Automated Robustness Testing of Off-the-Shelf Software Components

Nathan P. Kropp, Philip J. Koopman, Daniel P. Siewiorek
Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Fault Tolerant Computing Symposium

June 23-25, 1998 in Munich, Germany

Contact author:
Philip Koopman
Carnegie Mellon University, ECE Dept.
Hamerschlag D-202
5000 Forbes Ave
Pittsburgh PA 15213-3890
koopman@cmu.edu
Phone: +1 - 412 / 268-5225
Fax: +1 - 412 / 268-6353

Automated Robustness Testing of Off-the-Shelf Software Components

Nathan P. Kropp, Philip J. Koopman, Daniel P. Siewiorek
Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Abstract

Mission-critical system designers may have to use a Commercial Off-The-Shelf (COTS) approach to reduce costs and shorten development time, even though COTS software components may not specifically be designed for robust operation. Automated testing can assess component robustness without sacrificing the advantages of a COTS approach. This paper describes the Ballista methodology for scalable, portable, automated robustness testing of 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 ten POSIX systems. Between 42% and 63% of components tested had robustness problems, with a normalized failure rate ranging from 10% to 23% 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.

1. Introduction

Mission-critical system designers are being pressed to use a Commercial Off-The-Shelf (COTS) approach to reduce costs and shorten development time. Even though using available COTS components may be required for business success, there is a risk that resultant systems may not be robust. For example, 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 a presumed ability of operators to work around problems.

Mission-critical applications may not be as forgiving of robustness problems as traditional desktop applications. Even worse, components which are 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, which proved to have robustness problems when operating under the different flight conditions encountered on Ariane 5. [1] With the current trend toward 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 gracefully dealing with such exceptional conditions.

The robustness of a software component is the degree to which it functions correctly in the presence of exceptional inputs or stressful environmental conditions. [2] In some mature test suites so-called “dirty” tests that measure responses to invalid conditions can outnumber tests for correct conditions by a ratio of 4:1 or 5:1. [3] The focus of this paper is the Ballista methodology for automatic creation and execution of a portion of these numerous invalid input robustness tests — in particular tests designed to detect crashes and hangs caused by invalid inputs to function calls. The results presented indicate that robustness vulnerabilities to invalid inputs are common in at least one class of mature COTS software components.

The goal of Ballista is to automatically test for and harden against software component failures caused by exceptional inputs. In this first phase of the project the emphasis is on testing, with creation of protective hardening “software wrappers” to be done in the future. The approach presented here 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 even source code, but 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 creating robustness hardening wrappers.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

The Ballista approach is intended to be generically applicable to a wide range of application areas. 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*). [4] A demonstration of this system is available on the World Wide Web at *http://www.ices.cmu.edu/ballista*

In the balance of this paper, Section 2 describes the testing methodology and how it relates to previous work in the areas of software testing and fault injection. Section 3 describes the implementation, while Section 4 describes experimental results for operating system (OS) testing. Section 5 evaluates the effectiveness of the methodology, and Section 6 provides conclusions.

2. Methodology

A software component, for our purposes, is any piece of software that can be invoked as a procedure, function, or method with a non-null set of input parameters. While that is not a universal definition of all software interfaces, it is sufficiently broad to be of interest. In the Ballista approach, robustness testing of such a software component (a Module under Test, or *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 combinations 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.

Ballista draws upon ideas from the areas of both software testing and fault injection. A key idea is the use of an object-oriented approach driven by parameter list data type information to achieve scalability and automated initialization of system state for each test case.

2.1. 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 as a way of assessing whether a module has been thoroughly tested. [5] 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 are useful for attaining high test coverage of programs. But they typically focus on the control flow of a program rather than 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 a test is missing altogether. Additionally, structural testing typically requires access to source code, which may be unavailable when using COTS software components. A complementary approach is black-box testing, also called behavioral testing. [3] Black-box testing techniques are designed to demonstrate correct response to various input values regardless of the software implementation, and seem more appropriate for robustness testing.

Two types of black-box testing are particularly useful as starting points 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 are among the approaches used in Ballista as described in Section 3 on implementation.

Automatically generating software tests requires three things: a MuT, a machine-understandable specification of correct behavior, and an automatic way to compare results of executing the MuT with the 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 hangs. Thus, robustness testing can be performed on modules (that don’t intentionally crash or hang) in the absence of a behavioral specification.

