

# Data Generation Techniques for Automated Software Robustness Testing\*

Matthew Schmid & Frank Hill  
Reliable Software Technologies Corporation  
21515 Ridgetop Circle #250, Sterling, VA 20166  
phone: (703) 404-9293, fax: (703) 404-9295  
email: mschmid@rstcorp.com  
<http://www.rstcorp.com>

## Abstract

Commercial software components are being used in an increasingly large number of critical applications. Hospitals, military organizations, banks, and others are relying on the robust behavior of software they did not write. Due to the high cost of manual software testing, automated software testing is a desirable, yet difficult goal. One of the difficulties of automated software testing is the generation of data used as input to the component under test. This paper explores two techniques of generating data that can be used for automated software robustness testing. The goal of this research is to analyze the effectiveness of these two techniques, and explore their usefulness in automated software robustness testing.

## 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. 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.

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 [2,3] and Ballista [4], have examined using automated testing techniques to identify robustness failures [5, 6]. 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 applications be tested with the most effective data possible.

In this paper we examine 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

---

\* 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.

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 yield.

## 2. Related Work

Two research projects have independently defined the prior art in assessing system software robustness: Fuzz [2] and Ballista [4]. 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 [2]. In 1990, the group published a study of the reliability of standard Unix utility programs [2]. 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 were still between 18 and 23% [3].

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 [4]. 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 [4]. 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.

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 some-times real-time systems. The results from testing each of the commercial OSs are categorized according to a severity scale and a comparison of the OSs are found in [4].

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 in this paper. The goal of the work presented in this paper 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. 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. In this paper, the practice of generating data that is specific to the component currently under test is referred to as intelligent data generation.

#### 3.1 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 command line utilities, generic data consists of randomly generated strings. There are three attributes that can be altered during generic command line utility testing. They are string length, character set, and the number of strings passed as parameters. The same data generators are used to test each command line utility. A utility that expects a file name as a parameter will be tested the same way as a utility that expects the name of a printer as an argument. The test data that the data generator produces is independent of the utility being tested.

#### 3.2 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 current command line utility being tested takes two parameters: a printer name, and a file name. This would require the use of two intelligent data generators (one for generating printer names, the

other for generating file names). The intelligent file name generator will produce strings that correspond to existing files. Additionally it will produce other strings that test known boundary conditions associated with file names. For example, on Windows NT there is a limit of 255 characters as the length of a file name. The intelligent data generator will be designed to produce strings that explore this boundary condition. Furthermore, the generator might produce strings that correspond to files with different attributes (read only, system, or hidden), or even directory names. The intelligent printer name generator would produce input data that explores similar aspects of a printer name.

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. This can be explained by continuing the example of the command line utility that takes a printer name and a file name as its parameters. If the first thing that this utility did was to exit immediately if the specified printer 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.

## 4. The Experiment

In this experiment, we perform robustness testing of Windows NT software components. The two types of components that we test are command line utilities, and Win32 API functions. Both types of components are tested using both generic and intelligent testing techniques.

### 4.1 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 it 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.

### 4.2 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.

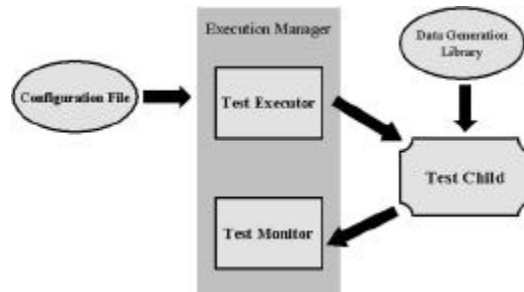


Figure 1: Testing Framework

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. In the case of the command line utilities, the utility itself constitutes the test child. When testing the Win32 API functions, however, 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 generated.

### 4.3 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 [7].

#### 4.3.1 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).

#### 4.3.2 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.

### 4.3.3 Win32 API Testing Results

The results of both the generic and intelligent Win32 API experimentation are summarized in Figure 2. 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.

