Robustness Testing of the Microsoft Win32 API

Charles P. Shelton
ECE Department & ICES
Carnegie Mellon University
Pittsburgh, PA, USA
cshelton@cmu.edu

Philip Koopman
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Kobey DeVale
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
kdevale@ece.cmu.edu

Abstract

Although Microsoft Windows is being deployed in mission-critical applications, little quantitative data has been published about its robustness. We present the results of executing over two million Ballista-generated exception handling tests across 237 functions and system calls involving six Windows variants, as well as similar tests conducted on the Linux operating system. Windows 95, Windows 98, and Windows CE were found to be vulnerable to complete system crashes caused by very simple C programs for several different functions. No system crashes were observed on Windows NT, Windows 2000, and Linux. Linux was significantly more graceful at handling exceptions from system calls in a program-recoverable manner than Windows NT and Windows 2000, but those Windows variants were more robust than Linux (with glibc) at handling C library exceptions. While the choice of operating systems cannot be made solely on the basis of one set of tests, it is hoped that such results will form a starting point for comparing dependability across heterogeneous platforms.

1. Introduction

Different versions of the Microsoft Windows operating system (OS) are becoming popular for mission- and safety-critical applications. The Windows 95/98 OS family is the dominant OS used in personal computer systems, and Windows NT 4.0 has become increasingly popular in business applications. The United States Navy has adopted Windows NT as the official OS to be incorporated into onboard computer systems [15]. Windows CE and Windows NT Embedded are new alternatives for embedded operating systems. Thus, there is considerable market and economic pressure to adopt Windows systems for critical applications.

Unfortunately, Windows operating systems have acquired a general reputation of being less dependable than Unix-based operating systems. In particular, the infamous "Blue Screen Of Death" that is displayed as a result of Windows system crashes is perceived by many as being far more prevalent than the equivalent kernel panics of Unix operating systems. Additionally, it is a common (although meagerly documented) experience that Windows systems need to be rebooted more often than Unix systems. However, there is little if any quantitative data published on the dependability of Windows, and no objective way to predict whether the impending move to Windows 2000 will actually improve dependability over either Windows 98 or Windows NT.

Beyond the dependability of Windows itself, the comparative dependability of Windows and Unix-based systems such as Linux has become a recurring theme of discussion in the media and Internet forums. While the most that can usually be quantified is mean time between reboots (anecdotally, Unix systems are generally said to operate longer between reboots than Windows NT), issues such as system administration, machine usage, the behavior of application programs, and even the stability of underlying hardware typically make such comparisons problematic. It would be useful to have a comparison of reliability between Windows and Unix systems based on direct, reproducible measurements on a reasonably level playing field.

The success of many critical systems requires dependable operation, and a significant component of system dependability can be a robust operating system. (Robustness is formally defined as the degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions [6].) Of particular concern is the behavior of the system when confronted with exceptional operating conditions and consequent exceptional data values. Because these instances are by definition not within the scope of designed operation, it is crucial that the system as a whole, and the OS in particular, react gracefully to prevent compromising critical operating requirements. (Some systems, such as clustered web servers, can be architected to withstand single-node failures; however there are many critical systems in the embedded computing world and mission-critical systems on desktops which cannot afford such redundancy, and which require highly dependable individual nodes.)

This paper presents a quantitative comparison of the vulnerability of six different versions of the Windows Win32 Application Programming Interface (API) to robustness failures caused by exceptional function or system call parameter values. These results are compared to the results of similarly testing Linux for exception handling robustness.

Exception handling tests are performed using the Ballista robustness testing harness [1] for both Windows and Linux. In order to perform a reasonable Windows-to-Linux comparison, 237 calls were selected for testing from the Win32 API, and matched with 183 calls of comparable functionality from the Linux API. Of these calls, 94 were C library functions that were tested with identical test cases in both APIs, with the balance of calls being system calls. Beyond C library functions, the calls selected for testing were common services used by many application programs such as memory management, file and directory system management, input/output (I/O), and process execution/control. The results are reported in groups rather than as individual functions to provide a reasonable basis for comparison in those areas where the APIs differ in the number and type of calls provided.

2. Background

The Ballista software testing methodology has been described in detail elsewhere [3], [9] and is publicly available as an Internet-based testing service [1] involving a central testing server and a portable testing client that was ported to Windows NT and Windows CE for this research. Thus, only a brief summary of Ballista testing operation will be given.

The Ballista testing methodology is a combination of software testing and fault injection approaches. Specifically selected exceptional values (selected via typical software testing strategies) are used to inject faults into a system via an API. For testing an OS, this involves selecting a set of functions and system calls to test, with each such Module under Test (MuT) being exercised in turn until a desired portion of the API is tested. Parameter test values are distinct values for a parameter of a certain data type that are randomly drawn from pools of predefined tests, with a separate pool defined for each data type being tested. These pools of values contain exceptional as well as non-exceptional cases to avoid successful exception handling on one parameter from masking the potential effects of unsuccessful exception handling on some other parameter value. Each test case (the execution of a single MuT with a single test value selected for each required parameter in the call) is executed as a separate task to minimize the occurrence of cross-test interference. A single Ballista test case involves selecting a set of test values, executing constructors associated with those test values to initialize essential system state, executing a call to the MuT with the selected test values in its parameter list, measuring whether the MuT behaves in a robust manner in that situation, and cleaning up any lingering system state in preparation for the next test (including freeing memory and deleting temporary files).

