CMU/ICES Technical Report #01-27-98

Automatic Robustness Testing of Off-the-Shelf Software Components

Nathan P. Kropp

Institute for Complex Engineered Systems

Carnegie Institute of Technology

Carnegie Mellon University

Abstract

Mission-critical system designers are turning towards Commercial Off-The-Shelf (COTS) software to reduce costs and shorten development time even though COTS software components may not specifically be designed for robust operation. (Systems are robust if they can function correctly despite exceptional inputs or stressful conditions.) Automated testing can assess component robustness without sacrificing the cost and time advantages of using COTS software. This report describes a scalable, portable, automated robustness testing tool for component interfaces. An object-oriented approach based on parameter data types rather than component functionality essentially eliminates the need for function-specific test scaffolding. A full-scale implementation that automatically tests the robustness of 233 operating system software components has been ported to nine POSIX systems. Between 42% and 63% of components on the POSIX systems measured had robustness problems, with a normalized failure rate ranging from 10% to 21% of tests conducted. Robustness testing could be used by developers to measure and improve robustness, or by consumers to compare the robustness of competing COTS component libraries.

Acknowledgments

This research was supported under DARPA contract DABT63-96-C-0064 (the Ballista project) and ONR contract N00014-96-1-0202. Thanks to John DeVale and Jiantao Pan for their help in collecting experimental data.

1 Introduction

1.1 Motivation

Use of Commercial Off-The-Shelf (COTS) software components in computing systems is becoming more popular with system designers as an alternative to costly and time consuming custom development. Unfortunately many COTS components do not provide the robustness necessary for safe use in mission-critical systems. For instance, a component originally intended for use in a desktop computing environment may have been developed with robustness as only a secondary goal because of the relatively low cost of a system crash and the ability of operators to work around known problems.

Even components specifically designed for mission-critical applications may prove to have problems with robustness if reused in a different context. For example, a root cause of the loss of Ariane 5 flight 501 was the reuse of Ariane 4 inertial navigation software [1]. The software proved to have robustness problems due to an overflow on a float-to-integer conversion when operating under the different conditions found on Ariane 5. With the current trend to an increased use of COTS components, opportunities for bad data values to be circulated within a system are likely to multiply, increasing the importance of dealing with such exceptional conditions gracefully.

Robustness of COTS software components is therefore an important consideration when building mission-critical systems. To a mission-critical system designer, a measure of robustness might be useful to determine whether a COTS component is appropriate for a given application. This is similar to the use of component performance metrics in designing a time-critical system. The ability to measure component robustness could also be useful to COTS component developers, in order to evaluate and improve the robustness of their products. Such robustness measurement, whether by the producer or the consumer, must be simple, automated, and fast. It was with these goals in mind that the Ballista approach to robustness testing, presented in this report, was created.

1.2 Background

Robustness is defined as “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [2]. Nonrobust behavior of a software component may or may not be the result of buggy software; it is often caused by code that simply neglects to test for invalid inputs to an algorithm. In fact, one could envision a system comprised of a number of COTS components, each bug-free (*i.e.*, operating as intended for every valid input), yet the system being nonrobust if, for example, a correct output of one component is invalid as an input to the next component. In this case the nonrobustness of the system stems not from the individual components but from the interconnection of them. Such a scenario is not uncommon, because systems can be built from various COTS components from different vendors, each with their own idea of invalid inputs and outputs. Hence, even if a component were bug-free and formally correct, its robustness could still be in question.

One approach to robustness testing, therefore, is to measure the response of a software component to invalid inputs. (A software component is any piece of software that can be invoked as a procedure, function, or method taking one or more arguments.) The focus of Ballista is the automatic creation and execution of invalid input robustness tests. Specifically, these tests are designed to detect crashes and hangs caused by invalid inputs to function calls. The Ballista methodology focuses on only the first part of the definition of robustness, concerning “invalid inputs;” system behavior under “stressful environmental conditions” is not considered. Nevertheless, the results presented here indicate that robustness vulnerabilities to invalid inputs are common in at least one class of mature COTS software components.

The Ballista approach has the following benefits:

- Only a description of the component interface in terms of parameters and data types is required. COTS or legacy software may not come with complete function specifications or perhaps not even source code, so these are not required by the Ballista robustness testing approach.
- Creation and execution of individual tests is automated, and the investment in creating test database information is prorated across many modules. In particular, no per-module test scaffolding, script, or other driver program need be written.
- The test results are highly repeatable, and permit isolating individual test cases for use in bug reports or for debugging purposes.

The Ballista approach is intended to be generic. In order to demonstrate feasibility on a full-scale example, automated robustness testing has been performed on several implementations of the POSIX operating system C language Application Programming Interface (API) [3].

1.3 Scope

Ballista robustness testing draws from fault injection methods as well as software testing techniques (Section 2, Prior Work) to develop a high-level, repeatable way to test robustness of COTS software (Section 3, Methodology). This has been applied in a full-scale implementation, testing POSIX operating system calls (Section 4, Implementation), and the results show that Ballista can be effective in identifying nonrobustness (Section 5, Experimental Results). The Ballista methodology in general is highly scalable due to an object-oriented approach (Section 6, Generic Applicability of the Methodology). The novelty of the Ballista approach generates substantial opportunities for further research (Section 7, Future Work). Conclusions are presented last (Section 8, Conclusions).

2 Prior Work

2.1 Use of fault injection concepts

Fault injection is a technique for testing the response of a system to a fault that is artificially induced in order to evaluate its robustness. Faults can be injected into a system via hardware or software. Hardware fault injection usually involves manipulating pins, lines, or chips electrically. Such manipulations affect the whole system and are therefore not useful for isolating the robustness of a particular software component *(Module under Test,* or *MuT)* running on that system.

Injecting faults via software, on the other hand, allows different parts of a system to be targeted. For example, FTAPE [4] injects faults such as parity errors into memory chips via software, which may affect only the owner of that memory location. FIAT [5] injects faults by making changes to the binary image of a MuT, and it measures the ability of both the MuT and the system as a whole to recover from such faults. FAUST [6] performs mutation on the source code of a MuT and is therefore targeted specifically at the MuT.

However, each of these software fault injection techniques has drawbacks for robustness testing of COTS software components. For FTAPE to target fault injections to a MuT, the memory layout of the system must be known (furthermore, specialized hardware is needed), so the technique is not portable. Since binary image changes may have global effects, FIAT is more a measure of overall system robustness than of that of a specific MuT. Code mutation techniques like FAUST modify source code (which may not be available for a COTS component) and are in general more suitable for test set coverage analysis than robustness testing.

Two portable software approaches are targeted specifically at testing the robustness of software components. The University of Wisconsin Fuzz approach [7] tests user programs (*e.g.,* UNIX command-line utilities) with both random and crafted input streams, looking for program crashes and system hangs. Its goal is to quantify and describe the robustness of UNIX systems from the typical interactive user's point of view.

The Carnegie Mellon University robustness benchmarking approach [8] [9] tests individual operating system calls with specific input values, to detect crashes and hangs. Fault injection is performed by passing combinations of acceptable and exceptional inputs as a parameter list to a MuT via a normal function call. Thus fault injection is done through the API.

The work reported herein is a generalization of previous Carnegie Mellon efforts, with a completely new implementation on a full-size application to demonstrate scalability.

2.2 Use of software testing concepts

Software testing for the purpose of determining reliability is often carried out by exercising a software system under representative workload conditions and measuring failure rates. In addition, emphasis is placed on code coverage (*i.e.*, portion of code exercised during testing) as a way of assessing whether a module has been thoroughly tested [10]. Unfortunately, traditional software reliability testing may not uncover robustness problems that occur because of unexpected input values generated by bugs in other modules, or because of an encounter with atypical operating conditions.

Structural, or white-box, testing techniques [10] are useful for attaining a high test coverage of programs. However, they typically focus on the control flow of a program rather than the handling of exceptional data values. For example, structural testing ascertains whether code designed to detect invalid data is executed by a test suite, but may not detect if such code is missing altogether. Additionally, structural testing typically requires access to source code, which is often unavailable when using COTS software components.

An alternative approach is black-box testing, also called behavioral testing [11]. This type of testing ignores the internal operation of the system being tested and instead focuses on whether the system produces the correct response to various input values. This is appropriate for testing COTS software since source code is often unavailable. Finally, black-box testing enables easy comparison of two MuTs with the same interface but different implementations. This allows, for example, competing COTS components which perform the same function to be readily compared in terms of robustness.

Two types of black-box testing are particularly useful for robustness testing: domain testing and syntax testing. Domain testing locates and probes points around extrema and discontinuities in the input domain. Syntax testing constructs character strings that are designed to test the robustness of string lexing and parsing systems. Both types of testing and more are used in Ballista as described in the next section.

3 Methodology

Automatically generating software tests requires four things: a MuT, a machine-understandable specification of correct behavior, a way to generate test cases, and an automatic way to compare the specification with the results of executing the MuT with those test cases.

3.1 Behavioral specification

Unfortunately, obtaining or creating a behavioral specification for a COTS or legacy software component is often impractical due to unavailability or cost. Fortunately, robustness testing need not use a detailed behavioral specification. Instead, the almost trivial specification of “doesn’t crash, doesn’t hang” suffices. Determining whether a MuT meets this specification is straightforward—the operating system can be queried to see if a test program terminates abnormally, and a watchdog timer can be used to detect infinite loops. Thus, robustness testing of any module that is not intentionally designed to crash or hang can be performed in the absence of a behavioral specification.

