

Techniques for Evaluating the Robustness of Windows NT Software^{*}

Matthew Schmid, Anup Ghosh, Frank Hill
Reliable Software Technologies
21351 Ridgetop Circle, Suite 400, Dulles VA 20166
{mschmid, aghosh, fhill}@rstcorp.com

Abstract

Windows NT is rapidly becoming the platform of choice for organizations engaging in commerce, engineering, and research. The Windows NT operating system and its software are being relied upon for an increasing number of critical applications in both the military and civilian arenas. It is essential that software testing techniques are created that will enable the development of software that is capable of functioning in such roles. This paper presents two approaches that can be used to aid in the robustness testing of Windows NT software. The first approach uses a test data generator to analyze the robustness of Windows NT Dynamic Link Libraries. The second approach uses binary wrapping and fault injection techniques to study the effects of operating system failures on an application. A Failure Simulation Tool has been developed to this end.

1 Introduction

An increasingly large number of mission critical applications are relying on the robustness of Commercial Off The Shelf (COTS) software. The military, for one, uses commercially available architectures as the basis for 90% of its systems[1]. Many commercial products are not fully prepared for use in high assurance situations. In spite of the criticality of these applications, there currently exists a dearth of software assurance techniques to assess the robustness of both the application and the operating system under strenuous conditions. The testing practices that ordinary commercial products undergo are not thorough enough to guarantee reliability, yet many of these products are being incorporated in critical systems.

For example, the U.S. Navy requires that its ships migrate to Windows NT workstations and servers under the Information Technology in the 21st century (IT-21) directive[2]. While modernizing the fleet's technology base is appropriate, the risks of migrating to new platforms are great, particularly in mission-critical systems. One widely publicized early casualty of this directive involved the USS Yorktown, a U.S. Navy Aegis missile cruiser. The cruiser suffered a significant software problem in the Windows NT systems that control the ship's propulsion system. An application crash resulting from an unhandled exception reportedly caused the ship's propulsion system to fail, requiring the boat to be towed back to the Norfolk Naval Base shipyard[3].

High assurance applications require software components that can function correctly even when faced with improper usage or stressful environmental conditions. The degree of tolerance to such situations is referred to as a component's robustness. Most commercial products are not targeted for high assurance applications. These products, which include most desktop applications and operating systems, have not been extensively tested for use in mission critical applications. Despite this fact, many of these products are used as essential components of critical systems.

Given the use of COTS software components in critical systems, it is important that the robustness of these components be evaluated and improved. Studies, including Fuzz[4,5] and Ballista[6], have examined using automated testing techniques to identify robustness failures. Automated testing has the advantage of being low-cost and efficient, however its effectiveness depends largely on the data that is used as test input. The input to a component under test will determine which robustness failures (if any) will be discovered, and which will remain hidden. It is therefore essential that high assurance

^{*} This work is sponsored under the Defense Advanced Research Projects Agency (DARPA) Contract F30602-97-C-0117. THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTATIVE OF THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OF THE U.S. GOVERNMENT.

applications be tested with the most effective data possible.

This paper examines two techniques that we have developed as means of performing robustness tests on COTS software. First is an approach to generating data that can be used for the testing of data-driven software and operating system components. We present the results of using this data to perform robustness tests on the Windows NT Application Program Interface. Further results of our testing can be found in [7] and [8].

The second robustness testing technique that we present is best suited for the interactive testing of COTS applications. We present a prototype tool, the Failure Simulation Tool (FST), that can be used to analyze the robustness of an application to the unexpected failure of functions on which it depends. A description of the tool is given, and its usage is discussed.

2 Related work

Two research projects have independently defined the prior art in assessing system software robustness: Fuzz[4] and Ballista[6]. Both of these research projects have studied the robustness of Unix system software. Fuzz, a University of Wisconsin research project, studied the robustness of Unix system utilities. Ballista, a Carnegie Mellon University research project, studied the robustness of different Unix operating systems when handling exceptional conditions. The methodologies and results from these studies are briefly summarized here to establish the prior art in robustness testing.

2.1 Fuzz

One of the first noted research studies on the robustness of software was performed by a group out of the University of Wisconsin[4]. In 1990, the group published a study of the reliability of standard Unix utility programs[4]. Using a random black-box testing tool called Fuzz, the group found that 25-33% of standard Unix utilities crashed or hung when tested using Fuzz. Five years later, the group repeated and extended the study of Unix utilities using the same basic techniques. The 1995 study found that in spite of advances in software, the failure rate of the systems they tested was still between 18 and 23%[5].

The study also noted differences in the failure rate between commercially developed software versus freely-distributed software such as GNU and Linux. Nine different operating system platforms were tested. Seven out of nine were commercial, while the other two were free software distributions. If one expected higher reliability out of commercial software development processes, then one would be in for a surprise in the results from the Fuzz study. The failure rates of system

utilities on commercial versions of Unix ranged from 15-43% while the failure rates of GNU utilities were only 6%.

Though the results from Fuzz analysis were quite revealing, the methodology employed by Fuzz is appealingly simple. Fuzz merely subjects a program to random input streams. The criteria for failure is very coarse, too. The program is considered to fail if it dumps a core file or if it hangs. After submitting a program to random input, Fuzz checks for the presence of a core file or a hung process. If a core file is detected, a "crash" entry is recorded in a log file. In this fashion, the group was able to study the robustness of Unix utilities to unexpected input.

2.2 Ballista

Ballista is a research project out of Carnegie Mellon University that is attempting to harden COTS software by analyzing its robustness gaps. Ballista automatically tests operating system software using combinations of both valid and invalid input. By determining where gaps in robustness exist, one goal of the Ballista project is to automatically generate software "wrappers" to filter dangerous inputs before reaching vulnerable COTS operating system (OS) software.

A robustness gap is defined as the failure of the OS to handle exceptional conditions[6]. Because real-world software is often rife with bugs that can generate unexpected or exception conditions, the goal of Ballista research is to assess the robustness of commercial OSs to handle exception conditions that may be generated by application software.

Unlike the Fuzz research, Ballista focused on assessing the robustness of operating system calls made frequently from desktop software. Empirical results from Ballista research found that `read()`, `write()`, `open()`, `close()`, `fstat()`, `stat()`, and `select()` were most often called. Rather than generating inputs to the application software that made these system calls, the Ballista research generated test harnesses for these system calls that allowed generation of both valid and invalid input.

