©1999, IEEE. *Post proceedings of the Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98)*, 11-13 November 1998, Washington, D.C.

# Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security

Philip Koopman
*Department of Electrical and Computer Engineering*
*& Institute for Complex Engineered Systems*
*Carnegie Mellon University, Pittsburgh, PA*
*koopman@cmu.edu*

**Abstract**

*Quantitative assessment tools are urgently needed in the areas of fault tolerance, software assurance, and computer security. Assessment methods typically employed in various combinations are fault injection, formal verification, and testing. However, these methods are expensive because they are labor-intensive, with costs scaling at least linearly with the number of software modules tested. Additionally, they are subject to human lapses and oversights because they require two different representations for each system, and then base results on a direct or an indirect representation comparison.*

*The Ballista project has found that robustness testing forms a niche in which scalable quantitative assessment can be achieved at low cost. This scalability stems from two techniques: associating state-setting information with test cases based on data types, and using one generic, but narrow, behavioral specification for all modules. Given that this approach has succeeded in comparing the robustness of various operating systems, it is natural to ask if it can be made more generally applicable.*

*It appears that Ballista-like testing can be used in the fault tolerance area to measure the generic robustness of a variety of API implementations, and in particular to identify reproducible ways to crash and hang software. In software assurance, it can be used as a quality check on exception handling, and in particular as a means to augment black box testing. Applying it to computer security appears more problematic, but might be possible if there is a way to orthogonally decompose various aspects of security-relevant system state into analogs of Ballista data types.*

*While Ballista-like testing is no substitute for traditional methods, it can serve to provide a useful quality assurance check that augments existing practice at relatively low cost. Alternately, it can serve to quantify the extent of potential problems, enabling better informed decisions by both developers and customers.*

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you can not measure it, when you can not express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thought, advanced to the stage of science.
>
> — Lord Kelvin

## 1. Introduction

One of the more difficult aspects of working in the areas of fault tolerance, software assurance, and computer security is quantifying results. Quantification is absolutely essential attain a mature capability for creating systems of any kind. And yet, despite significant research efforts, the attainment of quantification is proving elusive in the area of software-intensive systems. While there is no doubt that progress has been made in the area of general software development methodology, the focus has been on the primary attributes of productivity and software correctness. Less attention has been paid to what are often considered secondary attributes, such as fault tolerance, software safety, and security. These three areas seem to share a common problem in that it is difficult to attain quantification, largely because of the interaction of four problems.

First, the properties of software-intensive systems that need to be measured are typically very dependent on the details of the system specification or usage. Examples include functional defect rates, whether a system is "safe," and whether a system is "secure." This means that any measurement tool must be provided with a wealth of information about the system, and in general this requires access to a substantial portion of the system specification. Complicating the situation is that the specification, or indeed even the detailed implementation, may be unavailable for software components purchased off-the-shelf. Obtaining such information costs time and money. To illustrate how difficult this makes things, compare the difficulty of determining whether a computer system is secure to determining its speed at executing a prepackaged benchmark suite.

Second, the desired measurement outcome is often desired to be perfection (attaining a zero defect rate, completely safe, or never fails). Even though such perfection is generally unattainable in practice, it means that tests are focussed on finding deviations from a specification of some sort. This is in sharp contrast to more traditional computer system metrics such as computation speed or achieved network bandwidth. It also raises issues if, as is often (always?) the case, the specification itself is imperfect.

Third, it is common for the same tests that are used to measure the system to be used directly to fix any shortcomings found. For example, a project measuring bug reports per week of testing would naturally feed all bug reports to the maintenance group for correction. Thus, it is common for the measurement process itself to be coupled to changes in the system being measured. While it is clear that traditional speed measurements result in systems being optimized to perform better, the situation is different here in that there is no fundamental quantity to be measured (such as a particular workload to complete). Rather, deviations from perfection are being both measured and corrected by the same process.

Fourth, and perhaps most important from a practical implementation point of view, is that the process of measuring "bad things" such as bugs is often seen as a destructive act, as opposed to measuring "good things" such as performance. The outcome of a metric value of less than perfection can be seen as a label implying defective goods, whereas the outcome of a positive metric such as speed is merely a measurement of an inherent property.

Despite these problems, it is imperative that we find ways to quantify properties of systems that we wish to understand and to improve over time. (Of course there is always the risk of inventing numbers just for the sake of having numbers, so it is also important to find metrics that are at least correlated to actual desirable properties.) Thus, it would be desirable to have a measurement tool or methodology that does not require much detailed information about the system it is measuring, does not attempt to measure quantities that deviate very slightly from perfection, uses novel tests for each measurement or otherwise disentangles measurement from defect correction, and is somehow seen as a constructive measurement rather than a destructive

activity.

The work discussed herein does not satisfy all the criteria for providing high-quality numbers that will stand the test of time. Thus, we still seek the true numbers that Lord Kelvin exhorts us to use. However, on the principle that progress in measurement is a generally worthy goal, our approach might be considered a step in the right direction. In particular, this work demonstrates that it is possible in some cases to automate testing a large amount of software without simply turning a large testing effort into a similarly large exercise in providing detailed information to the automated testing system.

### 1.1. General existing techniques for software measurement

It is, of course, currently possible to measure many properties of software. The most mature of these approaches were born of the so-called software crisis that has somehow managed to last, at this point, several decades. These metrics attempt to measure productivity and thus evaluate ways to improve that productivity. They include the venerable source lines of code (SLOC), the widely used cyclomatic complexity metrics [27], and other more advanced metrics (*e.g.,* as discussed in [16]). While these metrics variously use measurements of the software itself, the software is not being evaluated for inherent utility. Instead, software is measured to determine complexity, which is then used to estimate development and maintenance costs (and, perhaps indirectly, to generate an estimated defect rate).

More recently, a wide variety of metrics and measurement methodologies have been proposed for the software reliability area. These metrics tend to use varying combinations of software complexity, development-phase defect discovery, testing-phase defect discovery, and in-service defect discovery to make predictions about residual defect rates for new and existing software systems (a broad discussion of such approaches can be found in [16] and [26]).

While software reliability metrics are definitely a productive step in the right direction, their use is not without problems. Testing-based approaches amount to requiring a software test suite development effort proportional to developing the original software in the first place, and thus are expensive. While in most cases testing of normal functionality is seen as necessary, there is a problem in that exception handling code and other portions of the system can overshadow the basic functionality in terms of testing complexity [5]. This means that in all but the most extreme safety-critical cases, limited time, budget, and manpower often result in testing effort being concentrated on "normal" functionality, giving short shrift to dependability aspects. In other words, reliability testing can fall victim to the problems of cost and not being seen as a primary value-added business activity beyond a certain point.

An alternate approach, that is often combined with a testing-based approach, is one based on monitoring defect discovery rates. If one assumes that defects are discovered at low marginal cost (*e.g.*, through an extensive beta testing program or a mandated functional correctness testing program), then such monitoring becomes a reasonably low cost exercise. However, this approach does suffer from the potential problem of delaying measurement feedback until downstream in the development cycle, when problems are potentially more expensive to fix. Additionally, defect rate monitoring has the problem that (presumably) the important defects are fixed, altering the system under test over time.

### 1.2. Robustness testing as an example of potentially general approach

This paper describes the attainment of a relatively generic, low cost, scalable, portable testing and measurement methodology for robustness. In particular, the methodology measures the robustness of an Application Programming Interface (API) with respect to exceptional

parameter values. The current results have implications for fault tolerance, software assurance, and computer security, and the methodology might be extended further.

Robustness was selected for measurement because system crashes are a way of life in any real-world system, no matter how carefully designed. Software is increasingly becoming the source of system failures, and the majority of software failures in practice seem to be due to problems with robustness [10]. Thirty years ago, the Apollo 11 mission experienced three computer crashes and reboots during descent to lunar landing, caused by exceptional radar configuration settings that resulted in the system running out of memory buffers [19]. Decades later, the maiden flight of the Ariane 5 heavy lifting rocket was lost due to events arising from a floating point-to-integer conversion exception [24]. Today, exceptional conditions routinely cause system failures in telecommunication systems, desktop computers, and elsewhere. Given that such problems have persisted in mission-critical space computers for decades, it is not difficult to predict they will still be with us several decades hence in everyday computing. However, the problem then will be that everyday computing will have completed its change from luxury to necessity, making such failures potentially devastating instead of merely inconvenient or expensive.

Thus, the Ballista project was created to measure the robustness of systems. Because operating system (OS) software underlies most large systems, the first application area that was chosen was to produce quantified results for robustness testing of full-scale, off-the-shelf operating systems. Automated testing was performed on fifteen POSIX [18] operating system versions from ten different vendors across a variety of hardware platforms. More than one million tests were executed in all, covering up to 233 distinct functions and system calls for each OS. Many of the tests resulted in robustness failures, ranging in severity from complete system crashes to false indication of successful operation in unspecified circumstances.

In brief, the Ballista testing methodology involves automatically generating sets of exceptional parameter values to be used in calling software modules. The results of these calls are examined to determine whether the module detected and notified the calling program of an error, and whether the task or even system suffered a crash or hang. There are two key approaches that make this approach inexpensive and scalable: the use of an object-oriented testing approach based on data types rather than functionality, and the fact that testing is performed for a relatively simple property of robustness rather than for the more complex property of correctness.