Any existing specification for a MuT might define inputs as falling into three categories: valid inputs, inputs which are specified to be handled as exceptions, and inputs for which the behavior is unspecified (Figure 1). Ballista testing, because it is not concerned with the specified behavior, collapses the unspecified and specified exceptional inputs into a single invalid input space. The focus is on overall robustness, not on whether written specifications have officially exempted certain cases of nonrobust operation. The robustness of the responses of the MuT can be characterized as robust (neither crashes nor hangs, but is not necessarily correct from a detailed behavioral view), having a reproducible failure (a crash or hang that is consistently reproduced within the Ballista single-call fault assumption), and an unreproducible failure (a robustness failure that is not readily

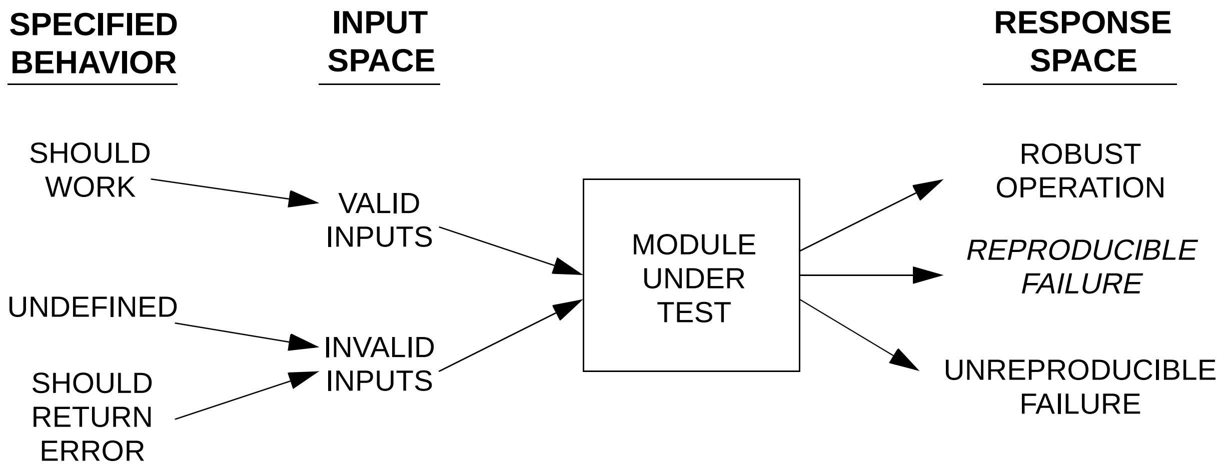


Figure 1. Ballista performs fault injection at the API level using combinations of valid and exceptional inputs.

reproducible, or that requires a sequence of calls to reproduce). The objective of Ballista is to identify reproducible failures.

3.2 Test case generation

In the Ballista approach, robustness testing of a MuT consists of establishing an initial system state, executing a single call to the MuT, determining whether a robustness problem occurred, and then restoring system state to pre-test conditions in preparation for the next test. Although executing sequences of calls to one or more MuTs during a test can be useful in some situations, we have found that even the simple approach of testing a single call at a time provides a rich set of tests, and uncovers a significant number of robustness problems.

A key concept of Ballista is that tests are based on the values of parameters passed to the MuT and not on the behavioral details of the MuT. Ballista uses an objected-oriented approach to define test cases based on the data types of the parameters for the MuT. The set of test cases used to test a MuT is completely determined by the data types of the parameter list of the MuT and in no way depends on the actual behavioral specification.

Figure 2 shows the Ballista approach to generating test cases for a MuT. Before conducting tests, a set of test values must be created for each data type used in the MuT. For example, if one or more modules to be tested require an integer data type as an input parameter, test values must be created for testing integers. Values to test integers might include 0, 1, and INT_MAX (maximum integer value). Additionally, if a pointer data type is used within the MuT, values of NULL and -1, among others, might be created as test cases.

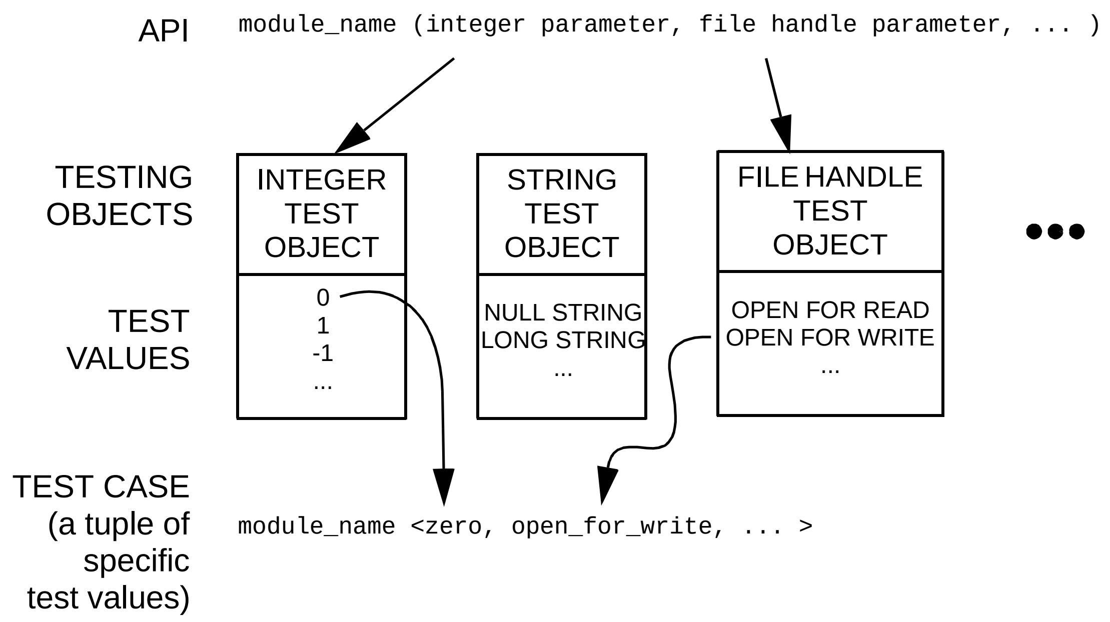


Figure 2. Refinement of a module within an API into a particular test case.

A module cannot be tested until test values are created for each of its parameter data types. Automatic testing generates module test cases by drawing from the pools of defined data-type-specific test values.

3.2.1 Data type test value selection

In choosing values to implement for testing, several criteria should be used. All require some knowledge of the typical uses of the associated data type. Without such knowledge, testing is still possible by choosing values at random, but the coverage is likely to be poor, and therefore the results may not accurately reflect a MuT's robustness.

The criteria for choosing data values rely on the notion of valid and invalid values. A properly constructed value is not in itself valid or invalid; the validity of a value is imposed by the module to which the value is passed. Therefore a given value may be valid when passed to a certain module but invalid as an input to a different module. If possible, at least one valid value should be identified for each intended use of a data type.

Also, different invalid values often elicit different system responses from a given MuT. With some knowledge of the typical uses of a data type, the implementor should attempt to identify different ways in which values can be invalid.

Fault masking is also an important consideration. In modules with more than one input parameter, masking can occur if a module performs error checking for one input but not another. For example, consider the case in which a module checks only its first parameter for validity. (Assume also that the module performs this check before any other operations.) Then issuing the call `module1 (<invalid>, <don't care>)` returns an error code (robust behavior). However, if invoking `module1 (<valid>, <invalid>)` causes abnormal termination (nonrobust behavior), the invalid first parameter in the previous call is said to mask a second-parameter robustness failure. To ensure that testing results adequately reflect a MuT's robustness, the possibility of masking should be kept in mind when choosing data values.

Another important concept is that of boundary values. Boundaries could be places where valid values meet invalid values (in the input space; for example, the integer zero, which is where positives meet negatives) or important system values (virtual memory page size, for example). Boundaries are identified largely from experience.

Thus, there are three criteria that should be followed when choosing data values.

1. Implement at least one valid value. This is to overcome possible fault masking. If a data type has multiple intended uses, ensure that for each intended use, there is at least one value that is valid for that use.

2. Implement at least one of each type of invalid value. This gives a rich set of values to test and is an attempt to span the space of input values.

3. Implement boundary values since errors and exceptions often occur when system boundaries are encountered.

3.2.2 Test value implementation

Each set of test values (one set per data type) is implemented as a testing object having a pair of constructor and destructor functions for each defined test value for the object's data type. Instantiation of a testing object (which requires selecting a test value from the list of available values) executes the appropriate constructor function that builds any required testing infrastructure. For example, an integer test constructor would simply return an integer value. But, a file descriptor test constructor might create a file, place information in it, set appropriate access permissions, then open the file for read or write operations. An example of a test constructor to create a file open for reading is shown in Figure 3.

```
case FD_OPEN_RD:
  create_file (fd_testfilename);
  fd_tempfd = open (fd_testfilename, O_RDONLY);
  *param = fd_tempfd;
  break;
```

Figure 3. Code for an example constructor. `fd_testfilename` is a standard test file name used by all constructors for file descriptors, `*param` is the parameter used in the subsequent call to the MuT, and the variable `fd_tempfd` is used later by the destructor.

