynamic frequency of operating system service calls. The tools commonly used on that workstation were exercised, including the EMACS Text Editor, GNU C Compiler, GNU Debugger, Bitmap X-Window image editor, and XV graphics file viewer. The intent of the study was to create a reasonable workload for testing purposes, rather than a comprehensive set of execution frequency data. The results of the study were that read(), write(), open(), close(), fstat(), stat(), and select() were the system calls most frequently executed, and were thus chosen as the area of concentration for the robustness benchmarks. [4]

Each of the system calls selected for testing takes a set of parameter values. These calls can be exercised by selecting combinations of valid and invalid parameter values for each test case. Table 1 shows the input parameters required by each OS function tested. In the portable robustness benchmarks, each parameter is tested with several input values (Table 2). The parameter values in Table 2 were chosen in order to exercise both plausible bugs (*e.g.*, mismatch between file handle access request and file access permissions) as well as exercise memory protection mechanisms that might not be properly handled (*e.g.*, accessing a memory location beyond allocated memory to trigger a page fault and corresponding protection violation). The current values were chosen using intuition. As time goes on experience will be used to build a larger "library" of test case values for each parameter type.

Each block in Table 2 indicates a set of test cases for each particular parameter type. All combinations of parameter values are tested for each function, giving to up to several hundred test cases. For example, read() is tested with all combinations of: 7 different file handle test cases, 9 different memory buffers, and 8 different lengths, for a total of 7x9x8 = 504 test cases. For read(), the first test case would be have the value: {FILE HANDLE=handle for an existing file that has been opened and then closed; BUFFER=buffer address

SYSTEM CALL	# CASES	PARAMETERS
read	504	FILE HANDLE BUFFER LENGTH
write	504	FILE HANDLE BUFFER LENGTH
open	495	FILE NAME FILE ACCESS FLAG FILE PERMIT MODE
close	7	FILE HANDLE
stat	81	FILE NAME BUFFER
fstat	63	FILE HANDLE BUFFER

Table 1. Parameters and number of combinations for a total of 2059 system call tests.

PARAMETER	#	VALUES
FILE HANDLE	7	VALID FILE, CLOSED VALID, OPEN FOR READ VALID, OPEN FOR R/W DELETED FILE ALTERED FILE HANDLE NULL -1
BUFFER	9	START OF 16 BYTE BUFFER START OF 4KB BUFFER START OF 64KB BUFFER 32KB IN TO 64KB BUFFER LAST IN 16 BYTE BUFFER 4KB BEYOND 4KB BUFFER NULL 4K BUFFER FREE()'d AFTER MALLOC()ing -1
LENGTH	8	1 16 VIRTUAL MEM. PAGE SIZE 4 * VM PAGE SIZE 16 * VM PAGE SIZE 16 * VM PAGE SIZE + 1 0 -1
FILE PERMIT MODE	5	WRITE ONLY READ ONLY READ WRITE READ WRITE EXECUTE EXECUTE
TIMEOUT VALUE	3	0 VALID TIMEOUT -1
FILE NAME	9	VALID, NON-EXISTENT INVALID — WHITE SPACE INVALID — LEADING SPACE VALID, CLOSED FILE VALID, OPEN FOR READ VALID, OPEN FOR R/W VALID SYNTAX FOR NON-EXISTENT DIR. INVALID, NON-ASCII CHAR NAME > 255 CHARACTERS
FILE ACCESS FLAG	11	O_RDONLY O_RDONLY|O_CREAT O_WRONLY O_WRONLY|O_CREAT O_RDWR O_RDWR|O_CREAT O_RDWR | O_CREAT | O_TRUNC O_RDONLY | O_WRONLY O_TRUNC O_CREAT O_EXCL
R/W/E FILE DESCRPT	3	NULL VALID DESCRIPTOR -1

Table 2. Parameters and values used for exhaustive testing.

with 16 bytes allocated; LENGTH=read length of 1 byte}. Testing the combinations of values is automatic. The total of 2059 test cases for all functions takes less than an hour to run, and is dominated by a conservative time-out value for detecting task "hangs" that could be fine-tuned if execution speed were to become an issue.