In the following sections we shall discuss previous work in this area, the Ballista testing methodology, and potential applications of that methodology to the areas of fault tolerance, software assurance, and computer security. While part of the story is simply the application of existing Ballista results and methodologies to these areas, the rest of the discussion will be more speculative, and hypothesize ways in which the approach might be generalized and extended.

## 2. Previous work

While the Ballista robustness testing method described in this paper 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. In software testing terms, Ballista performs tests for responses to exceptional input conditions (sometimes called "dirty" tests, which involve exceptional situations, as opposed to "clean" tests of correct functionality in normal situations). The test ideas used are based on "black box," or functional testing techniques [Bezier95] in which only functionality is of concern, not the actual structure of the source code. However, Ballista in its current form is not concerned with validating functionality for ordinary operating conditions, but rather determining whether or not a software module is

robust.

In common usage, the term robustness might refer to the time between operating system crashes under some usage profile. However, the authoritative definition of *robustness* that will be used in the current discussion is "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [17]. This expands the notion of robustness to be more than catastrophic system crashes, and encompasses situations in which small, recoverable failures might also occur. Ballista work concentrates on the portion of robustness dealing with invalid inputs. While 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 a single invocation of a software module from a single execution thread (however, as will be discussed, this is far less limiting an approach than one might think).

### 2.1. Crashme

An early and well known method for automatically testing operating systems for robustness was the development of the Crashme program [7]. Crashme operates by writing randomized data values to memory, then spawning large numbers of tasks that attempted to execute those random bytes as programs. While 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 it may eventually find some way to crash the system).

The crashme approach has the distinct advantage that it is relatively easy to implement, and can be ported readily to a wide variety of systems. It also has the compelling virtue that it actually does find ways to crash operating systems. However, as a technique it is not readily applied to other application areas, and does not produce repeatable, deterministic results that are desirable for concise bug reports. Finally, it is not really designed to quantify and compare systems so much as discover problems in a particular system.

### 2.2. Fault injection

Other work in the fault injection area has also tested aspects of robustness. The Fiat system [3] uses probes placed by the programmer to alter the binary process image in memory during execution. The Ferrari system [20] 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). The FTAPE system [37] injects faults into a system being exercised with a random workload generator by using a platform-specific device driver to inject the faults. While all of these systems have produced interesting results, none was intended to quantify robustness on the scale of an entire OS API.

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 [29], and then discovered that the problems were still present in operating systems several years later [30]. The Fuzz approach tested specific OS elements and interfaces (compared to the completely random approach of Crashme), although it still relied on random data injection.

The Ballista testing approach builds upon a legacy of work in the fault injection area performed by Siewiorek and his students at Carnegie Mellon University. This work has long sought a combination of repeatability, high-level implementation, and portability (quantification was not an explicit goal, but can be accomplished given that the other goals have been met).

A first effort at Carnegie Mellon was the creation of a descendent of crashme called CMU-crashme. Instead of executing random data, CMU-crashme restricted randomness to parameter values for concurrently executed system calls. This had the effect of concentrating testing on the OS interface, and eliminating a large number of tests that simply resulted in illegal instruction exceptions. It still, however, relied on serendipity to discover problems and was not repeatable.

The next generation of work was called robustness benchmarking, in which carefully selected data values were sent to selected function calls, with one call executed at a time. The effort was labor-intensive in that it required developing a test suite, but attained repeatability. And, perhaps surprisingly, even with a single-call model it found ways to crash operating systems from user mode, including a space-qualified fault tolerant aerospace computer [14]. Eventually, several operating systems were tested with a handful of calls, demonstrating portability, and revealing that robustness failures were prevalent across various systems [22].

Once repeatability had been achieved, the obvious next step was to genericize the tests and attain scalability for larger APIs. Thus, work was done to extend the robustness benchmarks by genericizing different operations within a function and attempting to associate them with different test data types [31]. For example, many functions serve to create a data object, whether it be a file, an allocated block of memory, or a data structure. Similarly, other functions modify, delete, and copy such objects. Thus, one could think of genericizing the function of a particular module into one of a preset category, and then direct testing based on a combination of generic function and data type. While this approach did work, attempts to scale it to full-size systems were unsuccessful because the effort required to specify generic functionality scaled linearly with the number of functions being tested. The approach remains promising, but is not without further issues to be resolved.

After attempting to genericize functions, a fresh approach to the problem was taken by eliminating the involvement of functionality altogether, and creating test cases based on data type information alone. This approach was the source of the Ballista testing methodology [23], which attained repeatability, portability, and scalability, and is described in more detail in later sections. As an additional benefit, the ability to perform over a million tests on more than a dozen operating systems provided enough test data to create a strategy for quantification of robustness.

### 2.3. Software testing

Given a desire to provide crashme-like testing in a more deterministic, repeatable setting, one can turn to software testing to borrow 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 often placed on code coverage as a way of assessing whether a module has been thoroughly tested. 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 whether a check for invalid data is missing from the program 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 [5]. 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.

Automatically generating software tests requires three things: a Module under Test (*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 individual behavioral specifications.

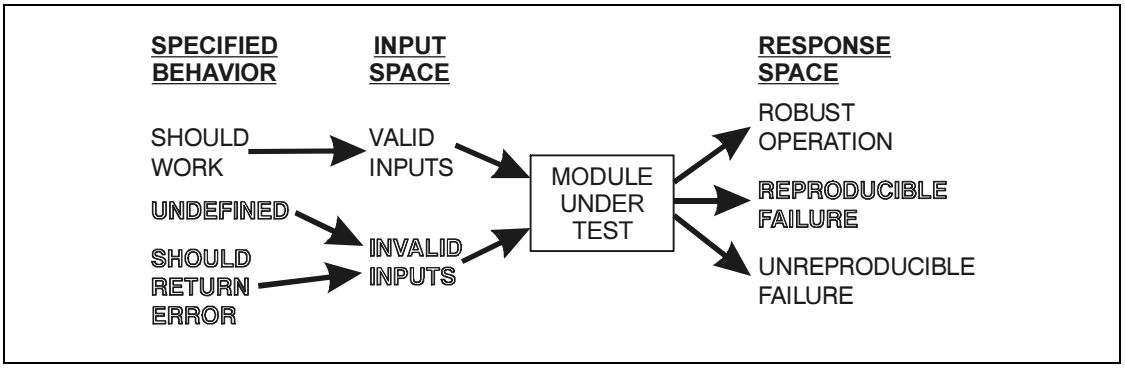


Figure 1. Ballista has a goal of identifying reproducible failures for invalid inputs to a module under test.

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.

### 2.4. Instrumentation approaches

There are several commercial products that help in developing robust code, such as Purify [34] and Boundschecker[32], which instrument software to detect exceptional situations. They work by detecting exceptions that arise during development testing or usage of the software. However, they are not able to find robustness failures that might occur in situations which are not tested (and, even with what would normally be called 100% test coverage, it is unlikely in

practice that every exceptional condition which will be encountered in the field is included in the software test suite). The Ballista approach differs from, and complements, these approaches by actively seeking out robustness failures; rather than being an instrumentation tool, it actually generates tests for exception handling ability and feeds them directly into software modules. Thus, Ballista is likely to find robustness failures that would otherwise be missed during normal software testing, even with available instrumentation tools.

## 3. The Ballista scalable robustness testing framework

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 this 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.

### 3.1. Ballista approach

The Ballista robustness testing methodology is based on combinational tests of valid and invalid parameter values for system calls and functions. In each *test case*, a single software MuT is called a single time to determine whether it is robust 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. 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 could be described as a tuple: *{MuT_name, test_value1, test_value2, ...}* corresponding to a procedure call of the form: *MuT_name(test_value1, test_value2, ...)* ) . Thus, the general approach to Ballista testing is to test the robustness of a single call to a MuT for a single tuple of test values, and then repeat this process for multiple test cases that each have different combinations of valid and invalid test values. A detailed discussion follows.

In Ballista, tests are based on the values of parameters and not on the behavioral details of 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 written 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