When a testing object is discarded, the corresponding destructor for that test case performs appropriate actions to free, remove, or otherwise undo whatever testing infrastructure may remain after the MuT has executed. For example, a destructor for an integer value does nothing. On the other hand, a destructor for a file descriptor might ensure that a file created by the constructor is deleted.

A natural result of defining test cases by objects based on data type instead of by behavior is that large numbers of test cases can be generated for functions that have multiple parameters in their input lists. Combinations of parameter test values are tested exhaustively by nested iteration. For example, testing a three-parameter function is illustrated in the simplified pseudocode shown in Figure 4. The corresponding real code is automatically generated given just a function name and a typed parameter list. In real testing a separate process is spawned for each test to facilitate the measurement of system response.

3.2.3 Per-function test scaffolding

An important benefit of the parameter-based test case generation approach used by Ballista is that no per-function test scaffolding is necessary. In the pseudocode in Figure 4 any test taking the parameter types `(fd, buf, len)` could be tested simply by changing the `read` to some other function name. All test scaffolding creation is both independent of the behavior of the function being tested, and completely encapsulated in the testing objects.

3.3 Robustness measurement

The response of the MuT is measured in terms of the CRASH scale. [9] In this scale the response lies in one of six categories: Catastrophic (the system crashes or hangs), Restart (the test process hangs), Abort (the test process terminates abnormally, i.e. “core dump”),

```
/* test function read (fd, buf, len) */
foreach ( fd_case ) {
  foreach ( buf_case ) {
    foreach ( len_case ) {
      fd_type  fd (fd_case);    /* constructors create instances */
      buf_type buf (buf_case);
      len_type len (len_case);

      puts (“starting test...”);
      read (fd, buf, len);
      puts (“...test completed”);

     ~fd();  ~buf();  ~len();        /* destructors - clean up */
    } } }
```

Figure 4. Example code for executing all tests for the function *read*(). In each iteration constructors create system state, the test is executed, and destructors restore system state to pre-test conditions.

Silent (the test process exits without an error code, when one should have been returned), Hindering (the test process exits with an error code not relevant to any exceptional input parameter value), and Pass (the module exits properly, with a correct error code if appropriate). In order to achieve automated testing in the absence of specification information, Silent and Hindering failures are not differentiated from Passes. Restarts and Aborts are detected by checking the status of the spawned test processes (using *wait*()). Catastrophic failures are detected when the tester is restarted after having been interrupted in the middle of a test cycle.

4 Implementation

The Ballista approach to robustness testing has been implemented for a set of 233 POSIX calls, including realtime extensions for C. Specifically, all system calls defined in the IEEE 1003.1b standard [3] (“POSIX.1b,” or “POSIX with realtime extensions”) were tested except for calls that take no arguments, such as *getpid*(); calls that do not return, such as *exit*(); and calls that send signals, such as *kill*(). POSIX calls were chosen as an example application because they form a reasonably complex set of functionality, and are widely available in a number of mature commercial implementations.

4.1 Test case database

Table 1 shows that only 20 data types were necessary for testing the 233 POSIX calls. The constructor and destructor code for each test value is typically from one to fifteen lines of C code. (See the Appendix for an example of code for the `filename` data type, as well as a sample function specification file and a sample test result file.) Current test values were chosen based on the Ballista programming team's experience with software defects and knowledge of compiler and operating system behavior, using the criteria described in Section 3.2.1. Testing objects fall into the categories of base type objects and specialized objects.

The only base type objects required to test the POSIX functions are integers, floats, and pointers to memory space. Test values for these data types include:

- Integer data type: 0, 1, -1, INT_MIN, INT_MAX, selected powers of two, powers of two minus one, and powers of two plus one.
- Float data type: 0, 1.0, -1.0, ±DBL_MIN, ±DBL_MAX, pi, and *e*
- Pointer data type: NULL, -1 (cast to a pointer), pointer to *free*()'ed memory, pointers to *malloc*()'ed buffers of various powers of two in size including 2^{31} bytes (if that much can be successfully allocated by *malloc*()). Some pointers are placed near the end of

Data Type	Number of Functions Requiring	Number of Test Cases
string	71	9
buffer	63	15
integer	55	16
bit masks	35	4
filename	32	9
file descriptor	27	13
FILE pointer	25	11
float	22	9
process ID	13	9
file mode	10	7
semaphore	7	8
AIO cntrl block	6	20
message queue	6	6
file open flags	6	9
signal set	5	7
simplified int	4	11
pointer to int	3	6
DIR pointer	3	7
timeout	3	4
size	2	9

Table 1. Data types used in POSIX testing. Only 20 data types were necessary for testing 233 POSIX system calls.

allocated memory to test the effects of accessing memory on virtual memory pages just past valid addresses.

Specialized testing objects build upon base type object test values but add in special information to create and initialize data structures or other system state such as files. Some examples include:

- String data type (based on the pointer base type): includes NULL, -1 (cast to a pointer), pointer to an empty string, a string as large as a virtual memory page, a string 64K bytes in length, a string having randomly selected characters, a string with pernicious file open permissions (*e.g.*, “rwb+-x”), and a string with a pernicious *printf*() format (*e.g.*, “%99999d%999.999f%999s”).
- File descriptor (based on the integer base type): includes -1, INT_MAX, and various descriptors: to a file open for reading, to a file open for writing, to a file whose offset is set to end of file, to an empty file, and to a file deleted after the file descriptor was assigned.

The above test values by no means represent all possible exceptional conditions and were chosen simply to explore a reasonably large input space. Future work could include studying ways to expand and automate the exploration of the exceptional input space. Nonetheless, experimental results show that even these relatively simple test values expose a significant number of robustness problems with mature software components.

A special feature of the test value database is that it is organized for automatic extraction of single-test-case programs. In other words, code for the various constructors and destructors for the particular test values of interest can be automatically extracted and placed in a single simple program that contains information for producing exactly one test case. This ability makes it easier to reproduce a robustness failure in isolation and facilitates creation of bug reports.

4.2 Test generation

In its simplest operating mode Ballista generates an exhaustive set of test cases that spans the cross product of all test values for each module input parameter. So, for example, the function *read*() would combine (per Table 1) 13 test values for the file descriptor, 15 test values for the buffer, and 16 test values for the integer length parameter, for a total of 13x15x16=3120 test cases.

Thus, the number of test cases for a particular MuT is determined by the number and type of input parameters and is exponential with the number of parameters. Most functions have fewer than 5000 tests; however, the seven POSIX functions listed in Table 2 exceed that, so combinations of parameter test values are pseudorandomly selected up to an arbitrary limit of 5000 test cases. In order to ensure comparability across systems and between different runs on the same system, the random number generator is seeded based on the function name, so for a given function the same tests are always run. A comparison of the results of random sampling to exhaustive testing (see Table 2) shows that the results can be expected to be very close (within less than one percentage point).

Table 2: Random Sampling vs. Exhaustive Testing
(Digital UNIX 3.2)

Function	Total Tests	Failure Rate from Complete Testing	Tests Run Randomly	Failure Rate from Random Testing	Percentage Point Difference
fread	49,152	29.9%	5000	30.8%	0.9%
fwrite	30,720	23.2%	5000	23.1%	0.1%
mmap	2,809,856	0%	5000	0%	0%
mq_receive	28,672	0%	5000	0%	0%
mq_send	19,712	0%	5000	0%	0%
strftime	25,600	75.6%	5000	75.5%	0.1%
timer_settime	7840	21.4%	5000	21.7%	0.3%

5 Experimental Results

The Ballista POSIX robustness test suite has been ported to the nine operating systems listed in Table 3 with no code modification. (In two cases the operating systems tested are significantly different versions from the same vendor.) On each operating system (OS) as many of the 233 POSIX calls were tested as were provided by the vendor. The compiler and libraries used were those supplied by the system vendor (in the case of Linux these were the GNU C tools).

Table 3: Summary of robustness testing results

System	POSIX Functions Tested	Fns. with Catastr. Failures	Fns. with Restart Failures	Fns. with Abort Failures	Fns. with No Failures	Number of Tests	Restart Failures	Abort Failures	Normal-ized Failure Rate
AIX 4.1	186	0	4	77	108 (58%)	64,009	13	11,559	10.0%
Digital UNIX 3.2	232	0	2	136	96 (41%)	92,628	17	18,074	15.6%
Digital UNIX 4.0	233	0	2	124	109 (47%)	92,658	17	18,316	15.0%
HP-UX A.09.05	186	0	3	87	98 (53%)	63,913	13	11,208	11.3%
IRIX 6.2	226	1	0	94	131 (58%)	91,470	0	15,086	12.6%
Linux 2.0.18	190	0	3	86	104 (55%)	64,513	9	11,986	12.5%
QNX 4.22	205	2	6	125	75 (37%)	73,508	505	20,068	20.7%
SunOS 4.1.3	189	0	2	104	85 (45%)	64,503	7	14,227	15.8%
SunOS 5.5.1	233	0	2	103	129 (55%)	92,658	28	15,376	14.6%

5.1 Results of testing POSIX operating systems

Table 3 shows that the combinational use of test values over a number of functions produced a reasonably large number of tests, ranging from 92,658 for the two OSes that supported all 233 POSIX functions to 63,913 for HP-UX. None of the OSes took more than three hours to test, so on average less than a minute was required for each function.

Catastrophic failures occurred in one function in IRIX, *munmap*(), requiring rebooting the workstation, and in two functions in QNX, *munmap*() and *mprotect*(). All OSes had relatively few Restart failures (task hangs). On the other hand, every OS exhibited a significant number of Abort failures (abnormal task terminations).