Ballista testing looks only for non-robust responses from software, and does not test for correct functionality. This, combined with a data type-based testing strategy, rather than a functional testing strategy, results in a highly scalable testing approach in which the effort spent on test development tends to grow sub-linearly with the number of MuTs to be tested. An additional property of Ballista testing results is that in practice they have proven to be highly repeatable. Virtually all test results reproduce the same robustness problems every time a brief single-test program representing a single test case is executed.

Ballista uses the CRASH scale [9] to measure robust or non-robust responses from MuTs. CRASH is an acronym for the different robustness failures that can occur. In **C**atastrophic failures, the most severe robustness failure type, the application causes a complete system crash that requires an OS reboot for recovery. In **R**estart failures, the application enters a state where it "hangs" and will not continue normal operation, requiring an application restart for recovery. **A**bort failures are an abnormal termination of an application task as the result of a signal or thrown exception that is not specific enough to constitute a recoverable error condition unless the task elects (or is forced by default) to terminate and restart. **S**ilent failures occur when a function or call is performed with invalid parameter values, but the system reports that it was completed successfully instead of returning an error indication. Finally, **H**indering failures report an incorrect error indication such as the wrong error reporting code. Ballista can automatically detect Catastrophic, Restart, and Abort failures; Silent failures and Hindering failures currently can be detected in only some situations, and require manual analysis.

Earlier Ballista publications (e.g., [3], [8], [9]) describe the software testing and fault injection heritage of this approach. Ballista can be thought of as using software testing principles to perform fault injection at the API level instead of the source code or object code level. The most closely related current research effort is the work done at Reliable Software Technologies on testing Windows NT [4], [5] in light of Ballista results on Unix systems. That work focuses on a broad coverage of functions for a single OS version with relatively simple testing values. Nonetheless, their results found many Abort-type failures in Windows NT, and a few Catastrophic failures that were caused by very complex execution sequences that could not be isolated for bug-reporting purposes. Other recent related work is the Fuzz project at the University of Wisconsin [12], [13], which has concentrated on Unix systems. There does not appear to be any previously published work that performs testing-oriented dependability comparisons of multiple Windows versions, nor comparisons of Windows to Unix robustness.

3. Implementation

The existing Ballista testing system ran only on Unix systems. Thus, testing Windows required porting the client-side testing harness to Windows as well as creating Windows-specific test values and an inter-API comparison methodology.

3. 1. Porting to Windows Desktop Systems

Porting the Ballista testing client software to the Windows platform faced many difficulties, chief among them the fact that Windows has no simple analog to the fork() system call implemented on POSIX systems (POSIX [7] is the standard for Unix). Thus it is more difficult to spawn a child process for each test case being executed. To overcome this, the Windows version of the Ballista test harness creates a memory-mapped file for each test case, writes data for that particular test case's parameters to this file, and then spawns the testing process. The testing process retrieves the data for the current test case from the memory location created by the calling process, and reports results for the test to that same memory location.

The Win32 API uses a thrown-exception error reporting model in addition to the error return code model (using the POSIX "errno" variable or the Win32 GetLastError() function) used by the POSIX API. While on POSIX systems Abort failures can be detected by simply monitoring the system for the occurrence of signals (most often `SIGSEGV` or `SIGBUS`), in Windows systems there are both legitimate and non-robust occurrences of thrown error reporting conditions. The Win32 API documentation [11], [14] does not provide sufficient information to make a per-function list of permissible and non-permissible thrown exceptions. For Windows testing, the Ballista test harness intercepted all integer and string exception values, and to be more than fair in evaluation, assumed that all such exceptions were valid and recoverable. In normal operation, any unrecoverable exceptions trigger the Windows top-level exception filter and display an "Application Error" message window before terminating the program. We disabled this exception filter and replaced it with code that would record such an unrecoverable exception as an Abort failure. (This technique could in fact be used to improve the robustness of an application program, but only by restarting abnormally terminated tasks. That approach might be sufficiently robust for many users, but is considered to be non-robust at the application level by most of the critical-system designers we have had discussions with.)

There were additional challenges involved in porting the Ballista testing client to a Windows environment, such as obtaining a remote procedure call (RPC) package that was compatible with the Unix-based Ballista testing server's RPC implementation. Most UNIX systems use ONC RPC, but Windows only supports DCE RPC, so a third party ONC RPC Windows client had to be used. Most porting issues were related to differing OS interface architectures, and were not fundamental to the Ballista approach.

Because many Win32 calls have four or more parameters, a very large number of test cases could be generated without exhausting all potential combinations of test values for a single MuT. Therefore, testing was capped at 5000 randomly selected test cases per MuT. 72 Windows MuTs and 34 POSIX MuTs were capped at 5000 tests each (per OS) in this manner. All other MuTs performed exhaustive testing of all combinations with fewer than 5000 tests. In order to fairly compare the desktop Windows variants, the same pseudorandom sampling of test cases was performed in the same order for each system call or C function tested across the different Windows variants. Previous findings have indicated that this random sampling gives accurate results when compared to exhaustive testing of all combinations [9].

The Win32 and POSIX APIs use different data types. However, most of the Windows data types required were minor specializations of fairly generic C data types. In those cases, the same test values used in POSIX were simply used for testing Windows. The only major data type for which new test values had to be created for testing Windows was the HANDLE type. The tests for this type were largely created by inheriting tests from existing types and adding test cases in the same general vein as existing data type tests. Overall, the data values used for testing were selected based on experience with previous Ballista testing and a general background knowledge from the software testing literature [2].