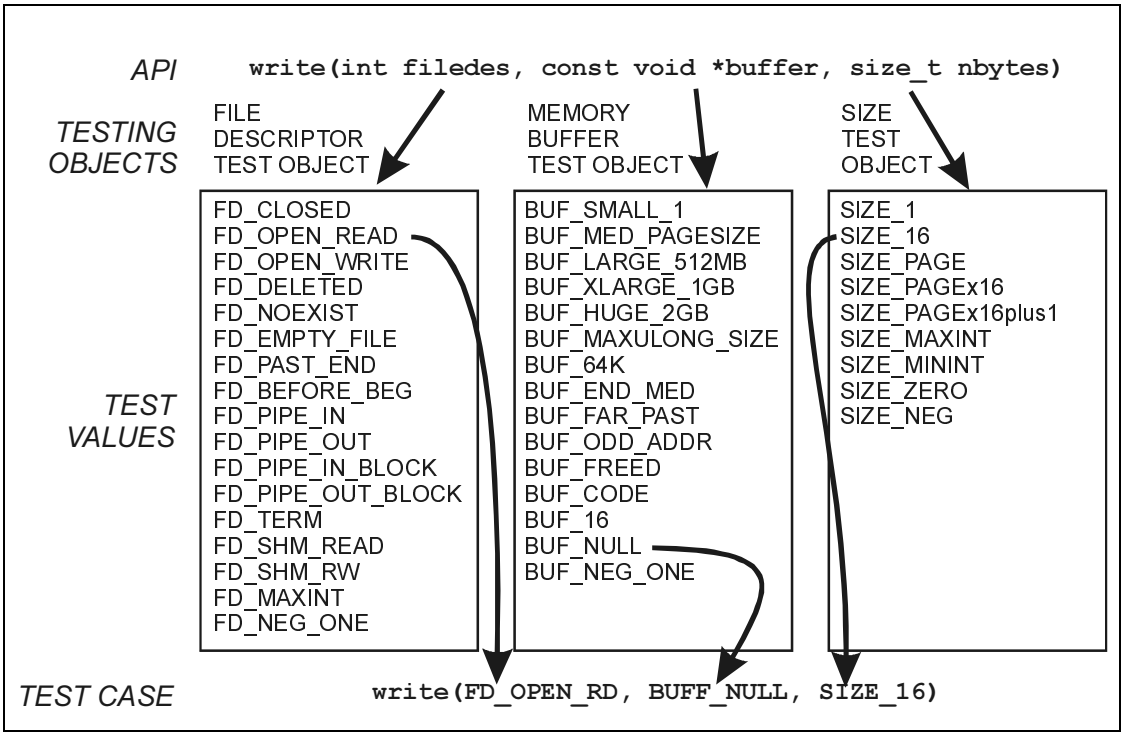


Figure 2. Each test case is composed of a function name with a tuple of test values. The test values are drawn from testing objects based on the data types of the API call being tested.

object (which includes selecting a test value from the list of available values) executes the appropriate constructor function that builds any required testing infrastructure (often called "scaffolding"). 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, then open the file requesting read permission. An example of a test constructor to create a file open for read is shown in Figure 2.

When a test case has completed execution, the corresponding destructors for the test values perform 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.

Inheritance of test values is also possible, although not shown explicitly in Figure 2. In particular, test values for native machine data types (*e.g.,* integer, character, and pointer) are predefined as base test values. Other data types inherit these test values and add additional test values that are more specific to their particular interpretation. Additionally, data structures inherit the test values of their constituent element data types.

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 without regard to the functionality of the MuT. In Figure 2, the name of any module taking the parameter types `(fd, buf, len)` could be tested simply by changing the `write` to some other function name. All test scaffolding creation is both independent of the behavior of the function being tested, and completely encapsulated in the testing objects.

### 3.2. Test harness

As a matter of efficiency, a single program for each MuT is compiled which is capable of generating all required test cases. A main program spawns a task which executes a single test case, with the main program using a watchdog timer to detect hangs and monitoring the task status to detect abnormal task terminations. The main program iterates through combinations of test values to execute all desired test cases. While one might think that executing tests in this manner would lead to failures caused by interactions among sequential tests, in practice essentially all failures observed in OS testing have been found to be reproducible in isolation. This independence of tests is attained because the constructor/destructor approach sets and restores a fresh copy of relevant system state for each test. (The independence may not hold in all situations, and especially in systems lacking hardware memory protection. However, understanding that issue in more detail is a subject of future research.)

While automatic generation of a large variety of test values is described in the next section, particular values used for the results reported in this section are hand-picked. An average of ten test values were selected by hand for each test object based on experience gained both in general programming and from previous generations of fault injection experiments. For most MuTs, an exhaustive testing approach is used in which all combinations of test values are used to create test cases. For a half-dozen of the POSIX calls tested, the number of parameters is large enough to yield too many test cases for reasonable exhaustive coverage; in these cases a pseudo-random sampling of 5000 test cases is used (based on a comparison to a run with exhaustive searching on one OS, the sampling gives results accurate to within 1 percentage point at a small fraction of the execution time).

It is important to note that this testing methodology does not generate test cases based on a description of MuT functionality, but rather on the data types of the MuT's arguments. This means that, for example, the set of test cases used to test write() would be identical to the test cases used to test read() because they take identical data types. This testing approach means that customized test scaffolding code does not need to be written for each MuT -- instead the amount of testing software written is proportional to the number of data types. As a result, the Ballista testing method was found to be highly scalable with respect to the amount of effort required per function, needing only 20 data types to test 233 POSIX function calls. An average data type has 10 test cases, each of 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).

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 only a dozen lines or fewer of executable code in size, these short programs contain the constructors for each parameter, the actual function call, and destructors. These single test cases can be used to reproduce robustness failures in isolation for use by developers for verifying problems, by users as "bug reports," or to verify test results in isolation.

### 3.3. Categorizing test results

After each test case is executed, the Ballista test harness categorizes the test results according to the CRASH severity scale: [23]

- **Catastrophic** failures occur when the entire OS becomes corrupted or the machine crashes or reboots. In other words, this is a complete system crash. These failures are identified manually because of difficulties encountered with loss of newly written, committed, file data across system crashes on several operating systems.

- **Restart** failures occur when a function call to an OS function never returns, resulting in a task that has "hung" and must be terminated using a command such as "kill -9". These failures are identified by a watchdog timer which times out after several seconds of waiting for a test case to complete.
- **Abort** failures tend to be the most prevalent, and result in abnormal termination (a "core dump") of a task caused by a signal generated within an OS function. Abort failures are identified by monitoring the status of the child process executing the test case.
- **Silent** failures occur when an OS returns no indication of error on an exceptional operation which clearly cannot be performed (for example, writing to a read-only file). These failures are not directly measured, but can be inferred as discussed in Section 6.4.
- **Hindering** failures occur when an incorrect error code is returned from a MuT, which could make it more difficult to execute appropriate error recovery. Hindering failures have been observed as fairly common (forming a substantial fraction of cases which returned error codes) in previous work [22], but are not further discussed in this paper due to lack of a way to perform automated identification of these robustness failures.

There are two additional possible outcomes of executing a test case. It is possible that a test case returns with 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 desired 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 no so far past as to trigger a memory protection exception by touching an unallocated virtual memory page).

One of the tradeoffs made to attain scalability in Ballista testing is that the test harness has no way to tell which test cases are valid or invalid for any particular MuT. Thus, some tests returning no error code are Silent failures, while others are actually a set of valid parameter values which should legitimately return with no error indication. This has the effect of "watering down" the test cases with non-exceptional tests, making the raw failure rates underestimates. We estimate that 12% of tests are of this non-exceptional type for OS tests; but even the raw failure rates are significant enough to make the point that Ballista testing is effective in finding robustness failures.

## 4. Results

In all 1,082,541 data points were collected. Operating systems which supported all of the 233 selected POSIX functions and system calls each had 92,658 total test cases, but those supporting a subset of the functionality tested had fewer test cases. The final number was dependent on which functions were supported and the number of combinations of tests for each supported function.

### 4.1. Measured failure rates

The compilers and libraries used to generate the test suite were those provided by the OS vendor. In the case of FreeBSD, NetBSD, Linux, and LynxOS, GNU C version 2.7.2.3 was used to build the test suite.

There were five different functions that could be made to cause entire operating system crashes on some OS (either automatic reboots or system hangs). Restart failures were relatively scarce, but present in all but two operating systems. Abort failures were common, indicating that in all operating systems it is relatively straightforward to elicit a core dump from an

instruction within a function or system call. (Abort failures do not have to do with subsequent use of an exceptional value returned from a system call -- they happen in response to an instruction within the vendor-provided software itself.) A check was made to ensure that Abort failures were not due to corruption of stack values and subsequent corruption/misdirection of calling program operation.

MuTs that underwent a catastrophic failure could not be completely tested, and resulted in no data on that MuT other than the presence of a catastrophic failure. Since the testing suite is automated, a system crash leaves it in an unrecoverable state with respect to the function in question. Further testing a function which suffered a catastrophic test would require either manual execution of individual cases, or adding tricky waiting and synchronization code into the test benchmark. Manual execution was performed for the Irix 6.2 catastrophic failure in munmap, and allowed the identification of this single line of user code which can crash the entire OS, requiring a manual reboot:

```
munmap(malloc((1<<30+1)),MAXINT);
```

### 4.2. Normalized failure rates

While it is tempting to simply use the raw number of tests that fail as a comparative metric, this approach is problematic. Most OS implementations do not support the full set of POSIX real-time extensions, so the raw number of failures cannot be used for comparisons. In addition, the number of tests executed per MuT is determined by the number and types of the arguments. So, a single MuT with a large number of test cases could significantly affect both the number of failures and the ratio of failures to tests executed. Similarly, an OS function with few test cases would have minimal effect on raw failure rates even if it demonstrated a large percentage of failures. Thus, some sort of normalized failure rate metric is called for, and is reported in the last column of Table 1.