The main trend to notice in Table 3 is that from 37% to 58% of functions did not exhibit robustness failures under testing. This indicates that, even in the best case, about half the functions had at least one robustness failure.

5.2 Detailed per-function results

It would be simple to list the number of test cases that produced different types of robustness failures, but it is difficult to draw conclusions from such a listing because some functions have far more tests than other functions as a result of the combinatorial explosion of test cases with multiparametered functions. Instead, the number of failures are reported as a percentage of tests on a per-function basis. Figures 5a and 5b graph the per-function percent of failed test cases for the 233 functions of Digital UNIX 4.0 and SunOS 5.5.1, respectively. Providing normalized failure rates conveys a sense of the probability of failure of a function when presented with exceptional inputs, independent of the varying number of test cases executed on each function.

The two functions in both Figures 5a and 5b with 100% failure rates are *longjmp*() and *siglongjmp*(), which perform control flow transfers to a target address. These functions are not required by the POSIX standard to recover from exceptional target addresses, and it is easy to see why such a function would abort on almost any invalid address provided to it. Nonetheless, one could envision a version of this function that could recover from such a situation. Similarly, one can argue that most of the remaining functions should return error codes rather than failing for a broad range of exceptional inputs.

The other function in Figure 5b with a 100% failure rate is *asctime*(), whose sole parameter is of type `struct tm *`. Due to the difficulty of writing tests for `structs` in the current implementation (see Section 7, Future Work), *asctime*() was tested with a generic pointer type. Since none of the generic tests generate a parameter conforming to the type `struct tm *`, many of the tests looked the same to *asctime*(), which explains all the responses being the same.

Graphs similar to those in Figure 5, for the remainder of the OSes, can be found in the Appendix. The gray areas denote functions not available on an OS. A comparison of all these graphs shows that the functions failing were not necessarily the same across different OSes. However, two groups of functions that generally failed on all systems are the C

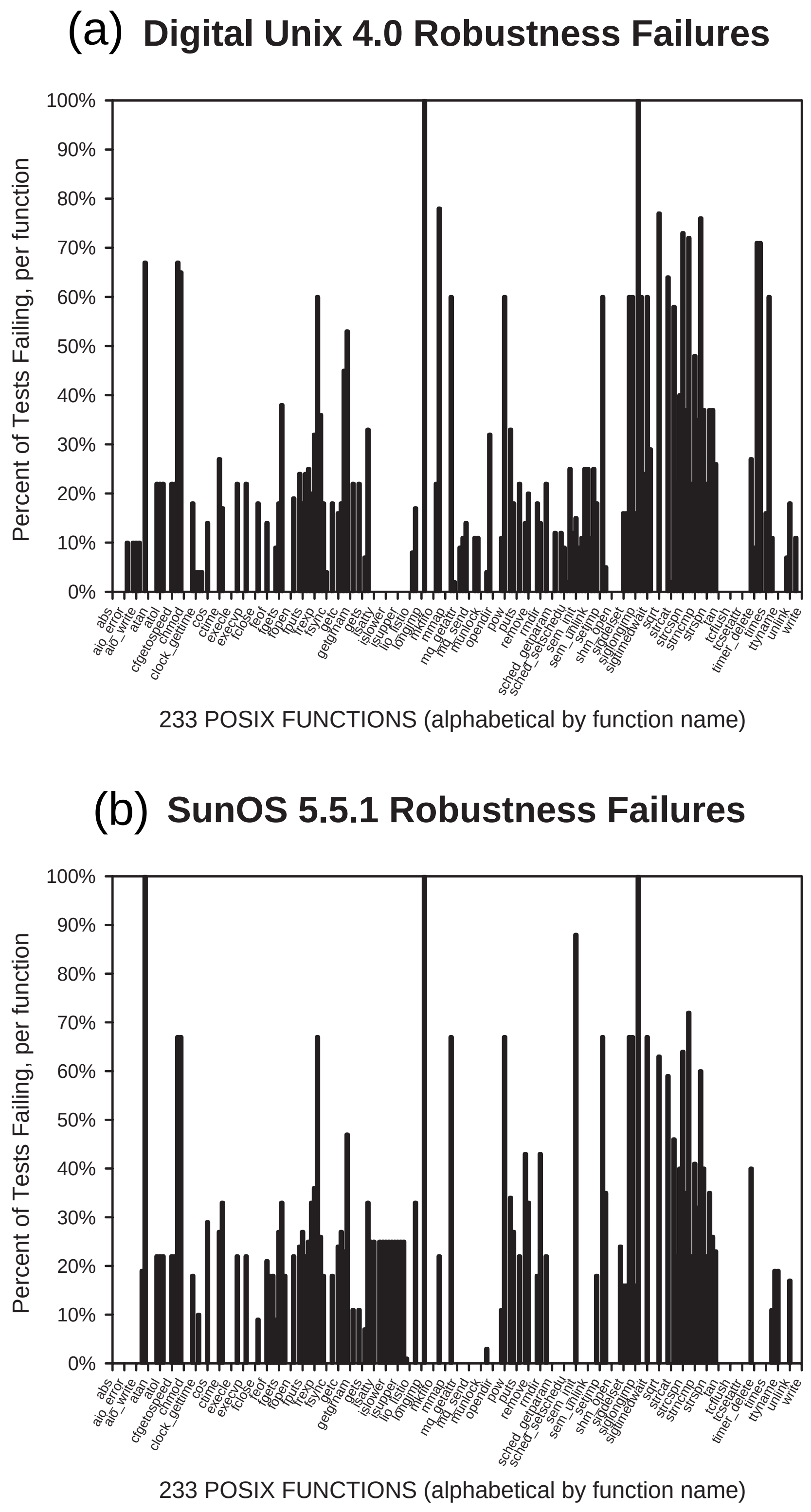


Figure 5. Normalized failure rates for 233 POSIX functions on Digital UNIX 4.0 and SunOS 5.5.1. The data represent 92,658 tests spanning the 233 functions.

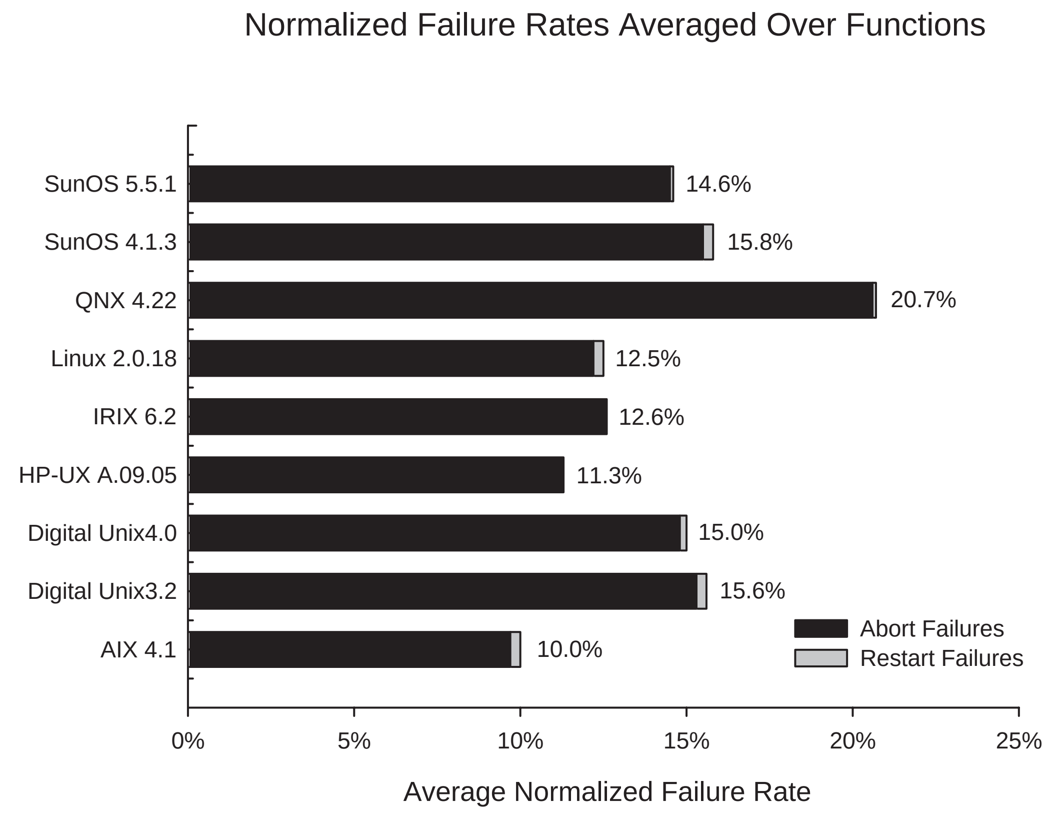


Figure 6. Normalized failure rates for nine POSIX operating systems. Catastrophic failures are not included due to the difficulty of retaining test result information across system crashes.

standard library string and file functions, often due to the same invalid parameters: NULL and other invalid pointers.

5.3 Comparing results among implementations

One possible use for robustness testing results is to compare different implementations of the same API. For example, one might be deciding which off-the-shelf operating system to use. It might be useful to compare the robustness results of different operating systems and avoid those that were significantly less robust than others. In application areas other than operating systems there might still be possible sources of identical or roughly equivalent APIs that could be similarly evaluated, such as graphic libraries or database engines.