3. 2. Porting to Windows CE

The Ballista client for Windows NT does not work on the Windows CE platform because Windows CE is designed to be an embedded operating system that runs on specialized hardware for consumer electronics and mission-critical systems. These systems have tighter memory constraints than a normal desktop PC. Also, Windows CE programs must be compiled and linked for specific hardware using tools that run on Windows NT, and then downloaded to the Windows CE device.

To overcome this problem, the Ballista client was split into two components: the test generation and reporting functions that run on a Windows NT PC, and the test execution and control functions that run on the target Windows CE platform. For each system call or function tested, the test execution and control portion is compiled on the PC and downloaded to the Windows CE machine via a serial port connection. The test generation component running on the PC initiates each test case by starting the test execution process on the target and passing the parameter list via the command line arguments.

Windows CE provides a remote API that allows Windows NT applications to communicate with the Windows CE target using file I/O and process creation, but does not provide mechanisms for process synchronization or control. Therefore, the test execution component running on the target must create another process that actually runs the test and records the result in the target's file system. The NT process must remain idle and wait for this file to appear on the target to get the results of the current test case and report them. Unfortunately this means tests are several orders of magnitude slower than tests run on the other Windows OS versions, taking five to ten seconds per test case.

Error classification was also a problem on Windows CE. Windows CE does not support the normal C++ `try/catch` exception handling scheme, so we had to use the Win32 structured exception handling constructs, `__try/__except` and `__try/__finally`. We did not use these on the other Windows platforms because the Microsoft documentation [11] recommends using C++ `try/catch` whenever possible, and states that the two exception handling methods are mutually exclusive.

The exceptions that we observed on Windows CE appeared to be analogous to the signals thrown in POSIX systems. For example, the exception `EXCEPTION_ACCESS_VIOLATION` thrown in Windows CE is comparable to a `SIGSEGV` signal thrown in UNIX. Therefore, we classified these exceptions as abort failures. The only exceptions observed were `EXCEPTION_ACCESS_VIOLATION`, `EXCEPTION_DATATYPE_MISALIGNMENT`, and `EXCEPTION_STACK_OVERFLOW`.

3. 3. Comparison methodology

Perhaps the greatest challenge in testing Windows systems and then comparing results to Linux was creating a reasonable comparison methodology. While C library functions are identical on both systems, the system calls have different functionality, different numbers of parameters, and somewhat different data types. However, the Ballista techniques of basing tests on data types and of normalized failure rate reporting were used to create an arguably fair comparison of exception handling test results.

Basing tests on data types rather than MuT functionality permits comparing APIs with similar functionality but different interfaces. Because the data type test definitions are nearly identical for both Windows and Linux, the same general tests in the same general proportions are being run regardless of functionality. Of course there is always the possibility of accidental bias. But, because the tests were originally developed specifically to find problems with POSIX systems by students who had no Windows programming experience, if anything the tests would be biased toward finding problems on the previous testing target of POSIX rather than specifically stressing Windows features.

Normalized failure rate data was used to permit comparison among different interfaces to similar functionality between Windows and Linux. Normalization is performed by computing the robustness failure rate on a per-MuT basis (number of test cases failed divided by number of test cases executed for each individual MuT). Then, the MuTs are grouped into comparable classes by functionality, such as all MuTs that perform memory management. The individual failure rates within each such group are averaged with uniform weights to provide a group failure rate, permitting relative comparisons among groups for all OS implementations. As an example, the I/O Primitives group consists of `{close dup dup2 fcntl fdatasync fsync lseek pipe read write}` for POSIX and `{AttachThreadInput CloseHandle DuplicateHandle FlushFileBuffers GetStdHandle LockFile LockFileEx ReadFile ReadFileEx SetFilePointer SetStdHandle UnlockFile UnlockFileEx WriteFile WriteFileEx}` for Win32. Robustness failure rates for the I/O Primitives group are computed by averaging the 10 individual failure rates for the POSIX calls, and comparing against the averaged result for the 15 individual Win32 call failure rates for some particular Windows implementation. While even this level of comparison obviously is not perfect, it has the virtue of encompassing the same set of higher-level functionality across two different APIs. For the purposes of achieving generic-level functionality comparisons, calls that did not have an obvious grouping counterpart for both POSIX and Windows were discarded.

Table 1. Robustness Failure rates by Module under Test (MuT) for Windows versions and Linux.

	System Calls Tested	System Calls with Catastrophic Failures	System Calls with Calculated Failure Rates	System Percent Restart Failures by Call	System Percent Abort Failures by Call	C Library Functions Tested	C Library Functions with Catastrophic Failures	C Library Functions with Calculated Failure Rates	C Library Percent Restart Failures by Function	C Library Percent Abort Failures by Function
Linux	91	0	91	0.2%	7.1%	94	0	94	0.8%	34.9%
Windows 95	133	7	126	0.1%	11.6%	94	1	93	0.02%	24.7%
Windows 98	143	5	138	0.1%	13.3%	94	2	92	0.0%	24.6%
Windows 98 SE	143	6	137	0.1%	12.9%	94	1	93	0.0%	25.0%
Windows NT	143	0	143	0.3%	23.5%	94	0	94	0.01%	24.6%
Windows 2000	143	0	143	0.4%	22.7%	94	0	94	0.05%	24.1%
Windows CE	71	10	61	0.1%	13.3%	82 (108)	18 (27)	64	0.0%	14.0%

