An Empirical Study of the Robustness of Windows NT Applications Using Random Testing[1]

Justin E. Forrester Barton P. Miller
{jforrest,bart}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

We report on the third in a series of studies on the reliability of application programs in the face of random input. In 1990 and 1995, we studied the reliability of UNIX application programs, both command line and X-Window based (GUI). In this study, we apply our testing techniques to applications running on the Window NT operating system. Our testing is simple black-box random testing; by any measure, it is a crude technique, but seems to be effective at locating bugs in real programs.

We tested almost 30 GUI-based applications by subjecting them to two kinds of random input: (1) streams of valid keyboard and mouse events and (2) streams of random Win32 messages. We have built a tool that helps automate the testing of Windows NT applications. With a few simple parameters, any application can be tested.

Our previous UNIX-based studies showed that, using our random testing techniques, we could crash a wide variety of command-line and X-window based applications on several UNIX platforms. The test results are similar for NT-based applications. When subjected to random valid input that could be produced by using the mouse and keyboard, we crashed 21% of applications that we tested and hung an additional 21% of these applications. When subjected to raw random Win32 messages, we crashed all the applications that we tested. We report which applications failed under which tests, and provide some analysis of these results.

1 INTRODUCTION

We report on the third in a series of studies on the reliability of application programs in the face of random input. In 1990 and 1995, we studied the reliability of UNIX command line and X-Window based (GUI) application programs[8,9]. In this study, we apply our techniques to applications running on the Window NT operating system.

The 1990 study evaluated the reliability of standard UNIX command line utilities. It showed that 25-33% of such applications crashed or hung when reading random input. The 1995 study evaluated a larger collection of applications than the first study, including some common X-Window applications. This newer study found failure rates similar to the original study. Specifically, up to 40% of standard command line UNIX utilities crashed or hung when given random input and 25% of the X-Window applications tested failed to deal with the random input. In our current (2000) study, we find a similar situation for applications running on Windows NT.

Our measure of reliability is a primitive and simple one. A program passes the test if it responds to the input and exits normally; it fails if it crashes (terminated abnormally) or hangs (stops responding to input within a reasonable length of time). The application did not have to respond sensibly or according to any formal specification. While the criterion is crude, it offers a mechanism that is easy to apply to any application, and crashing or hanging should not be ignored in any program. Such testing is a simple tool in the development and release cycle of a program. Certainly,

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

more extensive testing, beyond simple random input, should be applied to application programs. But curiously, our simple testing technique seems to find some bugs that are not found in normal testing. To support wider use of our random black box testing technique, in the past we have made our UNIX-based testing tools and scripts freely available; we continue that tradition with our new Windows NT tools.

Our study of X-Window applications provided the direction for the current study. To test X-Window applications, we interposed our testing program between the application (client) and the X-window display server. This allowed us to have full control of the input to any application program. We were able to both send completely random messages to the application, and to send random streams of valid keyboard and mouse events. In our current Windows NT study, we are able to accomplish the same level of input control of an application by using the Windows NT event mechanisms (described in Section 2).

Subjecting an application to streams of random valid keyboard and mouse events tests the application under conditions that it should definitely tolerate, as they could occur in normal use of the software. Subjecting an application to completely random input messages is a test of the general strength of error checking. This might be considered an evaluation of the software engineering discipline, with respect to error handling, used in producing the application.

Five years have passed since our last study, during which time Windows-based applications have clearly come to dominate the desktop environment. Windows NT (and now Windows 2000) offers the full power of a modern operating system, including virtual memory, processes, file protection, and networking. We felt it was time to do a comparable study of the reliability of applications in this environment.

Our current study has produced several main results:

- 21% of the applications that we tested *crashed* when presented with random, valid keyboard and mouse events.
- An additional 21% of the applications that we tested *hung* when presented with random valid keyboard and mouse events.
- Up to 100% of the applications that we tested failed (crashed or hung) when presented with completely random input streams.
- We noted (as a result of our completely random input testing) that applications running on Windows platforms are vulnerable to random input streams from any other application running on the same system.

The results of our study are significant for several reasons. First, reliability is the foundation of security[4]; our results offer an informal measure of the reliability of commonly used software. Second, we expose several bugs that could be examined with other more rigorous testing and debugging techniques, potentially enhancing software producers' ability to ship bug free software. Third, they expose the vulnerability of applications that use the Windows interfaces.