2.2. Benchmark structure

A benchmark system consists of two computers running three processes (Figure 1). Neither specialized hardware nor modifications to the OS are required for operation. The Test Computer runs the OS being tested, a Benchmark process, and a Starter process. The Monitor Computer runs a Watchdog process in order to oversee operation of the Test Computer.

The Starter task opens two socket communications channels with the Watchdog (one for its own use, and one for the Benchmark task to communicate results directly to the Watchdog), and sends periodic "I'm Alive" health check messages to the Watchdog task. Once the sockets are functioning, the Starter starts the separate Benchmark process. After the Benchmark is running, the Starter's only job is to continue sending health checks, and follow any re-start instructions from the Watchdog. The Starter task is used in order to conveniently re-start the Benchmark task after it crashes. Additionally, loss of communication with the Starter task indirectly informs the Watchdog task of any Test Computer system crash.

The Benchmark task contains a number of tests to be performed on a list of system calls as discussed in the previous section. Each test consists of a selected OS call and a set of parameters passed to that call. Upon completion of each test, the resultant OS error (or success) code is communicated to the Watchdog task for logging. The Benchmark task can be restarted to resume testing anywhere in the test list in order to recover from a task or system crash.

The Watchdog process tracks the status of the processes executing on the Test Computer and logs all test results to

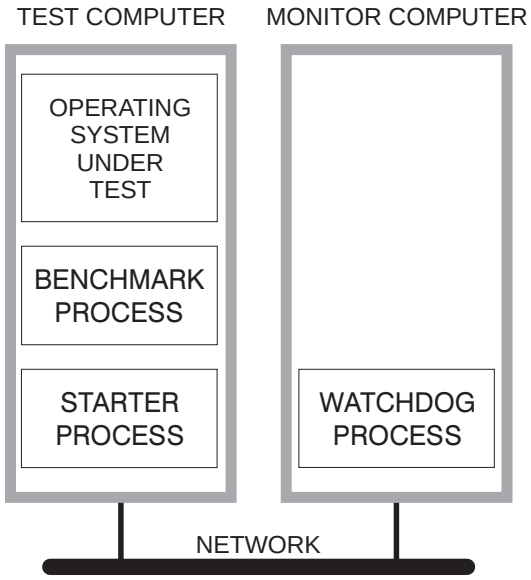

Figure 1. The robustness benchmarks consist of three tasks plus the operating system being tested, run using two computers.

a file. When a benchmark task fails, the Watchdog determines whether the task is active (and thus hung), or if the task aborted. If the Benchmark task has aborted, a message is sent to the Starter to restart it. If the Starter task has also hung or aborted, the Test Computer is declared to have suffered a system crash.

In the function calls tested, the return value is checked by the Benchmark task to determine if it indicated the occurrence of an error condition. If so, the error number variable (*errno*) is queried to gain additional information. For example, a read() call returns the number of bytes read, zero if the end-of-file marker is reached, or -1 if an error occurred. If read() call returns -1, the *errno* variable is queried to provide the error code.

3. A robustness gap severity scale

Applying the robustness benchmarks to several machines resulted in a substantial number of failures during execution. A failure was defined to be an incorrect error/success return code, abnormal termination, or loss of program control (*i.e.*, a "hung" system). Each failure revealed a robustness gap, in that the system failed to respond correctly to a set of inputs.

While one might be tempted to analyze the exact root cause of each failure in terms of a software defect in the OS, it was decided to take a more pragmatic view. In particular, it is uncommon for the user of an off-the-shelf OS to have access to the OS source code. So, identifying the root cause of a gap is of little practical interest for most users (as opposed to OS developers, who presumably have access to debugging tools and detailed implementation information).

