11

INTERFACE ROBUSTNESS TESTING: EXPERIENCES AND LESSONS LEARNED FROM THE BALLISTA PROJECT

Philip Koopman, Kobey DeVale, and John DeVale

11.1. INTRODUCTION

When the Ballista project started in 1996 as a 3-year DARPA-funded research project, the original goal was to create a Web-based testing service to identify robustness faults in software running on client computers via the Internet. Previous experience suggested that such tests would find interesting problems but it was unclear how to make robustness testing scalable to large interfaces. A major challenge was finding a way to test something as large and complex as an operating system (OS) application programming interface (API) without having to resort to labor-intensive manual test construction for each API function to be tested. In the end, a scalable approach was found and was successfully applied not only to operating system APIs but several other nonoperating system APIs as well.

The robustness testing methodology Ballista is based upon using combinational tests of valid and invalid parameter values for system calls and functions. In each test case, a single software module under test (or MuT) is called once. An MuT can be a stand-alone program, function, system call, method, or any other software that can be invoked with a procedure call. (The term MuT is similar in meaning to the more recent term dependability benchmark target [DBench 2004].) In most cases, MuTs are calling points into an API. Each invocation of a test case determines whether a particular MuT provides robust exception handling when called with a particular set of parameter values. These parameter values, or *test values,* are drawn from a pool of normal and exceptional values based on the data type of each argument passed to the MuT. Each test value has an associated *test object,* which holds code to create and clean up the related system state for a test (for example, a file handle test object has code to create a file, return a file handle test value, and subsequently delete the file after the test case has been executed). A test case, therefore, consists of the name of the MuT and a tuple of test values that are passed as parameters [i.e., a test case would be a procedure call of the form: returntype MuT_name(test_value1, test_value2, . . . , test_valueN)]. Thus, the general Ballista approach is to test the robustness of a single call to an MuT for a single tuple of test values, and then iterate this process for multiple test cases that each have different combinations of valid and invalid test values.

The Ballista testing method is highly scalable with respect to the amount of effort required per MuT, needing only 20 data types to test 233 POSIX functions and system calls [Kropp 1998]. An average data type for testing the POSIX API has 10 test cases, each having 10 lines of C code, meaning that the entire test suite required only 2000 lines of C code for test cases (in addition, of course, to the general testing harness code used for all test cases and various analysis scripts). Robustness testing of other APIs was similarly scalable in most cases.

This chapter describes the key design tradeoffs made during the creation and evolution of the Ballista robustness testing approach and associated toolset. It summarizes some previous results to draw together high-level lessons learned. It also contains previously unpublished robustness testing results, and key intermediate results that influenced the course of the project.

11.2. PREVIOUS ROBUSTNESS TESTING WORK

Although the Ballista robustness testing method described here is a form of software testing, its heritage traces back not only to the software testing community but also to the fault tolerance community as a form of software-based fault injection. Ballista builds upon more than 15 years of fault injection work at Carnegie Mellon University, including [Schuette 1986], [Czeck 1986], [Barton 1990], [Siewiorek 1993], [Dingman 1995], and [Dingman 1997], and makes the contribution of attaining scalability for cost-effective application to a reasonably large API. In software testing terms, Ballista tests responses to exceptional input conditions; they are sometimes called “dirty” tests, which involve exceptional situations, as opposed to “clean” tests of correct functionality in normal situations. The Ballista approach can be thought of as a “black box,” or functional testing technique [Bezier 1995] in which only functionality is of concern, not the actual structure of the source code. Among other things, this permits Ballista to be used on already-compiled software for which no source code is available. However, unlike the usual testing approaches, Ballista is concerned with determining how well a software module handles exceptions rather than with functional correctness.

Some people only use the term robustness to refer to the time between operating system crashes under some usage profile. However, the authoritative definition of robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [IEEE 1990].” This expands the notion of robustness to be more than just catastrophic system crashes, and encompasses situations in which small, recoverable failures might occur. Although robustness under stressful environmental conditions is indeed an important issue, a desire to attain highly repeatable results has led the Ballista project to consider only robustness issues dealing with invalid inputs for a single invocation of a software module from a single execution thread. It can be conjectured, based on anecdotal evidence, that improving exception handling will reduce stress-related system failures, but that remains an open area of research.

Robustness testing in general has, of course, existed for many years, including manual testing to overload systems, but most software product development efforts have very limited time and budget for testing, often leading to testing focused on correct functionality rather than robustness. An automated robustness testing approach could make it practical to obtain much better robustness information within a limited testing budget.

An early method for automatically testing operating systems for robustness was the development of the Crashme program [Carrette 1996]. Crashme operates by writing random data values to memory, then spawning large numbers of tasks that attempt to execute those random bytes as concurrent programs. Although many tasks terminate almost immediately due to illegal instruction exceptions, on occasion a single task or a confluence of multiple tasks can cause an operating system to fail. The effectiveness of the Crashme approach relies upon serendipity; in other words, if run long enough Crashme may eventually get lucky and find some way to crash the system.

Similarly, the Fuzz project at the University of Wisconsin has used random noise (or "fuzz") injection to discover robustness problems in operating systems. That work documented the source of several problems [Miller 1990], and then discovered that the problems were still present in operating systems several years later [Miller 1998]. The Fuzz approach tested specific OS elements and interfaces (as opposed to the completely random approach of Crashme), although it still relied on random data injection.

Other work in the fault-injection area has also tested limited aspects of robustness. The FIAT system [Barton 1990] uses probes placed by the programmer to alter the binary process image in memory during execution. The FERRARI system [Kanawati 1992] is similar in intent to FIAT, but uses software traps in a manner similar to debugger break points to permit emulation of specific system-level hardware faults (e.g., data address lines, condition codes). Xception [Carreira 1998] similarly performs software injection of faults to emulate transient hardware faults. The FTAPE system [Tsai 1995] injects faults into a system being exercised with a random workload generator by using a platform-specific device driver to inject the faults. MALFADA-RT [Rodriguez 2002] uses software to perform bit flips on memory values and run-time parameters. Although all of these systems have produced interesting results, none was intended to quantify robustness on the scale of an entire OS API.

While the hardware fault tolerance community has been investigating robustness mechanisms, the software engineering community has been working on ways to implement robust interfaces. As early as the 1970s, it was known that there are multiple ways to handle an exception [Hill 1971, Goodenough 1975]. The two methods that have become widely used are the signal-based model (also known as the termination model) and the error-return-code model (also known as the resumption model).

In an error-return-code model, function calls return an out-of-band value to indicate that an exceptional situation has occurred (for example, a NULL pointer might be returned upon failure to create a data structure in the C programming language). This approach is the supported mechanism for creating portable, robust systems in the POSIX API [IEEE 1993, lines 2368–2377].

On the other hand, in a signal-based model, the flow of control for a program does not address exceptional situations, but instead describes what will happen in the normal case. Exceptions cause signals to be "thrown" when they occur, and redirect the flow of control to separately written exception handlers. It has been argued that a signal-based approach is superior to an error-return-code approach based, in part, on performance concerns, and because of ease of programming [Gehani 1992, Cristian 1995]. Some APIs, such as CORBA, [OMG 1995] standardize this approach, whereas other APIs, such as POSIX, do not standardize it to a degree that it is useful in creating robust systems.

11.2.1. Previous Ballista Publications

Work on Ballista has previously been reported in several publications. Koopman and coworkers [Koopman 1997] described portable, although not scalable, robustness testing applied to six POSIX functions and the idea of using data types for scalability. They also introduced the CRASH severity scale, discussed later. Kropp and coworkers [Kropp 1998] later demonstrated that the ideas proposed in the previous paper actually could be scaled up to test a full-size API in the form of POSIX operating systems. Koopman and coworkers [Koop 1998] described the overall Ballista approach being developed, emphasizing the approaches to scalability and ability to quantify robustness results. The paper by DeVale and coworkers [DeVale 1999] was the initial description of the Ballista test harness approach, involving a testing server connected via the Internet to a testing client. (Ballista later evolved into a stand-alone test platform.) Koopman and DeVale [Koopman 1999] then presented refined testing results for POSIX, including subtracting the effects of nonexceptional tests and "silent" test faults. This paper also found that even independently developed versions of operating systems and C libraries frequently contain identical robustness defects. A subsequent paper by Koopman and DeVale [Koopman 2000] was a more thorough treatment of POSIX operating system robustness results. Pan and coworkers [Pan 1999] studied early Ballista testing results and concluded that most robustness failures observed were the result of either a single parameter value or a tuple of related parameter values rather than the result of complex interactions among parameter values. Fernsler and Koopman [Fernsler 1999] reported robustness testing results of the HLA distributed simulation backplane. Shelton and Koopman [Shelton 2000] reported Ballista testing results on several versions of the Microsoft Windows operating system. Pan and coworkers [Pan 2001] reported results of testing and hardening CORBA ORB implementations. DeVale and Koopman [DeVale 2001] described the exception handling performance of the STDIO library in the POSIX operating system, and additionally examined the robustness and performance of the SFIO library, which was designed to provide a more robust interface. They later [DeVale 2002] described techniques to reduce run-time overhead of robustness checks to a few percent, even for quite fine-grained checking, and additionally described a case study in which it was shown that some programmers wrote code that diverged significantly from their expectations of robustness.