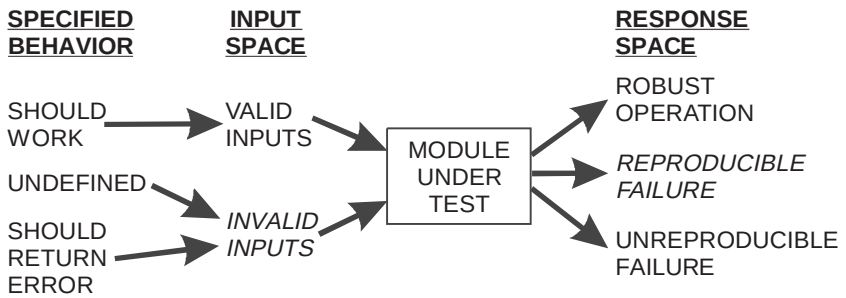

Figure 1. Ballista performs fault injection at the API level using combinations of valid and exceptional inputs.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

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 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), and an unreproducible failure (a robustness failure that is not readily reproducible). The objective of Ballista is to identify reproducible failures.

2.2. Use of fault injection concepts

Fault injection is a technique for evaluating robustness by artificially inducing a fault and observing the system's response. Some fault injection techniques require special-purpose hardware (*e.g.*, FTAPE [6]), and thus are not portable across commercially available systems. Even some Software-Implemented Fault Injection (SWIFI) approaches have exploited specific features of a particular hardware platform, reducing portability (*e.g.*, FIAT [7] and Ferrari [8]). Code mutation (*e.g.*, [9]) is a portable software testing technique that performs fault injection on source code, but is in general more suitable for test set coverage analysis than robustness testing.

Two portable SWIFI approaches are specifically targeted at testing the robustness of software components. The University of Wisconsin Fuzz approach [10] generates a random input stream to various Unix programs and detects crashes and hangs. The Carnegie Mellon robustness benchmarking approach [11][12] tests individual system calls with specific input values to detect crashes and hangs. Both techniques focus on robustness problems that can occur due to faulty data being passed among software modules.

The Ballista approach is a generalization of previous Carnegie Mellon work, and performs fault injection at the API level. Injection is performed by passing combinations of acceptable and exceptional inputs as a parameter list to the MuT via an ordinary function call. For example, a file handle to a file opened for reading might be passed to a file write function.

2.3. Elimination of per-function scaffolding

A key idea of Ballista is that tests are based on the values of parameters and not on the behavioral details of the MuT. In order to accomplish this, 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 does not depend on a behavioral specification.

Figure 2 shows the Ballista approach to generating test cases for a MuT. Before conducting testing, 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 MAXINT (maximum integer value). Additionally, if a pointer data type is used within the MuT, values of NULL and -1, among others, might be used. 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 pools of defined test values.

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. Instantiation of a testing object (which includes 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 a particular integer value. But, a file descriptor test constructor might create a file, place information in it, set appropriate access permissions,

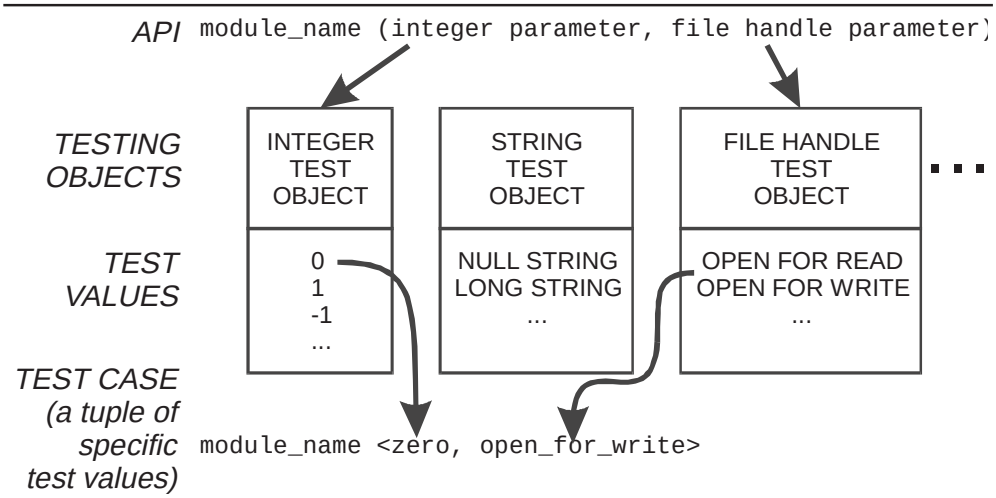

Figure 2. Refinement of a module within an API into a particular test case.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

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

Figure 3. Code for an example constructor. `fd_filename` 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.

then open the file requesting read permission. An example of a test constructor to create a file open for read is shown in Figure 3.