Therefore the robustness benchmark results are reported in terms of the *effect of the robustness gaps on the user* rather than the actual OS design defects contributing to any particular robustness gap(s). It is possible that this approach may emphasize the effects of a single design defect that is widely manifested, such as a single failure to handle a NULL pointer that is encountered in a large number of test cases. However, it can be argued that the scope of the effect of any defect is what is important to a user in any given software release, not how simple it is for the vendor to fix it.

3.1. *CRASH* scale categories

Once a user-centric view is taken, it is possible to group robustness test failures according to the severity of the effect on an end-use system. To this end, the results presented here are given according to the 5-point ***CRASH*** scale:

C - Catastrophic (OS crashes/multiple tasks affected)

R - Restart (task/process hangs, requiring restart)

A - Abort (task/process aborts, *e.g.*, segmentation violation)

S - Silent (no error code returned when one should be)

H - Hindering (incorrect error code returned)

The **Catastrophic** class of failure occurs when a failure is not contained within a single task. In other words, this level of failure means that a call to an OS function has caused other tasks, or even the system itself, to crash or hang. A Catastrophic failure typically requires a hardware reset of the entire system, but may possibly be limited to a warm restart of the OS. This class of failure is detected by the robustness benchmarks when both the Benchmark and Starter tasks become unresponsive to messages from the Watchdog task.

The **Restart** class of failure occurs when a single task hangs, resulting in the need to kill and restart that task to return to normal execution. This class of failure is detected by the robustness benchmarks when the Benchmark task fails to respond to messages after a timeout interval, but is still listed by the OS as an active task.

The **Abort** class of failure occurs when a single task experiences an abnormal termination. A typical abnormal termination is caused by a segmentation violation, in which the task attempts to access memory to which it does not have access permissions (for example, by dereferencing a null pointer). Note that if an error takes place during an OS call and the exception is handled appropriately by the OS before control returns to user code, it is not user-visible, and thus is not considered an Abort failure. An Abort failure only takes place if the user's task is aborted rather than having an error code returned to it, which would greatly complicate

having the user code react appropriately to the error condition. This class of failure is detected by the robustness benchmarks when the Benchmark task aborts.

The **Silent** class of failure occurs when invalid parameters are submitted to an OS call, but neither an error return code nor other task failure is generated. For example, a call to open a file with a NULL filename might return a success flag, instead of an error flag. This type of failure is detected by locating successes in the testing log when error return codes should have been generated.

The **Hindering** class is so named because the OS is hindering correct diagnosis of a problem by providing an incorrect error code. For example, an invalid memory access code returned when the only erroneous input is an invalid file handle value would be a hindering-class failure. A Hindering-class failure could potentially cause incorrect recovery action to be taken by the program, confounding efforts to provide dependable service. In the case of multiple invalid parameters, it is acceptable for any one of a number of error codes to be returned; the Hindering failure class is reserved for instances in which the error code lies entirely outside the set of reasonable values. This type of failure is detected by locating an incorrect return code in the testing log.

3.2. Interpretation of severity

The actual severity of each class on the 5-point scale depends on the application area. In a typical workstation environment, the C-R-A-S-H ordering represents decreasing order of severity in terms of operational impact (*i.e.*, Catastrophic is the most severe, and Hindering the least severe).

However, relative severity could vary depending on one's point of view. For example, software developers might perceive Silent failures as a particular concern because they indicate that an opportunity for software bug detection has been missed. A missed software bug may lead to an indirect error manifestation that is more difficult to track down, or may leave a latent bug to crop up after the software is fielded.

For fault-tolerant systems, the Restart, Silent, and some Catastrophic failures may be of the most concern. This is because fault-tolerant systems are typically constructed with a fail-fast assumption. But, a Restart failure or hanging types of Catastrophic failure would create an undetected failure that persists for a period of time, rather than a fast, overt system failure. Silent failures imply that software is operating incorrectly with no indication of failure whatsoever. Therefore, it is possible that a fault-tolerant OS or system designer would be most interested in correcting these types of failures.