11.3. EVOLUTION OF BALLISTA

The Ballista approach to robustness testing was built upon many years of precursor work on fault injection and software testing. It evolved and was further refined by many contributors over a number of years. Its refinement was very much a product of evolution and learning, as are many other research efforts.

A note on terminology is in order. We use the historical terminology developed as Ballista evolved rather than attempting a mapping of the rather imprecise understanding we had at the time to more precise current terminology. Our understanding of the problem space evolved only after many years of work, as did the terminology used by researchers in this area.

11.3.1. Early Results

The seeds of Ballista were sown by a combination of fault injection into parameter values using sentries [Russinovich 1993], and a switch from run-time fault injection to designing test cases for APIs. The initial work using sentries injected faults into procedure calls on the fly, inverting bits in data values. The general assumption was that software would display robustness vulnerabilities only in complex situations, and those vulnerabilities would depend on system state as well as, perhaps, timing situations. This corresponded with the collective wisdom from researchers in industry and universities that mature, commercial software would only be vulnerable to complex, subtle problems. Nobody really expected single-line tests to be able to crash mature, stable systems, so looking for such vulnerabilities systematically was not a primary objective at first. Even so, researchers at Carnegie Mellon were finding system-killer problems in high-integrity avionics systems [Dingman 1995] (the Advanced Spaceborne Computer module in this case).

11.3.2. Scalability Via Generic Functions

The detection of a system-killer call in deployed high-integrity software gave rise to the question as to whether such vulnerabilities were common in other software. Finding out whether software robustness was a significant issue in deployed code bases required testing significant APIs. The problem in doing that was one of scalability. The approach used to that point was essentially a type of software testing, and software testing effort is generally proportional to the number and complexity of functions being tested. Thus, robustness testing of large, complex APIs was impractical unless some scalable approach could be found.

Mukherjee and Siewiorek [Mukherjee 1997] tried to attain scalability by creating generic templates based on operations commonly performed by functions (e.g., reference an object before it is created, delete an active object, allocate objects until resources are exhausted). The approach was successful in creating 252 tests for Unix file operations by exercising combinations of test values, but did not fundamentally solve the scaling problem because effort was still required to map each function being tested to a generic hierarchy.

11.3.3. The Start of Ballista

The Ballista project per se was started under a DARPA research contract in the fall of 1996. This project proposed setting up a testing server that would connect to clients anywhere on the Internet and direct robustness tests. Although Ballista was always intended to be a generic API robustness testing technique, the first experimental results were obtained for POSIX operating systems because of the ready availability of a large number of implementations available for comparison purposes throughout the Carnegie Mellon campus.

Initial work on Ballista involved hand-creating tests based on the code used in [Dingman 1995] to see which test values would be most effective in exposing robustness failures. Within a few months, the team had confirmed (as reported in [Mukherjee 1997]) that machine crashes were not the only robustness failures that might matter. There were some failures that caused task hangs, some tests that should have failed but did not (e.g., a request to allocate 3 GB of memory on a system with fewer than 32 MB, but the return values indicated that memory was allocated as requested), and many tests that resulted in a task abnormal termination. One incident in particular that pointed out the need for broadening the measurement of robustness was a microkernel function in the QNX operating system (send/receive) that abnormally terminated the calling task by generating a signal, even though the user manual for that function unambiguously stated that an error code (EFAULT) should be returned to avoid incurring a segment violation signal.

Although initial results suggested that robustness failures seemed to be possible in a variety of commercial-grade software implementations, scalability remained a problem. Scalability was finally obtained through a combination of two approaches: creating a scalable oracle, and automated, scalable test-case generation.

11.3.4. The CRASH Scale

Normally, testing is the execution of software with a particular workload and comparison of the results of execution with an "oracle" to determine if the test matches the expected outputs (a "pass") or differs from the expected outputs (a "fail"). Generating random inputs to a function to test it could always be done for scalability. Unfortunately, generating detailed results for what should happen when executing a function did not scale well. (This problem with random testing is well known and discussed, for example, in [Duran 1984] and [Thevenod-Fosse 1991].)

The solution to this dilemma was constraining the properties that the oracle predicted to narrowly reflect robustness properties, rather than attempting to represent functionality in a broader sense. In particular, the Ballista approach to testing used the following fixed properties for every function being tested:

1. **A test should never crash the system.** Violation of this property is a "catastrophic" failure. System-killer tests had to be manually detected, but in most cases they were infrequent and readily detected by the person administering the tests. (For example, HP-UX machines had a "heartbeat" LED. System crashes amounted to "cardiac arrest" when that LED went dead.) Not many system killers were found, but for most systems they were treated by vendors as a high-priority problem.
2. **A test should never hang.** This is a "restart" failure, requiring a restart of the task that failed so as to recover the system. Monitoring for restart failures was automated by using a time-out in the test harness, and killing tasks that took more than several seconds to execute (normally, each test executed within a few hundred milliseconds, depending on disk drive usage). Task hangs were uncommon except in VxWorks testing, where they were almost as common as task crashes.
3. **A test should never crash the testing task.** This is an "abort" failure. Monitoring for abort failures was automated by catching signals generated by tasks via the test harness. Task crashes were common in most systems tested.
4. **A test should report exceptional situations.** Lack of reporting an exception when one should be reported is a "silent" failure. Some systems accepted exceptional inputs with no indication of error—neither a signal nor an error code. Automating detection of silent failures would involve creating an oracle that could predict whether a test case was an exception or not, which is impractical, so this was not done in the run-time test harness. Multiversion comparison techniques did allow us to deduce how frequent this situation was for POSIX operating systems [Koopman99]. Estimated silent failure rates ranged from 6% to 19% across POSIX operating systems tested. (This is a concept distinct from an injected fault that is not activated; in the case of a silent failure, an exceptional value was acted upon and failure to report it is clearly erroneous.)
5. **A test should not report incorrect error codes.** This is a "hindering" failure. Manual analysis of small-scale tests indicated that even when error codes were generated, they were often incorrect or misleading in some operating systems, but usu-

ally correct in others [Koopman 1997]. However, as with Silent failures, no practical way has yet been found to automate the analysis of this type of failure.

The names of the above five failure categories form the acronym for the "CRASH" severity scale [Koopman 1997], although Ballista is only able to automatically detect the "C," "R," and "A" type failures without additional data analysis. However, those three types of failures alone provide valuable robustness measurement results, and make Ballista scalable via use of a simple, automated oracle function of "doesn't crash, doesn't hang."

There are two additional possible outcomes of executing a test case. It is possible that a test case returns an error code that is appropriate for invalid parameters forming the test case. This is a case in which the test case passes; in other words, generating an error code is the correct response. Additionally, in some tests the MuT legitimately returns no error code and successfully completes the requested operation. This happens when the parameters in the test case happen to be all valid, or when it is unreasonable to expect the OS to detect an exceptional situation (such as pointing to an address past the end of a buffer, but not so far past as to go beyond a virtual memory page or other protection boundary). For operating systems, approximately 12% of all executed test cases were nonexceptional tests [Koopman 1999].

It is important to note that the concepts of "silent" failures and "nonexceptional" tests differ significantly from the notion of a "nonsignificant test" (e.g., as discussed in [Arlat 1990]). Nonsignificant tests are a result of an injection of a fault that is not activated during test execution (for example, inverting a bit in a memory variable that is never read by a program, or corrupting an instruction that is never executed). In contrast, a silent failure is one in which the test program did in fact involve the corrupted or exceptional value in a computation but failed to detect that a problem occurred. A nonexceptional test is one in which a fault or exception was not actually injected at all, meaning the MuT's failure to detect that nonexistent error is correct operation.

11.3.5. Test Cases Based on Data Types

Beyond classifying results, it was important to have a scalable way to generate test cases. This was done by organizing tests by *data types* of parameters rather than the functionality of the MuT itself. Many APIs use far fewer types than functions (for example, POSIX requires only 20 data types as arguments for 233 functions and system calls). Thus, Ballista completely ignores the purpose of a function, and instead builds test cases exclusively from data type information. This is done by creating a set of test values for each data type that is likely to contain exceptional values for functions. For example, an integer data type would contain –1, 0, 1, the maximum permitted integer value and the minimum permitted integer value. Although some of these values are likely to be nonexceptional for many MuTs, it was also likely that some of these values would be exceptional.

Given a dictionary of "interesting" test values for each data type, tests are composed by using all combinations of test values across all the parameters of the function (shown in Figure 11.1). Ballista thus ends up using an idea similar to category-partition testing [Ostrand 1988].