In the original studies, we had access to the source code of a large percentage of the programs that we tested, including applications running on several vendors' platforms and GNU and Linux applications. As a result, in addition to causing the programs to hang or crash, we were able to debug most applications to find the cause of the crash. These causes were then categorized and reported. These results were also passed to the software vendors/authors in the form of specific bug reports. In the Windows environment, we do not have such access to the source code of the applications. As a result, we can neither perform this step in our analysis nor can we provide such guidance to vendors.

Section 2 describes the details of how we perform random testing on Windows NT systems. Section 3 discusses experimental method and Section 4 presents the results from those experiments. Section 5 offers some analysis of the results and presents associated commentary. Related work is discussed in Section 6.

2 Random Testing on the Windows NT Platform

Our goal in using random testing is to stress the application program. This testing required us to simulate user input in the Windows NT environment. We first describe the components of the kernel and application that are involved with processing user input. Next, we describe how application programs can be tested in this environment.

In the 1995 study of X-Window applications, random user input was delivered to applications by inserting random input in the regular communication stream between the X-Window server and the application. Two types of random input were used: (1) random data streams and (2) random streams of valid keyboard and mouse events. The testing using random data streams sent completely random data (not necessarily conforming to the window system protocol) to an application. While this kind of input is unlikely under normal operating conditions, it provided some insight into the level of testing and robustness of an application. It is crucial for a properly constructed program to check return parameters from system calls and library routines. The testing with random valid keyboard and mouse events is essentially testing an application as though a monkey was at the keyboard and mouse. Any user could generate this input, and any failure in these circumstances represents a bug that can be encountered during normal use of the application.

We used the same basic principles and categories in the Windows NT environment, but the architecture is slightly different. Figure 1 provides a simplified view of the components used to support user input in the Windows NT environment[10,11,12].

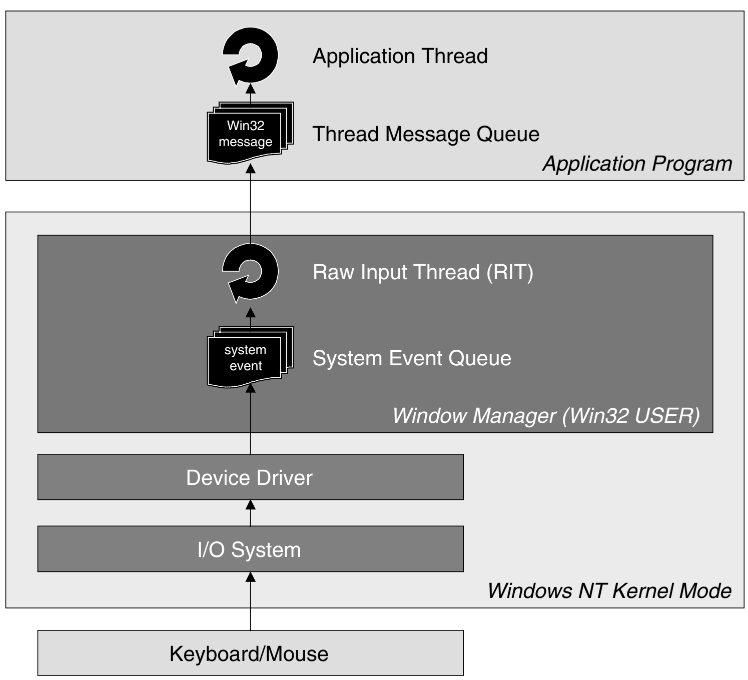

Figure 1: Windows NT Architectural Components for User Input

We use an example to explain the role of each component in Figure 1. Consider the case where a user clicks on a link in a web browser. This action sets into motion the Windows NT user input architecture. The mouse click first generates a processor interrupt. The interrupt is handled by the I/O System in the base of the Windows NT kernel.

The I/O System hands the mouse interrupt to the mouse device driver. The device driver then computes the parameters of the mouse click, such as which mouse button has been clicked, and adds an event to the System Event Queue (the event queue of the Window Manager) by calling the `mouse_event` function. At this point, the device driver's work is complete and the interrupt has been successfully handled.

After being placed in the System Event Queue, the mouse event awaits processing by the kernel's Raw Input Thread (RIT). The RIT first converts the raw system event to a Win32 message. A Win32 message is a generic message structure that is used to provide applications with input. The RIT next delivers the newly created Win32 message to the event queue associated with the window. In the case of the mouse click, the RIT will create a Win32 message with the `WM_LMOUSEBUTTONDOWN` identifier and current mouse coordinates, and then determine that the target window for the message is the web browser. Once the RIT has determined that the web browser window should receive this message, it will call the `PostMessage` function. This function will place the new Win32 message in the message queue belonging to the application thread that created the browser window.