The Silent class of failure presents an interpretation problem. In an ideally robust system, access to even one byte past the end of a data structure would generate a hardware exception, which would be converted by the OS into an error code. However, the reality of modern hardware is that access protection is typically provided on virtual memory page boundaries, and generally does not detect overruns beyond data structure boundaries if the resultant addresses are still within the valid data address space. Even though such an access is an erroroneous condition, it would be unreasonable to claim that failure to detect it must be caused by a defect in the operating system. Thus, only accesses to addresses 0 and -1 are considered when counting Silent failures. Similarly, in cases such as write() with a zero length parameter, invalid input parameters may be legitimately ignored if tested before other parameter values; these cases were left out when counting Silent failures.

4. Experimental data

Table 3 summarizes the robustness benchmark data collected for the Mach [5], HP-UX, QNX [6], LynxOS, and the FTX OS used on Stratus high-availability computers. The open() function was not testable under QNX due to lack of compiler support for a particular variable argument C function call form that was required by the test methodology used.

4.1. Results

Of the five OSs tested (Table 3), all except FTX had restart failures. While none of the data presented include Catastrophic failures, such failures have been observed previously. In particular, [3] reports a single Catastrophic failure for a special-purpose computer. Furthermore, 92 instances of Catastrophic failures were observed in a version of the Mach OS previous to the version reported here [4]. (The data presented in Table 3 are for the most recent full-production OS versions available on the test computers at the time of testing).

Two particular results illustrate the types of problems that this testing brings to light. In the case of QNX, the large number of abort failures is indicative of a product testing approach that had not previously placed emphasis on handling exceptional conditions, but rather looked upon such problems as bugs in the application code [7] (which is true from a certain point of view). In the case of FTX a careful emphasis on testing both normal and exceptional conditions seems to have resulted in substantially more robust software. However, the presence of Silent failures indicates that even the best current approaches to improving software robustness do not guarantee perfect results (in fact,

if Silent errors are of primary concern, then QNX performs better than FTX.)

4.2. On comparing OS robustness

It is tempting, and traditional, to propose some sort of weighted sum scheme in order to create a single benchmark number. However, this is not done because different consumers of the data may have significantly different needs as discussed above. Instead, results are reported as a 6-element vector as in Table 3.

The rationale for reporting the vector according to the CRASH categories is that it is hoped, over time, OS vendors will improve their products so that they have zero defects at the higher levels of the scale. Thus, as time goes on there may be a comparison at the highest level of non-zero failures. For example, with current technology it seems useful to compare the number of Restart failures as an important indicator of comparative robustness. In time, one can hope that most OS vendors will eliminate Restart failures. Thus, the next generation of comparisons might be done at the level of reducing Abort failures.

There is certainly room for controversy in comparing the number of observed failures rather than root cause bugs according to benchmark results. For example, it may well be that given the combinatorial nature of the testing performed, a single software bug accounts for a large number of failures. However, there are three responses to this that seem compelling to users of COTS software:

- These results take the point of view of a user who wants to evaluate the robustness of an OS before selecting it for use, not an OS developer. It is of little relevance to a user why the operating system is "broken." However, it is (arguably) of much more importance to know how many opportunities there are to encounter an OS robustness gap. Thus, robustness gaps are counted in terms of opportunities to elicit failure, not number of root causes of failure.
- Even if OS source code is available, it may be difficult to find root causes (the next section presents response regions as a way to potentially gain insight, but this is far from a definitive technique). Thus, it is impracticable to grade results according to root causes without significant vendor cooperation, and certainly cannot be automated for wide distribution.
- If a single OS defect is responsible for a large number of robustness gaps, then the next release of the OS should (if the defect is fixed) have a markedly better robustness rating in return for minimal effort on the part of the vendor. Thus, this approach does not permanently penalize a vendor for having a defect that happens to generate a large number of problems.