Based on the results from testing, a robustness gap severity scale was formulated. The scale categorized failures into the following categories: Crash, Restart, Abort, Silent, and Hinderer (CRASH). A failure is defined by the error or success return code, abnormal terminations, or loss of program control. The categorization of failures is more fine-grained than the Fuzz research that categorized failures as either crashes or hangs.

The Ballista robustness testing methodology was applied to five different commercial Unixes: Mach, HP-UX, QNX, LynxOS, and FTX OS that are often used in high-availability, and occasionally real-time systems. The results from testing each of the commercial OSs are

categorized by the CRASH severity scale and a comparison of the OSs are found in [6].

In summary, the Ballista research has been able to demonstrate robustness gaps in several commercial OSs that are used in mission-critical systems by employing black-box testing. These robustness gaps, in turn, can be used by software developers to improve the software. On the other hand, failing improvement in the software, software crackers may attempt to exploit vulnerabilities in the OS.

The research on Unix system software presented in this section serves as the basis for the robustness testing of the NT software system described herein. The goal of the work presented here is to assess the robustness of application software and system utilities that are commonly used on the NT operating system. By first identifying potential robustness gaps, this work will pave the road to isolating potential vulnerabilities in the Windows NT system.

3 Testing the Win32 API

This study examines two different approaches to generating data to be used for automated robustness testing. The two approaches differ in terms of the type of data that is generated, and in the amount of time and effort required to develop the data generation routines. The first type of data generation that is discussed is called generic data generation, and the second is called intelligent data generation. We will compare and contrast both the preparation needed to perform each type of data generation, and the testing results that each yields.

3.1 Input data generation

Both the Fuzz project and the Ballista project use automatically generated test data to perform automated robustness testing. The development of the data generators used by the researchers working on the Ballista project clearly required more time than did the development of the data generators used by researchers on the Fuzz project. This is because the Ballista team required a different data generator for each parameter type that they encountered, while the Fuzz team needed only one data generator for all of their experimentation. The data used for command line testing in the Fuzz project consisted simply of randomly generated strings of characters. These randomly generated strings were used to test all of the UNIX utilities, regardless of what the utility expected as its command line argument(s). Each utility, therefore, was treated in a generic manner, and only one data generator was needed. We refer to test data that is not dependent on the specific component being tested as generic data.

The Ballista team took a different approach to data generation. They tested UNIX operating system function calls, and generated function arguments based on the type declared in the function's specification. This approach required that a new data generator be written for each new type that is encountered in a function's specification. Although the number of elements in the set of data generators needed to test a group of functions is less than or equal to the number of functions, this may still require a large number of data generators. We refer to the practice of generating data that is specific to the component currently under test as intelligent data generation.

3.2 Generic data

The generation of generic test data is not dependent on the software component being tested. During generic testing, the same test data generator is used to test all components. This concept can be made clearer through an example. When testing a function, the component expects each parameter to be of a certain type, as specified in the API. In this example, the same data generator would be used to produce data for each function parameter, regardless of the expected type. An example of the generic data generators that we used for API testing is given in section 3.4.3.

3.3 Intelligent data

Intelligent test data differs from generic test data because it is tailored specifically to the component under test. The example above can be extended to show the differences between generic and intelligent data. Assume that the function being tested takes two parameters: a file pointer and a string. This would require the use of two intelligent data generators (one for generating file pointers, the other for generating strings). The intelligent file pointer generator will produce valid file pointers, and the intelligent string generator would produce actual strings. Furthermore, these generators can be tweaked to produce data that tests well-known boundary conditions or unusual situations. For example, the file pointer generator could produce pointers to files that are read only, files that are large, files that are empty, or files that are closed.

The purpose of using intelligent data generators is to take advantage of our knowledge of what type of input the component under test is expecting. We use this knowledge to produce data that we believe will exercise the component in ways that generic data cannot. Intelligent testing involves combining the use of intelligent data generators with the use of generic data

generators. The reason that tests that combine intelligent data with generic data will exercise more of a component's functionality is because the component may be able to screen out tests that use purely generic data. To continue the example from above, if the first thing that a function being tested performed was to exit immediately if the specified file did not exist, then testing with generic data would never cause the utility to execute any further. This would hide any potential flaws that might be found through continued execution of the utility.

3.4 Component robustness

The IEEE Standard Glossary of Software Engineering Terminology defines robustness as “The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.” (IEEE Std 610.12.1990) Applying this definition of robustness to the two classes of components that we are testing allows us to make two claims.

1. Neither an application, nor a function, should hang, crash, or disrupt the system unless this is a specified behavior.
2. A function that throws an exception that is not documented as being capable of throwing is committing a non-robust action.

The first statement is a fairly straightforward application of the definition of robustness. The second statement requires some more explanation. Exceptions are messages used within a program to indicate that an event outside of the normal flow of execution has occurred. Programmers often make use of exceptions to perform error-handling routines. The danger of using exceptions arises when they are not properly handled. If a function throws an exception, and the application does not catch this exception, then the application will crash. In order to catch an exception, a programmer must put exception-handling code around areas that he or she knows could throw an exception. This will only be done if the programmer knows that it is possible that a function can throw an exception. Because uncaught exceptions are dangerous, it is important that a function only throws exceptions that are documented.

A function that throws an exception when it is not specified that it can throw an exception is committing a non-robust action. The function does not necessarily contain a bug, but it is not performing as robustly as it should. Robustness failures like this can easily lead to non-robust applications.

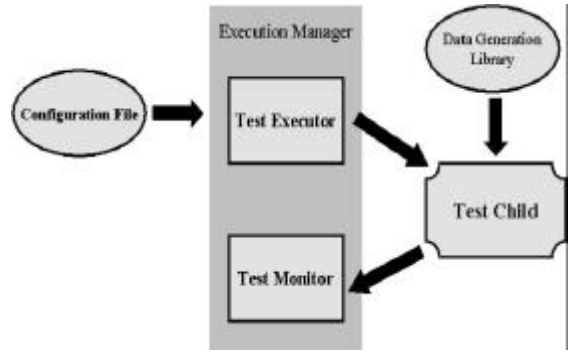


Figure 1: Testing Framework

3.5 Test framework

To perform our automated robustness testing we began by developing a simple test framework (Figure 1). The framework consists of four important components: the configuration file, the execution manager, the test child, and the data generation library.