At this point, the application can receive and process the message. The Win32 Application Program Interface (API) provides the `GetMessage` function for applications to retrieve messages that have been posted to their message queues. Application threads that create windows generally enter a "message loop". This loop usually retrieves a message, does preliminary processing, and dispatches the message to a registered callback function (sometimes called a *window procedure*) that is defined to process input for a specific window. In the case of the web browser example, the Win32 message concerning the mouse click would be retrieved by the application via a call to `GetMessage` and then dispatched to the window procedure for the web browser window. The window procedure would then examine the parameters of the `WM_LMOUSEBUTTONDOWN` message to determine that the user had clicked the left mouse button at a given set of coordinates in the window and that the click had occurred over the web link.

Given the above architecture, it is possible to test applications using both random events and random Win32 messages. Testing with random events entails inserting random system events into the system event queue. Random system events simulate actual keystroke or mouse events. They are added to the system via the same mechanism that the related device driver uses, namely the `mouse_event` and `keybd_event` functions.

The second random testing mechanism involves sending random Win32 messages to an application. Random Win32 messages combine random but valid message types with completely random contents (parameters). Delivering these messages is possible by using the Win32 API function `PostMessage`. The `PostMessage` function delivers a Win32 message to a message queue corresponding to a selected window and returns. Note that there is similar function to `PostMessage`, called `SendMessage`, that delivers a Win32 message and waits for the message to be processed fully before returning. Win32 messages are of a fixed size and format. These messages have three fields, a message ID field and two integer parameters. Our testing produced random values in each of these fields, constraining the first field (message ID) to the range of valid message ID's.

Figure 2 shows where each random testing mechanism fits into the Windows NT user input architecture.

Notice in Figure 2 that under both testing conditions, the target application is unable to distinguish messages sent by our testing mechanisms from those actually sent by the system. This distinction is essential to create an authentic test environment.

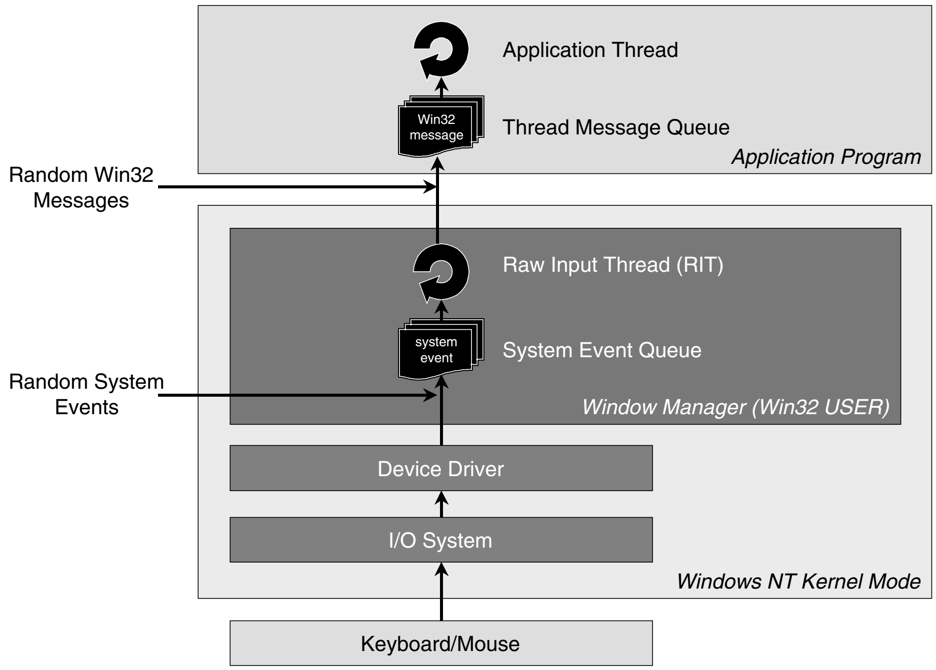

Figure 2: Insertion of Random Input

3 Experimental Method

We describe the applications that we tested, the test environment, our new testing tool (called *fuzz*), and the tests that we performed. We then discuss how the data was collected and analyzed.

3.1 Applications and Platform

We selected a group of almost 30 application programs. While we tried to select applications that were representative of a variety of computing tasks, the selection was also influenced by what software was commonly used in the Computer Sciences Department at the University of Wisconsin. The software includes word processors, Web browsers, presentation graphics editors, network utilities, spread sheets, software development environments, and others. In addition to functional variety, we also strove to test applications from a variety of vendors, including both commercial and free software.