	Call	Catastrophic	Restart	Abort	Silent	Hindering	None
Mach 2.6	read	0	0	0	11	0	493
	write	0	0	0	2	0	502
	open	0	0	0	0	0	495
	close	0	0	0	0	2	5
	stat	0	0	0	0	0	81
	fstat	0	0	0	0	0	63
	select	0	15	0	52	0	338
	TOTAL:	0	15	0	65	2	1977
HP-UX 10.0	read	0	0	0	18	0	486
	write	0	0	0	0	0	504
	open	0	0	0	0	0	495
	close	0	0	0	0	0	7
	stat	0	0	0	0	0	81
	fstat	0	0	0	0	0	63
	select	0	30	0	40	0	335
	TOTAL:	0	30	0	58	0	1971
QNX 4.22	read	0	0	0	0	0	504
	write	0	9	0	45	0	450
	open	*	*	*	*	*	*
	close	0	0	0	0	0	7
	stat	0	0	6	0	0	75
	fstat	0	0	6	0	0	57
	select	0	26	157	40	162	20
	TOTAL:	0	35	169	85	162	1113
LynxOS 2.4	read	0	0	0	0	0	504
	write	0	0	0	0	0	504
	open	0	0	0	50	5	440
	close	0	0	0	0	0	7
	stat	0	0	0	14	0	67
	fstat	0	0	0	0	0	63
	select	0	51	0	47	184	123
	TOTAL:	0	51	0	111	189	1708
FTX	read	0	0	0	6	0	498
	write	0	0	0	1	0	503
	open	0	0	0	0	0	495
	close	0	0	0	0	0	7
	stat	0	0	0	0	0	81
	fstat	0	0	0	0	0	63
	select	0	0	0	271	0	134
	TOTAL:	0	0	0	278	0	1781

Table 3. Robustness Benchmark experiment results: Mach 2.6 on DEC 5000/100; HP-UX 10.10 on HP 9000/series 750; QNX 4.22 on Pentium Pro; LynxOS 2.4 on Pentium; FTX on Stratus.

It seems reasonable, within the limits of the number of functions tested and the test cases performed, to compare operating systems in terms of the total number of robustness gaps observed. At a higher level there is a binary comparison made (*e.g.*, an operating system that has no Catastrophic robustness gaps is considered more robust than one that has one or more Catastrophic robustness gaps). At a more detailed level, a large difference in the number of robustness gaps in a particular class may well indicate that user code could experience a substantially different level of dependability with respect to ideal OS exception handling. It is recommended at this point that only large differences in number of gaps found be considered significant when comparing operating systems

5. Response regions

Whenever there are a number of robustness gaps found with a particular system call, it is reasonable to ask whether there is a pattern to the way the system responds to input stimuli. In many cases a set of contiguous input data values can tend to cause the same fault to produce multiple failures.[8] In the past, the value ranges for which failures occur have been represented by error crystals.[9]

The experimental results presented here, along with the ***CRASH*** severity scale, lead to an extension of the error crystal concept to form *response regions*. A response region is a range of input values for which the same class of robustness gap or other failure response prevails.

When a system call is invoked, it produces both a return value and, if unsuccessful, an error code. For the purposes of this discussion, this pair of values is termed the response of the system call. Graphically representing the responses of the