	Total MuTs (Functions + Calls) Tested	Overall MuTs with Catastrophic Failures	Overall MuTs with Calculated Failure Rates	Overall Percent Restart Failures by MuT	Overall Percent Abort Failures by MuT
Linux	183	0	183	0.5%	21.9%
Windows 95	227	8	219	0.08%	17.2%
Windows 98	237	7	230	0.06%	17.8%
Windows 98 SE	237	7	230	0.06%	17.8%
Windows NT	237	0	237	0.20%	23.9%
Windows 2000	237	0	237	0.23%	23.3%
Windows CE	153 (179)	28 (37)	125	0.04%	13.7%

In all, 3,430 distinct test values incorporated into 37 data types were available for testing POSIX, and 1,073 distinct test values incorporated into 43 data types were available for testing Windows. Given the cap of 5000 tests per MuT, a total of over 148,000 tests were run on each implementation of the C library, plus an additional 380,000 tests on each implementation of the Win32 API compared to 210,000 tests on the Linux system calls.

We did not test any functions in the Graphical Device Interface (GDI) or any Windows device driver specific code. Similarly, although we did not detect any obvious resource "leakage" during testing, we did not specifically target that type of failure mode for testing, nor did we test the systems under heavy loading conditions. While these are clearly potential sources of robustness problems, we elected to limit testing to comparable situations between Windows and Linux, and to restrict results to include only highly repeatable situations to lend confidence to the accuracy of the conclusions.

4. Experimental Results

Ballista robustness testing was performed on the following operating systems on comparable Pentium-class computers with at least 64 megabytes of RAM:

- Windows 95 revision B
- Windows 98 with Service Pack 1 installed
- Windows 98 Second Edition (SE)/Service Pack 1
- Windows NT 4.0 Workstation/Service Pack 5
- Windows 2000 Professional Beta 3 (Build 2031)
- Windows CE 2.11 running on a Hewlett Packard Jornada 820 Handheld PC
- RedHat Linux 6.0 (Kernel version 2.2.5)

The Microsoft Visual C++ compiler (version 6.0) was used for all Windows systems, and the GNU C compiler (version 2.91.66) was used for the Linux system. (Technically the results for C library testing are the result of the GNU development team and not Linux developers, but they are so prevalently used as a pair to implement POSIX functionality with the C binding that this seems a reasonable approach.)

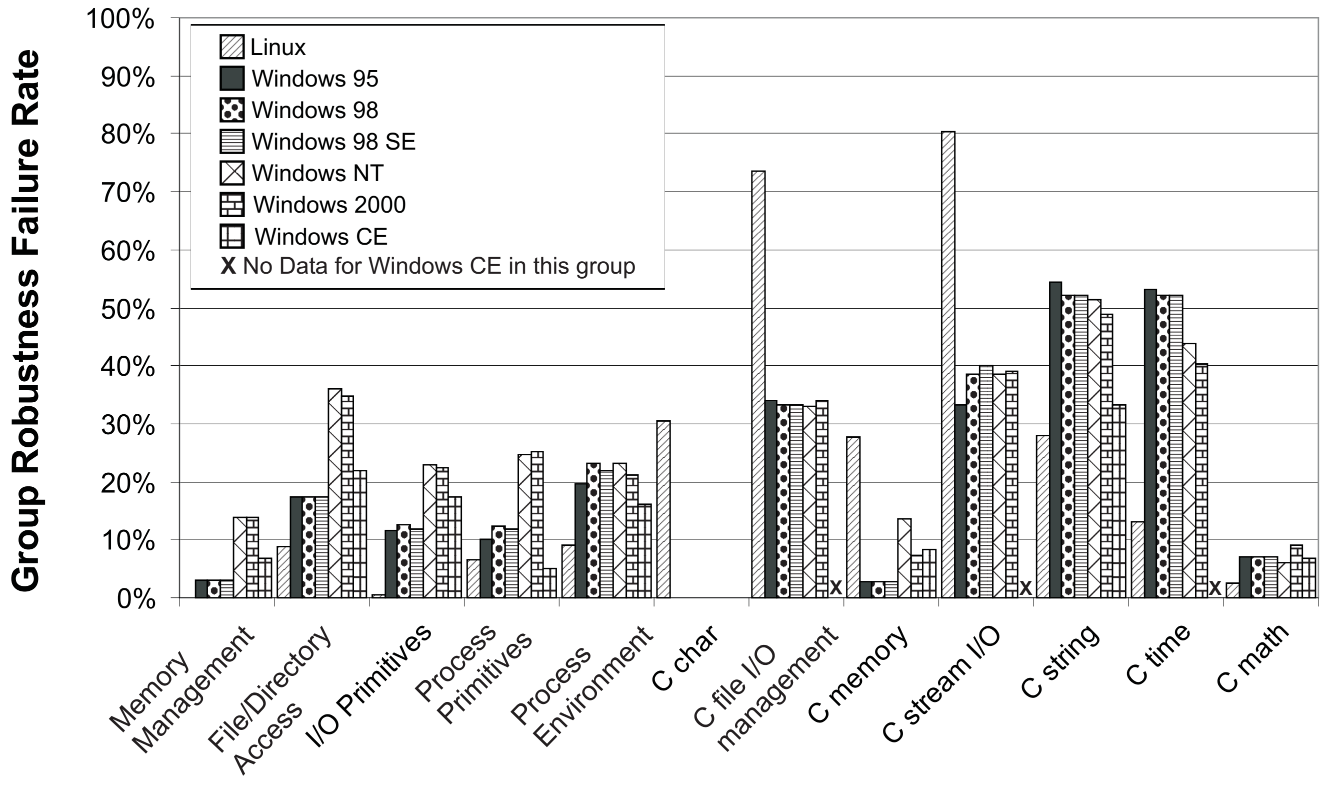

Figure 1. Comparative Windows and Linux robustness failure rates by functional category.