Table 1: Measured failure rates for fifteen POSIX operating systems.

| System | POSIX Calls Tested | Calls with Catastrophic Failures | Number of Tests | Abort Failures | Restart Failures | Normalized Abort + Restart Rate |
|---|---|---|---|---|---|---|
| AIX 4.1 | 186 | 0 | 64009 | 11559 | 13 | 9.99% |
| FreeBSD 2.2.5 | 175 | 0 | 57755 | 14794 | 83 | 20.28 |
| HPUX 9.05 | 186 | 0 | 63913 | 11208 | 13 | 11.39 |
| HPUX 10.20 | 185 | 1 | 54996 | 10717 | 7 | 13.05 |
| IRIX 5.3 | 189 | 0 | 57967 | 10642 | 6 | 14.45 |
| IRIX 6.2 | 225 | 1 | 91470 | 15086 | 0 | 12.62 |
| Linux 2.0.18 | 190 | 0 | 64513 | 11986 | 9 | 12.54 |
| Lynx 2.4.0 | 222 | 1 | 76462 | 14612 | 0 | 11.89 |
| NetBSD 1.3 | 182 | 0 | 60627 | 14904 | 49 | 16.39 |
| Digital Unix 3.2 | 232 | 1 | 92628 | 18074 | 17 | 15.63 |
| Digital Unix 4.0 | 233 | 0 | 92658 | 18316 | 17 | 15.07 |
| QNX 4.22 | 203 | 2 | 73488 | 20068 | 505 | 20.99 |
| QNX 4.24 | 206 | 0 | 74893 | 22265 | 655 | 22.69 |
| SunOS 4.13 | 189 | 0 | 64503 | 14227 | 7 | 15.84 |
| SunOS 5.5 | 233 | 0 | 92658 | 15376 | 28 | 14.55 |

We define the normalized failure rate for a particular operating system to be:

$$F = \sum_{i=1}^{N} w_i \frac{f_i}{t_i}$$ with a range of values from 0 to 1 inclusive, where:

$N$ = number of functions tested

$wi$ is a weighting of importance or relative execution frequency of that function where

$$\sum w_i = 1$$

$f_i$ is the number of tests which produced robustness failure for function $i$

$t_i$ is the number of tests executed for function $i$

This definition produces a sort of exposure metric, in which the failure rate within each function is weighted and averaged across all functions tested for a particular OS. This metric has the advantage of removing the effects of differing number of tests per function, and also permits comparing OS implementations with differing numbers of functions implemented according to a single normalized metric. For the results given in Table 1 and the remainder of this paper, an equal weighting is used (*i.e.*, $w_i = 1/N$) to produce a generically applicable result. However, if an OS were being tested with regard to a specific application, weightings should be developed to reflect the dynamic frequency of calling each function to give a more accurate exposure metric.

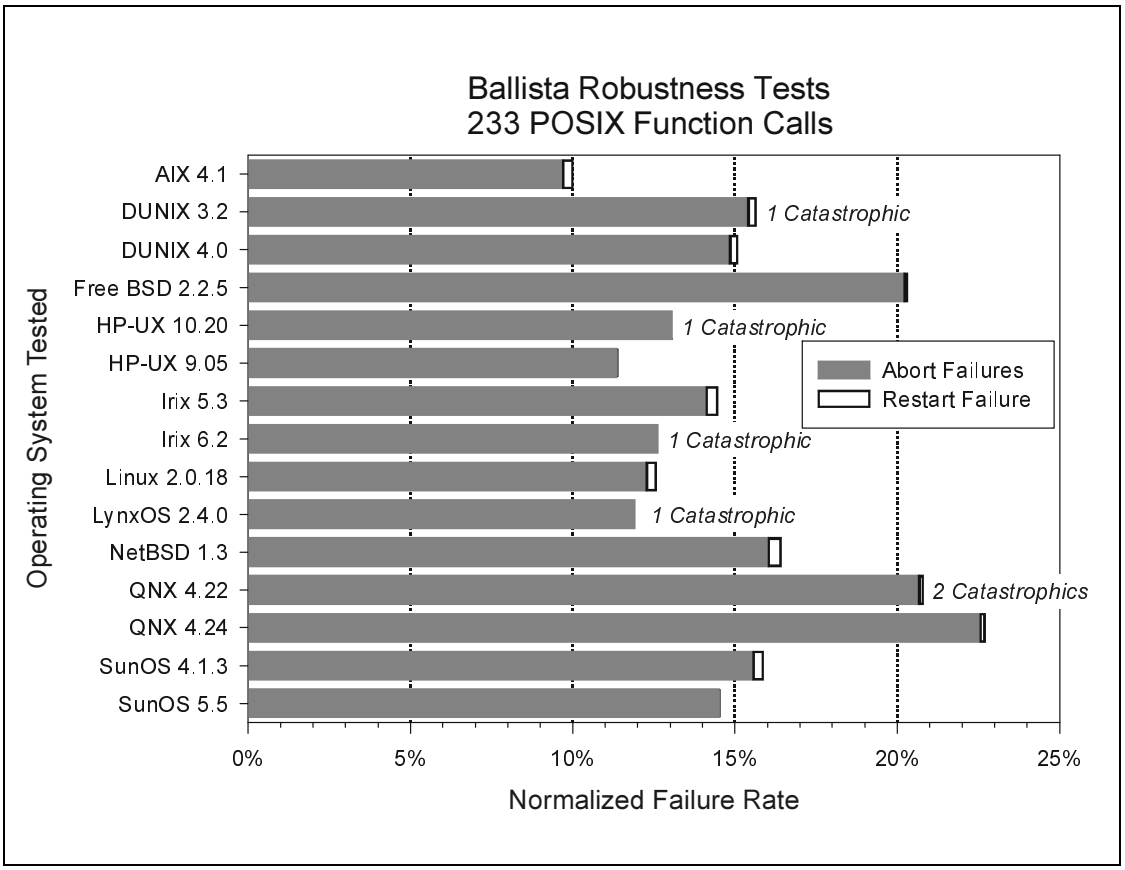


Figure 3. Normalized robustness failure rates for fifteen POSIX operating systems.

Figure 3 shows the normalized failures rates as calculated for each operating system (this is the same data as the rightmost column in Table 1). Overall failure rates for both Abort and Restart failures range from the low of 9.99% (AIX) to a high of 22.69% (QNX 4.22). The mean failure rate is 15.158% with a standard deviation of 3.678 percentage points. As mentioned previously, these are raw failure rates that include non-exceptional test cases which dilute the failure rates, and do not include Silent failures.

## 5. Generalizing Ballista & applying to other interfaces

The initial approach to Ballista and its predecessors was focussed on testing an increasingly larger set of operating system calls. It is, of course, natural to be skeptical that the approach can scale to increased thoroughness and other applications. To date, implementations have been completed to the proof-of-concept point in a simulation API and a mechanism has been created for performing finer-grained testing.

### 5.1. Experience with distributed simulation API

Testing operating system calls requires the ability to test traditional subroutine-call-plus-parameter-list interfaces with single calls and a minimum of system state. However, other APIs may have significantly different requirements. To probe the extensibility of Ballista, an entirely different API is being tested, based on a distributed simulation system.

The High Level Architecture (HLA) Run Time Interface (RTI) is an API used for a distributed simulation system by the U.S. Department of Defense [11]. This system is written in C++ in an entirely object-oriented style, and has significantly more complicated preconditions for function calls. Differences from OS testing include:

- An object-oriented implementation, meaning that methods are to be tested rather than function calls. This can largely be treated as a syntactic difference.
- A system which requires support for call-backs to objects for testing rather than a simple call to a function to be tested. This was handled with a slight modification to the standard Ballista client test scaffolding.
- Significantly more complex data types. It was feared that a proliferation of data type variants would increase the difficulty of testing them. However, in the end only 13 data types were required for 87 interfaces. And of these data types, 2 were completely re-used from operating system testing, while the rest inherited existing test cases. Thus, performing tests on data types proved moderately scalable. Perhaps just as importantly, these same data types would be used heavily by any HLA application program, and could thus be reused for Ballista testing of such application programs at a future date, resulting in even more effective scaling.
- A system with moderately complicated preconditions for each and every method (for example: to perform an operation, first a so-called federation must be created; then a federate must be created and instructed to join the federation; then the constructors for that federate must be executed depending on data types). Upon further examination, however, it was found that the different methods being tested could be collapsed into only twelve different equivalence classes with respect to test scaffolding. Thus only 10 sets of scaffolding are required (each only a few lines of code long) for 86 modules under test, which as with the data types is a sub-linear scaling with respect to number of methods being tested.
- Support for error reporting by throwing exceptions rather than error return codes. This was done by adding an exception handler to the Ballista test harness, with the requirement that the tester provide a way to distinguish between acceptable exception values and "unknown" exceptions that, when thrown, amount to Abort failures.

### 5.2. Structure testing

Testing of data structures can quickly become tedious if separate data types must be created to test each different structure. Fortunately, this can be overcome automatically by creating a system that can parse structure definitions and create composite constructors in terms of the primitive components of a structure. As a simple example, a complex number can be treated as a structure containing two floating point numbers, and tests for that data type can be generated simply by providing the structure information, given that floating point numbers can already be tested as a base data type.

It must be kept in mind that a data structure might be pointed to as well as referenced directly, so test values for a structure inherit both pointer tests (*e.g.*, null pointer instead of a pointer to the structure) as well as various data value tests (*e.g.*, pointer to a structure with particular values).

### 5.3. Finer grain testing -- dials & logical structures

One of the original (and current) goals of Ballista is to automatically generate hardening wrappers based on test results. While that work is still ongoing and beyond the scope of this discussion, it led to an early focus on a need for using testing as a characterization mechanism in addition to merely a quantification mechanism. For example, it is not enough to simply say that a particular module exhibits robustness failures at a certain rate, but also one must know exactly what triggers these failures. Being able to do this requires fairly fine-grained testing. As an example, the file handle tests described earlier are quite course grain, and certainly do not test all possible combinations (for example, they test a file open for write with no data, but not a file open for write with some data already in it). It can be readily seen that without some sort of further decomposition the number of possible test cases grows in a combinational explosion, and loses scalability.