Although Ballista evolved into a general-purpose API robustness testing tool, the initial application of Ballista was for a set of 233 POSIX functions and calls defined in the IEEE 1003.1b standard [IEEE93] ("POSIX.1b" or "POSIX with real-time extensions with C language binding"). All standard calls and functions supported by each OS implementation 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().

For each POSIX function tested, an interface description was created with the function name and type information for each argument. In some cases, a more specific type was created to result in better testing (for example, a file descriptor might be of type int, but was described to Ballista as a more specific file descriptor data type).

As an example, Figure 11.1 shows test values used to test write(int filedes, const void *buffer, size_t nbytes), which takes parameters specifying a file descriptor, a memory buffer, and a number of bytes to be written. Because write() takes three parameters of three different data types, Ballista draws test values from separate test objects established for each of the three data types. In Figure 11.1, the arrows indicate that the particular test case being constructed will test a file descriptor for a file that has been opened with only read access, a NULL pointer to the buffer, and a size of 16 bytes. Other combinations of test values are assembled to create other test cases. In the usual case, all combinations of test values are generated to create a combinatorial number of test cases. For a half-dozen POSIX calls, the number of parameters is large enough to yield too many test cases for exhaustive coverage within a reasonable execution time. In these cases, pseudorandom sampling is used. (Based on a comparison to a run with exhaustive searching on one OS, sampling 5000 test cases gives results accurate to within 1 percentage point for each function [Kropp 1998].)

11.3.6. Test Case Construction: Parameter Constructors and Destructors

Even with a dictionary of test values, there were still difficulties in sequencing the operations associated with some tests. For example, tests requiring access to files might rely

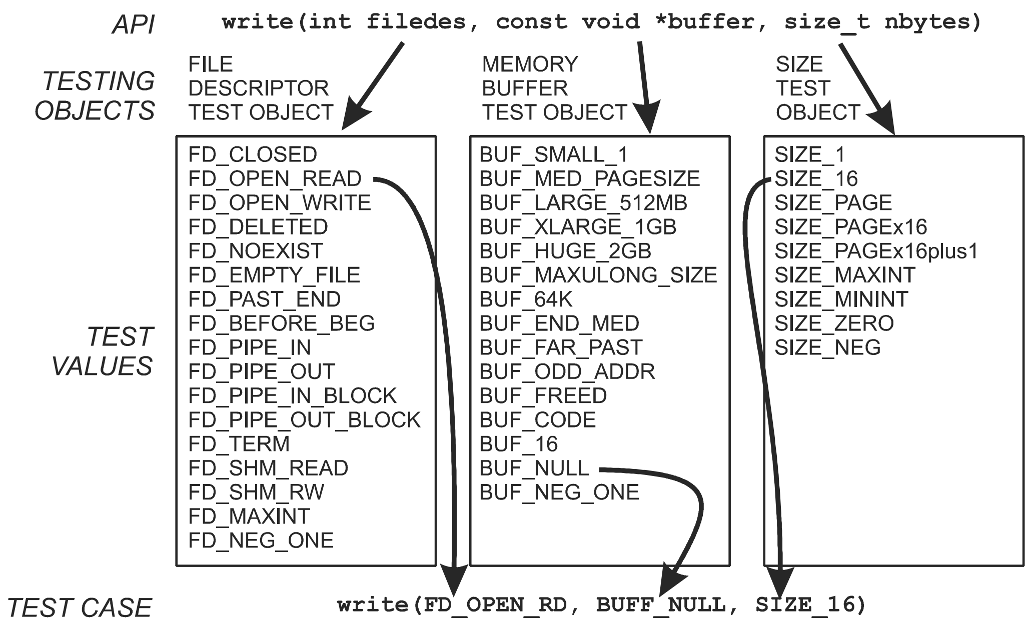

Figure 11.1. Ballista test case generation for the write() function. The arrows show a single test case being generated from three particular test values; in general, all combinations of test values are tried in the course of testing.

upon a set of files with various properties precreated by the test harness. However, such an approach was inefficient because, typically, only one or two files out of the pool were needed for any particular test. Moreover, sometimes multiple copies of a particular preinitialized data structure or file were needed to execute a test when a MuT's parameter list contained more than one instance of a particular data type. An additional issue was ensuring that any data structures or files that were created for a test were properly cleaned up, to avoid unintentionally running out of memory or filling up a disk as tests were executed. This was solved by associating constructors and destructors with test values.

Each test value (such as FD_OPEN_READ in Figure 11.1) is associated with a triplet of test object code fragments that are kept in a simple database in the form of a specially formatted text file, comprising a test object associated with each test value. The first fragment for each test object, the *create* fragment, is a constructor that is called before the test case is executed (it is not literally a C++ constructor but, rather, a code fragment identified to the test harness as constructing the instance of a test value). The create code may simply return a value (such as a NULL), but may also do something more complicated that initializes system state. For example, the create code for FD_OPEN_READ creates a file, puts a predetermined set of bytes into the file, opens the file for reading, then returns a file descriptor for that file.

Create code fragments are not permitted to release resources. As an example, a create code fragment to set up system state for a "file created, closed, and then deleted" test value would create a file, but not delete it. This prevents situations in which successive invocations of create code fragments reclaim and reuse resources. Consider an example situation in which a first file handle points to a closed, deleted file, while a second file handle points to a file open for reading. If the first file handle is released by closing the file while setting up the first test object, it is possible that the second test object will reuse system resources for the first file handle when preparing the second file handle. The result could be the first file handle referring to a file open for reading (i.e., pointing to the same file state as the second file handle). To avoid unintentional sharing via recycling of resources, no system resources are released until after all system resources required have been obtained.

The second fragment in each test object performs a commit function. This modifies the system state to release resources before a test case is executed. All create code fragments are executed for a test case before any commit code fragments are executed. This avoids unintended sharing of reclaimed resources within a single test case. Examples in which this commit code are required include releasing allocated memory, closing file handles, deleting files, and releasing federation members in a distributed system.

The third code fragment for each test value is a destructor that deletes any data structures or files created by the corresponding constructor (for example, the destructor for FD_OPEN_READ closes and deletes the file created by its matching constructor). Tests are executed from within a test harness by having a parent task fork a fresh child process for every test case. The child process first executes create code fragments for all test values used in a selected test case, then executes all commit code fragments, then executes a call to the MuT with those test values, then calls destructors for all the test values used. Special care is taken to ensure that any robustness failure is a result of the MuT, and not attributable to the constructors or destructors themselves. Functions implemented as macros are tested using the same technique, and require no special treatment.

The test values used in the experiments were a combination of values suggested in the testing literature (e.g., [Marick 1995]), values that found problems in precursor research, and values selected based on personal experience. For example, consider file descriptor test values. File descriptor test values include descriptors to existing files, negative one, the maximum integer number (MAXINT), and zero. Situations that are likely to be exceptional in only some contexts are tested, including file open only for read and file open only for write. File descriptors are also tested for inherently exceptional situations such as file created and opened for read, but then deleted from the file system without the program's knowledge.

The guideline for test value selection for all data types were to include, as appropriate: zero, negative one, maximum/minimum representable values, null pointers, pointers to nonexistent memory, lengths near virtual memory page size, pointers to heap-allocated memory, files open for combinations of read/write with and without exceptional permission settings, and files/data structures that had been released before the test itself was executed. While creating generically applicable rules for thorough test value selection remains an open subject, this experience-driven approach was sufficient to produce useful results and uncover common robustness vulnerabilities across many different systems.

It is important that both valid as well as invalid parameters be used for each parameter of a MuT. This is partly to help in identifying which bad parameter is actually responsible for a robustness failure. If a particular MuT has a large number of failures, those failures often are insensitive to the value of most parameters. The easiest way to pin down which parameters really matter is to find individual test cases in which all parameters but one or two are valid. A more subtle reason for including valid parameters is that many MuTs have partial robustness checking. For example, a module that takes two pointers as inputs might check the first pointer for validity but not the second pointer. If the first pointer is invalid for all tests, then the module will catch that and mask the fact that the MuT is vulnerable to invalid second parameter values. To more fully test this MuT, at least some test cases must include a valid first pointer (the one that the MuT checks) and an invalid second pointer (the one that the MuT does not check).

An important benefit derived from the Ballista testing implementation is the ability to automatically generate the source code for any single test case the suite is capable of running. In many cases that are only a dozen lines or fewer of executable code in size, these short programs contain the constructors (create code fragments and commit code fragments) for each parameter, the actual function call, and destructors. These single-test-case programs have been used to reproduce robustness failures in isolation for use by OS developers and to verify test result data.