Table 2. Overall robustness failure rates by functional category. Catastrophic failure rates are excluded from numbers, but their presence is indicated by a "*".

	System Calls					C Library						
	Memory Management	File/Directory Access	I/O Primitives	Process Primitives	Process Environment	C char	C file I/O management	C memory	C I/O stream	C string	C time	C math
Linux	0.08%	8.8%	0.4%	6.6%	9.1%	30.4%	73.5%	27.8%	80.4%	28.0%	13.2%	2.6%
Windows 95	*3.0%	*17.3%	*11.6%	*10.2%	*19.6%	0.0%	33.9%	2.7%	*33.2%	54.4%	53.2%	7.1%
Windows 98	3.1%	*17.3%	*12.5%	*12.4%	*23.1%	0.0%	33.3%	2.7%	*38.6%	*52.1%	52.0%	7.1%
Windows 98 SE	3.1%	*17.3%	*11.9%	*11.9%	*22.0%	0.0%	33.3%	2.7%	*39.9%	*52.2%	52.0%	14.8%
Windows NT	13.9%	36.1%	23.0%	24.8%	17.4%	0.0%	32.9%	13.5%	38.7%	51.3%	43.8%	6.2%
Windows 2000	13.9%	34.7%	22.4%	25.3%	13.8%	0.0%	34.0%	7.4%	39.0%	48.9%	40.3%	9.0%
Windows CE	*6.7%	22.0%	17.5%	*5.0%	*16.1%	0.0%	*	8.2%	*	*33.4%	N/A	6.9%

In all, 91 POSIX system calls, 143 Win32 system calls, and 94 C library functions were tested on all desktop operating systems. (10 Win32 system calls were not supported by Windows 95, but were tested on the other desktop Windows platforms.) Because it implements a subset of the Win32 API, only 71 Win32 system calls and 82 C library functions were tested on Windows CE. Table 1 shows the results of robustness testing. The percentages of failures are uniformly weighted averages across all functions tested for that OS. Functions with Catastrophic failures are excluded because the system crash interrupts the testing process, and the set of test cases run for that function is incomplete.

Windows CE gives preferred support to the UNICODE 16-bit character set as opposed to the ASCII 8-bit character set that is used on both UNIX and other Windows platforms. There were 26 C functions that had both an ASCII and a UNICODE implementation. The failure rates for both versions were comparable with the exception of strncpy, which had a Catastrophic failure in the UNICODE version but not in the ASCII version. Since Windows CE uses the UNICODE character set as a default, we only report the failure rates for the UNICODE versions of these C functions. The numbers in parentheses in the Windows CE rows in Table 1 represent the number of functions tested when counting both ASCII and UNICODE functions separately.

In order to compare Windows results to Linux results, the different calls and functions were divided into twelve groupings as shown in Table 2 and Figure 1. These groupings not only serve to permit comparing failure rates across different APIs, but also give a summary of failures for different types of functions. Each failure rate is a uniformly weighted average across all functions tested for that particular OS; the total failure rates give each group's failure rate an even weighting to compensate for the effects caused by different APIs having different numbers of functions to implement each function category. Again, functions with Catastrophic failures are excluded from this calculation. Functions tested on Windows CE in the C file I/O management and the C stream I/O groups had too many functions with Catastrophic failures to report accurate group failure rates; 6 out of 10 in the former and 11 out of 14 in the latter. Windows CE does not support functions in the C time group, so no results for that group are reported.

Table 2 and Figure 1 show that there are significant differences in the robustness failure rates of Linux and Windows, as well as between the Windows 95/98 family and the Windows NT/2000 family of operating systems. Windows CE was unlike either family of desktop Windows variants. (It should be noted that the dominant source of robustness failures is Abort failures, so these results should be interpreted in light of the degree to which those failures affect any particular application.)

Listing 1. A line of code that produces Catastrophic failures on Windows 95, Windows 98 and Windows CE

```
GetThreadContext(GetCurrentThread(),
                              NULL);
```

Windows 95, Windows 98, and Windows 98 SE exhibited similar failure rates, including a number of functions that caused repeatable Catastrophic system crash failures. Five of the Win32 API system calls: DuplicateHandle(), GetFileInformationByHandle(), GetThreadContext(), MsgWaitForMultipleObjects(), and MsgWaitForMultipleObjectsEx(), plus two C library functions, fwrite() and strncpy(), caused Catastrophic failures for certain test cases in Windows 98. Listing 1 shows a representative test case that has crashed Windows 98 every time it has been run on two different desktop machines, a Windows 95 machine, a Windows 98 laptop computer, and our Windows CE device.

Windows 98 SE had Catastrophic failures in the same five Win32 API system calls as Windows 98, plus another in the CreateThread() call, but eliminated the Catastrophic failure in the C library function fwrite(). Windows 95 had all the Catastrophic failures of Windows 98 except for MsgWaitForMultipleObjectsEx(), which was not implemented in Windows 95. Windows 95 also did not exhibit Catastrophic failures in the C library function strncpy(). Windows 95 did, however, have three additional calls with Catastrophic failures: FileTimeToSystemTime(), HeapCreate(), and ReadProcessMemory().