Figure 6 shows normalized failure rates for the OSes measured. Each failure rate is the arithmetic mean of the normalized failure rates for each function, including both functions that fail and functions that are failure-free. So, the normalized failure rates represent a failure probability metric for an OS implementation conditional upon the actual exceptional input distribution of the current Ballista test cases. As such, they are probably most useful as relative measures of the robustness of an entire API.

It is important to note that the results do not purport to report the number of software defects ("bugs") in the modules that have been tested. Rather, they report the number of times that inputs elicit faulty responses due to one or more robustness deficiencies within the software being tested. From a user's perspective it is not as important how many bugs are within COTS software as the likelihood of triggering a failure response due to a robustness deficiency.

6 Generic applicability of the methodology

The successful experience of the Ballista methodology in testing implementations of the POSIX API suggests that it may be a useful technique for robustness testing of generic COTS software modules. Different aspects of the Ballista methodology that are important for generic applicability include: scalability, portability, cost of implementation, and effectiveness.

6.1 Scalability

Testing a new software module with Ballista often incurs no incremental test case development cost. In cases where the data types used by a new software module are already included in the test database, testing is accomplished simply by defining the interface to the module in terms of data types and running a test. For example, once tests for a file descriptor, buffer, and length are created to enable testing the function *read*(), other functions such as *write*(), *dup*(), and *close*() can be tested using the same data types. Furthermore, the data types buffer and length would have already been defined if these functions were tested after tests had been created for functions such as *memcpy*(). Even when data types are not available it may be possible to substitute a more generic data type or base data type for an initial but limited assessment (for example, a generic memory pointer may be somewhat useful for testing a pointer to a special data structure).

In addition, specific data type test values and even new data types can be added without invalidating previous results. Such additions can augment existing test results, making rerunning all previous tests unnecessary. Furthermore, the system reorganizes itself automatically whenever additions or other changes are made, eliminating the need to revise the test harness infrastructure after each change.

The number of tests to be run can be limited using random sampling or specific parameter variation. Random sampling allows the execution time of testing a module to be kept low even with a large search space. Parameter variation can be used to produce a specific subset of the complete tests for a module; one or more parameters are held constant while the others are varied. As the degenerate case, a single test case can be run individually. This can be useful for obtaining one specific result, possibly to compare it to the same test case run as a part of a complete test run, or run under different system conditions (to verify repeatability). (Note that the ability to run a single test case is separate from the capability to automatically extract standalone code that runs a single test case.)

6.2 Portability

The Ballista approach has proven portable across platforms, and promises to be portable across applications. The Ballista tests have been ported to nine processor/operating system pairs. This demonstrates that high-level robustness testing can be conducted without any hardware or operating system modifications. Furthermore, the use of normalized failure reporting supports direct comparisons among different implementations of an API executing on different platforms.

In a somewhat different sense, Ballista seems to be portable across different applications. The POSIX API encompasses functions including file handling, string handling, I/O, task handling, and even mathematical functions. No changes or exceptions to the Ballista approach were necessary in spanning this large range of functionality, so it seems likely that Ballista will be useful for a significant number of other applications as well.

6.3 Testing cost

One of the biggest unknowns when embarking upon a full-scale demonstration of the Ballista methodology was the amount of test scaffolding that would have to be erected for each function tested. In the worst case, special-purpose code would have been necessary for each of the 233 POSIX functions tested. If that had been the case, it would have resulted in a significant cost for constructing tests for automatic execution (a testing cost linear with the number of modules to be tested).

However, the adoption of an object-oriented approach based on data type yielded an expense for creating test cases that was sublinear with the number of modules tested. Figure 7 is a graph of functions testable versus data types implemented, for the POSIX test set. The figure shows that implementing just a few data types enables most of the functions to be testable. Eventually the curve levels off, but the incremental cost is still linear at worst. The key observation is that in a typical program there may be fewer data types than functions—the same data types are used over and over when creating function declarations. In the case of POSIX calls, only 20 data types were used by 233 functions, so the effort in creating the test suite was driven by the 20 data types, not by the number of functions.

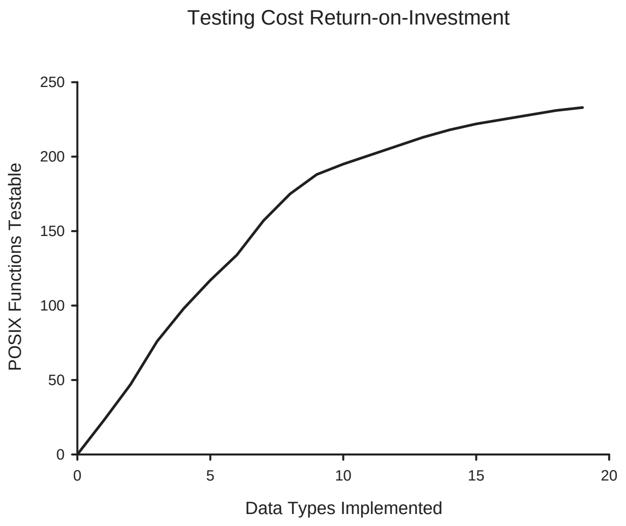


Figure 7. Benefits of an object-oriented approach based on data type. Only 20 data types are needed to test 233 POSIX functions.

Although we have not conducted the exercise, it seems likely the Ballista testing approach will also work with an object-oriented software system. The effort involved in preparing for automated testing would be proportional to the number of object classes (data types) rather than the number of methods within each class. In fact, one could envision robustness testing information being added as a standard part of programming practice when creating a new class, just as debugging print statements might be added. Thus, a transition to object-oriented programming should have little effect on the cost and effectiveness of the Ballista testing methodology.

6.4 Effectiveness

The Ballista testing fault model is fairly simplistic: single function calls that result in a crash or hang. It specifically does not encompass sequences of calls. Nonetheless, it is sufficient to uncover a significant number of robustness problems. Part of this may be that such problems are easy to uncover, but part of it may also be that the object-oriented testing approach is more powerful than it appears upon first thought.

In particular, a significant amount of system state may be set by the constructor for each instance of a data type test value. For example, a file descriptor test value might create a particular file with associated permissions, access mode, and contents as part of its constructor operation. Thus, a single test case can in many cases replace a sequence of tests that would otherwise have to be executed to create and test a function in the context of a particular system state. In other words, a series of calls to achieve a given system state can be simulated by a constructor that in effect jumps directly to a desired system state without need for a sequence of calls in a test.

A high emphasis has been placed on reproducibility within Ballista. In 99% of the 80,232 cases attempted, extracting a single test case into a standalone test program leads to a reproduction of robustness failures. In a few cases having to do with the location of buffers the failure is reproducible only by executing a single test case within the testing harness (but, is reproducible in the harness and presumably has to do with how the data structures have been arranged in memory).

The only situation in which Ballista results have been found to lack reproducibility is in some Catastrophic failures (complete system crashes). On two systems (IRIX and QNX, as per the results presented above), system crashes were completely reproducible. On Digital UNIX 3.2 with an external swap partition mounted (different conditions from those under which the results presented above were obtained), it appeared that a succession of two or three test cases could produce a system crash from the function *mq_receive*(), probably having to do either with internal operating system state being damaged by one call resulting in the crash of a second call, or with latent manifestation of the error.

In all cases, however, robustness failures have been reproducible by rerunning the Ballista test programs, and could be recreated under varying system loads including otherwise idle systems.

7 Future Work

The robustness results reported here are unweighted normalized averages. It may be useful to be able to weight the data according to relative frequencies of module calls in actual programs, so that the robustness results better reflect the likelihood of a specific program or application encountering nonrobust behavior. Modules used more often could be weighted more heavily than seldom-used modules. A more extensive analysis could determine relative frequencies of actual parameters passed to modules. With these weights applied to robustness data, a more practically accurate measure of robustness could be available.

Current Ballista testing searches for robustness faults using heuristically created test cases. Future work could include both random and patterned coverage of the entire function input space in order to produce better information about the size and shape of input regions producing error responses, and to generate statistical information about test coverage.

In the Ballista implementation presented here, the data type test databases were built as objects, but without taking advantage of the hierarchy present in many data types (*e.g.*, a filename is a type of character string, which is a type of pointer (`char *`)). Building the test database hierarchically could make database creation and expansion even easier, as well as enabling hierarchical data types (*e.g.*, `structs` in C) to be built with very little extra effort.

If COTS software is to be used in mission-critical applications, it may be beneficial to provide a mechanism that could keep a software component from behaving in a nonrobust manner. One way to do this could be to create a software wrapper to encapsulate a COTS component. The wrapper would discard calls to the component that would cause the component to behave nonrobustly. Ballista-type testing could be used in creating the wrapper, generating a list of which function/parameter combinations are dangerous (eliciting Catastrophic, Restart, or Abort failures during Ballista testing) so that those calls are not passed to the component.

8 Conclusions

The Ballista methodology can automatically assess the robustness of software components in response to exceptional input parameter values. This has been demonstrated by a full-scale implementation and application to POSIX operating system calls, in which as many as 233 functions were tested on each of nine commercially available operating systems. Even in these mature sets of software, about half the functions tested exhibited robustness failures with a normalized failure rate of from 10% to 21%, and even Catastrophic failures were found in several operating systems.

An object-oriented approach based on data types rather than component functionality is the key to the Ballista methodology. This was found to be inexpensive to implement because the test database development was proportional to the number of data types (20 data types) instead of the number of functions tested (233 functions) or the number of tests executed (up to 92,658 tests).