11.3.7. Repeatability

One concern with early Ballista results was the repeatability of individual tests. For example, an early implementation included pseudorandom values for some data types, including pointers. To alleviate concerns that random pointers were valid but incorrect (e.g., pointing to inappropriate locations in the heap or stack, which could be difficult to check for correctness at run time in any software system), those tests were eliminated in favor of deterministic test values. The general characteristics of the test results were unchanged by the elimination of pseudorandom variables.

An additional issue arose because Ballista ran tests in large batches rather than one at a time for reasons of efficiency. It was possible that problems created by one test might carry over to another test, meaning that it was possible that problems being seen were not the result of a test being run but, rather, the result of activating a fault created dozens or even thousands of tests earlier. This concern was initially addressed by adding a capability to the Ballista test harness of generating a stand-alone single-test program for any desired test. An experiment involving generating and executing every single stand-alone test compared to batch testing for Digital Unix revealed that almost all cases (well above

99%) created programs that correctly reproduced robustness problems seen in large test runs. (What happened in the few other cases is unclear, but the difference was attributed to the fact that the large test harness had a much different execution environment than the small single-test programs.) The main benefit of this approach was the availability of automatically generated bug-report programs to help vendors identify and correct robustness defects. The biggest problem was that it was very time-consuming to generate, compile, and execute more than 100,000 individual test programs to confirm results. And even if individual test programs were executed, it still might be the case that there was residual system damage from one stand-alone test to the next that caused carryover of problems. This batch approach worked well for systems with hardware memory protection, but repeatability was an issue revisited for each set of tests conducted.

A better approach for ensuring repeatability than executing tests one at a time was found in the form of reordering tests. Two complete sets of tests can be executed on an identical platform, but in reverse testing order. If the result of each and every test matches from run to run regardless of order, then it is unlikely that the results are being affected by carryover from one test to the next identically in both forward and reverse order. The results of reordering tests (as well as one experiment in which tests were conducted in randomized order) indicated that carryover effects were exceedingly rare for all operating systems except VxWorks (discussed later). Indeed, test reordering first detected that VxWorks had significant carryover problems that were an issue for both batch tests and tests executed one at a time.

The only other significant order-dependent problem that was noted was for Digital Unix 3.2 when running an external swap partition, which caused a system crash when three test cases in a row executed mq_receive() with very large buffer sizes.

11.3.8. Concurrency

It came as a surprise that the vast majority of OS robustness vulnerabilities, including system killers, could be reproduced via simple test programs involving a single function call. The community's consensus at the time was that the only system killers to be found in mature commercial operating systems would be caused by concurrency, timing problems, or "wearout" problems. To see if Ballista was missing even more problems, some experiments were performed trying to extend Ballista to explore concurrency issues.

One attempt to explore concurrent testing was performed as a graduate course project [Pientka 1999]. The approach was to launch multiple concurrent copies of Ballista tests on Red Hat Linux 5.2 (kernel 2.0.36). Some resource depletion problems were found when executing repeated calls to memory management functions, but the only true concurrency-based problem found was when some threads executing munmap with exceptional values got stuck in the kernel and could not be terminated without rebooting the system. The problem was traced to a nonreentrant exception handler. Once the failure mode was understood, the effect was highly reproducible by launching multiple parallel copies of exceptional calls to that function.

Other follow-on research had some initial success by systematically attempting to deplete resources such as memory. It is clear that vulnerabilities to such system stresses exist, but unmodified methods used by Ballista have not been very effective in revealing them. The more important result from Ballista, however, is that there appear to be substantive problems that are *not* a result of timing or concurrency problems, even though system designers and researchers alike initially thought such problems were unlikely to exist.

11.3.9. Dimensionality

Related to the notion that robustness problems would be found only in complex timing situations was the notion that problems would also involve subtle interactions of parameter values. This lead to exploring the response regions of faults [Koopman97] to determine what patterns of test values appeared in robustness failures. The expectation was that robustness problems would emerge only when test cases had particular values for each parameter (such as, for example, a file that happens to be open for reading, *and* a null pointer used as a buffer, *and* a string count greater than 64KB, *and* all zero data values, or some similar complex situation).

Instead, in the vast majority of cases, only one or two data values in a single system call were necessary to cause an abort failure. This lead us to analyze the *dimensionality* of faults in terms of number of parameters that affected whether a particular test result was robust or nonrobust [Pan 1999]. The result was that the vast majority of robustness failures were due to one or two parameter values. Moreover, the common two-parameter sensitivities were based on related values, such as a buffer pointer and associated buffer length.

The result that only single calls of low dimensionality (a single or two related exceptional parameter values) rather than high dimensionality (multiple unrelated parameters having a specific set of coincident values) could expose significant robustness defects was counter to initial expectations and feedback from others in the field, but the general rule for the results of almost all Ballista work is that robustness defects tend to have low dimensionality and are in general the result of fairly simple problems such as omitting a check for null pointer values.

11.3.10. Fine-Grain Data Structures

One of the aspects of the initial Ballista approach that had to be improved over time was the way in which parameter tests were created. Some parameter values were relatively simple, such as integers, whereas some were much more complex, such as files. The initial Ballista implementation picked selected values of complex data structures such as files (e.g., a file to which the test program has both read and write access permissions is buffered and is currently opened for reading). The combinations inherent in such complex data structures were a small impediment to scalability but were still viable.

As APIs beyond operating systems were tested, it became clear that more sophisticated notions of complex data structures were needed to deal with data structures defined by those other APIs. As an example, the SFIO library [Korn 1991] defines a file parameter that has a file state, a buffer type, and a flag value. To handle this case, Ballista was expanded to include a concept of tupled test values. Tupled test values are independently selected for testing by the test harness but combine to create a single parameter value passed to the MuT.

An example is different aspects of a file state being combined to create a single file handle value for testing purposes (Figure 11.2). Testing Sfseek requires test values for two data types: Sfio_t* and integer. The integer parameter is the position within the file for the seek operation and is tested with various integer test values used for all Ballista integer tests. In Figure 11.2, it so happens that the integer value of ZERO is selected for a particular test case but, in general, any of the IntValues might be selected for a test case. The first parameter is a complex test data type that includes values for file state (open, closed, etc.), buffer type (mapped, buffered, or nonbuffered), and various combinations of flag values. One test value from each of the subtypes (File State, Buffer Type, and Flags)

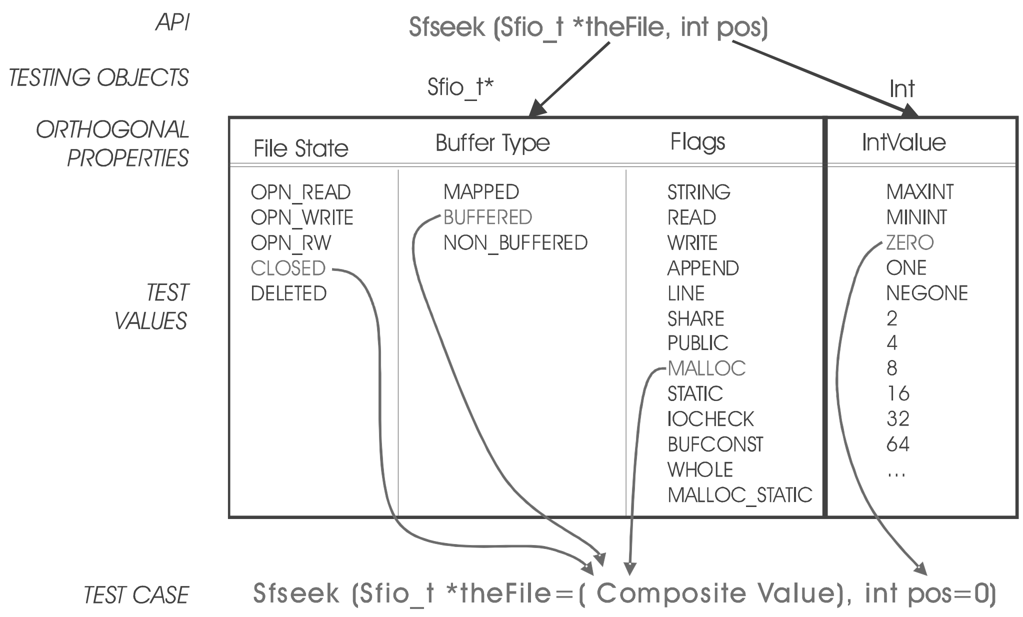

Figure 11.2. Example of fine-grain tupled test values for an SFIO function test.

is selected for a particular test case to form a composite test value. In Figure 11.2, the selection happens to be CLOSED, BUFFERED, and MALLOC. It would have also been possible to further make Flags a composite test type, but the developers of the tests thought that there were few enough interesting Flag test values that simply enumerating them all was manageable.

11.3.11. Initialization of System State

An additional challenge that became apparent as successive API implementations were tested was creating a simple approach to initialize the state behind the API interface. As a simple example, the POSIX random number generator rand(void) returns a random number based on a seed set with the call srand(unsigned seed). Attempting to test rand for robustness is only really meaningful if it is paired with a call to srand to set an exceptional seed value.