Additionally, it is desirable to have a very large set of potential tests for Ballista to draw upon without having to manually create too many tests. A particular reason to want to do this is to implement randomized testing for the purposes of characterization, as well as adding an ability to discover new, heuristically useful, data type values for testing.

Fortunately a technique to genericize data testing is available in the form of decomposing the constructors for any particular data type test value creation. An analogy can be made to a set of dials that are set in a particular position for a particular test value. The dials are orthogonal (or at least loosely coupled in terms of meaningful sets of values), and each dial is backed by a set of enumerated constructors for particular aspects of a test value. (This approach is similar to the category/choice approach described in [33], but Ballista requires neither inter-category constraints nor specifically described test frames.) As an example, some of the dials for file creations and their settings include:

- File existence: exists, does not exist, deleted after file handle obtained
- File read permissions: read permission, read permission to other than current task, no read permission
- File access: all combinations of open for read, write, read+write, *etc.*

Note that the above dials are influenced by a desire to decouple various aspects of the data value, but remain coupled to the extent required by conveniently writing constructors. Thus, file permissions can be decoupled because they can be set independently after file creation, but file access modes are placed into a single dial because it was found to be awkward to manipulate them independently. In particular applications, the file data type might be expanded to include other dials such as the type of data present in the file.

While the addition of dials seems to complicate the testing situation, it turns out that there is a

unifying approach available -- logical structures. Every data type that is decomposed into dials can be logically treated as a data structure for testing purposes (the only difference from a physical structure point of view is that there is no test for a pointer to that data structure, since it only exists logically for the test program, not physically within the MuT). So, to continue the file handle example, testing file handles is done by declaring a file handle to be a logical structure containing a set of dials, and then exploring various combinations of dials in exactly the same way that Ballista would test combinations of values within a physical data structure.

### 5.4. Potential improvements in characterization

Beyond simply measuring failure rates, Ballista testing has the potential to characterize failure patterns through the use of patterned testing on fine-grain dial-based representations of data values. Previous work in the area of test exploration includes the AETG system [9], which permits testers to specify which sets of parameters in a multi-parameter function calls are closely coupled functionally, and thus should be tested in concert. The TOFU system [6] takes this a step further by adding interaction weightings to the identified parameter tuples.

Clearly the testing approaches from AETG and TOFU can be applied in part to Ballista testing in the form of encouraging testing of one or a few parameters at a time. It is probably undesirable to have a programmer give explicit hints to Ballista with respect to coupling (this reduces scalability, and there is always the chance that the tester will fail to see a subtle coupling and provide an inaccurate hint). However, heuristics are being developed to permit Ballista to automatically discover what couplings may be present, to do exploration around general regions of interest in the boundary between tests that succeed and that fail, and to generally do "smart" testing to characterize failures with a reasonable number of tests.

The current mechanism for making Ballista available is as a web-based testing service rather than as a piece of software that is executed locally. Part of the reason for this strategy is to enable data collection that can be used to improve the quality of characterization approaches.

## 6. Potential applications to fault tolerance

Traditional fault tolerance techniques are often based on the notion of creating dependable systems out of individually undependable components. The general idea is that if individual component failure rates are known, then a variety of techniques can be used to ensure that a composite system is highly dependable. It is of course unlikely that Ballista testing will produce highly accurate reliability estimates in terms of failures per million operating hours as is done for fielded hardware. Nonetheless, it is probably a worthy goal to produce some relative measure of software component reliability, and a rating of exception handling ability might be a useful starting point.

There are four areas in which a Ballista testing might provide improved quantification for fault tolerance of software-intensive systems: detecting system crashes, detecting violations of fail-stop assumptions, providing information for creating software wrappers, and providing an assessment of software diversity.

In general, it is considered a good idea to build systems that are impossible to crash from "normal" software applications. This is an extension of the traditional fault tolerance approach of having a "hard core" that is as small as possible, and upon which resources are lavished in both design and implementation to ensure its dependability. Outside of that hard core of a processor or operating system kernel, it is to be expected that bugs will occur on a regular basis, generating exceptional situations that must be handled robustly with respect to the system architecture. The particular area of emphasis for Ballista is on exercising exception handling

techniques, since it is assumed that normal, correct values are well exercised by normal operational testing.

### 6.1. Test for overall system crash situations

The most obvious application of Ballista to fault tolerance is to ferret out ways to completely crash a system from unprivileged application software. As the previous results indicate, Ballista can locate some ways to crash an operating system even with the current rather simplistic tests available. Application of a similar methodology should help quantify the extent to which exceptional conditions can trickle down through an application program to reach the operating system as well.

There are four approaches that can improve the effectiveness of Ballista to identify system crashes. The basic techniques required are those discussed in the previous section.

- Use finer grain testing using the dial approach to decomposing parameter test values
- Test primitive functions within the operating system rather than just the POSIX-specified functions.
- Test concurrent function calls, to identify timing-dependent problems such as resource locking bugs.
- Test for the results of resource exhaustion effects, such behaving gracefully when the process table is full or when a memory leak is encountered.

### 6.2. Test for violations of fail-stop assumption

A second way to apply Ballista to measuring and improving Fault Tolerance is to use it to detect problems with respect to Restart and Silent failures. It is common to base fault tolerance on a fail-fast assumption. However, a Restart failure means that a task is "fail-slow" in that its failure is not recognized until, typically, a time-out period has elapsed (failure to detect a periodic "heartbeat" message, failure to reset a watchdog timer, *etc.*) Furthermore, a Silent failure means that a system has failed somehow, but that the failure is undetected from some period of time. It might be that the Silent failure is not activated, or it might be that it results in a "time bomb" inside the system, such as a NULL pointer to a data structure that is not dereferenced except in an error handling routine.

Restart failures are moderately easy to deal with, and basically are searched for in the same way that catastrophic system failures are searched for as discussed above. A particularly nasty restart failure that was found on a real system was in AIX, in which the core dump facility waited until enough disk space on a nearly full disk was available to dump core, even if core dumping was disabled (the check for disk space came before the check for the maximum amount to dump, which in our case was set to zero).

Silent failures can be identified by using multi-version comparison among different OS test results. This scheme determines declares a result to be a Silent failure if that test case on that OS returns no error indication while that same test case results in an error indication on another OS. This approach works with reasonable accuracy in practice (approximately 80% accurate) [12], but of course requires that there be multiple implementations available. Improving performance for Silent failures takes one of two approaches. In some cases hardware is available that could detect a Silent failure, but is not being used (for example, because memory protection is turned off for page zero of memory to support legacy software). In other cases it is simply that the exceptional parameters are not being checked, and in addition evade detection by hardware mechanisms.

### 6.3. Generating software wrappers

A possible way to improve software fault tolerance is by using software "wrappers" that encapsulate the modules of a software system. While it is aesthetically more pleasing to fix any exception handling deficiencies in a software module directly, there are times when that is not possible either because of lack of access to the source code or time/budget constraints. Therefore it may be attractive to automatically or semi-automatically generate software wrappers that are interposed between the API implementation and the application. These wrappers can automatically check for exceptional conditions and take action to report them or, in some cases, recover from them.

While a generic capability to create universal software wrappers remains a research issue, there are several strategies that may be employed within a wrapper, such as reporting error return codes, performing retry, and invoking alternate algorithms. A particularly interesting use of wrappers in connection with robustness testing is to use a robustness testing result as a warning that a particular operation is hazardous, and to perform a checkpoint against the need for a later rollback if that operation should fail. This would use failure predictions from robustness testing to lessen the typical cost of checkpointing to those times when it would appear to be most fruitful.

The Xept approach [38] uses software "wrappers" around procedure calls as a way to encapsulate error checking and error handling within the context of a readable program. Given that the fault tolerance community has found that transient system failure rates far outnumber manifestations of permanent system faults/design errors in practice, even as simple a strategy as retrying failed operations from within a software wrapper has the potential to significantly improve system robustness. One could therefore envision using the results of Ballista testing to provide information for error checking within an Xept software wrapper.

### 6.4. Potential use as measure of diversity for multi-version software systems

A method for using Ballista for Fault Tolerance that does not directly involve robustness testing *per se* is in the area of assessing implementation diversity for multi-version systems. One way to improve dependability is to use multiple, independently-developed versions of software in the hope that different versions are diverse in nature, and thus have differing failure modes [8][1]. This is a way of exploiting inherent design diversity in separately designed systems, in an analogy to exploiting diversity among multiple independently-manufactured hardware subsystems.

One issue with using multi-version software is whether or not such software is truly diverse. There are studies and arguments for both sides of this issue (*e.g.*, [2][4][15][21][25][28]). However, Ballista testing gives a way to shed additional light onto the diversity question. In particular, it can measure the diversity of multiple software versions with respect to exception handling capabilities. (Diversity with respect to correctness would hinge on implementing software assurance abilities discussed later.) It can do so by comparing the results of identical test cases on multiple implementations, and detecting instances in which at least one version detected an exception, indicating that a notional multi-version system could have detected such an exception and responded gracefully even if all systems did not. This is a bit different than traditional multi-version voting in that it suffices for a single version of software to detect an exception, whereas voting typically requires a majority of versions to agree on a correct computational result.