This high-level approach has enabled Ballista testing to be highly repeatable, portable, and extendable. The implementation presented here tested a wide variety of system calls on a number of different operating systems, without modification. The results obtained are able to be recreated by running the tests again. Finally, by basing tests on parameter data types, new functions can be added to the test set for low (often zero) cost.

The major contribution of this Master's project has been to develop and implement the Ballista approach, which has been shown to be an effective method for robustness testing. It has produced practical results that could help evaluate and improve current software, and it also shows promise as the basis for more sophisticated and comprehensive test methods.

9 References

[1] Lions, J. (Chair), *Ariane 5 Flight 501 Failure, European Space Agency,* Paris, 19 July 1996. *http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.* Accessed 29 Apr 1998.

[2] *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990), IEEE Computer Society, 10 Dec 1990.

[3] *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.

[4] Tsai, T., and R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," *Proc. Eighth Intl. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Germany, 20-22 Sep 1995, Springer-Verlag, pp. 26-40.

[5] Segall, Z., D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT - Fault Injection Based Automated Testing Environment," *Proc. Eighteenth Intl. Symp. on Fault-Tolerant Computing,* Tokyo, 27-30 Jun 1988, IEEE Computer Society, pp. 102-107.

[6] Suh, B., C. Fineman, and Z. Segall, "FAUST - Fault Injection Based Automated Software Testing," *Proc. 1991 Systems Design Synthesis Technology Workshop,* Silver Spring, MD, 10-13 Sep 1991, NSWC.

[7] Miller, B., D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services,* Computer Sciences Technical Report 1268, University of Wisconsin–Madison, 1995.

[8] Dingman, C., *Portable Robustness Benchmarks,* Ph.D. Thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 1997.

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

[10] Horgan, J., and A. Mathur, “Software Testing and Reliability,” in: Lyu, M., ed., *Handbook of Software Reliability Engineering,* IEEE Computer Society, 1995, pp. 531-566.

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

Appendix

Code for filename data type

```
/* fname.c
 *
 * Data object constructor for filenames
 * Written by Nathan Kropp; Test cases by Nathan Kropp and Chris Dingman
 * For Master’s project, CMU, 1997
 * 4 June 1997
 *
 */

#include “types.h”
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include “support.h”

#define FNAME_NOEXIST   0
#define FNAME_EMBED_SPC 1
#define FNAME_LONG      2
#define FNAME_CLOSED    3
#define FNAME_OPEN_RD   4
#define FNAME_OPEN_WR   5
#define FNAME_EMPTY_STR 6
#define FNAME_RAND      7
#define FNAME_NEG       8
#define FNAME_NULL      9
#define NUM_VALUES      10      /* Must equal prev line + 1 (and  */
                               /*  prev line must be the max)    */

int Get_FNAME (char *param_name[PARAM_NAME_LEN],
                        char **param,
                        int value)
{
   /* The following table must match the above #defines */
   char *param_name_table[] = {
      “FNAME_NOEXIST”,
      “FNAME_EMBED_SPC”,
      “FNAME_LONG”,
      “FNAME_CLOSED”,
      “FNAME_OPEN_RD”,
      “FNAME_OPEN_WR”,
      “FNAME_EMPTY_STR”,
      “FNAME_RAND”,
      “FNAME_NEG”,
      “FNAME_NULL”
   };

   /* local var declarations go here */
   /* VARS */
   int fname_tempfd;
   char *fname_testfilename;
   static int fname_count;      /* count for multiple instances */
   char fname_count_str [127];  /* count, in string format      */
```

```
/* end VARS */

if ( param == NULL ) {
   /* initialization stuff here */
   /* INIT */
   fname_count = 0;
   /* end INIT */

   return (NUM_VALUES);
}
assert (param_name != NULL);

/* global setup here */
/* SETUP */
fname_count++;
sprintf (fname_count_str, “_fname%d”, fname_count);
fname_testfilename = (char *) malloc (strlen (TESTFILE) + strlen (fname_count_str) + 2048);
                                             /* + 2048 to be safe below */
strcpy (fname_testfilename, TESTFILE);
strcat (fname_testfilename, fname_count_str);
/* end SETUP */

switch ( value )
{
 case FNAME_NOEXIST:
   unlink (fname_testfilename);
   *param = fname_testfilename;
   break;

 case FNAME_EMBED_SPC:
   strcpy (fname_testfilename, TESTDIR);
   strcat (fname_testfilename, “ space here”);
   unlink (fname_testfilename);
   *param = fname_testfilename;
   break;

 case FNAME_LONG:
   strcpy (fname_testfilename, TESTDIR);
   sup_fill (fname_testfilename + strlen (fname_testfilename), 1028);
   fname_testfilename [1027] = ‘\0’;
   *param = fname_testfilename;
   break;

 case FNAME_CLOSED:
   sup_createfile (fname_testfilename);
   fname_tempfd = open (fname_testfilename, O_RDONLY);
   close (fname_tempfd);
   *param = fname_testfilename;
   break;

 case FNAME_OPEN_RD:
   sup_createfile (fname_testfilename);
   fname_tempfd = open (fname_testfilename, O_RDONLY);
   *param = fname_testfilename;
   break;

 case FNAME_OPEN_WR:
   sup_createfile (fname_testfilename);
   fname_tempfd = open (fname_testfilename, O_WRONLY);
   *param = fname_testfilename;
   break;

 case FNAME_EMPTY_STR:
   *fname_testfilename = ‘\0’;
   *param = fname_testfilename;
```

```
      break;

     case FNAME_RAND:
      *param = (char *) ( (unsigned long) fname_testfilename < 194670 ?
                          (unsigned long) fname_testfilename+194675 :
                          (unsigned long) fname_testfilename-194675 );
         /* that keeps it positive */
      break;

     case FNAME_NEG:
      *param = (char *) -1;
      break;

     case FNAME_NULL:
      *param = NULL;
      break;

     default:
      fprintf (stderr, “Error: Unknown filename type [%d]\n”, value);
    }

   /* global teardown here */

   *param_name = param_name_table [value];
   return (0);
}
```

Sample function specification file

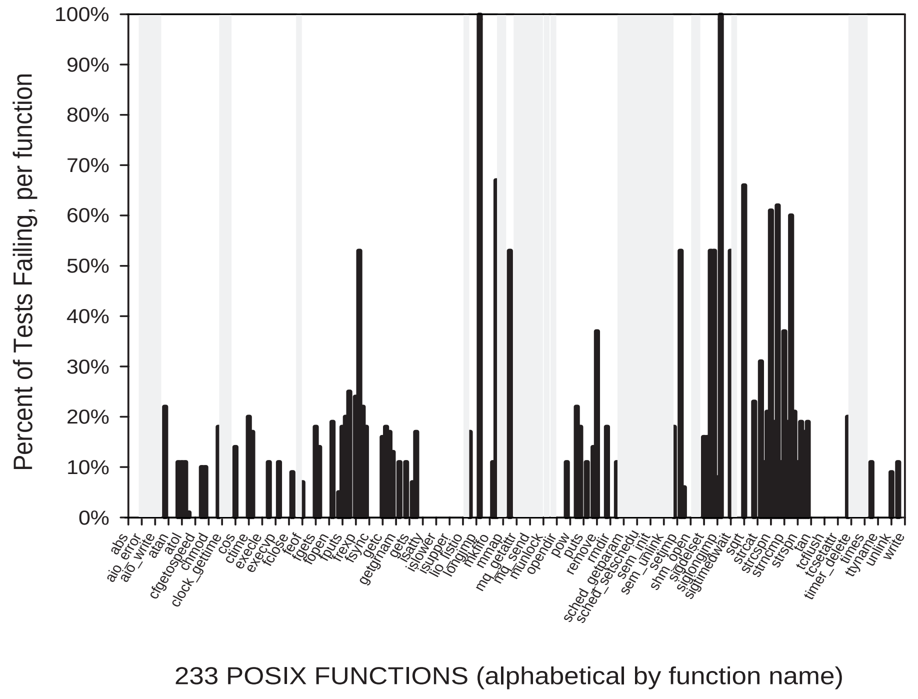
```
# Format of a function specification line:
#
# <#include file>  <return type>  <FUNCTION NAME>  <param1 type>  <param2 type>  ...
#

file_group:
unistd     int        access     fname      mode
unistd     int        chdir      fname
sys/stat   int        chmod      fname      fmode
unistd     int        chown      fname      pid        pid
unistd     int        close      fd
fcntl      int        creat      fname      fmode
unistd     int        dup        fd
unistd     int        dup2       fd         fd
sys/stat   int        fchmod     fd         fmode
fcntl      int        fcntl      fd         mode       int
fcntl      int        fcntl      fd         mode       oflags
unistd     int        fdatasync  fd
sys/stat   int        fstat      fd         buf
unistd     int        fsync      fd
unistd     int        ftruncate  fd         int
unistd     char*      getcwd     buf        int
unistd     int        link       fname      fname
unistd     int        lseek      fd         int        mode
sys/stat   int        mkdir      fname      fmode
sys/stat   int        mkfifo     fname      fmode
fcntl      int        open       fname      oflags
fcntl      int        open       fname      oflags     fmode
unistd     int        read       fd         buf        int
unistd     int        rename     fname      fname
unistd     int        rmdir      fname
sys/stat   int        stat       fname      buf
sys/stat   int        umask      fmode
unistd     int        unlink     fname
unistd     int        write      fd         str        int
```