The initial solution to system state initialization was to have a per-function initialization code (for example, call srand before testing the rand function). Although this worked for POSIX and a few other APIs, it was in general an unsatisfactory solution because it required an additional mechanism beyond the central parameter-based approach of the rest of Ballista.

Eventually, a cleaner way of initializing system state was discovered by permitting some parameters specified in a Ballista test to be declared as *phantom* parameters. Phantom parameters are processed as regular parameters for Ballista test harness purposes, having their constructor and destructor code called in normal sequence. However, phantom parameters are omitted from the parameter list of the MuT. So, for example, the Ballista test signature for a random-number generator might be rand (+phantom_seed), where

"+" means that the parameter is a phantom parameter, and is thus omitted when calling rand() for test. However, the test values associated with phantom_seed are still executed, and in particular the constructor for phantom_seed can call srand to set an appropriate exceptional value for testing to be used by rand.

The need for system state initialization increased as the amount of inherent state in APIs being tested increased. CORBA, the HLA simulation backplane, and especially the industrial process automation software made extensive use of system state initialization, with five different general initialization states sufficing for all HLA tests executed. That experience lead to the idea of using phantom parameters as a generic state initialization technique. (The HLA testing itself [Fernsler 1999] did not use phantom parameters in this manner; that idea came from the HLA work, but only near the end of that work, as a retrospective lesson about how it could have been done better.)

11.3.12. Quantifying the Results

Because Ballista testing often produces as many as 100,000 individual test results for an API implementation, a summary metric for the robustness of an API is essential. Based on experience testing several APIs, and a number of intermediate techniques, two elements emerged as useful robustness metrics:

1. Number of functions tested that had a least one catastrophic result. Attempting to report the number of catastrophic results encountered is difficult because of the manual intervention needed to restore operation after a system crash. Additionally, many crashes could be due to a single exceptional value problem that appears in multiple tests. Since system crashes affect few functions in most APIs, a simple report of the number of functions that have crashed (regardless of how many times they crash) suffices.
2. The normalized mean abort failure rate across functions. The percent of abort failures is equally weighted across all functions tested, then averaged to give a general representation of the API. Although in practice some functions are used more than others [Musa96], workloads make such different use of each API that no single weighting would give an accurate picture. (We recognize that each percentage is in fact a ratio involving test failures compared to tests conducted, and that taking an arithmetic mean across such ratios is problematic in a pure mathematical sense. However, in terms of benchmarking, the approach is that each individual function's failure ratio is a robustness "score." Averaging scores across tested functions gives an average score that, in our experience, corresponds quite closely with our informal observations about the overall robustness of tested systems. Until a useful methodology is found to map these scores onto the frequency of function executions in application programs, there does not seem to be much point digging deeper into mathematical analysis of the ratios.)

Restart failures are usually so rare (except for VxWorks) that they do not really affect results and can either be combined with abort failure rates as an aggregate failure rate or simply omitted.

As a practical matter, most APIs also have a few functions that have much higher than average abort failure rates. It is useful to report which functions those are so they can either be repaired by vendors or avoided (to the degree possible) by application programs. To permit identification of functions with very high abort failure rates and enable application-specific weighting of failure rates, Ballista testing records the failure rate of each call in the API. More sophisticated methods of quantifying robustness testing results are evolving over time (e.g., [Duraes 2002]), but the above robustness metrics sufficed for the customers of the Ballista project.

11.3.13. System Killer Vulnerabilities

Some of the Ballista robustness-testing results, particularly those that deal with tasks abnormally terminating via throwing a signal, are controversial, as discussed in subsequent sections. However, the one set of results that is generally accepted as important is when Ballista finds "system killer" robustness vulnerabilities. A *system killer* is a single robustness test case that causes a complete system crash or hang, necessitating a system reset to recover. Depending on the operating system being used, a system crash might manifest as a "kernel panic" for Unix systems, a "blue screen of death" for Windows systems, or simply a system so locked up that only a hardware reset or power-cycle operation will recover the system.

System killer defects are surprisingly tedious to isolate, because test logs tend to be lost or corrupted by such crashes despite concerted efforts to flush writes to file systems and otherwise capture system state just before the test that caused a crash. Nonetheless, a tedious search through sets of tests usually revealed an exact, single system call that caused a crash. Perhaps surprisingly, crashes can often be caused by very simple user programs such as these examples:

- Irix 6.2: munmap(malloc((1<<30)+1), ((1<<31)-1)))
- Digital UNIX (OSF 1) 4.0D: mprotect(malloc((1 << 29)+1), 65537, 0)
- Windows 98: GetThreadContext(GetCurrentThread(), NULL)
- Windows 98: MsgWaitForMultipleObjects(INT_MAX, (LPHANDLE)((void*) -1), 0, 129, 64)

Table 11.1 summarizes the systems tested within the Ballista project, including how many distinct functions had system killer vulnerabilities and an overall normalized abort robustness failure rate (discussed later). In some cases, precise data is not available, so the best available data is presented.

It is recognized that software versions tested significantly predate the writing of this chapter. During the course of our research, system developers uniformly said that a more recent version of software (which was not available to the Ballista team at the time of testing) would be dramatically better. But the new version usually looked pretty much like the old version, and sometimes was worse unless the developers incorporated Ballista-style testing results into their own testing process or specifically fixed system killers previously identified for correction. Since the point of this chapter is to relate lessons learned from Ballista, there has been no attempt to perform the extensive work required to remeasure systems.

11.4. LESSONS LEARNED

Some of the technical lessons learned in developing Ballista have already been discussed. But beyond the lessons learned in engineering the Ballista test tools are the technical and nontechnical lessons revealed by the results of the tests.

TABLE 11.1. System killer vulnerabilities found by Ballista testing

API and implementation	Functions tested	System killer functions	System killer function names	Normalized robustness failure rate, %
AIX 4.1	186	0	n/a	10.0%
FreeBSD 2.2.5	175	0	n/a	20.3
HPUX 9.05	186	0	n/a	11.4
HPUX 10.20	185	1	mmap	13.1
Irix 5.3	189	0	n/a	14.5
Irix 6.2	225	1	munmap	12.6
Linux 2.0.18	190	0	n/a	12.5
Red Hat Linux 2.2.5	183	0	n/a	21.9
LynxOS 2.4.0	222	1	setpgid	11.9
NetBSD 1.3	182	0	n/a	16.4
OSF-1 3.2	232	1	mqreceive	15.6
OSF-1 4.0B	233	0	mprotect (4.0D only, not 4.0B)	15.1
QNX 4.22	203	2	munmap, mprotect	21.0
QNX 4.24	206	0	n/a	22.7
SunOS 4.13	189	0	n/a	15.8
SunOS 5.5	233	0	n/a	14.6
Windows NT SP 5	237	0	n/a	23.9
Window 2000 Pro Beta Build 2031	237	0	n/a	23.3
Windows XP RC 1	235	0	n/a	23.1
Windows 95 rev B	227	8	DuplicateHandle, FileTimeToSystemTime, GetFileInformationByHandle, GetThreadContext, HeapCreate, MsgWaitForMultipleObjectsEx, ReadProcessMemory, fwrite	17.2
Windows 98 SP 1	237	7	DuplicateHandle, GetFileInformationByHandle, GetThreadContext, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, fwrite, strncpy	17.8
Windows 98 SE SP 1	237	7	CreateThread, DuplicateHandle, GetFileInformationByHandle, GetThreadContext, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, strncpy	17.8
Windows CE 2.11	179	28	See [Shelton00]	13.7
Sun JVM 1.3.1-04 (Red Hat Linux 2.4.18-3)	226	0	n/a	4.7
Timesys J2ME Foundation 1.0 (Linux 2.4.7; Timesys GPL 3.1.214)	226	Unknown (JVM crashes)	"Process received signal 11, suspending" message causing JVM crashes.	Approx. 5%

TABLE 11.1. *Continued*

API and implementation	Functions tested	System killer functions	System killer function names	Normalized robustness failure rate, %
VxWorks (without memory protection)	75	2	bzero; gmtime_r	13.0
RTI-HLA 1.3.5 (Digital Unix 4.0)	86	0	(found a server crash in a pre-1.3.5 version)	10.2
RTI-HLA 1.3.5 (Sun OS 5.6)	86	0	n/a	10.0
SFIO	36	0	n/a	5.6
Industrial Process Automation Object Manager (COM/DCOM)	52	0	n/a	5 methods had abort failures

11.4.1. Mature Commercial Software Has Robustness Vulnerabilities

A key result of the Ballista robustness testing project is that even mature commercial software such as operating systems can and often does have catastrophic responses to even very simple exceptional parameter values sent to an API. Moreover, it is common to have a significant fraction of robustness tests generate abort failures, although abort failure robustness varies significantly depending on the API and implementation.