The operating system on which we ran and tested the applications was Windows NT 4.0 (build 1381, service pack 5). To insure that our results were timely, we tested a subset of the applications on the new Windows 2000 system (build 2128, release candidate 2). For the 9 applications that we re-tested on Windows 2000, we obtained similar results to those tested under NT 4.0[2]. The hardware platform used for testing was a collection of standard PCs, each containing a 400 MHz Intel Pentium II processor and 128 MB of RAM.

2. In the near future, we will have a complete re-testing of all applications on Windows 2000.

3.2 The Fuzz Testing Tool

The mechanism we used for testing applications was a new tool, called *fuzz*, that we built for applications running on the Windows NT platform. Fuzz produces repeatable sequences of random input and delivers them as input to running applications via the mechanisms described in Section 2. Its basic operation is as follows:

1. Obtain the process ID of the application to be tested (either by launching the application itself or by an explicit command line parameter).
2. Determine the main window of the target application along with its desktop placement coordinates.
3. Using one of `SendMessage`, `PostMessage`, or `keybd_event` and `mouse_event`, deliver random input to the running application.

Fuzz is invoked from a command line; it does not use a GUI so that our interactions with the tool do no interfere with the testing of the applications. The first version of our Windows NT fuzz tool had a GUI interface but the use of the GUI for the testing tool interfered with the testing of the applications. As a result, we changed fuzz to operate from a command line. The fuzz command has the following format:

```
fuzz [-ws] [-wp] [-v] [-i pid] [-n #msgs] [-c] [-l] [-e seed] [-a appl cmd line]
```

Where `-ws` is random Win32 messages using `SendMessage`, `-wp` is random Win32 messages using `PostMessage`, and `-v` is random valid mouse and keyboard events. One of these three options must be specified.

The `-i` option is used to start testing an already-running application with the specified process ID, and `-a` tells fuzz to launch the application itself. The `-n` option controls the maximum number of messages that will be sent to the application, and `-e` allows the seed for the random number generator to be set.

The `-l` and `-c` options provide finer control of the `SendMessage` and `PostMessage` tests. Null parameters can be included in the tests with `-l` and `WM_COMMAND` messages (control activation messages such as button clicks) can be included with `-c`. These parameters were not used in the tests that we report in this paper.

3.3 The Tests

Our tests were divided into three categories according to the different input techniques described in Section 2. As such, the application underwent a battery of random tests that included the following:

- 500,000 random Win32 messages sent via the `SendMessage` API call,
- 500,000 random Win32 messages sent via the `PostMessage` API call, and
- 25,000 random system events introduced via the `mouse_event` and `keybd_event` API calls.

The first two cases use completely random input and the third case uses streams of valid keyboard and mouse events.

The quantity of messages to send was determined during preliminary testing. During that testing, it appeared that if the application was going to fail at all, it would do so within the above number of messages or events. Each of the three tests detailed above was performed with two distinct sequences of random input (with different random seeds), and three test trials were conduced for each application and random sequence, for a total of 18 runs for each application. The same random input streams were used for each application.

4 Results

The outcome of each test was divided into three categories: the application crashed completely, the application hung (stopped responding), or the application processed the input and we were able to close the application via normal application mechanisms. Since the categories are simple and few, we were able to categorize the success or failure of an application through simple inspection. In addition to the quantitative results, we report on the diagnosis of the cause of the crash from GNU emacs (for which we have the source code).

Figure 3 summarizes the results of the experiments for Windows NT 4.0 and Figure 4 has preliminary results for a subset of the applications run on Windows 2000. If an application failed on any of the runs in a particular category (column), the result is listed in the table. If the application neither crashed nor hung, it passed the tests (and has no mark in the corresponding column).

The overall results of the tests show that a large number of applications failed to reasonably deal with random input. Overall, the failure rates for the Win32 message tests were much greater than those for the random valid keyboard and mouse event tests. This was to be expected, since several Win32 message types include pointers as parameters, which the applications apparently de-reference blindly. The NT 4.0 tests using the `SendMessage` API function produced a crash rate of over 71%, 7% of the applications hung, and a scant 21% successfully dealt with the random input. The tests using the `PostMessage` API function produced a slightly higher crash rate of 93% and a hang rate of 7%. No application was able to successfully withstand the `PostMessage` test.