The configuration file specifies what is being tested, and where the test data will come from. It is a flat text file that is read in one line at a time. Each line includes the name of the component to be tested, and the names of the data generators that should be used to supply the input for each parameter. Each parameter that is required by the component under test is specified individually. This is an example of what a line of the configuration file might look like during intelligent testing. In this example, the utility “print” expects the name of a printer followed by the name of a file.

```
print $PRINTER $FILENAME
```

Here is what a line from the generic testing configuration file might look like:

```
print $GENERIC $GENERIC
```

The data generation library contains all of the routines needed for generating both generic and intelligent data (these are called data generators). Each data generator generates a fixed number of pieces of data. The number of data elements that a data generator will produce can be returned by the data generator if it is queried. The data element that a data generator returns can be controlled by the parameters that are passed to it.

The test child is a process that is executed as an individual test. When testing the Win32 API functions the test child is a special process that will perform one execution of the function under test. This allows each run of a function test to begin in a newly created address

space. This reduces the chance that a buildup of system state will affect a test.

The execution manager is the heart of the framework. It is responsible for reading the configuration file, executing a test child, and monitoring the results of the test. After reading a line from the configuration file, the execution manager uses functions in the data generation library to determine how many tests will be run for a component. This number represents all possible combinations of the data produced by the specified data generators. For example, the line from the intelligent testing configuration file mentioned above specifies one file name generator, and one printer name generator. If the \$FILENAME data generator produces 10 different values, and the \$PRINTER data generator produces 5 values, then the execution manager would know that it has to run 50 (10 x 5) test cases. The execution manager then prepares the test child so that it will execute the correct test. Finally the execution manager executes the test child.

The test monitor is the part of the execution manager that gathers and analyzes the results of an individual test case. The test monitor is able to determine the conditions under which the test child has terminated. Some possible ends to a test case include the test child exiting normally, the test child not exiting (hanging), the test child exiting due to an uncaught exception (program crash), and the test child exiting due to a system crash. In the event of a system crash, after restarting the computer the testing framework is able to continue testing at the point that it left off. The results that the test monitor gathers are used to produce a report that details any robustness failures that were detected during testing.

This framework enables us to configure a set of tests, and then execute them and gather the results automatically. The results are stored as a report that can easily be compared to other reports that the utility has

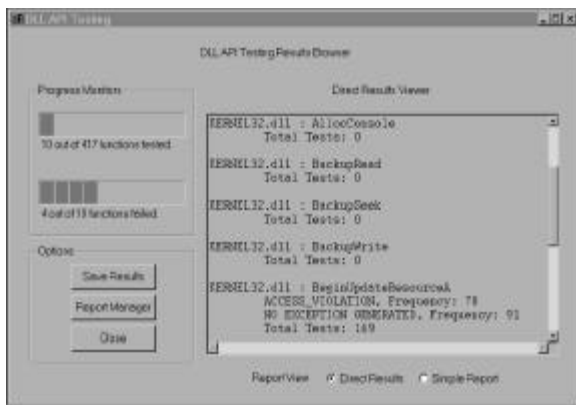


Figure 2: Win32 API function testing GUI – test configuration screen

generated.

The complete functionality of the test framework is most easily managed through its Graphical User Interface. The GUI can be used to configure and automatically execute a set of test cases, as well as to produce results reports. Status bars, including those that can be seen in Figure 2, give the tester an idea of how many tests have been run, and how many are remaining. The results of the testing are computed on the fly, and can be viewed in the results window even before all tests have completed. The reports manager, shown in Figure 3, is used to analyze the results of more than one experiment. It enables the user to select any number of previously generated reports and combine them into a comprehensive report suitable for experimental results comparisons.

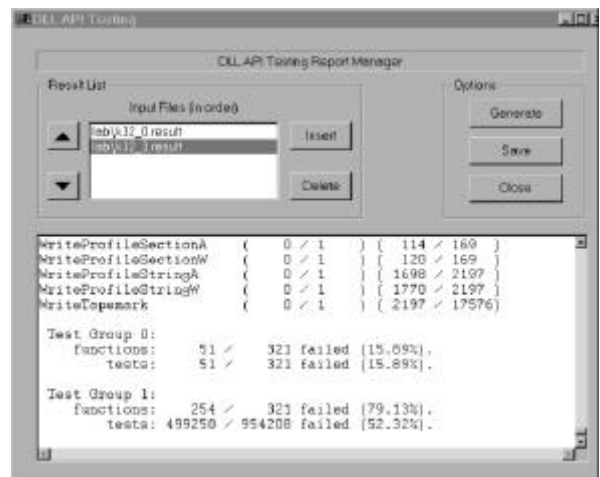


Figure 3: Win32 API function testing GUI – report manager screen

3.6 Win32 API function testing

The Win32 API is a set of functions that is standard across the Windows NT, Windows 95/98, Win32s, and Windows CE platforms (although not all functions are fully implemented on each of these platforms). These functions are located in Dynamic Link Libraries (DLLs), and represent a programmer's interface to the Windows operating system. For this experiment, we chose to concentrate on three of the most important Windows DLLs: USER32.DLL, KERNEL32.DLL, and GDI32.DLL. The USER32 DLL contains functions for performing user-interface tasks such as window creation and message sending, KERNEL32 consists of functions for managing memory, processes, and threads, and GDI32 contains functions for drawing graphical images and displaying text[9].

3.6.1 Generic Win32 API testing

The generic data generators that we used for testing the Win32 API functions were all integer based. This was done because all possible types can be represented through integers. For example, the char * type (a pointer to an array of characters) is simply an integer value that tells where in memory the beginning of the character array is located. The type float is a 32 bit value (just like an integer), and differs only in its interpretation by an application. Since the premise behind generic data generation is that there is no distinction made between argument types, the generic data generator used during the Win32 API testing generates only integers.