Multi-version assessment of operating systems was performed with Ballista test data [12], and found that OS kernel calls are moderately diverse (but not completely diverse). However, it was

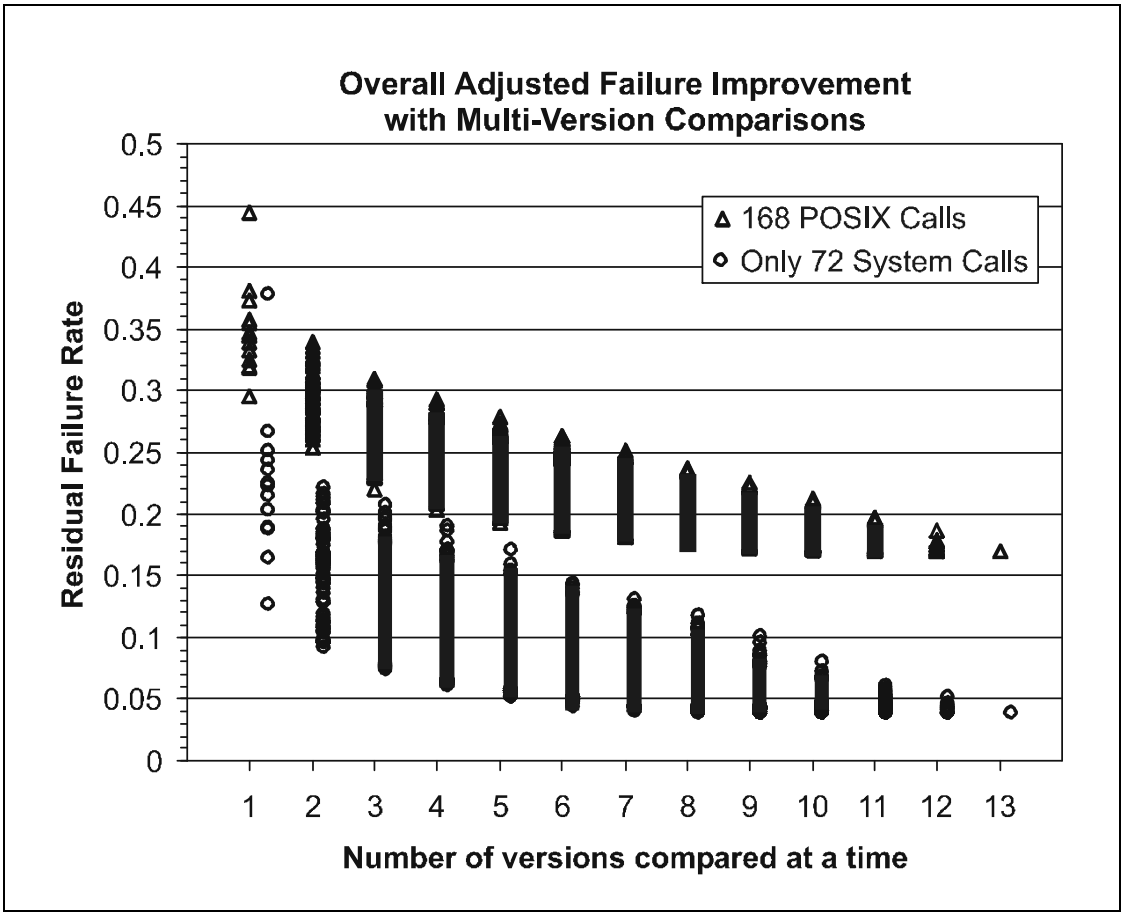


Figure 4. Overall adjusted failure rates for thirteen POSIX operating systems using multi-version failure detection comparison.

also found that C library function calls display a relatively poor level of diversity. Figure 4 shows a scatter plot of the results attained by using all combinations of multi-version detection among thirteen operating systems across 168 function calls [12]. As can be seen, the overall OS interface cannot be improved beyond a 17% failure rate even if all thirteen OS versions are used for comparison. However, if the system call subset is considered in isolation (*i.e.*, those calls not part of the C library), then the failure rate is reduced dramatically (but not eliminated completely) via comparison. This suggests that, at least with respect to exception handling, the C library functions are not particularly diverse.

## 7. Potential generalization to Software Assurance

Software assurance research has produced many metrics. Some of these metrics are based on measuring the complexity of the software itself. Some of the metrics are more process-based, such as software shipping decision heuristics. And some of the approaches are not metric-based, but rather based on assuring complete correctness, such as with formal methods.

Within the realm of software assurance, Ballista testing is akin to an automated black box testing approach. Since black box testing is about functionality, the question becomes one of how much functionality can be put into the testing while retaining scalability.

### 7.1. Robustness metric for COTS software component libraries

In a large number of software systems, exception handling is dealt with incompletely in the software specification. This is done intentionally, and usually indicated by phrases such as a behavior being "undefined" or "unspecified."

In some systems, robustness is not an overriding concern, but in others (and in particular ones in which security and fault tolerance are significant concerns), it may be important. In these latter cases Ballista testing can serve to exercise exception handling code as already described, and create a metric for exception handling ability. It should be noted that robustness failures found by Ballista are not necessarily "bugs" or the results of "defects", because those terms can be construed to be with respect to the original software specification, which may indeed state that behavior in a particular situation is "undefined." However, if a general-purpose software component library is being used in a critical system, it may be advantageous to select a library with the best robustness properties, or at least have some measure available of the robustness of the library selected.

### 7.2. A quality assurance metric for systems with rigorous exception handling

Some systems are specifically designed to be robust. An example is the earlier one of the HLA RTI, which is designed to report error codes for any possible exception. Testing results to date have found that most exceptions are in fact reported, but most software modules do have a few exceptions that go unreported. And, in an interesting result, a few modules were found to have a much higher failure rate than the other modules, suggesting areas of particular concern with respect to system robustness.

Thus, an additional use of Ballista testing for software assurance is a Quality Assurance check on software that is specifically written to implement rigorous exception handling. One way to employ this testing is to generate bug lists that must be fixed. But, especially in cost-sensitive systems, in can be used instead to find the particular modules that for some reason have far worse than average robustness and concentrate available resources on them first. Thus, Ballista testing might be considered a screening tool for which areas to concentrate corrective efforts upon, even if it does not find all existing problems.

### 7.3. Adaptation to black box testing with customized corner values, etc.

A potentially high impact, but also very difficult, extension of Ballista testing would be to use it to partially automate traditional software testing.

In traditional black box testing, every module is tested with respect to a specification. This approach tends to scale poorly because extra work must be performed to define a specification for every module to be tested, thus scaling effort linearly with the size of the API to be tested. The obvious way to attain scalability is to somehow genericize functionality so that it becomes more object oriented. And, in fact, that approach was taken with pre-Ballista robustness testing [31], but did not achieve scalability. It is possible, however, that in light of work on Ballista, the generic functionality approach could be made to work.

Another approach is to limit correctness testing to properties that can be represented as a linear superposition of test cases for individual parameters. In this approach, correctness tests with postcondition assertions would be added to Ballista testing, and the postconditions would be tested after each test was run. It is difficult to see how this approach could be made to deal with changes to system state caused by a function, because the changes would of necessity be related to functionality. But, the assertions could test for invariants across changes, such as a

string remaining null-terminated after any operation.

A simpler, less ambitious, extension would be to included customized test values for boundary testing for a particular module or data type in addition to current Ballista robustness tests. For example, a struct containing an X,Y coordinate pair to a graphics display routine might be tested with the coordinate pair 1024,768 to see whether an appropriate error code is returned for an off-screen point plot attempt. Such an approach might still scale because it is based on a data type that is (presumably) used by multiple software modules.

### 7.4. Generic behavioral descriptions other than for robustness?

The point at which applying Ballista to software assurance becomes tricky is when one desires to test for properties other than robustness. Beyond trying to apply data-type driven testing to correctness testing, a different way to extend Ballista testing is to find properties that have generic specifications applicable to all (or almost all) software modules, or very simple specifications that can easily be stated for each module. Potential examples of these include:

- Execution time ("module returns in less than X microseconds") where perhaps each module test will need to be annotated with a particular time.
- Memory consumption ("module consumes no more than X bytes of memory from the heap").
- Safety critical state change ("module does not alter any of a predefined set of system variables").

A further type of testing that might be performed would require machine-readable interface specifications to be available, and might thus cross the line into more general software assurance. However, if such specifications are available anyway, the incremental cost to perform Ballista-based testing using them might be reasonable. Example applications might include:

- Checking that all exceptions specified can be generated (failure to generate a documented exception either suggests an addition to the test set or points out something that is not consistent with the current implementation, which may or may not be a problem).
- Checking that exceptions generated are all in the specification (generating an undocumented exception is either a documentation defect or a software defect).
- Using specified pre-condition information to help distinguish between exceptional and non-exceptional tests. Flag any tests which satisfy preconditions but produce exceptional results.
- Using post-condition information to help assess correctness of tests executed under satisfied preconditions (this quickly begins to cross the line into traditional software testing).

If the above examples begin to sound like traditional software testing -- that is because they are. The only significant difference that Ballista testing seems to offer at this point is an object-oriented view to performing testing even on non-object-oriented software.

## 8. Potential generalization to security

Much of current security testing involves looking for known system vulnerabilities that have been a problem in the past (*e.g.*, the SATAN tool discussed in [35]). Scripts to exercise such problems are useful for ensuring that systems retain a capability to resist previously identified attacks, and can be thought of as a kind of regression testing for new revisions as well as a certification suite for new software.