When a testing object is discarded, the corresponding destructor for that test case performs appropriate actions to free, remove, or otherwise undo whatever system state may remain in place 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 code is automatically generated given just a function name and a typed parameter list. In actual testing a separate process is spawned for each test case to facilitate failure detection.

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.

Ballista is related in some respects to other automated combinational testing approaches such as AETG [13] and TOFU [14] in that test cases are organized by parameter fields and then combined to create tests of functions. However, the Ballista approach does not require the addition of any information specific to the function under test, such as relations among fields and unallowed tests required by AETG, and the further addition of interaction weightings by TOFU. While such information about parameters could be added to the Ballista approach, doing so is not required to attain effective operation. Additionally, approaches that require weighted relationships may not be

```
/* test function read(fd, buf, len) */
foreach ( fd_case )
{ foreach ( buf_case )
  { foreach ( len_case )
    { /* constructors create instances */
      fd_test fd(fd_case);
      buf_test buf(buf_case);
      len_test len(len_case);

      /* test performed */
      puts("starting test...");
      read(fd, buf, len);
      puts("...test completed");

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

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.

appropriate in robustness testing, where one is testing the behavior of software in exceptional operating conditions. For example, one could argue that it is the interactions programmers and testers didn't think important or possible (and thus are exceptional) that are precisely the ones that Ballista should hope to find.

2.4. Robustness measurement

The response of the MuT is measured in terms of the CRASH scale. [14] 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"), Silent (the test process exits without an error code, but one should have been returned), Hindering (the test process exits with an error code not relevant to the situation), and Pass (the module exits properly, possibly with an appropriate error code). Silent and Hindering failures are currently not found by Ballista. While it would be desirable to extend Ballista to include Silent and Hindering failures, it is unclear at this time how to do so in a scalable fashion without requiring detailed information about each function tested.

3. Implementation

The Ballista approach to robustness testing has been implemented for a set of 233 POSIX calls, including real-time extensions for C. All system calls defined in the IEEE 1003.1b standard [4] ("POSIX.1b", or "POSIX with

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

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 intentionally send signals, such as `kill`. POSIX calls were chosen as an example application because they encompass a reasonably complex set of functionality, and are widely available in multiple mature commercial implementations.

3.1. Test value database

Table 1 shows the 20 data types which were necessary for testing the 233 POSIX calls. The code for the 190 test values across the 20 data types typically totals between one and fifteen lines of C code per test value. Current test values were chosen based on the Ballista programming team's experience with software defects and knowledge of compiler and operating system behavior.

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, MAXINT, -MAXINT, selected powers of two, powers of two minus one, and powers of two plus one.
- Float data type: 0, 1, -1, +/-DBL_MIN, +/-DBL_MAX, pi, and *e*
- Pointer data type: NULL, -1 (cast to a pointer), pointer to `free`'d memory, and 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 pointer values are set near the end of allocated memory to test the effects of accessing memory on virtual memory pages just past valid addresses.