The generic testing that we performed on the Win32 API was done in three stages (referred to as Generic 0, Generic 1, and Generic 2). These stages are distinguished by the sets of integers that we used. Each set of integers is a superset of the previous set. The first set of integers that we used consisted only of { 0 }. The second consisted of { -1, 0, 1 }, and the third contained { -2^{31} , -2^{15} , -1, 0, 1, $2^{15} - 1$, $2^{31} - 1$ }. A test consisted of executing a function using all combinations of the numbers in these sets. For example, during the first stage of testing we called all of the functions and passed the value zero as each of the required parameters (resulting in only one test case per function). The second stage of testing consisted of running 3^x test cases, where x is the number of parameters that the function expects. The final stage of testing required 7^x test cases. Due to the time intensive nature of the testing that we are conducting, we limited our experiment to test only functions that contained four or fewer parameters (a maximum of $7^4 = 2401$ tests per function during generic testing).

3.6.2 Intelligent Win32 API testing

Intelligent testing of the Win32 API involved the development of over 40 distinct data generators. Each data generator produced data that is specific to a particular parameter type. One data generator was often capable of producing multiple pieces of data related to the data type for which it was written. Furthermore, each intelligent data generator also produced all of the data items output by the third generic data generator. An example of an intelligent data generator is the data generator that produces character strings. In addition to the data produced by the third generic data generator, this data generator produces a number of valid strings of various lengths and the null string. Other examples of Win32 API intelligent data generators are those that produce handles to files, handles to various system objects (i.e., module handles), and certain data structures.

3.7 Win32 API testing results

The results of both the generic and intelligent Win32 API experimentation are summarized in Figure 4. The bars represent the percentage of functions that demonstrated robustness failures. The robustness failures that are charted are almost entirely due to exceptions that the functions are throwing. Any one of these exceptions could cause an application to crash if they are not caught. The most common exception that we found (representative of an estimated 99% of all exceptions) is an ACCESS_VIOLATION. This is a type of memory exception that is caused by a program referencing a portion of memory that it has not been allocated. An example of this would be trying to write to a null pointer. "Access violations" frequently occur when an incorrect value is passed to a function that expects some sort of pointer. The number of functions that throw access violations when they are passed all zeros underscores this point. Keep in mind that the undocumented throwing of an exception does not necessarily indicate that a function contains a bug, however this is often not the most robust course of action that the function could take.

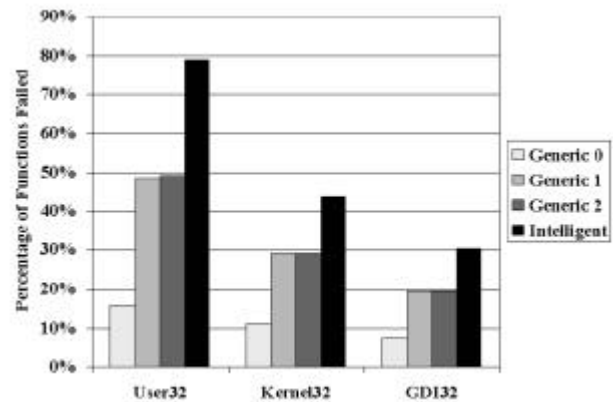


Figure 4: Win32 Function Testing Results

There were 321 USER32 functions tested, 307 KERNEL32 functions tested, and 231 GDI32 functions tested. In the analysis that we present here, we focus on the number of functions in each DLL that demonstrated robustness failures. There are a couple of reasons that we chose to concentrate on the number of functions that had failures, not the number of failures per function. One reason for this is that a function only has to suffer a robustness failure once to potentially harm a mission critical system. If it can be shown that a function is capable of failing in at least one circumstance, then the robustness of this function is called into question.

Another reason for focusing on the percentage of functions that failed, not the percentage of failures per

function, is that the number of tests that are run for each function is subjective. This subjectivity arises from the development of the data generators. There is no practical way to write a data generator (generic or intelligent) that will produce an exhaustive set of tests. Look, for example, at the intelligent data generator that produces strings. We use this generator to produce a number of strings of varying lengths. There is, however, a near infinite number of string lengths and character patterns that we could produce. Instead of attempting to exhaustively generate all of these possibilities (an intractable task), we instead select a small sampling of strings that we hope will test a function in different ways.

The way that this subjectivity could affect our data gathering is if we examine the number or percentage of failures per function. We would not even be able to give an accurate percentage of failures on a per function basis. If function X fails 50% of the time when tested with the data used during the third round of generic testing, we could easily add or remove data values to alter this percentage. What is most important to us is not the number of times that we can cause a function to fail, but whether or not a function failed during our testing.

Each progressive level of testing, from Generic 0 to intelligent, is a superset of the previous testing level. Generic testing accounted for over 60% of the exceptions found in each of the three DLLs tested. Notice that the percentage of functions that fails rises sharply between the first and second rounds of generic testing, and then again between the third round of generic testing and the round of intelligent testing. These results indicate two things to us. First, they show that despite its simplicity, generic testing is a worthwhile activity. Second, the results indicate that intelligent testing is a more comprehensive, and thus necessary part of automated robustness testing.

The process of setting up the generic tests that we conducted on the Win32 API functions was a fairly inexpensive task. The generic data generators that we used were simple to design and implement. Additionally, it appears that further expanding the sets of integers used during generic testing will not bring many new results. As the set of integers was changed from three elements to seven elements, the number of additional functions that failed was zero for KERNEL32 and GDI32, and only two for USER32. The generic data generation approach to automated testing could certainly be valuable to a tester that simply wanted to take a first step towards evaluating the robustness of a set of functions.

The significant increase between the number of functions found to exhibit robustness failures during generic testing, and the number of functions found to fail during intelligent testing underscores the importance of intelligent software testing. This data supports the claim that as the level of intelligence that is used during testing

increases, so does the number of problems discovered. For critical systems, this indicates that a significant amount of time and effort needs to be put into software testing.

As a final note, we found a number of serious operating system robustness failures during the testing of GDI32 and KERNEL32. Each of these DLLs contains a few functions that were capable of crashing the operating system. All of these operating system crashes occurred during intelligent testing. These OS crashes are significant because they represent robustness failures that could not be trapped by an application in any way. They are dangerous because they could either occur accidentally during normal system use, or could be caused intentionally by a malicious user (as in a Denial Of Service attack).

3.8 Win32 API testing conclusions

The experimental results of the Win32 API function testing demonstrates the usefulness of performing automated robustness tests with generic data, as well as the importance of using intelligent robustness testing techniques. Despite its simplicity, generic testing has proven to provide valuable results for the robustness testing performed in this experiment. Automated generic testing is an inexpensive yet useful testing technique.