The version of an implementation of an API has little effect on the overall abort failure robustness level. It is just as common for abort failures to increase as decrease in a later version of a piece of software. It is also just as common for system killers to appear and disappear between software versions, even with seemingly minor changes. For example, the system killer reported in Table 11.1 for OSF-1 4.0D was not present in OSF-1 4.0B.

Other evidence suggests that system killer robustness defects can come and go depending on small changes to software, lurking until a chance set of circumstances in a system build exposes them to being activated by the API. The most striking example was a vulnerability to system crashes in Windows 95 via the system call FileTimeToSystemTime. While Microsoft fixed all the system crashes the Ballista team reported for Windows 98, they did not search for the cause of the FileTimeToSystemTime problem in Windows 95, because it did not cause a crash in Windows 98. Unfortunately, in a beta version for Windows ME, that same system killer had reappeared, causing system crashes. From this experience and other observations discussed in the next section, it seems important to track down and fix any system killers revealed via robustness testing, regardless of whether they go into remission in any particular release. One can reasonably expect that a system killer that mysteriously goes away on its own will reappear in some future release if not specifically corrected.

Most APIs studied have a few functions with highly elevated abort failure rates. In other words, usually it is only a handful of functions that are severely nonrobust. For example, in the Timesys Linux-RT GPL + RT-Java testing, three methods out of 266 methods tested had elevated robustness failure rates. String(), convValueOf(), and append() all had had robustness failure rates above 70%, whereas all other methods tested had robustness failure rates below 25%. Moreover, the three methods with elevated failure rates were the most interesting to the spacecraft system designers requesting the tests, because they were the only methods having failures due to problems other than missing null pointer checks.

11.4.2. Robustness Soft Spots Sometimes Contain System Killers

Based on several anecdotal observations, there appears to be some truth to the notion that modules with high rates of abort robustness failures can harbor system killer defects. In general, functions that had system killers tended to have high abort failure rates, but exact data is not available because machines suffering catastrophic failures usually corrupt test data during the crash. The following incidents, while somewhat circumstantial, support this notion.

The difference between abort failure rates in QNX 4.22 and QNX 4.24 is in large part due to a near-100% abort failure rate for the functions munmap and mprotect in version 4.24. These are the two functions that had system killer defects in version 4.22. Although the system killers were eliminated in version 4.24 (after the Ballista team had reported them to QNX), it is clear that they came from functions that were overall nonrobust.

HPUX 10.20 had a higher abort failure rate than HPUX 9.05 and, in particular, the abort failure rate for memory management functions went from 0% to essentially 100%. A discussion with the system designers revealed that the memory management functions had been rewritten for HPUX version 10. It so happened that the one system killer found in that operating system was mmap, one of the functions with a high abort failure rate.

It is not always the case that system killers lurk in areas of high abort failure rates (for example, the munmap system killer in Irix 6.2 was in a set of functions with near-0% abort failure rates). Nonetheless, looking for system killers in otherwise nonrobust software seems like a productive strategy.

11.4.3. Dynamic Tension between Fault Masking and Fault Detection

During discussions with various developers, it became clear that there were two opposing perspectives on the notion of whether signals and other exceptions that normally cause task termination were acceptable responses to exceptional conditions. The division hinged upon whether or not the primary goal was fault-detection or fault tolerant operation.

Organizations that primarily exist for the purpose of software development tended to say that generating unrecoverable exceptions was desirable. In that way, developers cannot miss a defect, and have incentive to fix it so as to avoid embarrassing code crashes. Some developers went so far as to say that abort failures were the right thing to do in all cases, and asserted that their code intentionally threw signals in any situation in which an exceptional parameter value was encountered. (This claim was not borne out by Ballista test results.) They also claimed that code with low abort failure rates probably had high silent failure rates. (This claim was true in a sense, but FreeBSD 2.2.5, which was said to be designed to intentionally generate abort failures, had even more silent failure rates than AIX 4.1, which had the lowest abort failure rate of POSIX systems tested.)

Organizations that primarily survive by supporting fielded equipment tended to say that generating unrecoverable exceptions was never desirable. In fact, one had gone to some lengths to intentionally generate silent failures in preference to abort failures for some legacy applications. The developers stated that this was to ensure no field failures of software, which would result in a situation in which customers were unequipped to diagnose, repair, recompile, and reinstall software that crashed. (However, they still had significant abort failure rates in their code.)

The results of these observations are twofold. First, there is truth to both sides of the division as to whether abort failures are good or bad. Developers want to know that something is wrong in their code. Maintainers do not want software crashing in the field. A challenge is to somehow help both sets of stakeholders in real systems. Second, most software has both abort and silent robustness failures, even in cases in which developers stated that they intended to eliminate one of those categories of exception responses.

11.4.4. Programmers Do Not Necessarily Understand the Trade-offs They Are Making

During the course of the Ballista project, developers repeatedly stated that making software robust would cause it to be too inefficient to be practicable. However, it appears that this perception may be incorrect. For one thing, it seems that at least some developers overstate the true execution cost of increasing robustness. [Devale 2001a] reports an experiment in manually hardening the math libraries of FreeBSD 3.0. In that experiment, hardening the code against all robustness failures identified by Ballista reduced abort failure rates from 12.5% to zero. Attaining this robustness improvement cost only a 1% slowdown compared to the unmodified library code, but developers had already incurred a 13% speed penalty to reduce abort failure rates from 15.3% (with all checks manually removed) to 12.5% (for released code). Thus the developers were paying almost all the necessary speed penalty without actually gaining much robustness in return compared to what they could have had.

In addition to overstating performance penalties for improving robustness, developers who intended to implement all-abort or all-silent robustness responses did not succeed at doing so. This gave rise to the question as to whether developers in general are making well-informed and effectively executed engineering decisions when they design an intended level of robustness into their software.

A case study tested the hypothesis that developers put the intended level of robustness into their software, reported in [Devale 2001a]. That study examined three sets of industry code modules from a large corporation and found that none of the developers achieved exactly the robustness they set out to. One of the three development teams came very close, but the other two teams had many more abort failures in their Java code than they expected.

An additional data point on this topic was obtained by testing the robustness of the SFIO (Safe, Fast I/O) library. An interview with the developers, who are very experienced industry professionals, established that they believed they had incorporated all the robustness that was practical into SFIO, and that any remaining nonrobust behaviors would either be impractical to resolve (e.g., situations too complex to reasonably test for) or would take an unacceptable run-time overhead to resolve. DeVale and Koopman [DeVale01] report the results of selecting eight functions from the SFIO library for hand improvement of robustness. Abort failure rates for those eight functions were reduced from 2.86% to 0.44%–0.78% (depending on the underlying operating system), with an average speed penalty of only 2%.

The lesson learned from these experiences is that even highly experienced industry practitioners probably cannot implement the level of robustness they intend to without gaps in the absence of a robustness measurement tool such as Ballista. Moreover, intuition as to the likely speed penalty of robustness checking is often incorrect. A measurement tool helps provide feedback as to whether a desired level of robustness quality has been reached. Beyond that, actual speed measurement is required to determine whether checking for exceptional parameters incurs too much overhead, and in every case we have seen, it doesn't.

11.4.5. Programmers Often Make the Same Robustness Mistakes

Some approaches to software fault tolerance rely on the use of diverse sets of software to help provide independent software failures (e.g., [Avizienis 1985]). The topic of how diversity can be measured and ensured is a difficult one and has been discussed in many previous publications (e.g., [Eckhardt 1991] and [Avizienis 1988]). Contemporaneous with Ballista work, Littlewood and Strigini [Littlewood 2000] introduced the concept of "robustness diversity" to describe the concept of two different implementations of a function producing different responses when subject to exceptional values. The ability to measure the robustness of many different implementations of the same operating operating system API with Ballista provided an excellent chance to gather information about robustness diversity on commercially available software.

Koopman and DeVale [Koopman 1999] examined the question of whether independently developed software was likely to have similar robustness vulnerabilities, especially to abort failures. The result was that 3.8% of robustness failures occurred in all thirteen POSIX operating systems examined. The two most diverse (from this point of view) operating systems (AIX and FreeBSD) had 9.7% of the same system call robustness failures and 25.4% overall common-mode robustness failures for the entire POSIX API, including C library calls. Based on this, it is clear that independently developed software cannot, without further qualification, be considered perfectly diverse with respect to robustness vulnerabilities. (Whether or not this lack of perfect diversity matters in practice, of course, depends on the intended application.)

An additional data point is that Windows CE had many of the same system killer vulnerabilities as Windows 95 and Windows 98, despite the fact that they were said to be developed by different teams within Microsoft and have different code bases.

The evidence to date indicates that one must assume that independent programmers will leave similar robustness vulnerabilities in their code unless some specific and effective countermeasure is in place to prevent it.