Specialized testing objects build upon base type test values, but 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 a mixture of various characters, a string with pernicious file modes, and a string with a pernicious printf format.
- File descriptor (based on integer base type): includes -1; MAXINT; 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.

In all cases it is desired to include test values that encompass both valid and exceptional values for their use in the POSIX API so that one correctly handled exception would not mask other incorrectly handled exceptions. For example, in several operating systems a buffer length of zero causes the `write` function to return normally, regardless of the values of the other parameters, whereas calling `write` with a valid non-zero length and an invalid NULL buffer pointer causes a segmentation violation. Without the inclusion of the valid non-zero value in the length data type, this robustness failure would not be uncovered.

Data Type	# of Funcs.	# Test Values
string	71	9
buffer	63	15
integer	55	16
bit masks	35	4
file name	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
directory pointer	3	7
timeout	3	5
size	2	9

Table 1. Data types used in POSIX testing.
Only 20 data types sufficed to test 233 POSIX functions.

The above test values by no means represent all possible exceptional conditions. Future work will include surveying published lists of test values and studying ways to 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 using a set of shell scripts. In other words, the various constructors and destructors for the particular test values of interest can be automatically extracted and placed in a single 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.

3.2. Test generation

The simplest Ballista operating mode generates an exhaustive set of test cases that spans the cross-product of all test values for each module input parameter. 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.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

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. For most functions the number of tests is less than 5000, which corresponds to less than a minute of test time. However, for seven POSIX functions the number of tests is larger than 5000, so combinations of parameter test values are (deterministically) pseudo-randomly selected up to an arbitrary limit of 5000 test cases. A comparison of the results of pseudo-random sampling to actual exhaustive testing showed that the results can be expected to be very close (within less than a one percentage point difference) with respect to failure rate information. In the future a more sophisticated sampling approach such as concentrating on varying pairs and triples of parameters (*e.g.*, based on testing pairs and triples of parameters as in [13]) could be used, although the pseudo-random approach appears to suffice for current purposes.

4. Experimental Results

The Ballista POSIX robustness test suite has been ported to the ten operating systems listed in Table 2 with no code modifications. On each 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 and LynxOS these were GNU C tools).

4.1. Results of testing POSIX operating systems

Table 2 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 OSs that supported all 233 POSIX functions to 54,996 for HP-UX. One function in Irix, `munmap`, suffered Catastrophic failures causing a system crash that required manual rebooting. The `mmap` function in HP-UX caused a system panic followed by automatic reboot. Similarly the `setpgid` function in LynxOS caused a system crash. All OSs had relatively few Restart failures (task hangs).

The main trend to notice in Table 2 is that only 37% to 58% of functions exhibited no robustness failures under testing. This indicates that, even in the best case, about half the functions had at least one robustness failure. (Ballista does not currently test for Silent or Hindering failures, but these might be present in functions that are indicated to be failure-free in Table 2.)

4.2. Normalizing robustness test results