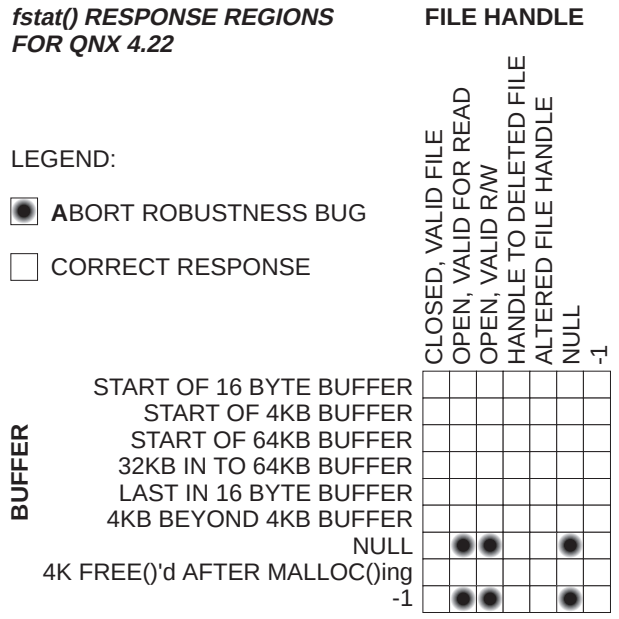

Figure 2. Response regions for fstat() system call on QNX 4.22. Abort robustness gaps occur in three different File Handle cases.

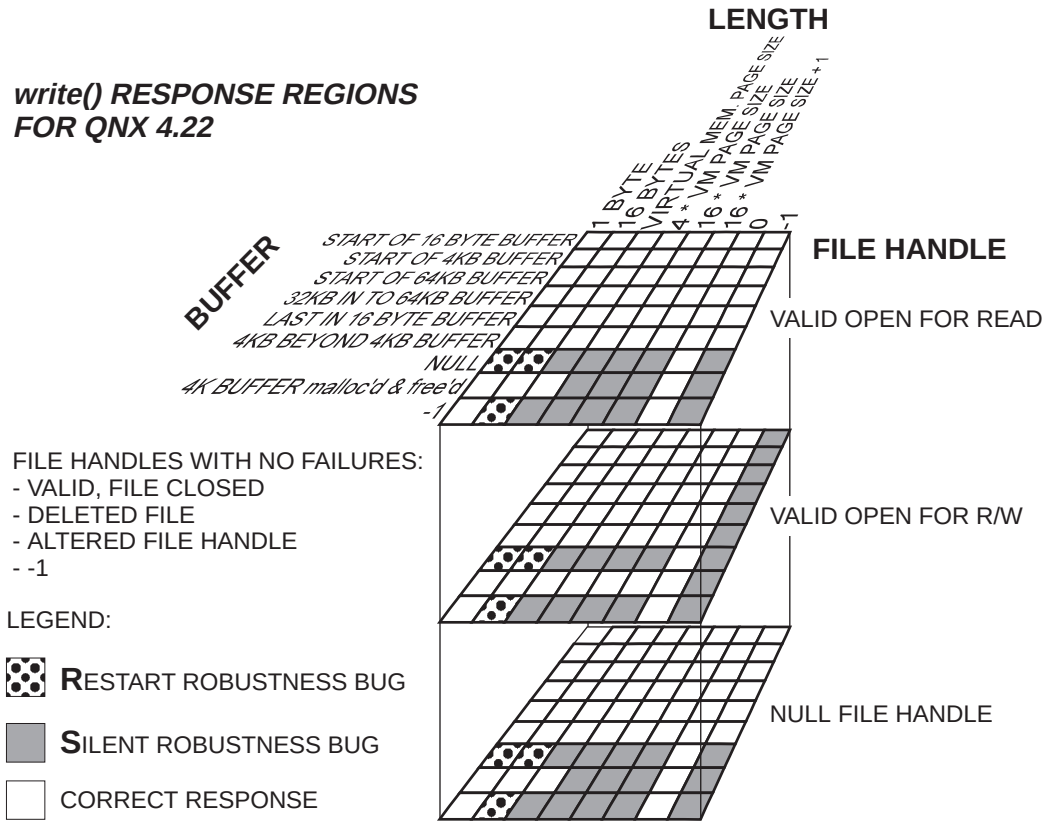

Figure 3. Response regions for write() system call on QNX 4.22. Both Restart and Silent robustness gaps occur in three different File Handle cases.

system to a range of *n* input parameters leads to a map of response regions in *n*-dimensional space. The concept of a response region is an extension of the error crystal concept in that it illustrates not only the presence of a robustness gap, but also the severity of the response for each response region.