11.4.6. Memory Protection Really Works

As has been mentioned in previous sections, Ballista testing of the VxWorks operating system produced some anomalous behaviors. The version of VxWorks tested did not have memory protection. (Historically, VxWorks was designed for resource-constrained systems and did not include memory protection as a supported feature, although newer versions of VxWorks are said to do so.) Testing results included 9,944 tests on 37 functions and caused 2,920 abort and restart failures (in approximately equal proportions). The results included 360 system crashes. (An additional 38 functions were tested by an industry partner, with similar results.)

VxWorks testing was made very difficult by the fact that test results were highly sensitive to the order in which they were run. Changing the order of test execution significantly changed which tests suffered abort and restart failures, indicating a high carryover of faults from one test to another. Given a lack of memory protection, this was not really a surprise. The carryover was so pervasive that the only way repeatable results could be obtained was to install an automated reset box that completely reset the test hardware between each and every single Ballista test run. The fact that other systems had minimal carryover suggests that the memory protection in place in other systems is a valuable fault containment mechanism.

The startling (to us, at the time) result was not that there was carryover with VxWorks but, rather, that complete crashes were so rare in the tests run. Eliciting and isolating tests that caused clean system crashes was quite difficult. Most often, the system would degrade in some way and keep working to some degree for an extended period of time after tests rather than failing quickly. Failing "fast" was a rare outcome. This suggests that a lack of memory protection reduces the validity of "fail-fast" assumptions in the absence of any specifically designed fail-fast enforcement mechanism.

11.4.7. Ballista Scaled Well, Within Some Practical Limits

In the end, the parameter-based testing approach used with Ballista scaled well across several APIs of varying characteristics. The use of fine-grain data types and phantom parameters solidified a single mechanism (a parameter list with data-type-based test values) for addressing a wide variety of situations. The main limitations to Ballista testing are scalability with respect to internal system state behind the API and difficulty in exercising "software aging" phenomena.

Ballista-style testing scales best with APIs having little state, such as operating system calls and programming libraries. It scales less well as the system requires more state behind the API. CORBA client testing and HLA RTI simulation backplane testing worked moderately well, because the number of different types of system start-up operations necessary to perform testing were much smaller than the number of functions to be tested, and could readily be represented as phantom parameters that performed operations such as initializing the system, forming a federation (an HLA operation [DoD 1998]), and so on. However, testing the industrial process control automation object manager became difficult because of the significant amount of work needed to set up a clean copy of a database before every test. In general, Ballista-style testing is not suitable for APIs that access a database.

A second significant limitation to Ballista-style testing is that it is not designed to stress system vulnerabilities related to concurrency or resource leaks (especially "software aging" phenomena). Some concurrency problems can be searched for in a straightforward manner by looking for nonreentrant exception handlers. And some aging problems can be revealed simply by running Ballista test sequences many times over. But, it remains unclear how to effectively test for other such problems. Such problems do exist, but Ballista is not really designed to find them.

A third limitation to Ballista testing is nontechnical. Some people believe that the abort failures it finds are important, especially as an indicator of general system robustness, and some people do not. This remains partly an issue of culture (see also Section 11.4.3), and partly an issue of whether anyone can demonstrate with certainty that code suffering high levels of Abort failures really is prone to catastrophic failures as well. Fortunately, most people believe that finding system killers is useful, and Ballista is effective enough at finding them to be worth using on many systems, regardless of opinion on the importance of abort failures.

11.4.8. Parameter Checking Need Not Be Slow

A commonly cited technical impediment to improving system robustness is the execution cost of performing parameter-value checks to permit so-called wrappers to detect and reject calls having exceptional values (for example, using the techniques in Xept [Vo 1997]). As has already been discussed, usually parameter checking is fast enough to not be a problem (see Section 11.4.4), but one can do even better.

More advanced CPU hardware could tend to further reduce the cost of performing exceptional parameter-value checking. If compiler hints help the CPU speculate that values are nonexceptional, then the critical path of the computation can proceed, and checking can take place as hardware resources permit before the critical path instruction results are retired. This has the potential to effectively eliminate parameter-value checking execution costs on superscalar, speculative, out-of-order execution hardware.

DeVale and Koopman [DeVale 2002] addressed this problem further by developing a novel scheme to cache checks and reuse them to reduce the overhead of performing parameter checks. The insight is that many parameter values are used multiple times (e.g., a file handle for character I/O operations). Rather than check the validity of such a parameter every time an API function is called, the API can remember that it has checked the value once, and simply reuse the results of that check, stored in a software cache, until the value is altered. Using this technique reduced parameter checking overhead to less than 5%, even for the worst case of very short functions being called, and to 2% for most other cases studied.

11.5. CONCLUSIONS

In the past decade, robustness testing and related areas have grown into a thriving area within the dependability research community. Researchers involved in extending, adapting, augmenting, and creating alternative ideas to Ballista-style testing are too numerous to mention individually.

Two collaborative efforts are, however, worth specific mention. The first is the Special Interest Group on Dependability Benchmarking, which is the sponsoring organization of this book. The second is the DBench Consortium [DBench 2004], which performed a thorough exploration of the subject of dependability benchmarking in Europe, looking at it as a combination of many related techniques, including fault injection and API robustness testing.

With the hindsight of 10 years of experience, the early decisions of Ballista worked out well. Especially fortuitous was the choice of basing test values on parameter data types, which extended to a number of APIs well beyond the originally envisioned operating systems tests. In the end, system killer robustness vulnerabilities lurk in even the most mature commercial software. Nonrobust behavior in the form of abort failures is prevalent, with anything less than a 10% abort failure rate being achieved only by developers who take special care. From a technical point of view, exploring concurrency-related problems and wear-out phenomena requires a significant step beyond Ballista.

Perhaps the biggest open challenges are nontechnical. Developers vary widely in their belief that software robustness beyond absence of system killer vulnerabilities matters, and the research community has yet to produce compelling evidence that they are wrong to think so. For developers who do strive to reduce noncatastrophic robustness vulnerabilities, it appears that a tool such as Ballista is essential to double check their efforts and help prevent missing vulnerabilities.

A copy of the Ballista toolset remains available from the authors and, for the indefinite future, at http://ballista.org.

ACKNOWLEDGMENTS

Many people have contributed to Ballista over the years. Dan Siewiorek has been very active in advising and supporting the project throughout. The late Asad Zaidi created the first Ballista testing Web-based server, and others ran tests on various APIs, contributed ideas, coauthored publications, and otherwise helped this project succeed: Roy Maxion, Charles Shelton, Jintao Pan, Kim Fernsler, Nathan Kropp, Geoff Hendrey, David Guttendorf, Chris Dingman, John Sung, Michael Abowd, Keith Arner, Bemina Atanacio, Dave Hanekamp, Mark Hsu, Ying Shi, Scott Wen, Kanda Runapongsa, Orna Raz, Brigette Pientka, Meredith Beveridge, and Ted Marz.

Ballista has also benefited from the collaboration with and sponsorship from many organizations, the most significant of which was DARPA (contract DABT63-96-C-0064). Other sponsors include ABB, Emerson, Microsoft, AT&T, Cisco, Office of Naval Research, Lucent, IBM, and in-kind support from DRC. We also received equipment donations from Intel, Digital Equipment Corporation/Compaq, the U.S. Navy, and QNX.

REFERENCES

[Arlat 1990] Arlat, J., Aguera, M., Amat, L., Crouzet, Y, Fabre, J.-C., Laprie, J.-C., Martins, E., and Powell, D., "Fault Injection for Dependability Validation—A Methodology and Some Applications," *IEEE Transactions on Software Engineering,* vol. 16, no. 2, pp. 166–182, February 1990.

[Avizienis 1985] Avizienis, A., "The N-version Approach to Fault-tolerant Software," *IEEE Transactions on Software Engineering,* vol. SE-11, no.12, pp. 1491–1501, 1985.

[Avizienis 1988] Avizienis, A., Lyu, M. R., and Schutz, W., In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software, *Eighteenth International Symposium on Fault-Tolerant Computing,* Tokyo, Japan, pp. 15–22, 1988.

[Barton 1990] Barton, J., Czeck, E., Segall, Z., and Siewiorek, D., "Fault Injection Experiments Using FIAT," *IEEE Transactions on Computers,* vol. 39, no. 4, pp. 575–582.

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

[Carreira 1998] Carreira, J., Madeira H., and Silva J., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering,* vol. 24, no. 2, pp. 125–136, February 1998.

[Carrette 1996] Carrette, G., "CRASHME: Random Input Testing," http://people.delphi.com/gjc/crashme.html, accessed July 6, 1998.

[Cristian 1995] Cristian, F., "Exception Handling and Tolerance of Software Faults," in *Software Fault Tolerance,* Michael R. Lyu (Ed.), pp. 81–107, Chichester, UK: Wiley, 1995.