Sample test result file

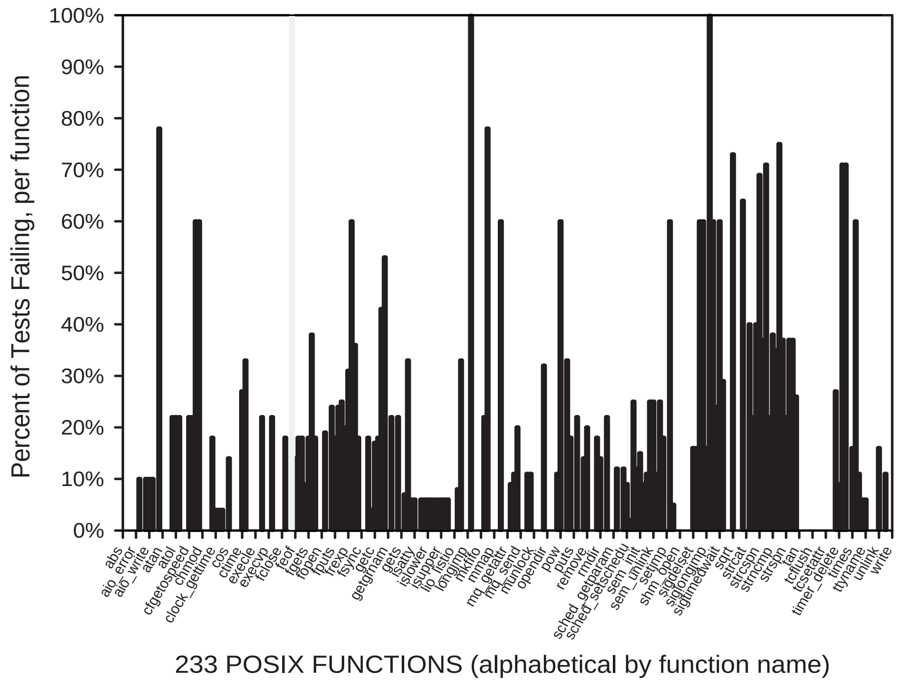
```
Results from CRASHmarks OS robustness test suite

OS under test:  OSF1 V3.2
Run date:  Thu 22 Jan 1998  09:16:52 EST
CRASHmarks version:  0.90


System call chmod 40/40
FNAME_NOEXIST      FMODE_ZERO     Done - Pass    2
FNAME_NOEXIST      FMODE_ONE      Done - Pass    2
FNAME_NOEXIST      FMODE_ALL      Done - Pass    2
FNAME_NOEXIST      FMODE_NEG_ONE  Done - Pass    2
FNAME_EMBED_SPC    FMODE_ZERO     Done - Pass    2
FNAME_EMBED_SPC    FMODE_ONE      Done - Pass    2
FNAME_EMBED_SPC    FMODE_ALL      Done - Pass    2
FNAME_EMBED_SPC    FMODE_NEG_ONE  Done - Pass    2
FNAME_LONG         FMODE_ZERO     Done - Pass    63
FNAME_LONG         FMODE_ONE      Done - Pass    63
FNAME_LONG         FMODE_ALL      Done - Pass    63
FNAME_LONG         FMODE_NEG_ONE  Done - Pass    63
FNAME_CLOSED       FMODE_ZERO     Done - Pass    0
FNAME_CLOSED       FMODE_ONE      Done - Pass    0
FNAME_CLOSED       FMODE_ALL      Done - Pass    0
FNAME_CLOSED       FMODE_NEG_ONE  Done - Pass    0
FNAME_OPEN_RD      FMODE_ZERO     Done - Pass    0
FNAME_OPEN_RD      FMODE_ONE      Done - Pass    0
FNAME_OPEN_RD      FMODE_ALL      Done - Pass    0
FNAME_OPEN_RD      FMODE_NEG_ONE  Done - Pass    0
FNAME_OPEN_WR      FMODE_ZERO     Done - Pass    0
FNAME_OPEN_WR      FMODE_ONE      Done - Pass    0
FNAME_OPEN_WR      FMODE_ALL      Done - Pass    0
FNAME_OPEN_WR      FMODE_NEG_ONE  Done - Pass    0
FNAME_EMPTY_STR    FMODE_ZERO     Done - Pass    2
FNAME_EMPTY_STR    FMODE_ONE      Done - Pass    2
FNAME_EMPTY_STR    FMODE_ALL      Done - Pass    2
FNAME_EMPTY_STR    FMODE_NEG_ONE  Done - Pass    2
FNAME_NEG          FMODE_ZERO     Done - Pass    14
FNAME_NEG          FMODE_ONE      Done - Pass    14
FNAME_NEG          FMODE_ALL      Done - Pass    14
FNAME_NEG          FMODE_NEG_ONE  Done - Pass    14
FNAME_NULL         FMODE_ZERO     Done - Pass    14
FNAME_NULL         FMODE_ONE      Done - Pass    14
FNAME_NULL         FMODE_ALL      Done - Pass    14
FNAME_NULL         FMODE_NEG_ONE  Done - Pass    14

System call pipe 16/16
BUF_SMALL          Done - Pass    0
BUF_MED            Done - Pass    0
BUF_LARGE          Done - Abort   -1
BUF_XLARGE         Done - Abort   -1
BUF_HUGE           Done - Abort   -1
BUF_MAX            Done - Abort   -1
BUF_64K            Done - Pass    0
BUF_END_MED        Done - Pass    0
BUF_FAR_PAST       Done - Abort   -1
BUF_ODD            Done - Pass    0
BUF_FREED          Done - Pass    0
BUF_CODE           Done - Abort   -1
BUF_LOW            Done - Abort   -1
BUF_NULL           Done - Abort   -1
BUF_NEG            Done - Abort   -1
```