Not surprisingly, intelligent testing uncovered more robustness failures than the automated Win32 API function robustness testing. These results confirm our belief that the use of more intelligent testing techniques and improved data generators uncovers more robustness failures. Furthermore, intelligent testing uncovers robustness failures that could never be discovered using only generic testing.

The desire to build fault tolerant computer systems using commercial software necessitates better testing of software components. This involves the testing of both existing components, and of the system as a whole. The experiment conducted indicates that automated robustness testing using generic testing techniques can yield impressive results, but that mission critical applications will require significantly more intelligent testing.

The remainder of this paper discusses a testing technique that has developed out of the work performed on Win32 API testing. This approach relies on our knowledge of robustness weaknesses in the Win32 API to examine how such vulnerabilities can affect a system that depends on this API.

4 Application testing with the failure simulation tool

The results that we obtained from our Win32 API robustness testing led us to the development of the Failure

Simulation Tool (FST). The FST provides a means of uncovering the impact of the failure of Win32 API functions on an application. This section of the paper discusses the motivation for and development of the FST.

4.1 Motivation for the FST

The purpose of this research was to develop an approach and technology for artificially forcing exceptions and error conditions from third-party COTS software when invoked by the software application under study. The goal in developing this technology is to support testing of critical applications under unusual, but known, failure conditions in an application's environment. In particular, we simulate the failure of operating system functions; however, the approach and tool can be used to simulate the failure of other third party software such as imported libraries and software development kits.

We have focused on the Win32 platform because we believe this to be the platform to which most critical applications are migrating, and it is the platform for which the least amount of research on dependability assessment has been performed, in spite of its growing adoption. While the general approach developed here is platform independent, the technology we have built and the implementation details of the approach are specific to the Win32 platform.

The approach is briefly summarized here, then developed in Section 4.4. Because we are working in the domain of COTS software, we do not assume access to program source code; instead, we work with executable program binaries. The approach is to employ fault injection functions in the interface between the software application under study and the operating system (or third party) software functions the application uses. The fault injection functions simulate the failure of these resources, specifically by throwing exceptions or returning error values from the third-party functions. The simulated failures are not arbitrary, but rather based on actual observed failures from OS functions determined in our previous study of the Windows NT platform (see [8]), or based on specifications of exceptions and error values that are produced by the function being used. In addition, the approach does not work on models of systems, but on actual system software itself. Therefore, we are not simulating in the traditional sense, but rather forcing certain conditions to occur via fault injection testing that would otherwise be very difficult to obtain in traditional testing of the application. The analysis studies the behavior of the software application under these stressful conditions and poses the questions: is the application robust to this type of OS function failure? does the application crash when presented with this exception or error value? or does the application handle the anomalous condition gracefully?

In the remainder of this section, we present some background in robustness testing research, provide motivation for why handling errors and exceptions is critical, then develop the methodology and tool for testing COTS software under these types of stressful conditions.

4.2 Background

Robustness testing is now being recognized within the dependability research community as an important part of dependability assessment. To date, robustness testing has focused on different variants of Unix software. In section 2.1.2 we discussed the work performed by B.P. Miller[4,5] and P. Koopman[6,10].

In this study, we are concerned with testing the robustness of application software --- specifically mission-critical applications --- that run on the Win32 platform. Unlike nominal testing approaches (see [11,12,13,14,15]) that focus on function feature testing, we are concerned with testing the software application under stressful conditions. Robustness testing aims to show the ability, or conversely, the inability, of a program to continue to operate under anomalous input conditions.

Testing an application's robustness to unusual or stressful conditions is generally the domain of fault injection analysis. To date, fault injection analysis of software has generally required access to source code for instrumentation and mutation (see [16] for an overview of software fault injection). In addition, fault injection analysis to date has been performed on Unix-based systems. We seek to develop technologies that will work on COTS-based systems and for the Win32 platform.

To define the problem domain of this work better: an application is robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions. Applications can be vulnerable to non-robust behavior from the operating system. For example, if an OS function throws an unspecified exception, then an application will have little chance of recovering, unless it is designed specifically to handle unspecified exceptions. As a result, application robustness is compromised by non-robust OS behavior.

In our previous studies of the Windows NT platform, we analyzed the robustness of Windows NT OS functions to unexpected or anomalous inputs[7,8]. We developed test harnesses and test data generators for testing OS functions with combinations of valid and anomalous inputs in three core Dynamically Linked Libraries (DLLs) of the Win32 Application Program Interface (API): USER32.DLL, KERNEL32.DLL, and GDI32.DLL. Results from these studies show non-robust behavior from a large percentage of tested DLL functions. This information is particularly relevant to application developers that use these functions. That is, unless

application developers are building in robustness to handle exceptions thrown by these functions, their applications may crash if they use these functions in unexpected ways.

We know from our testing of the Win32 platform that the OS functions can throw exceptions and return error values when presented with unusual or ill-formed input. In fact, we know exactly which exceptions and error codes a given OS function will return based on function specifications and our previous experimentation. However, even though this anomalous behavior from the OS is possible (as demonstrated), it is actually unusual during the normal course of events. That is, during normal operation, the OS will rarely behave in this way. Using nominal testing approaches to test the application might take an extremely long time (and a great many test cases) before the OS exhibits this kind of behavior. Thus, testing the robustness of the application to stressful environmental conditions is very difficult using nominal testing approaches.

However, using fault injection, we force these unusual conditions from the OS or from other third-party software to occur. Rather than randomly selecting state values to inject, we inject known exception and error conditions. This approach then forces these rare events that can occur to occur. Thus, this approach enables testing of applications to rare, but real failure modes in the system. The approach does not completely address the problem of covering all failure modes (especially unknown ones), but it does exploit the fact that third-party software does fail in known ways, even if infrequently. At a minimum, a mission-critical application must account for these known failure modes from third-party software. However, many applications never do, because software designers are mostly concerned about the application they are developing and assume the rest of the environment works as advertised. The approach and tool developed here tests the validity of that assumption by forcing anomalous conditions to occur.