While it is simple to list the number of test cases that produced different types of robustness failures, 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 multi-parametered functions. Instead, the number of failures are reported as a percentage of tests on a per-function basis. Figure 5 graphs the per-function percent of failed tests cases for the 233 functions tested in Digital

System	POSIX Functions Tested	Fns. with Catastr. Failures	Fns. with Restart Failures	Fns. with Abort Failures	Fns. with No Failures	Number of Tests	Abort Failures	Restart Failures	Normalized Failure Rate
AIX 4.1	186	0	4	77	108 (58%)	64,009	11,559	13	9.99%
Digital Unix 4.0	233	0	2	124	109 (47%)	92,658	18,316	17	15.07%
FreeBSD 2.2.5	175	0	4	98	77 (44%)	57,755	14,794	83	20.28%
HP-UX B.10.20	186	1	2	93	92 (49%)	54,996	10.717	7	13.05%
Irix 6.2	226	1	0	94	131 (58%)	91,470	15,086	0	12.62%
Linux 2.0.18	190	0	3	86	104 (55%)	64,513	11,986	9	12.54%
LynxOS 2.4.0	223	1	0	108	114 (51%)	76,462	14,612	0	11.89%
NetBSD 1.3	182	0	4	99	83 (46%)	60,627	14,904	49	16.39%
QNX 4.24	206	0	4	127	77 (37%)	74,893	22,265	655	22.69%
SunOS 5.5	233	0	2	103	129 (55%)	92,658	15,376	28	14.55%

Table 2. Summary of robustness testing results. The number of test cases and functions tested varies with the fraction of POSIX calls supported by the system. Some functions have multiple types of failures. Catastrophic failures are not included in failure rate because of difficulties retaining recorded data across system crashes.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

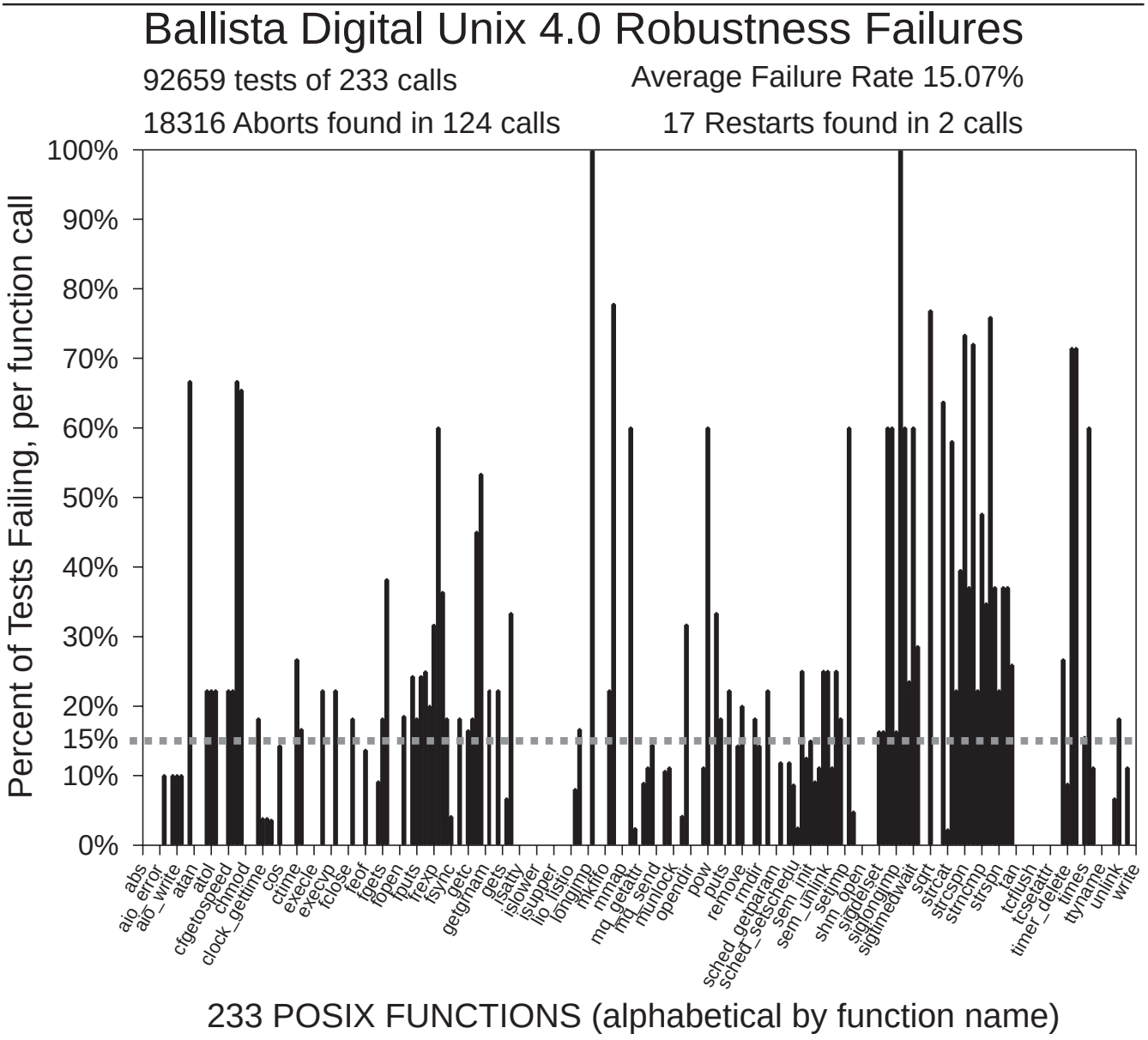

Figure 5. Normalized failure rates for 233 POSIX functions on Digital Unix 4.0.

Unix 4.0. 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 Figure 5 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 recovered from such a situation.) On the other hand, most of the remaining functions could plausibly return error codes rather than failing for a broad range of exceptional inputs.

4.3. 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, and it might be useful to compare the robustness results for different operating systems. (It should be realized, however, that these results measure only

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

one aspect of robustness, and do not take into account system-level robustness architectures that might adequately deal with function-call level robustness failures.)

Figure 6 shows normalized failure rates for the ten OS implementations tested. 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. Thus, it provides the notion of an unweighted exposure to robustness failures on a per-call basis. So, the normalized failure rates represent a failure probability metric for an OS implementation assuming a flat input distribution selected from the Ballista test values. As such, they are probably most useful as relative measures of the robustness of an entire API. If one particular application or set of applications is to be used with the API, it might be better to weigh the failure rates according to the frequency of use of each function call — but applications may vary so widely in their use of POSIX calls that it seems inappropriate to attempt such a weighting for generic robustness failure results such as those presented here.

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 opportunities for eliciting faulty responses due to one or more robustness problems within the software being tested. From a user's perspective it is not really important how many problems are within a piece of COTS software if the number is other than zero. What is important is the likelihood of triggering a failure response due to such a robustness problem.

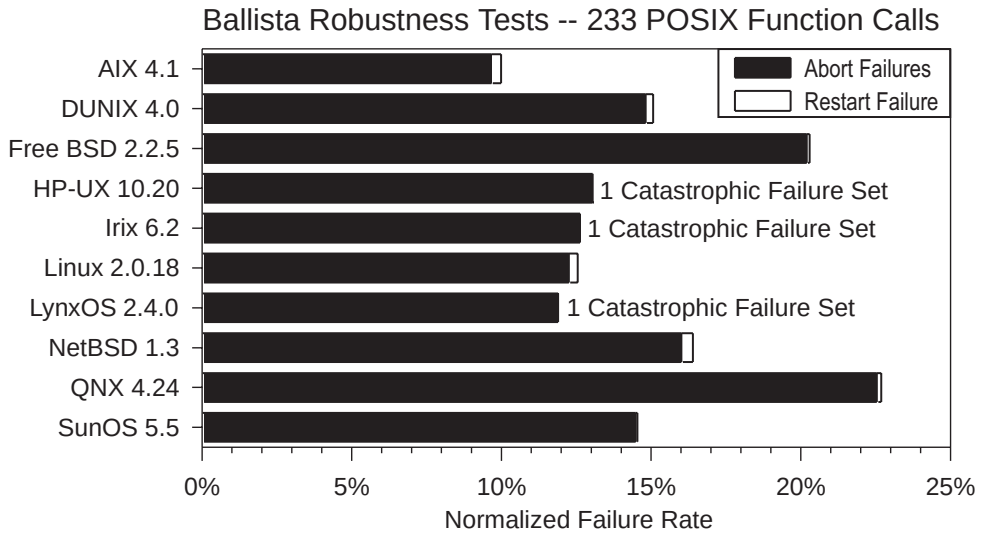

Figure 6. Normalized failure rates for ten POSIX operating systems. Aggregate failure rates range from a low of 9.99% for AIX to a high of 22.69% for QNX. The figures were obtained by taking the arithmetic mean of the percentage of test cases resulting in failure across all the functions tested on each particular operating system. Three implementations each had one function that caused catastrophic failures — an OS crash from user mode.

5. Generic applicability of the methodology

The successful experience of using the Ballista methodology to test 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.