Windows CE had abort failure rates that did not correspond to either the Windows 95/98 family, or the Windows NT/2000 family, and had significantly more functions with Catastrophic failures than any other OS tested, especially in the C library functions. Windows CE had Catastrophic failures in ten Win32 system calls: CreateThread(), GetThreadContext(), InterlockedDecrement(), InterlockedExchange(), InterlockedIncrement(), MsgWaitForMultipleObjects(), MsgWaitForMultipleObjectsEx(), ReadProcessMemory(), SetThreadContext(), and VirtualAlloc(). Windows CE also had 18 C library functions with Catastrophic failures (27 counting ASCII and UNICODE functions separately), 17 of which failed due to the same invalid C file pointer as a parameter.

For several of the functions with Catastrophic failures we could not isolate the system crash to a single test case. We could repeatedly crash the system by running the entire test harness for these functions, but could not reproduce it when running the test cases independently. These system crashes were probably due to inter-test interference, which indicates that system state was not properly cleaned be-

Table 3. Functions that exhibited Catastrophic failures by OS and function group. A "*" indicates that the failure could not be reproduced outside of the test harness.

	Windows 95	Windows 98	Windows 98 SE	Windows CE
Memory Management				
HeapCreate	X			
VirtualAlloc				X
File/Directory Access				
FileTimeToSystemTime	X			
GetFileInformationByHandle	X	X	X	
Process Primitives				
MsgWaitForMultipleObjects	X	X	X	X
*MsgWaitForMultipleObjectsEx		X	X	X
*ReadProcessMemory	X			X
*CreateThread			X	X
Process Environment				
GetThreadContext	X	X	X	X
*InterlockedDecrement, *InterlockedExchange				X
*InterlockedIncrement, SetThreadContext				X

	Windows 95	Windows 98	Windows 98 SE	Windows CE
I/O Primitives				
*DuplicateHandle	X	X	X	
C file I/O management				
clearerr, fclose, fflush				X
_wfreopen, fseek, ftell				X
C I/O stream				
*fwrite	X	X		X
*fread				X
(UNICODE and ASCII) fgetc, *fgets, fprintf,				X
(UNICODE and ASCII) fputc, fputs, fscanf,				X
(UNICODE and ASCII) getc, putc, ungetc				X
C string				
*strncpy		X	X	
(UNICODE) *_tcsncpy				X

tween test cases, even though each test is run in a separate process to minimize this effect. All system calls and functions with Catastrophic failures across all OS's are listed in Table 3 by function group.

Windows NT, Windows 2000, and Linux exhibited no Catastrophic failures during this testing. This is certainly not to say that they cannot be made to crash, but rather that they have reached a different plateau of overall robustness - it is, at a minimum, difficult to find a simple C program that crashes them when run as a single task in user mode. Thus, one can consider that there is some merit to Microsoft's claim that Windows NT is more reliable than Windows 98 (as, for example, stated on their Web site [10]).

Restart failures were relatively rare for all the OS implementations tested. However, they might be a critical problem for any system that assumes fail-fast semantics, including clustered servers that otherwise do not require ultra-high dependability hardware nodes. In, general Restart failures were too infrequent for comparisons to be meaningful.

The classification of Aborts as failures is controversial. In systems in which task termination is acceptable (systems requiring fail-fast operation that can withstand the latency of task restarts), or desirable (debugging scenarios), they may not be considered a problem. However, in some critical and embedded systems that either do not have time to accomplish a task restart or cannot withstand the loss of state information accompanying a task restart, Abort failures can be a significant problem. Our experience in talking with companies that require high levels of field reliability is that Aborts are indeed considered failures for those applications. For other applications that do not share the same philosophy, Abort numbers may not have significant meaning

Given that Abort failures are relevant for some applications, Figure 1 shows that there are striking differences in Abort failure rates across operating systems and functional groupings. For example, Linux has more than a 30% Abort failure rate for C character operations, whereas all the Windows systems have zero percent failure rates (this difference is presumably because Windows does boundary checking on character table-lookup operations). Linux also has higher failure rates on C file I/O management, C stream I/O, and C memory operations. For other groupings Linux has a much lower Abort failure rate. It is interesting to note that the similar code bases for the Windows 95/98 pairing and the Windows NT/2000 pairing show up in relatively similar Abort failure rates. Windows CE generally has lower abort failure rates than Windows NT and Windows 2000, but the significant number of functions that can cause complete system crashes indicates that despite this, Windows CE is less stable than Windows NT/2000.