Results of Robustness Testing of Other Operating Systems

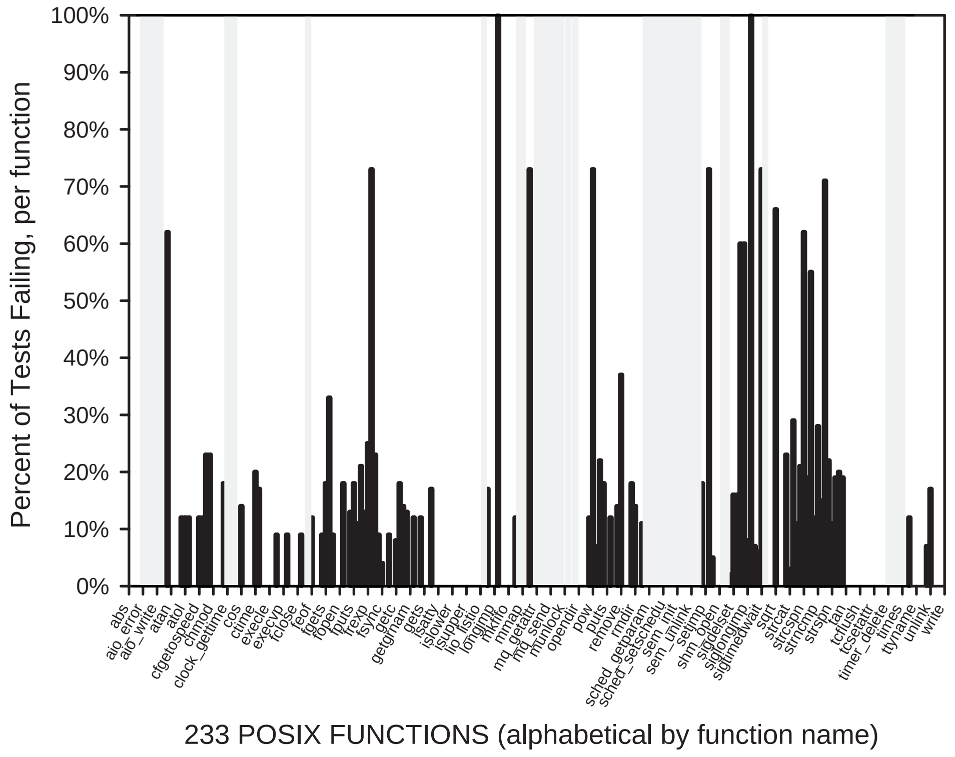
AIX 4.1 Robustness Failures



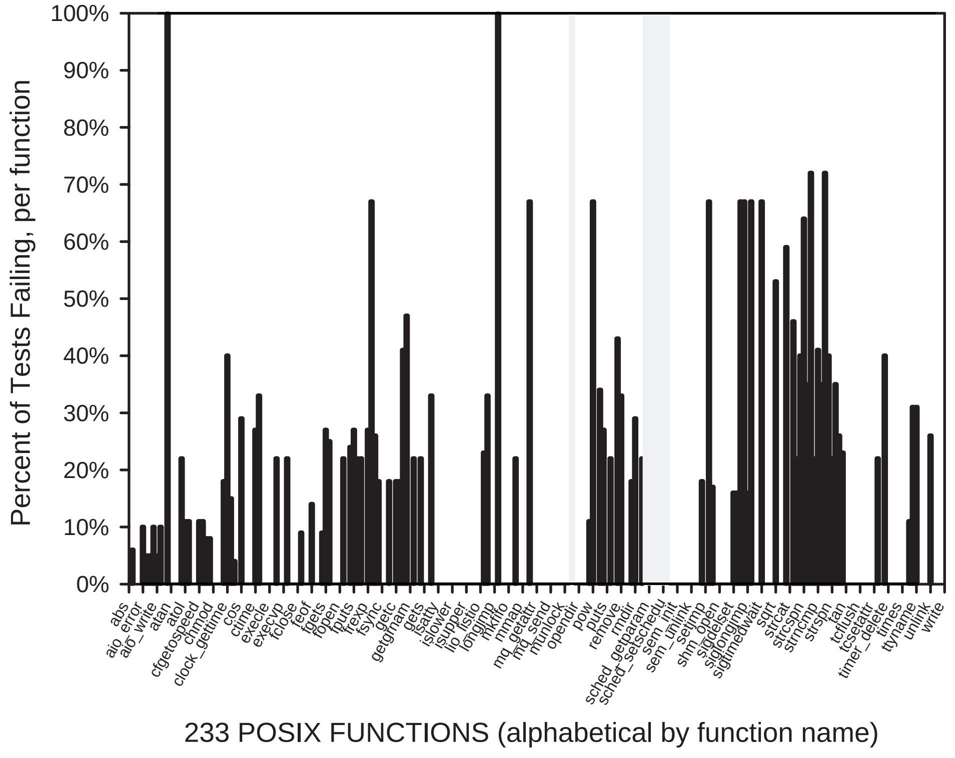
Digital Unix 3.2 Robustness Failures



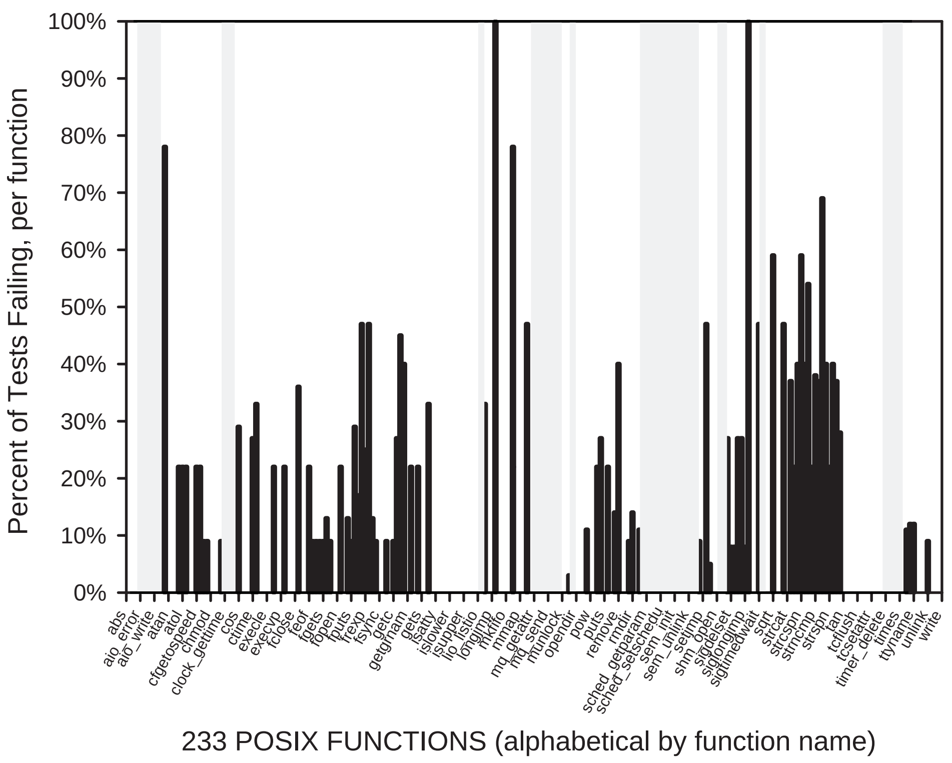
HP-UX A.09.05 Robustness Failures



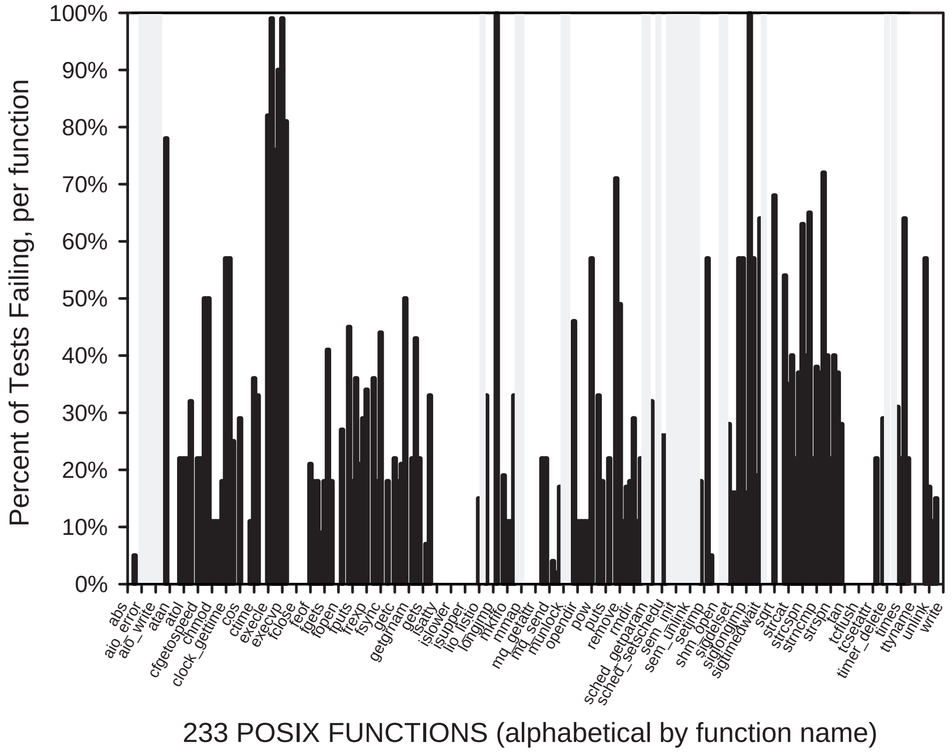
IRIX 6.2 Robustness Failures



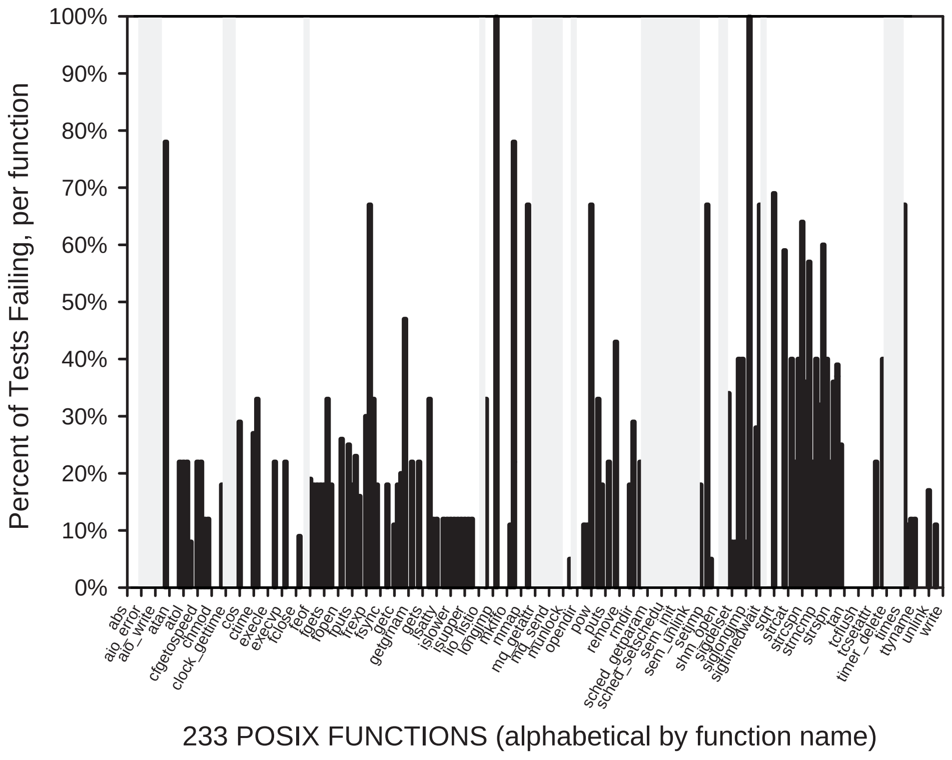
Linux 2.0.18 Robustness Failures



QNX 4.22 Robustness Failures



SunOS 4.1.3 Robustness Failures


Percent of Tests Failing, per function
abs
aio_error
aio_write
atan
atol
cfgetospeed
chmod
clock_gettime
cos
ctime
execle
execvp
fclose
feof
fgets
fopen
fputs
frexp
fsync
getc
getgrnam
gets
isatty
islower
isupper
ln
lfstio
longjmp
mkfifo
mmap
mq_getattr
mq_send
munlock
opendir
pow
puts
remove
rmdir
sched_getparam
sched_setschedu
sem_init
sem_unlink
setjmp
shm_open
sigdelset
siglongjmp
sigtimedwait
sqrt
strcat
strcspn
strncmp
strspn
tan
tcflush
tcsetattr
timer_delete
times
ttyname
unlink
write
233 POSIX FUNCTIONS (alphabetical by function name)