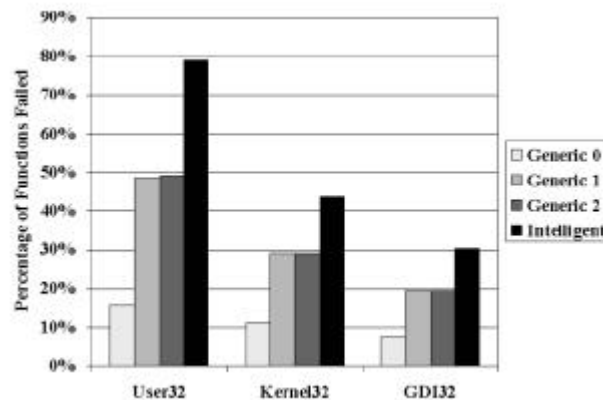


Figure 2: Win32 Function Testing Results

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).

#### 4.4 Win32 Command Line Utility Testing

For the second part to this experiment, we performed automated robustness testing of a number of Windows NT command line utilities. The primary difference between this experiment and the testing of the Win32 API functions is that we are now testing complete applications, not operating system functions. The utilities that we tested consisted of both a group of programs that are native to the Windows NT operating system, and a group of programs that were written by Cygnus for use on the Windows NT platform. Many of the command line utilities that we tested are analogous to the UNIX utilities tested by the researchers on the Fuzz project.

##### 4.4.1 Generic Command Line Utility Testing

The generic data generators that we used for testing the command line utilities were all string based. The parameters that command line utilities accept are always read from the command line as strings. Therefore, if a command line utility expects an integer as a parameter, it reads these parameters as a string (e.g., "10") and then converts it to the type that it requires.

There is another fundamental difference between the Win32 API function testing and the command line utility testing that we need to address. All of the functions that we tested accepted a fixed number of parameters. Many command line utilities will accept a varying number of parameters. Parameters are delimited by blank spaces. Instead of only judging whether or not a command line utility exhibited any robustness failures, we chose to distinguish between test cases that involved different numbers of parameters. Due to resource considerations, we limited the number of parameters that we would test to four.

The following is an example of what the configuration file for the command line utility "comp" looks like:

```

comp $GENERIC
comp $GENERIC $GENERIC
comp $GENERIC $GENERIC $GENERIC
comp $GENERIC $GENERIC $GENERIC $GENERIC

```

We refer to each of these lines as a template. Each utility is tested using 1, 2, 3, and 4 parameters, for a total of four templates. The generic data generator for the command line utility testing produced strings of varying lengths and character sets. The character sets that were used included alphanumeric, printable ASCII, and all ASCII except the null character. The null character (ASCII value 0) was avoided because it could be interpreted as the end of a line. Additionally, none of these strings contained spaces, so they should not be misinterpreted as representing more than one parameter.

#### 4.4.2 Intelligent Command Line Utility Testing

The intelligent data that was used as parameters to the command line utilities was based on syntactic and semantic information gathered from documentation for each utility. Although the data generators produce only strings, the strings that they produce have semantic meaning. For example, an intelligent data generator that produces file names chooses names of files that actually exist. There exist other data generators that produce strings corresponding to integers, directory names, printer names, etc. In addition to using these intelligent pieces of data, intelligent testing involved running test cases that combined intelligent data with generic data.

#### 4.4.3 Command Line Utility Testing Results

Table 1 summarizes the results of the command line utility testing. The data represents the number of templates that exhibited robustness failures (out of a possible four). The robustness failures charted here are due to the application abnormally terminating due to an uncaught exception that was thrown within the application. As stated earlier, an exception, such as those examined in the Win32 API function testing portion of this experiment, can cause an application to crash if it is not handled properly.