The issue of Silent failures is a potentially thorny one. Silent failures cannot be measured directly by Ballista be-

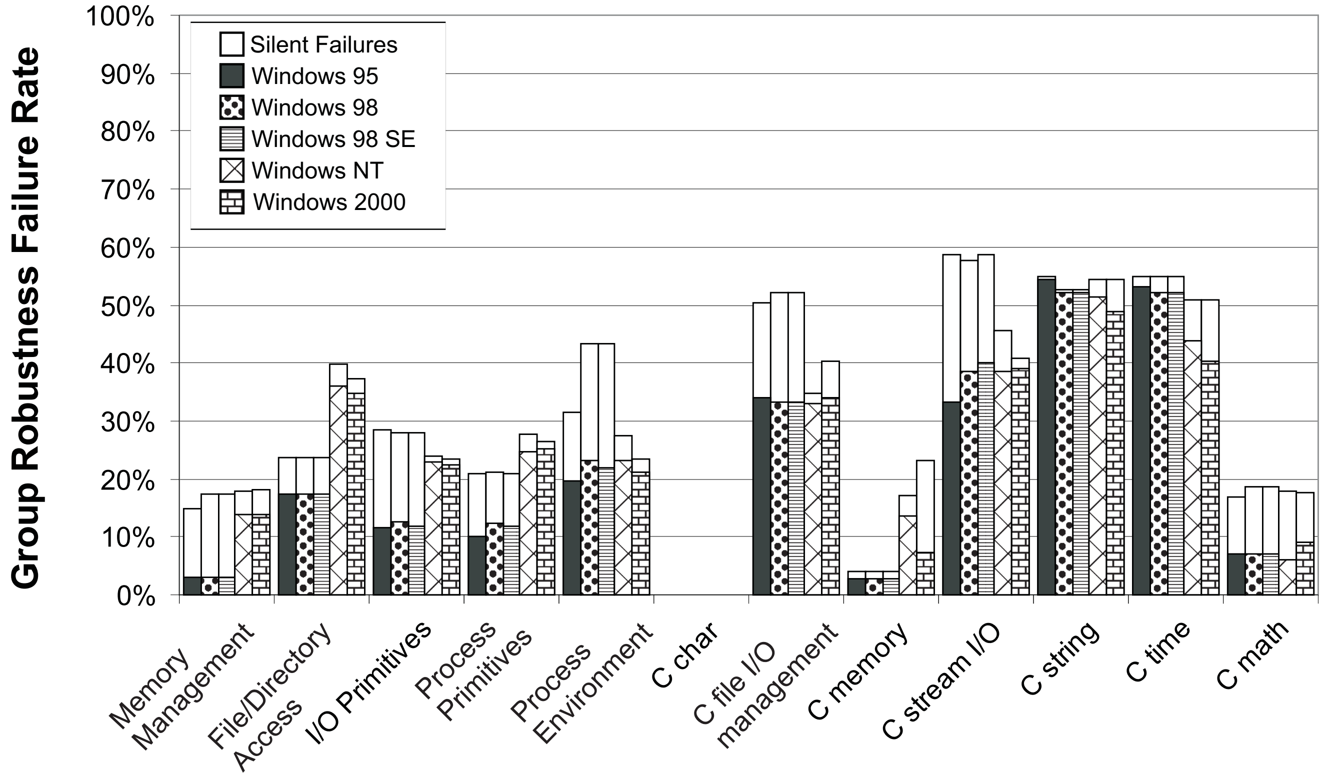

Figure 2. Abort, Restart, and estimated Silent failure rates for Windows desktop operating systems.

cause they involve situations in which there is no observable indication of a failure. (Note: this is not to say they are non-observable in the usual sense of non-activated injected faults that do not affect results. The problem is that there is an exceptional condition that ought to generate observable results to attain robust operation, but does not. As an example, a Silent failure might be a call that reads data from a non-existent file, but returns seemingly valid data bytes with no error indication.)

It is impractical to annotate millions of tests to identify Silent failures. However, we can estimate silent failure rates by voting results across different versions of the same API. Based on previous experience with POSIX [8], we would expect there to be approximately a 10% pass-with-non-exceptional test rate (but, this is a very gross approximation), with the rest of the test cases with a pass with no error reported being Silent failures. If one presumes that the Win32 API is supposed to be identical in exception handling as well as functionality across implementations, if one system reports a pass with no error reported for one particular test case and another system reports a pass with an error or a failure for that identical test case, then we can declare the system that reported no error as having a Silent failure. We wrote a script to automatically vote across identical test cases for each system to generate estimated Silent failure rates. (Note: this analysis does not apply to Linux because it is not an identical API.) Windows CE is not included in this analysis because although the API is similar, it is not identical. Some parameters are not used in Windows CE, and over half of the functions tested on the other Win32 platforms were not supported. Therefore, silent failure rates cannot be reported accurately for Windows CE.

Based on the estimated Silent failures, it seems that the Win32 calls for Windows 95/98/98 SE have a significantly higher Silent failure rate than Windows NT/2000. C library functions vary, with Windows 95/98/98 SE having both higher and lower Silent failure rates than Windows NT/2000 depending on the functional category. Figure 2 shows the overall robustness failure rates for the different Windows variants tested, including these estimated Silent failure rates. Based on these results, it appears that Windows NT and Windows 2000 suffer fewer robustness failures overall than Windows 95/98/98 SE. The only significant exceptions are for the File and Directory Access category as well as the C memory management category, which both suffer from higher Abort failure rates on Windows NT and Windows 2000. (A possible limitation of this approach is that it cannot find instances in which all versions of Windows suffer a Silent failure. This hidden Silent failure rate may be significant, but quantification is not practical.)

5. Conclusions and Future Work

This work demonstrates that it is possible to compare the robustness of different OS APIs on a relatively level playing field. The use of data type-based testing techniques and normalization of test results by functional groupings enables a detailed comparison of APIs having generally similar capabilities but different interfaces.