It is important to note that we are not necessarily identifying program bugs in the application, but rather we are assessing the ability of the application to handle stressful environmental conditions from third-party software. So, this approach is an off-nominal testing approach that is not a substitute for traditional testing and fault removal techniques. In fact, the approach tests the application's error and exception handling mechanisms --- the safety net for any application. If the application does not account for these types of unusual conditions, chances are that it will fail.

4.3 Errors and exceptions

Error and exception handling are critical functions in any application. In fact, error and exception handling

make up a significant percentage of the code written in today's applications. However, error and exception handling are rarely tested because: (1) programmers tend to assume well-behaved functionality from the environment, and (2) it is difficult to create these kinds of anomalous behaviors using nominal testing techniques.

In [17], Howell argues that error handling is one of the most crucial, but most often overlooked aspect of critical system design and analysis. For example, Howell cites four examples of the criticality of error handling [17]:

- an analysis of software defects by Hewlett-Packard's Scientific Instruments Division determined that error checking code was the third most frequent cause of defects in their software
- in a case study of a fault-tolerant electronic switching system, it was found that 2 out of 3 system failures were due to problems in the error handling code
- many of the safety-critical failures found in the final checks of the space shuttle avionics system were found to be associated with the exception handling and redundancy management software
- problems with the use of Ada exceptions were a key part of the loss of the first Ariane-5 rocket

Errors and exceptions are often used by vendors of third-party software components to signal when a resource request has failed or when a resource is being improperly used. For example, exceptions may be thrown when invalid parameters are sent to a function call, or when the requesting software does not have the appropriate permission to request the resource.

Error codes are returned by a function call when an error occurs during the execution of the function. For example, if a memory allocation function is unable to allocate memory, it may return an invalid pointer. Error codes are graceful exits from a function that also allow a programmer to debug the source of the problem. Exceptions can be thrown for similar reasons, but more often, exceptions are thrown for more severe problems such as hardware failures or for cases when it is not clear why a resource request failed. For example, an exception returned by a function call may have actually originated from another function that was called by the requested function. If an exception is not handled by one function, it is passed up the function call chain repeatedly until either some function handles the exception or the application crashes.

In using third-party software, the application developer must be aware of what exceptions can be thrown by the third-party function. Third-party functions

can be embedded in libraries such as the C run-time library, software development kits, commercial software APIs, or as part of the core operating system. In most cases, the source code to the third-party function is not available, but header files and API specifications are. The function API, header files, or documentation should declare what exceptions, if any, can be thrown by the third-party function (and under what circumstances). If the application developer does not write exception handlers for these specified cases, then the robustness or survivability of the application is placed at risk if an exception is thrown in practice.

4.4 Wrapping Win32 COTS software for failure simulation

In order to assess the robustness of COTS-based systems, we instrument the interfaces between the software application and the operating system with a software wrapper. The wrapper simulates the effect of failing system resources, such as memory allocation errors, network failures, file input/output (I/O) problems, as well as the range of exceptions that can be thrown by OS functions when improperly used. The analysis tests

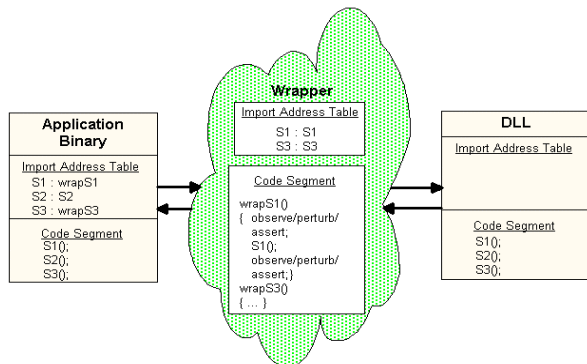


Figure 5: Wrapping Win32 Executable Programs

the robustness of the application to anomalous and stressful environment conditions. An application is considered robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions.

From our previous studies of the Windows NT platform, we found a large percentage of OS functions in the three core DLLs of the Win32 API that threw exceptions when presented anomalous input. If these functions are used similarly by an application, then the application must be prepared to handle these exceptions, specified or not. Because testing the application via nominal testing approaches is unlikely to trigger these anomalous OS conditions, we need some alternative method to test the robustness of these applications to OS

anomalies without requiring access to source code. To address this shortcoming in the state-of-the-art, we have developed the Failure Simulation Tool (FST) for Windows NT.

The approach that FST employs is to artificially inject an error or exception thrown by an OS function and determine if the application is robust to this type of unusual condition. FST instruments the interface between the application executable and the DLL functions it imports such that all interactions between the application and the operating system can be captured and manipulated.

Figure 5 illustrates how program executables are wrapped. The application's Import Address Table (IAT), which is used to look up the address of imported DLL functions, is modified for functions that are wrapped to point to the wrapper DLL. For instance, in Figure, functions S1 and S3 are wrapped by modifying the IAT of the application. When functions S1 and S3 are called by the application, the wrapper DLL is called instead. The wrapper DLL, in turn, executes, providing the ability to throw an exception or return an error code to the calling application. In addition, as the figure shows, we also have the ability to record usage profiles of function calls and the ability to execute assertions. The technique for performing this operation is based partially on work published by Matt Pietrek in [18].

There are several ways in which the wrapper can be used. First, the wrapper can be used as a pass-through recorder in which the function call is unchanged, but the function call and its parameters are recorded. This information can be useful in other problem domains such as for performance monitoring and for sandboxing programs. Second, the wrapper can be used to call alternative functions instead of the one requested. This approach can be used to customize COTS software for one's own purposes. For our purposes, we are interested in returning error codes and exceptions for specified function calls. Thus, we develop custom failure functions for each function we are interested in failing.

Three options for failing function calls are: (1) replace calling parameters with invalid parameters that are known to cause exceptions or error codes, (2) calling the function with the original parameters that are passed by the application, then replacing the returned result with an exception or error code, or (3) intercept the function call with the wrapper as before, but rather than calling the requested function, just returning the exception or error code. Option 3 is attractive in its simplicity. However, options (1) and (2) are attractive for maintaining consistent system state. In some cases, it is desirable to require the requested function to execute with invalid parameters to ensure that side effects from failing function calls are properly executed. Option (1) accounts

for this case. Using the information from our previous studies of the Win32 API[8], Option (1) can be implemented by using specifically those input parameters that resulted in OS function exceptions. Alternatively, specifications for which parameters are invalid when using a function (such as one might find in pre-condition assertions) can be used for causing the failure of the function in using Option (1). Option (2) is less rigorous about handling side effects from failing function calls, but will ensure side effects from normal function calls are properly executed. If, however, side effects from calling functions are not a concern, i.e., if the analyst is strictly concerned about how well the application handles exceptions or error codes returned from a function call, then Option 3 is sufficient.