5.1. Scalability

Testing a new software module with Ballista often incurs no incremental 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 tests. For example, once test values 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 for buffer and length would have already been defined if these functions were tested after tests had been created for functions such as `memcpy` which have nothing to do with the file system. 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).

The thoroughness of testing is enhanced by the combinatorial aspect of breaking test cases down by data type. Testing combinations of data types is performed automatically, generating potentially thousands of distinct composite tests, but requiring only up to a dozen or so specifically defined test values for each data type. Although the details are beyond the scope of this

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

paper, the experimental data demonstrate that in a number of instances the sweeping of the search space with this combinatorial testing finds robustness failures that would have been missed with simpler single-parameter or double-parameter search strategies.

Finally, the number of tests to be run can be limited using pseudo-random sampling (or, if necessary, other more sophisticated techniques such as pair-wise testing as mentioned earlier). Thus, the execution time of testing a module can be kept low even with a large search space as long as a statistical sampling of behavior is sufficient. Testing of the POSIX API takes approximately three hours on a workstation for the data presented in this paper.

5.2. Portability

The Ballista approach has proven portable across platforms, and promises to be portable across applications. The Ballista tests have been ported to ten 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 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. This claim of portability is further supported by initial experiences in applying Ballista to a high-level simulation backplane API, although it is too early to make definitive statements about that application.

5.3. Testing cost

One of the biggest unknowns when embarking upon a full-scale implementation 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. The key observation is that in a typical program there are 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 the number of functions.

Although we are just starting to apply the Ballista testing approach to an object-oriented API, it seems likely that it will be successful there as well. The effort involved in preparing for automated testing should 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 not adversely affect the cost and effectiveness of the Ballista testing methodology.

5.4. Effectiveness and system state

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 failures. Part of this may be that such problems are unexpectedly 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 can be set by the constructor for each test value. For example, a file descriptor test value might create a particular file with associated permissions, access mode, and contents with its constructor (and, erase the file with its destructor). Thus, a single test case can replace a sequence of tests that would otherwise have to be executed to create and test a function executed in the context of a particular system state. In other words, the end effect of 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 an explicit sequence of calls in the form of per-function test scaffolding.

A high emphasis has been placed on reproducibility within Ballista. In essentially every case checked, it was found that 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 the details of how data structures have been allocated in memory).

The only situation in which Ballista results have been found to lack reproducibility is in Catastrophic failures (complete system crashes, not just single-task crashes). On Irix 6.2, system crashes were caused by the execution of a

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany

single function call. On Digital Unix 3.2 with an external swap partition mounted, 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 a time-delayed manifestation of the error. In all cases, however, robustness failures were reproducible by rerunning a few tests within the Ballista test harness, and could be recreated under varying system loads including otherwise idle systems.

The current state of Ballista testing is that it searches for robustness faults using heuristically created test cases. Future work will 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. It is not the goal of robustness testing to predict software failure rates under normal operating conditions. Rather, the goal is to find failures caused by exceptional situations, which may correspond to a lack of robustness in the face of the unexpected.

6. Conclusions

The Ballista testing methodology can automatically assess the robustness of software components to exceptional input parameter values. Data taken on 233 function calls from the POSIX API demonstrate that approximately half the functions tested exhibited robustness failures. Additionally, a significant fraction of tests ranging from a low of 9.99% on AIX to a high of 22.69% on QNX resulted in robustness failures.

More important than the particulars of the OS tests executed are the results of using the Ballista testing methodology on a full-size application. The Ballista methodology was found to be inexpensive to implement because test database development effort 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 92,659 tests). The testing technique was demonstrated to be portable across systems while requiring no special-purpose fault injection support, and the experience suggests that is should be applicable to other significant APIs.

A specific advantage of the Ballista approach is the ability to set a rich system state before executing a test case, obviating the need for sequences of tests to produce a significant number of robustness failures.

Future work on Ballista will include increasing the number and variety of test values and data types supported, adding automatic generation of patterned and random tests in addition to the current heuristic-based testing, and applying the testing methodology to other application areas.

7. Acknowledgments

This research was sponsored by 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.

8. References

[1] Lions, J. (Chair), *Ariane 5 Flight 501 Failure*, European Space Agency, Paris, July 19, 1996. *http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html*, Accessed Dec. 4, 1997.

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

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

[4] *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.

[5] Horgan, J. & A. Mathur, "Software Testing and Reliability", In: Lyu, M. (Ed.) *Handbook of Software Reliability Engineering* , IEEE Computer Society, 1995, pp. 531-566.

[6] Tsai, T., & 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, Sept. 20-22 1995, Springer-Verlag, pp. 26-40.

[7] Segall, Z. *et al.*, "FIAT - Fault Injection Based Automated Testing Environment," *Eighteenth Intl. Symp. on Fault-Tolerant Computing (FTCS-18)*, Tokyo, June 27-30 1988, IEEE Computer Society, pp. 102-107.

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

[9] Chio, B., R. DeMillo, E. Krauser, A. Mathur, R. Martin, A. Offut, H. Pan, & E. Spafford, "The Mothra Toolset," *Proc. of Twenty-Second Hawaii Intl. Conf. on System Sciences*, Jan. 3-6 1989, pp. 275-284 volume 2.

[10] Miller, B., *et al.*, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, Computer Science Technical Report 1268, Univ. of Wisconsin-Madison, 1995.

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

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

[13] Cohen, D., S. Dalal, M. Fredman & G. Patton, "The AETG System: an approach to testing based on combinatorial design," *IEEE Trans. on Software Engr.*, 23(7), July 1997, pp. 437-444.

[14] Biyani, R. & P. Santhanam, "TOFU: Test Optimizer for Functional Usage," *Software Engineering Technical Brief,* 2(1), 1997, IBM T.J. Watson Research Center.

Published in the Proceedings of FTCS'98
June 23-25, 1998 in Munich, Germany