However, what the security community seems to lack at this point is a systematic ability to detect new security vulnerabilities before they are detected via suffering attacks to exploit them. (There is clearly work to build up a theoretical infrastructure and improve this situation; but that is a difficult problem that will take much effort.) While it is certainly no silver bullet, Ballista

testing may be able to add a few tricks to a generic security testing toolkit.

### 8.1. Bullet-proofing key modules

One classic way to gain entry into a system is to cause a key software subsystem to fail or provide inappropriate access to data, thus circumventing whatever access control may be in place. Current Ballista testing techniques can be used to harden the constituent software modules of such critical subsystems, improving crash resistance at the module level.

An application of existing Ballista testing techniques would be to ensure that interfaces to modules are reasonably hardened. For example, a technique that has enjoyed perennial success is to intentionally submit an overly long string or buffer value to a software module in an effort to get that module to overwrite other, unrelated data causing a security hole (*e.g.*, [36]). While such an attack probably would not be directly detected by Ballista testing, a Ballista test approach could identify that the module does not perform buffer length checking by identifying a separate test case in which an overly long buffer results in an Abort failure, thus indirectly suggesting a vulnerability to an oversized-buffer attack.

At the system level, Ballista has not yet been applied to command-line interfaces and the like. It might be possible to build a parameterized testing approach based on a set of legal and illegal commands, flag values, and so on, but this has not been explored within the Ballista project. (However, the Fuzz project[30] has demonstrated success in robustness testing of command-line interfaces, and should be considered for use in security testing.)

### 8.2. Generic security behavioral specifications?

As in software assurance, it is possible that Ballista testing can help with some security testing. Possible high-level generic behavioral descriptions to use in detecting failures might be:

- Grants access to a file with insufficient access permissions
- Permits modification of audit logs
- Accesses critical system information (should not be possible with software modules other than a certain few)

As an example, a Ballista test might be set up for the file read command in which the file handle used for testing is created with various forms of permission for access (read-only access, write-only access, no access). However in this case the test result of interest would not be the error return status of the module, but rather whether a read was actually performed, and (via a postprocessing script) whether read permissions were actually set in the test case executed. This amounts to a pre-condition/post-condition comparison, similar to what might be used for a software assurance application (*e.g.*, if pre-condition is "read access permitted" then the read() function satisfies the post-condition of "data read" in the absence of other exceptions).

## 9. Ballista in perspective: what it does and doesn't do

Although we attempt to suggest several possible extensions of Ballista testing beyond the domain of software robustness, it should be made clear that there are definite limitations on what it can do. And, furthermore, some of the suggestions above might not actually be practical. Bear in mind that software testing is in general a specialized form of software development. Thus, the question of whether software testing can be performed is in most cases of interest not theoretical, but rather, an issue of economics. What is different about Ballista testing is that it attempts to limit the scope of the testing in a way that significantly reduces development cost and effort, while still attempting to measure useful system properties. Thus, it is not a do-all

solution, but rather an exercise in attaining limited additional testing or characterization at comparatively low cost.

### 9.1. The goal: "quantification without tears"

Of course everyone would like to have complete testing or characterization without spending any money or time on it. But, obviously, that isn't going to happen any more than software is going to get developed effortlessly in the first place. So, the real question is, how well does Ballista "scale," where scalability has to do with incremental cost to test additional software modules once an initial set of modules has been tested.

The Ballista approach has proven portable across platforms, and promises to be portable across applications. The Ballista tests have been ported to ten different operating systems. 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. The HLA RTI testing encompasses a large range of object-oriented testing considerations, and required only minor enhancements to the Ballista testing approach. So it seems likely that Ballista will be useful for a significant number of other applications as well.

**9.1.1. 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.

In the case of the HLA RTI, results appear to be that a reasonable level of scalability has also been achieved. The effort involved in preparing for automated testing is 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.

A further step in reducing the effort for preparing tests is to add inheritance of test cases. This is conceptually present in the current version of Ballista, but is still being implemented. Each primitive machine type (such as integer or floating point number) has a set of pre-defined tests. Users adding more specific data types (such as "file mode bit pattern" or "enumeration index") can inherit existing generic tests and add more specific test cases as well. Multiple inheritance will help simple creation of pointer data types, for example by inheriting both basic tests for any pointer as well as tests for data values in test cases having a valid pointer.

**9.1.2. Effectiveness and system state:** The Ballista testing fault model is fairly simplistic: sin-

gle 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 stand-alone test program leads to a reproduction of robustness failures. The only situation in which Ballista results have been found to lack such single-call reproducibility is in Catastrophic failures (complete system crashes, not just single-task crashes), and even there results can be reproduced by running a given handful of test cases together.

### 9.2. Ballista quantifies some aspects of robustness

Ballista has succeeded in providing quantification of some aspects of robustness, specifically the reaction of individual software modules when presented with exceptional parameter values in a single-call context with a moderate amount of associated state information. While we speculate that improving robustness at this simple level will yield benefits at the system level, it is as yet unclear how significant an effect this will be. There is no denying that some system-level problems will be dynamic, involving such emergent behaviors as livelock. But, skeptics have, at various times over the course of the past decade, incorrectly predicted that Ballista and its predecessors would not find any way to crash a system. Given that such system crashes can be found with an admittedly simplistic testing model, it is difficult to say exactly how much remains beyond the reach of a single-threaded Ballista implementation (and, it remains a research topic to figure out how to create a multi-threaded Ballista testing system that retains the characteristics of repeatability and scalability).

There are, of course, not only benefits, but also dangers and pitfalls to be had. The benefits of being able to quantify robustness, albeit in a limited scope, stem from the fact that once something can be measured, it is more readily determined whether that metric becomes better or worse over time. This can lead to the following potential benefits:

- Customers that care about robustness can make more informed purchasing decisions.
- Developers can justify additional resources to improve robustness, since they can now measure the robustness of their system both on an absolute scale and in relation to competitors.
- Development managers can make better-informed decisions about how much effort to devote to robustness (based on measured robustness levels), and measure the effectiveness of those efforts based on resultant robustness improvements.
- From a market-driven point of view, historically only things that can be measured tend to be improved over time. Historically this has included hardware attributes (*e.g.*, MIPS, megabytes RAM, gigabytes disk, megabits/second I/O). For software this has, all too often, meant the length of feature check-off lists. For less performance or functionality-directed attributes, an ability to quantify may mean that they are driven to improvement by market forces as well.

There are, of course, dangers inherent in using any measurement system, and particularly one that produces a single number for comparison:

- A classic management danger is that "you get what you measure." Ballista certainly does not measure every aspect of robustness. Thus, there is a very real danger that consumers will purchase based on "robustness" metrics when the type of robustness they need is not what is being measured, that system designers will optimize for measured robustness at the expense of robustness attributes not measured, *etc*. While this is a very real danger, it is inherent in any measurement technique, and must be addressed by continually improving the measurements in response to such trends. Fear of creating a less-than-perfect measure is no excuse for deciding to measure nothing at all.
- Customers might insist on better robustness numbers even if they do not really need them. While this is a potential pitfall, it is probably no worse than a hobbyist who insists on having the latest CPU model without really needing the performance. And, in both cases, this behavior serves to drive improvement of the metric to the benefit of users who do in fact need it.
- In time, Ballista metrics may suffer from a problem known as the "inoculation effect."

### 9.3. The inoculation effect requires continuing improvement

A significant concern with Ballista testing is the effect it may have over time on the software it measures (called the "inoculation effect" with respect to software testing [5]). In particular, the issue is, what is to stop a software developer from ensuring that all Ballista tests pass 100%? The first, simple, answer is that reaching that state of affairs is probably desirable, and having a situation in which robustness or other qualities are taken for granted by the marketplace rather than being non-ideal is probably a good thing.

However, there is a more general answer to this issue in that Ballista testing abilities should be made to improve over time, and to adapt to whatever strategies may be used to "game" or otherwise defeat it as a reasonable measurement tool. There is ample experience with this issue in the performance metric arena, where compiler technology has long been used to "break" benchmark programs by using customized optimizations. The response should be not to declare that Ballista testing is pointless because of this almost inevitable outcome, but rather to take it as a challenge to continually improve testing capabilities. Improvement can come in any one of several forms, including:

- Creating richer sets of test data for each data type.
- Improving the abilities of the "dials" metaphor to increase fine-grain testing abilities.
- Improving the search abilities of the testing harness to seek out and identify patterns.
- Adding randomized testing.
- In time, adding concurrent testing abilities.
- In time, increasing the amount of system state that can be set for each test.

Thus, while dealing with the inoculation effect is a legitimate concern, it should be seen as a challenge in a process of ever-improving system quality rather than a stumbling block.

### 9.4. Scalability is attained by ignoring functional specifications

The key to Ballista's scalability is that it abstracts the functional specification of each module to an extremely high, almost trivial, level. As soon as functionality begins to creep into the testing process, scalability is placed at risk. An immediate reaction of some to seeing Ballista results is to want to apply the approach to displace traditional software testing. While this is naturally a powerful desire (because testing is so expensive), it is simply not what Ballista

testing is designed for. It might be possible to encroach on software testing a bit using techniques discussed previously, but ultimately there is a certain amount of essential complexity to software, and that must somehow be represented to the testing system if an automated test generator is to determine whether the actual software artifact meets the specification or not.