The FST modifies the executable program's IAT such that the address of imported DLL functions is replaced with the address to our wrapper functions. This modification occurs in memory rather than on disk, so the program is not changed permanently. The wrapper then makes the call to the intended OS function either with the program's data or with erroneous data. On the return from the OS function, the wrapper has the option to return the values unmodified, to return erroneous values, or to throw exceptions. We use this capability to throw exceptions from functions in the OS (which we found to be non-robust in our earlier studies) called by the program under analysis.

After instrumenting all of the relevant Import Address Table entries, the FST performs a search across the code segment of the module for call sites to the wrapped functions. A call site is identified as a call instruction followed by a one-word pointer into the IAT. This call site information is used for two purposes. First, it gives the user a sense of how many call sites there are for a particular function. Second, it allows the FST to tally the number of calls on per site basis instead of on a per function basis.

Finally, the FST tries to match call sites with linked or external debug information. For instance, if the FST has located a call to HeapAlloc at a specific location in memory, it attempts to determine the corresponding line of source. Though debugging information is not necessarily distributed with an application, it can be a great help to a tester on those occasions when it is present. Because of the difficulties involved with juggling the many formats in which debugging information can be distributed, the FST makes use of the Windows NT library ImageHlp.DLL which helps to abstract the details of the debug information format.

The FST is also able to intercept dynamic calls to exported functions. The FST provides a special wrapper for the KERNEL32.DLL function GetProcAddress, a function that can be used to retrieve the address of a function at run-time. For functions that the FST would

like to wrap, the special GetProcAddress wrapper will return the address of a wrapper function instead of the correct function.

4.5 Using the failure simulation tool

The prototype Failure Simulation Tool provides an interface that helps to simplify the testing of COTS software. There are three primary components to the Failure Simulation Tool: the Graphical User Interface (GUI), the configuration file, and the function wrapper DLLs.

A function wrapper DLL contains the wrappers that will be called in place of the functions that the user chooses to wrap. These wrappers are responsible for simulating the failure of a particular function. The FST includes a variety of wrappers for commonly used functions. Additional wrappers can be developed by a user and used by the Failure Simulation Tool.

The configuration file is used to specify the DLL functions that are going to be wrapped, and the function that is going to be doing the wrapping. The configuration file is also used to break the functions into groups that are displayed by the tree control in the GUI. Here is an example of the configuration file.

```
PAGE: Memory
KERNEL32:HeapAlloc:WRAPDLL:WrapHeapAlloc
KERNEL32:LocalAlloc:WRAPDLL:WrapLocalAlloc
```

This configuration file specifies that there should be a group named "Memory," and that there will be two functions in that group. The first function to wrap is HeapAlloc, located in KERNEL32.DLL, and it should be wrapped using the function WrapHeapAlloc, found in WRAPDLL.DLL. The second function is specified in the same manner.

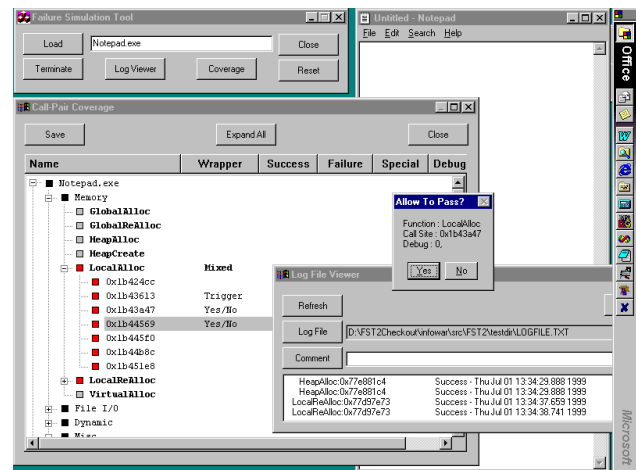


Figure 6: The Failure Simulation Tool GUI

Figure 6 shows the graphic interface to the failure simulation tool that allows selective failing of OS resources. The window in the upper left is the Execution Manager. This window is used to select an application for testing, to execute and terminate that application, and to access the other components of the FST. The largest window in Figure 9 is the interface that allows the user to control which functions are currently being wrapped, and what the behavior of those wrappers should be. The window titled "Log File Viewer" maintains a time-stamped listing of all of the function calls that have been made. The small window in the foreground is an example of a wrapper that allows the user to select whether a function should fail or not on a case-by-case basis. In the upper right of this figure is the application being tested – in this case the Notepad.exe application. For a more detailed description of how to use the Failure Simulation Tool, please see the documentation supplied as part of the System User's Manual.

In its current version, the FST is used to interactively fail system resources during execution. The dynamic binding to functions allows the tool to fail or succeed calls to OS functions on the fly in real time. The FST can be used to wrap any Win32 application (mission critical or not) in order to interactively fail system resources and to determine the impact of this failure. The performance overhead of the wrapping has not been measured, however, no visible degradation in performance of the application software has been observed. The tool is to be used for off-line analysis, however, in order to determine the effect of system failures prior to deployment. Thus, performance overhead in the analysis is not a large concern, unless it were to introduce unacceptable delays in testing.

4.6 Conclusions

This section on the Failure Simulation Tool provides an approach and tool for assessing the robustness of Win32 applications in the face of operating system anomalies. Two factors have motivated this work: first, more and more critical systems are being employed on the Win32 platforms such as Windows NT/95/CE/2000; second, the error/exception handling routines of software applications are rarely tested, but form the critical safety net for software applications. In addition, because most COTS software (such as third-party libraries or OS software) rarely provides access to source code, we constrain our approach to analyzing software in executable format.

The Win32 Failure Simulation Tool was developed to allow interactive failing of OS resources during testing. The tool allows the analyst to observe the effect of errors or exceptions returned from the OS on the application

under analysis. If the program fails to handle exceptions thrown by an OS function it will usually crash.