#### **Robustness Failures Discovered in Microsoft and Cygnus GNU Win32 Utilities**

<b>Microsoft</b>	Generic	Intelligent	<b>Cygnus</b>	Generic	Intelligent
findstr	3 / 4	4 / 4	diff	2 / 4	3 / 4
xcopy	3 / 4	4 / 4	gunzip	2 / 4	2 / 4
expand	2 / 4	3 / 4	ls	2 / 4	3 / 4
comp	0 / 4	1 / 4	cp	2 / 4	3 / 4
ftp	0 / 4	1 / 4	od	2 / 4	3 / 4
ping	0 / 4	0 / 4	grep	2 / 4	2 / 4

Table 1: Results from testing Microsoft Windows NT and Cygnus GNU Win32 command line utilities

The experimental results that we have gathered during the command line utility testing appear to support the conclusions that we came to after analyzing the Win32 API function testing. Simple generic testing was able to uncover robustness failures in a significant number of the utilities that we tested (9 out of 12). Intelligent testing was able to uncover two additional utilities that contained robustness failures.

Intelligent testing also proved to be better than generic testing at discovering more robustness failures for each utility. In all but four of the thirteen utilities tested, intelligent testing resulted in more failed templates than generic testing. Only two of the tested utilities did not demonstrate any robustness failures.

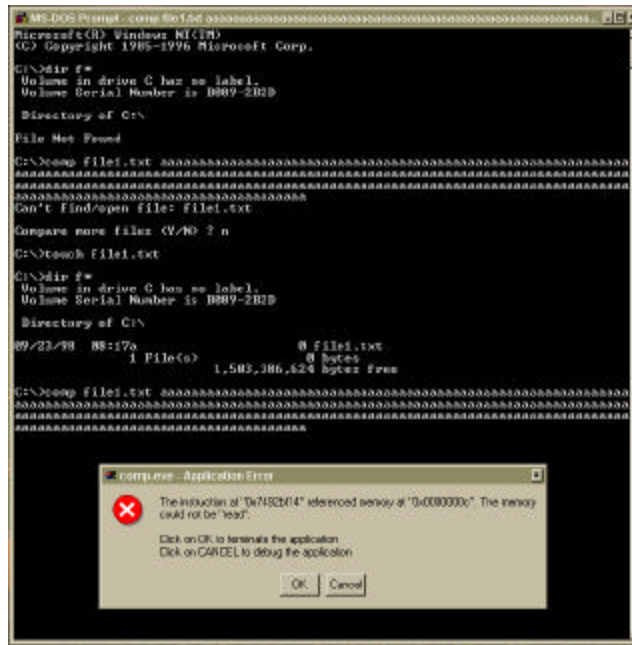


Figure 3: Screen capture of a robustness failure discovered in the native Windows NT command line utility `comp`.

Figure 3 contains a screen capture of a robustness failure that we discovered in the native Windows NT utility `comp`. This particular failure occurs only when the first parameter is a valid file name and the second parameter is a buffer of approximately 250 characters. This is an example of a robustness failure that could not be detected using only generic testing because it only appears when the first parameter is the name of an existing file.

## 5. Conclusions

The experimental results of both the Win32 API function testing and the Windows NT command line utility testing demonstrate 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 in both halves of this experiment. Automated generic testing is an inexpensive yet useful testing technique.

Intelligent testing uncovered more robustness failures for both the automated Win32 API function robustness testing, and the automated Windows NT command line utility testing. These results verify that more intelligent testing techniques uncover more robustness failures. Furthermore, intelligent testing uncovers robustness failures that could never be discovered using only generic testing (as described in the example of the `comp` utility).

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.

## 6. References

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. 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.
3. 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.
4. P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16<sup>th</sup> IEEE Symposium on Reliable Distributed Systems*, pages 72-79, October 1997.
5. A. Ghosh, M. Schmid, V. Shah. An Approach for Analyzing the Robustness of Windows NT Software. In *Proceedings of the 21<sup>st</sup> National Information Systems Security Conference*, October 5-8, 1998, p. 374-382. Crystal City, VA.
6. 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.
7. J. Richter. *Advanced Windows, Third Edition*. Microsoft Press, Redmond Washington, 1997. Page 529.