[Czeck 1986] Czeck, E., Feather, F., Grizzaffi, A., Finelli, G., Segall, Z., and Siewiorek, D., "Fault-free Performance Validation of Avionic Multiprocessors," in *Proceedings of the IEEE/AIAA 7th Digital Avionics Systems Conference,* Fort Worth, TX, 13–16 Oct. 1986, pp. 803, 670–677.

[DBench 2004] *DBench Project Final Report,* IST-2000-25425, LAAS-CNRS, May 2004, http://www.laas.fr/Dbench.

[DeVale 1999] DeVale, J., Koopman, P., and Guttendorf, D., "The Ballista Software Robustness

Testing Service," in *16th International Conference on Testing Computer Software,* pp. 33–42, 1999.

[DeVale 2001] DeVale, J., and Koopman, P., "Performance Evaluation of Exception Handling in I/O Libraries," in *International Conference on Dependable Systems and Networks (DSN),* July 2001, Göteborg, Sweden.

[DeVale 2001a] DeVale, J., *High Performance Robust Computer Systems,* Ph.D. dissertation, Carnegie Mellon University ECE Dept., Pittsburgh, PA, December 2001.

[DeVale 2002] DeVale, J., and Koopman, P., "Robust Software—No more excuses," in *International Conference on Dependable Systems and Networks (DSN),* Washington DC, July 2002.

[Dingman 1995] Dingman, C., "Measuring Robustness of a Fault Tolerant Aerospace System," in *25th International Symposium on Fault-Tolerant Computing,* June 1995. pp. 522–527.

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

[DoD 1998] U.S. Department of Defense, *High Level Architecture Run Time Infrastructure Programmer's Guide, RTI 1.3 Version 5,* Dec. 16, 1998, DMSO/SAIC/Virtual Technology Corp.

[Duran 1984] Duran, J., and Ntafos, S., "An Evaluation of Random Testing," *IEEE Trans. On Software Enginering,* Vol. SE-10, No. 4, July 1984, pp. 438–443.

[Duraes 2002] Durães J., and Madeira, H. "Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior," *IEICE Transactions on Information and Systems,* vol. E86-D, no. 12, pp. 2563–2570, December 2002.

[Eckhardt 1991] Eckhardt, D. E., Caglayan, A. K., Knight, J. C., Lee, L. D., McAllister, D. F., Vouk, M. A., and Kelly, J. P. J.; "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Transactions on Software Engineering,* vol. 17, no. 7, pp. 692–702, July 1991.

[Fernsler 1999] Fernsler, K., and Koopman, P., "Robustness Testing of a Distributed Simulation Backplane," in *ISSRE 99,* Boca Raton, FL, pp. 189–198, 1999.

[Gehani 1992] Gehani, N., "Exceptional C or C with Exceptions," *Software—Practice and Experience,* vol. 22, no. 10, pp. 827–848, 1992.

[Goodenough 1975] Goodenough, J., "Exception Handling: Issues and a Proposed Notation," *Communications of the ACM,*vol. 18, no. 12, pp. 683–696, December 1975.

[Hill 1971] Hill, I., "Faults in Functions, in ALGOL and FORTRAN," *The Computer Journal,* vol. 14, no. 3, pp. 315–316, August 1971.

[IEEE 1990] *IEEE Standard Glossary of Software Engineering Terminology,* IEEE Std 610.12-1990, IEEE Computer Society, Los Alamitos, CA, 1990.

[IEEE 1993] *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, Los Alamitos, CA, 1994.

[Kanawati 1992] Kanawati, G., Kanawati, N., and Abraham, J., "FERRARI: A Tool for the Validation of System Dependability Properties," in *1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems.* Amherst, MA, pp. 336–344, 1992.

[Koopman 1997] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., and Marz, T., "Comparing Operating Systems Using Robustness Benchmarks," in *Proceedings of Symposium on Reliable and Distributed Systems,* Durham, NC, pp. 72–79, 1997.

[Koopman 1998] Koopman, P., "Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security," in *Post proceedings of the Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98),* Washington, DC, pp. 103–131, November 1998.

[Koopman 1999] Koopman, P., and DeVale, J., "Comparing the Robustness of POSIX Operating Systems," in *28th Fault Tolerant Computing Symposium,* pp. 30–37, 1999.

[Koopman00] Koopman, P., and DeVale, J., "The Exception Handling Effectiveness of POSIX

Operating Systems," *IEEE Transactions on Software Engineering,* vol. 26, no. 9, pp. 837–848, September 2000.

[Korn 1991] Korn, D., and Vo, K.-P., "SFIO: Safe/Fast String/File IO," in *Proceedings of the Summer 1991 USENIX Conference,* pp. 235–256, 1991.

[Kropp 1998] Kropp, N., Koopman, P., and Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," in *28th Fault Tolerant Computing Symposium,* pp. 230–239, 1998.

[Littlewood 2000] Littlewood, B., and Strigini, L., *A Discussion of Practices for Enhancing Diversity in Software Designs,* DISPO LS-DI-TR-04_v1_1d, 2000.

[Marick 1995] Marick, B., *The Craft of Software Testing,* Prentice-Hall, Upper Saddle River, NJ, 1995.

[Miller 1990] Miller, B., Fredriksen, L., and So, B., "An Empirical Study of the Reliability of Operating System Utilities," *Communication of the ACM,* vol. 33, pp. 32–44, December 1990.

[Miller 1998] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J., *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services,* Computer Science Technical Report 1268, University of Wisconsin–Madison, May 1998.

[Mukherjee 1997] Mukherjee, A., and Siewiorek, D. P., "Measuring Software Dependability by Robustness Benchmarking," *IEEE Transactions on Software Engineering,* vol. 23, no. 6, pp. 366–378, June 1997.

[Musa 1996] Musa, J., Fuoco, G., Irving, N., Kropfl, D., and Juhlin, B., "The Operational Profile," in Lyu, M. (Ed.), *Handbook of Software Reliability Engineering,* pp. 167–216, McGraw-Hill/IEEE Computer Society Press, Los Alamitos, CA, 1996.

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Revision 2.0, July 1995.

[Ostrand 1988] Ostrand, T. J., and Balcer, M. J., "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM,* vol. 31, no. 6, pp. 676–686, 1988.

[Pan 1999] Pan, J., Koopman, P., and Siewiorek, D., "A Dimensionality Model Approach to Testing and Improving Software Robustness," in *Autotestcon99,* San Antonio, TX, 1999.

[Pan 2001] Pan, J., Koopman, P., Siewiorek, D., Huang, Y., Gruber, R., and Jiang, M., "Robustness Testing and Hardening of CORBA ORB Implementations," in *International Conference on Dependable Systems and Networks (DSN),* Göteborg, Sweden, pp. 141–150, 2001.

[Pientka 1999] Pientka, B., and Raz, O., *Robustness in Concurrent Testing,* Project Report, Carnegie Mellon University, Pittsburgh, PA. May 23, 1999.

[Rodriguez 2002] Rodríguez, M., Albinet, A., and Arlat, J., "MAFALDA-RT: A Tool for Dependability Assessment of Real Time Systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2002),* Washington, DC, 2002, pp. 267–272.

[Russinovich 1993] Russinovich, M., Segall, Z., and Siewiorek, D., "Application Transparent Fault Management in Fault Tolerant Mach," in *The Twenty-Third International Symposium on Fault-Tolerant Computing,* pp. 10–19, Toulouse, France, June 22–24 1993.

[Schuette 1986] Schuette, M., Shen, J., Siewiorek, D., and Zhu, Y., "Experimental Evaluation of Two Concurrent Error Detection Schemes," in *16th Annual International Symposium on Fault-Tolerant Computing Systems,* Vienna, Austria, pp. 138–43, 1986.

[Shelton 2000] Shelton, C., and Koopman, P., "Robustness Testing of the Microsoft Win32 API," in *International Conference on Dependable Systems and Networks (DSN),* New York City, 2000.

[Siewiorek 1993] Siewiorek, D., Hudak, J., Suh, B., and Segal, Z., "Development of a Benchmark to Measure System Robustness," in *23rd International Symposium on Fault-Tolerant Computing,* June 1993, Toulouse, France, pp. 88–97.

[Thevenod-Fosse91] Thévenod-Fosse, P., Waeselynck H., and Crouzet, Y., "An Experimental

Study of Software Structural Testing: Deterministic versus Random Input Generation," in *Proceedings of 21st IEEE International Symposium on Fault-Tolerant Computing (FTCS-21),* Montréal, Québec, Canada, pp. 410–417, 1991.

[Tsai 1995] Tsai, T., and Iyer, R., "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," in *Proceedings of Eighth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation,* pp. 26–40, Heidelberg, Germany, Springer-Verlag, London, 1995.

[Vo 1997] Vo, K-P., Wang, Y-M., Chung, P., and Huang, Y., "Xept: A Software Instrumentation Method for Exception Handling," in *The Eighth International Symposium on Software Reliability Engineering,* Albuquerque, NM, pp. 60–69, 1997.