In the example of the USS Yorktown, the approach described herein can be used to wrap the ship's propulsion system software in order to assess how robust it is to exceptions thrown by the OS. For instance, when a divide-by-zero exception is thrown, the analysis would show that the propulsion system will crash. This information can then be used to prevent such an error from occurring or to handle the divide-by-zero exception gracefully. The most pressing question now for this smart ship is what other exceptions is the ship's propulsion system and other critical systems non-robust to?

Currently, a limitation of the tool is its coarse-grained ability to monitor the effects of the fault injection on the target application. Our measure for robustness is crude: an application should not hang, crash, or disrupt the system in the presence of the failure conditions we force. However, it is possible that the third party failures we introduce slowly corrupt the program state (including memory and program registers) that remain latent until a later period after the testing has ceased. It is also possible that while the failure does not crash the program it could simply cause the incorrect execution of the program's functions. Neither of these cases is analyzed by our tool. To address the former problem, an extensive testing suite would be necessary to gain confidence that even after the failure was caused, the program remains robust. To address the latter problem, an oracle of correct behavior is necessary and a regression test suite would be required after the failure was forced in order to determine correctness of the program. In both of these cases, our tool would falsely label the program as robust to the failure of the third-party component, when in fact the failure introduced a latent error in the application, or the program's output is corrupted without effecting the execution capability of the program. Hence, the scope of our monitoring is limited to the ability of the program to continue to execute in the presence of third-party failures. It does not have the ability to judge the correctness of the functions computed by the application.

In summary, the testing approach and prototype presented is of value to consumers, integrators, and maintainers of critical systems who require high levels of confidence that the software will behave robustly in the face of anomalous system behavior.

5 Summary

In this paper, we discussed a key property of secure and dependable systems: robustness. Our research focused on developing techniques for analyzing the robustness of software due to unknown and anomalous events. The work presented was partitioned into two distinct and complementary threads of robustness

research: intelligent test case generation, and testing application software for robustness to failing operating system (OS) resources.

The Win32 API testing framework that we developed supports intelligent black-box testing of an API to unusual inputs. The work leveraged research in robustness testing of OS utilities previously performed on Unix systems, most notably by Barton Miller's research group at the University of Wisconsin. Robustness testing of OS utilities has previously found vulnerabilities in operating system software to unexpected, random input. Adopting this approach for the Win32 platform, we enhanced it further by intelligently combining valid input with anomalous input in order to unmask flaws that remain hidden to purely random testing. Our framework provides a test harness and library for automatically generating test data according to both random (generally invalid) and valid parameters. Our studies have shown empirically the benefit derived from combining intelligent test data generation with random test case generation for the purpose of testing robustness.

The other research topic presented was concerned with testing the robustness of Win32 applications under failing OS conditions. From our previous research with the Win32 API, we found that the three core libraries that compose the Win32 system more often than not throw memory access violation exceptions when presented unusual input. Thus, if application developers (particularly for mission-critical applications) do not account for these exceptions that the OS throws, then the application is likely to crash. The Failure Simulation Tool (FST) provides the ability to test Win32 executables (without requiring source code) for robustness to exceptions or errors returned by OS functions. FST instruments the interface between an application and the OS DLL in order to return errors or exceptions from an OS function. This is a far more efficient approach than black-box testing of an application in hopes of generating an OS exception. Experiments showed that Microsoft desktop applications had varying levels of non-robustness to exceptions and errors returned by OS functions. This type of non-robust behavior is typically expected from desktop applications. However, non-robustness to errors or exceptions returned from OS functions is typically not acceptable in a mission-critical application, such as a ship propulsion system. Thus, FST provides the ability to test mission-critical software for robustness to failing OS functions. In the final stage of this work, we used the instrumentation layer to provide protective wrappers for applications such that an exception can be caught by the wrapper and returned as an error when it is known a priori that the error is handled gracefully, while an exception is not.

In summary, the research discussed presents novel technologies for analyzing and improving the robustness

of Win32 systems under unusual conditions associated either with malicious attack or misbehaving operating system functions.

-
- [1] Gen. John J. Sheehan. A commander-in-chief's view of rear-area, home-front vulnerabilities and support options. In *Proceedings of the Fifth InfoWarCon*, September 1996. Presentation, September 5.
 - [2] M. Binderberger. Re: Navy Turns to Off-The-Shelf PCs to Power Ships (RISKS-19.75). *RISKS Digest*, 19(76), May 25, 1998.
 - [3] G. Slabodkin. Software Glitches Leave Navy Smart Ship Dead in the Water, July 13 1998. Available online: www.gcn.com/gcn/1998/July13/cov2.htm.
 - [4] B.P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32-44, December 1990.
 - [5] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.
 - [6] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, pages 72-79, October 1997.
 - [7] A. Ghosh, M. Schmid, V. Shah. An Approach for Analyzing the Robustness of Windows NT Software. In *Proceedings of the 21st National Information Systems Security Conference*, October 5-8, 1998, p. 374-382. Crystal City, VA.
 - [8] A. Ghosh, M. Schmid, V. Shah. Testing the Robustness of Windows NT Software. To appear in the *International Symposium on Software Reliability Engineering (ISSRE'98)*, November 4-7, 1998, Paderborn, GE.
 - [9] J. Richter. *Advanced Windows, Third Edition*. Microsoft Press, Redmond Washington, 1997. Page 529.
 - [10] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated Robustness Testing of Off-The-Shelf Software Components. In *Proceedings of the Fault Tolerant Computing Symposium*, June 23-25 1998.
 - [11] B. Beizer. *Software Testing Techniques*. Electrical Engineering / Computer Science and Engineering. Van Nostrand Reinhold, 1983.
 - [12] B. Beizer. *Black Box Testing*. Wiley, New York, 1995.
 - [13] G. Myers. *The Art of Software Testing*. Wiley, 1979.
 - [14] B. Marick. *The Craft of Software Testing*. Prentice-Hall, 1995.

[15] J. Duran and S. Ntafos. *An Evaluation of Random Testing*. IEEE Transactions on Software Engineering, SE-10:438-444, July 1984.

[16] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.

[17] C. Howell. Error Handling: When bad things happen to good infrastructures. In *Proceedings of the 2nd Annual Information Survivability Workshop*, pages 89-92, November 1998. Orlando, FL.

[18] M. Pietrek. *Windows 95 System Programming Secrets*. IDG Books Worldwide, Inc., 1995.