### 9.5. System "state" is a deeper issue than at first suspected

The whole issue of system state has turned out to be far more slippery than originally anticipated. In the original vision of what Ballista would do, system state was explicitly stated to be outside the scope of research. Additionally, it was feared that per-module scaffolding might be required for each function, but at least it was hoped that per-module functional specifications might be avoided.

However, the object-oriented testing approach has permitted per-module scaffolding to be largely eliminated, and a moderate amount of system state to be included in each test. Both of these goals were accomplished by the subsumption of all testing information into the data types. It is currently unclear how far these concepts can be pushed. The initial design did not permit setting any system state, yet that was eventually incorporated via constructors when complex data types such as file handles were incorporated. Preliminary results on operating systems did not show how the techniques could be expanded to encompass object-oriented software testing. Yet that was accomplished with the HLA RTI application. Later results did not show how parameterless functions could be tested, yet that was later accomplished with so-called phantom parameters, that are passed as dummy parameters by the Ballista testing harness, but stripped before calling the actual function [13]. And, although it has not yet been implemented, it is clear that phantom parameters to set things such as remaining disk space can help with larger chunks of system state than previously anticipated.

Finally, it is a commonly held notion that to find all but the most trivial of bugs a sequence of calls must be executed. The results on operating systems have shown that this is not necessarily true -- Ballista testing can find ways to crash an entire system with a single function call. But at a deeper level, what Ballista has done is replaced the notion of a sequence of calls (a control-flow thought model) with the notion of setting system state prior to a single call (an object-oriented thought model). In fact, Ballista *does* make a series of calls via test value constructors -- it is simply the case that these calls are hidden from the user and not considered part of the actual test sequence. Additionally, since Ballista executes tests in batches under a harness, a moderately large number of system calls are made in a row that might tend to flush out state-dependent failures. However, experience has shown that the sequences of calls across different test cases are largely irrelevant -- the single-call testing model has proven to be a useful one.

Thus, as research on Ballista has progressed, we have continually been surprised at the effectiveness of, and the amount of system state that can be set by, a deceptively simple "single-call" testing model.

## 10. Conclusions and future work

The Ballista testing approach has possible applications to fault tolerance, software assurance, and computer security. It is based on a pair of key approaches that can be applied to these three areas:

- Limiting the scope of testing to those areas in which a generic functional specification can be used across all modules being tested.
- Using an object-oriented approach to creating test values in which all tests are determined by the data types of parameters rather than any genericization of module functionality.

The results of Ballista testing on operating systems have indicated that this generic approach is sufficient to test for and identify significant numbers of robustness failures in a highly scalable manner. Automated testing identified ways to crash commercial operating systems, ways to hang tasks, ways to cause abnormal terminations, and "Silent" errors in which no indication of failure is given. These results have direct applicability to various areas of computing, and scale to applications beyond operating systems.

Beyond the current results, it may be possible to expand the idea of Ballista testing in several directions. In fault tolerance it may prove possible to use Ballista testing to help create software wrappers, to evaluate diversity for multi-version software applications, and to reduce the average cost of checkpointing by providing a warning of potentially hazardous operations. In software assurance, it can be used to evaluate exception handling abilities, and perhaps to provide partially automated testing for boundary conditions. In security it might be used to assure robustness of critical modules, and also to test for general security properties of all modules.

Potential future work already underway by the Ballista project team includes exploring the temporal aspects of robustness, porting Ballista to the popular Windows operating system, and generating software wrappers to improve robustness. The other potential applications may be explored in time, but in general are left as an opportunity for others to pursue.

## 11. Acknowledgments

The graduate students who have worked on the Ballista project have been crucial to its success, and spent many hours writing the system as well as collecting data. The students contributing to the work reported herein are: Nathan Kropp, John DeVale, Jiantao Pan, and Kim Fernsler. Additionally, Dan Siewiorek and his students laid the groundwork for Ballista in the early 1990s with previous fault injection projects. Dan in particular has offered sage advice and inspired several interesting twists in the research direction. This research was sponsored by DARPA ITO contract DABT63-96-C-0064.

## 12. References

[1] Avizienis, A., "The N-version approach to fault-tolerant software", *IEEE Transactions on Software Engineering*, **SE-11**(12): 1491-501, Dec. 1985.

[2] Avizienis, A., Lyu, M., Schutz, W., "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," *The Eighteenth International Symposium on Fault Tolerant Computing*, p. 15-22, 1988.

[3] Barton, J., Czeck, E., Segall, Z., Siewiorek, D., "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, **39**(4): 575-82, Apr. 1990.

[4] Brilliant, S.S., Knight, J.C., Leveson, N.G., "Analysis of Faults in an N-Version Software Experiment", *IEEE Transactions on Software Engineering*, **16**(2): 238-47, Feb. 1990.

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

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

[7] Carrette, G., "CRASHME: Random input testing," (no formal publication available) *http://people.delphi.com/gjc/crashme.html* accessed July 6, 1998.

[8] Chen, L., Avizienis, A., "N-Version Programming : A Fault Tolerance Approach to Reliability of Software Operation," *The Eighth International Symposium on Fault Tolerant Computing*, p. 3-9, 1978.

[9] Cohen, D., Dalal, S., Fredman, M. & Patton, G. "The AETG System: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, **23**(7): 437-44, Jul. 1997.

[10] Cristian, Flaviu, "Exception Handling and Tolerance of Software Faults," In: Michael R. Lyu (Ed.) *Software Fault Tolerance*. Chichester: Wiley, p. 81-107, Ch. 4, 1995.

[11] Dahmann, J., Fujimoto, R., & Weatherly, R, "The Department of Defense High Level Architecture," *Proceedings of the 1997 Winter Simulation Conference*, Winter Conference Board of Directors, San Diego, CA, p. 142-9, 1997.

[12] DeVale, J. & Koopman, P., "Comparing the Robustness of POSIX Operating Systems," *29th Symposium on Fault Tolerant Computing (FTCS-29)*, June, 1999, in press.

[13] DeVale, J., Koopman, P. & Guttendorf, D., "Ballista Software Robustness Testing Service," *Testing Computer Software conference (TCS-99)*, June 1999, in press.

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

[15] Eckhardt, D.E. & Lee, L.D., "Fundamental differences in the reliability of N-modular redundancy and N-version programming," *Journal of Systems and Software*, **8**(4): 313-318, Sept. 1988.

[16] Hartz, M., Walker, E. & Mahar, D., *Introduction to Software Reliability: A State of the Art Review*, Reliability Analysis Center report SWREL, Rome NY, December 1996.

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

[18] *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.

[19] Jones, E., (ed.) *The Apollo Lunar Surface Journal, Apollo 11 lunar landing*, entries at times 102:38:30, 102:42:22, and 102:42:41, National Aeronautics and Space Administration, Washington, DC, 1996.

[20] Kanawati, G., Kanawati, N. & Abraham, J., "FERRARI: a tool for the validation of system dependability properties," *1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. Amherst, MA, USA, p. 336-344, July 1992.

[21] Knight, J.C., Leveson, N.G., "A reply to the criticisms of the Knight and Leveson experiment", *SIGSOFT Software Engineering Notes*, **15**(1): 24-35, Jan. 1990.

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

[23] Kropp, N., Koopman, P. & Siewiorek, D., "Automated Robustness Testing of Off-the-Shelf Software Components," *28th Fault Tolerant Computing Symposium*, p. 230-9, June 23-25, 1998.

[24] Lions, J.L. (chairman) *Ariane 5 Flight 501 Failure: report by the inquiry board*, European Space Agency, Paris, July 19, 1996.

[25] Lyu, M.R., "Software Reliability Measurements in N-Version Software Execution Environment," *Proceedings of the Third International Symposium on Software Reliability and Engineering*, p. 254-63, 1992.

[26] Lyu, M. (ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press/ McGraw Hill, 1996.

[27] McCabe, T.J., *Structured testing: A software testing methodology using the cyclomatic complexity metric*, Nat. Bur. Stand., Washington, DC, USA, Dec. 1982.

[28] McAllister, D.F., Sun, C., Vouk, M.A., "Reliability of Voting in Fault-Tolerant Software Systems

for Small Output-Spaces," *IEEE Transactions on Reliability*, **39**(5): 524-34, Dec. 1990.

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

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

[31] Mukherjee, A. & Siewiorek, D.P., "Measuring software dependability by robustness benchmarking, *IEEE Transactions on Software Engineering*, **23**(6): 366-78, June 1997.

[32] Numega. BoundsChecker. *http://www.numega.com/products/aed/vc.shtml*, accessed 6/1/99.

[33] Ostrand, T.J. & Balcer, M.J., "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, **31**(6): 676-86, June 1988.

[34] Purify. *http://www.pureatria.com/products/purify/index.html*, accessed 8/28/98.

[35] Ram, P. & Rand, D.K., " Satan: double-edged sword," *Computer*, **28**(6): 82-3, June 1995.

[36] Spanbauer, S., "Pentium Bug, Meet the IE4 Flaw," *PC World*, p. 55, Feb. 1998.

[37] Tsai, T., & R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," *Proceedings Eighth International Conference. on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Germany, Springer-Verlag, p. 26-40, Sept. 20-22 1995.

[38] Vo, K-P., Wang, Y-M., Chung, P. & Huang, Y., "Xept: a software instrumentation method for exception handling," *The Eighth International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, p. 60-69, 2-5 Nov. 1997.