Figure 2 shows response regions for the fstat() system call under QNX 4.22. While source code to QNX was not available for inspection, one could speculate that Restart gaps are encountered for NULL and -1 file buffer pointers when accessing a valid and open file (the NULL file handle corresponds to *stdin*). An example that shows multiple response categories on the same response map is the 3-dimensional response for write() under QNX (Figure 3). In Figure 3, some NULL and -1 buffer pointer inputs produce Restart responses, while several other values and value combinations produce Silent responses. Note that in both Figures 2 and 3, a "correct" response includes both any case where the system call returns an appropriate error code, as well as an the return of an appropriate success flag, and does not judge whether the execution of the function call is in any other sense "correct."

There are at least two uses for response regions. Software maintainers may gain insight necessary for correcting OS defects by examining response region maps. For example, response to illegal buffer pointer values in QNX could be investigated based on the information in Figures 2 and 3.

A second use for response regions is to provide data for encapsulating OS service calls to shield user programs from incorrect responses. For example, under QNX the fstat() routine could be called from user code through a wrapper routine that first tests for NULL and -1 values in the buffer pointer, then calls the OS version of fstat(). Such a wrapper could return the correct error code instead of hanging when incorrect buffer pointers are used, thus providing a fix for the user until a future OS release corrects the root problem.

6. Conclusions

A set of portable robustness benchmarks has been developed that exercise several OS calls with combinations of valid and invalid input parameters. Results of experiments on five different operating systems suggest that significant failures in ability to gracefully and correctly handle exceptional conditions are commonplace. This is despite the fact that the operating systems tested are considered to be mature, and some are currently being used in safety- and mission-critical systems.

We are developing a portable robustness testing suite that will empower users of off-the-shelf software to test their OS of choice for vulnerabilities. Additionally, the 5-point ***CRASH*** scale provides the ability to compare different OSs at a high level, and may encourage competition among vendors to improve the robustness of their products. While the results presented here are not comprehensive in coverage of OS functions, they are of sufficient scope to demonstrate significant shortcomings in terms of failure prevention and handling.

It is important to point out that this work does not encompass all possible failure modes of an operating system, such as failures that arise only under particular timing conditions. However, it is a first step toward building repeatable, portable tools for comparing the robustness of offerings from different vendors.

7. Acknowledgments

This research was sponsored by: DARPA contract DABT63-96-C-0064 (Ballista project), ONR contract N00014-96-1-0202, and USDOT under Cooperative Agreement Number DTFH61-94-X-00001 as part of the National Automated Highway System Consortium.

8. References

1. Segall, Z., *et al.*, "FIAT - Fault Injection Based Automated Testing Environment," in *Proc. of the Eighteenth Intl. Symposium on Fault Tolerant Computing*, June 1988, pp. 102-107.

2. Kanawati, G.A., N.A. Kanawati, and J.A. Abraham, "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

3. Dingman, C.P., J. Marshall, D.P. Siewiorek, "Measuring Robustness of a Fault Tolerant Aerospace System," *Proc. Twenty-fifth Intl. Symposium on Fault Tolerant Computing*, June 1995, pp. 522-527.

4. Dingman, C.P., *Portable Robustness Benchmarks*, Ph.D. Thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA. May 1997.

5. Rashid, R., R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A Foundation for Open Systems", *Proc. Second Workshop Workstation Operating Systems*, September 1989.

6. *QNX Operating System: system architecture guide*, QNX Corporation, http://www.qnx.com/docs/qnx/sysarch/index.html, December 2, 1996.

7. Koopman, P.J., personal communications with QNX support staff, December 4-10 1996.

8. Bishop, P.G. "The Variation of Software Survival Time for Different Operational Input Profiles." *23rd International Symposium on Fault-Tolerant Computing*, pp. 98-107, 22-24 June 1993, Toulouse, France.

9. Finelli, G.B. "Results of Software Error-Data Experiments." *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference*. pp. 1-5. American Institute of Aeronautics and Astronautics, Washington, D.C. 7-9 September 1988.