Applying the Ballista testing methodology to several Microsoft Windows operating systems revealed a variety of robustness failures. The Windows CE OS and the Windows 95/98/98 SE family of operating systems were clearly vulnerable to robustness failures induced by exceptional parameter values, and could be crashed via a variety of functions. Additionally, the Windows 95/98/98 SE systems had a significant level of Silent failure rates in which exceptional operating situations produced neither abnormal termination nor any other indication of an exception when such an indication was demonstrated possible by other Windows variants. Windows NT and Windows 2000 proved as resistant to system crashes as Linux under these testing conditions, and in most cases had fewer Silent failures than the Windows 95/98/98 SE family (although only a relative comparison was possible; the absolute level of Silent failures is more difficult to determine).

An examination of Abort failures (exceptions too non-specific to be recoverable) and Restart failures (task "hangs") showed differences among the Windows variants and between Windows and Linux. Linux had a significantly lower Abort failure rate in eight out of twelve functional groupings, but was significantly higher in the remaining four. The four groupings for which Linux Abort failures are higher are entirely within the C library, for which the POSIX and Win32 APIs are identical.

Windows CE has abort failure rates comparable to Windows NT and Windows 2000, but has several functions that cause complete system crashes. This makes Windows CE a less attractive alternative for embedded systems, where dependability and reliability are of much higher importance than in desktop PC applications. While abort failures may be recoverable by task restarts, a complete OS crash will more than likely cause complete system failure. It should be noted that many of the catastrophic failures found in Windows CE were traceable to incorrect handling of a single bad parameter value, namely an invalid C file pointer (the actual parameter was a string buffer typecast to a file pointer). It could be argued that since we can trace problem to one underlying cause that we should not penalize Windows CE for seventeen functions that happen to take the same parameter. However, developers who wish to use Windows CE in their systems would have to generate software wrappers for each of the seventeen functions they use to protect against a system crash because they only have access to the interface, not the underlying implementation.

It is also interesting to note that several of the Win32 system calls that crashed on Windows CE also crashed on Windows 95/98/98 SE (some with the exact same parameter values, as in Listing 1), despite the fact that they were developed by different teams within Microsoft and have different code bases. One can speculate that this indicates the underlying causes of these errors may be in the specification rather than the implementation; however the problem may simply be that different programmers tend to make the same sorts of mistakes in similar situations.

While it is not appropriate to make sweeping claims about the dependability of Windows or Linux from these test results alone, a few observations seem warranted by the data presented. The marked difference in finding catastrophic failures in Windows CE and the Windows 95/98/98 SE family compared to the other OS families lends credibility to Microsoft's statement that the Windows NT/2000 systems are more reliable overall. A relative assessment of Linux vs. Windows NT reliability is less clear-cut. Linux seems more robust on system calls, but more susceptible to Abort failures on C library calls (which are actually part of the GNU C compiler suite for Linux) compared to Windows NT.

Future work on Windows testing will include looking for dependability problems caused by heavy load conditions, as well as state- and sequence-dependent failures. In particular, we will attempt to find ways to reproduce the elusive crashes that we have observed to occur in both Windows and Linux outside of the current robustness testing framework.

6. Acknowledgements

This work was supported by Emerson Electric, Lucent Technologies, Asea Brown Boveri (ABB), and Darpa (contract DABT63-96-C-0064). Equipment support was provided by Intel, Compaq and Microsoft. Thanks to Jiantao Pan and Meredith Beveridge for their assistance and to Dan Siewiorek for his continuing guidance.

7. References

[1] Ballista Robustness Testing Service, *http://www.ices.cmu.edu /ballista/index.html*, November 1999.

[2] Beizer, Boris, *Black Box Testing: Techniques for Functional Testing of Software Systems.* John Wiley & Sons, Inc., New York, 1995.

[3] DeVale, J., Koopman, P., Guttendorf, D., "The Ballista Software Robustness Testing Service," *Testing Computer Software Conference*, 1999.

[4] Ghosh, A., Schmid, M., Hill, F., "Wrapping Windows NT Software For Robustness," *29th Fault Tolerant Computing Symposium*, June15-18, 1999.

[5] Ghosh, A., Schmid, M., "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors," *10th International Symposium on Software Reliability Engineering*, November 1-4, 1999.

[6] *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std610.12-1990)*, IEEE Computer Soc., Dec. 10, 1990.

[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, 1994.

[8] Koopman, P., DeVale, J., "Comparing the Robustness of POSIX Operating Systems," *29th Fault Tolerant Computing Symposium*, June 15-18, 1999.

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

[10] Microsoft Corp., *Choosing the Best Windows Desktop Platform For Large and Medium-Sized Businesses and Organizations*, white paper, June 1998, http://www.microsoft.com/windows/platform/info/how2choose-mb.htm, November 1999.

[11] Microsoft Corp., *Microsoft Platform Software Development Kit Documentation*, 1999.

[12] Miller, B.P., Fredriksen, L. & So, B., "An empirical study of the reliability of Unix utilities," *Communications of the ACM*, 33(12): 32-43, December 1990.

[13] Miller, B.P., D. Koski, C. Pheow Lee, V. Maganty, R. Murthy, A. Natarajan & J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, University of Wisconsin, CS-TR-95-1268, April 1995.

[14] Simon, R., *Windows NT Win32 API SuperBible*. Waite Group Press, Corte Modera, CA, 1997.

[15] Slabodkin, Gregory, "Software glitches leave Navy Smart Ship dead in the water," *Government Computer News*, http://www.gcn.com/archives/gcn/1998/july13/cov2.htm, July 13, 1998.