The random valid keyboard and mouse event results, while somewhat improved over the random Win32 message test, produced a significant number of crashes. Fully 21% of the applications crashed and 21% hung, leaving only 57% of applications that were able to successfully deal with the random events. This result is especially troublesome because these random events could be introduced by any user of a Windows system using only the mouse and keyboard.

The Windows 2000 tests have similar results to those performed on NT 4.0. When we have completed the full retesting of applications on Windows 2000, we will include those results (including finding the cause for the subtle changes in some of the result details).

We examined the emacs application after it crashed from the random Win32 messages. The cause of the crash was simple: casting a parameter of the Win32 message to a pointer to a structure and then trying to de-reference the pointer to access a field of the structure. In the file `w32fns.c`, the message handler is a standard Win32 callback function. This callback function tries to de-reference its third parameter (`lparam`); note that there is no error checking or exception handling to protect this de-reference.

```
LRESULT CALLBACK
w32_wnd_proc (hwnd, msg, wParam, lParam)
{
    . . .
    POINT *pos;
    pos = (POINT *)lParam;
    . . .
    if (TrackPopupMenu ((HMENU)wParam, flags, pos->x, pos->y, 0, hwnd, NULL))
        . . .
}
```

The pointer was a random value produced by fuzz, and therefore was invalid; this de-reference caused an access violation. It is not uncommon to find failures caused by using an unsafe pointer; our previous studies found such cases,

Application	Vendor	SendMessage	PostMessage	Random Valid Events
Access 97	Microsoft	●	●	❍
Acrobat Reader 4.0	Adobe Systems	●	●	
Calculator 4.0	Microsoft		●	
CD-Player 4.0	Microsoft	●	●	
Codewarrior Pro 3.3	Metrowerks	●	●	●
Command AntiVirus 4.54	Command Software Systems	●	●	
Eudora Pro 3.0.5	Qualcomm	●	●	❍
Excel 97	Microsoft	●	●	
FrameMaker 5.5	Adobe Systems		●	
FreeCell 4.0	Microsoft	●	●	
Ghostscript 5.50	Aladdin Enterprises	●	●	
Ghostview 2.7	Ghostgum Software Pty	●	●	
GNU Emacs 20.3.1	Free Software Foundation	●	●	
Internet Explorer 4.0	Microsoft	●	●	●
Java Workshop 2.0a	Sun Microsystems		●	❍
Netscape Communicator 4.7	Netscape Communications	●	●	●
NotePad 4.0	Microsoft	●	●	
Paint 4.0	Microsoft	●	●	
Paint Shop Pro 5.03	Jasc Software		❍	
PowerPoint 97	Microsoft	❍	❍	❍
Secure CRT 2.4	Van Dyke Technologies	●	●	❍
Solitaire 4.0	Microsoft		●	
Telnet 5 for Windows	MIT Kerberos Group		●	
Visual C++ 6.0	Microsoft	●	●	●
Winamp 2.5c	Nullsoft	❍	●	
Word 97	Microsoft	●	●	●
WordPad 4.0	Microsoft	●	●	●
WS_FTP LE 4.50	Ipswitch	●	●	❍
Percent Crashed		71.4%	92.9%	21.4%
Percent Hung		7.1%	7.1%	21.4%
Total Percent Failed		78.5%	100.0%	42.8%

Figure 3: Summary of Windows NT 4.0 Test Results

● = *Crash,* ❍ = *Hang.*

Note that if an application both crashed and hung, only the crash is reported.

Application	Vendor	SendMessage	PostMessage	Random Valid Events
Access 97	Microsoft	●	●	
Codewarrior Pro 3.3	Metrowerks			●
Excel 97	Microsoft	●	●	
Netscape Communicator 4.7	Netscape Communications	●	●	●
Paint Shop Pro 5.03	Jasc Software			❍
PowerPoint 97	Microsoft	❍		❍
Secure CRT 2.4	Van Dyke Technologies	●	●	
Visual C++ 6.0	Microsoft	●	●	❍
Word 97	Microsoft	●	●	●
Percent Crashed		66.7%	66.7%	33.3%
Percent Hung		11.1%	0.0%	33.3%
Total Percent Failed		77.8%	66.7%	66.7%

Figure 4: Summary of Windows 2000 Test Results
● = *Crash,* ❍ = *Hang.*
Note that if an application both crashed and hung, only the crash is reported.

and these cases are also well-documented in the literature [13]. From our inspection of other crashes (based only on the machine code), it appears that this problem is the likely cause of many of the random Win32 message crashes.

5 Analysis and Conclusions

The goal of this study was to provide a first look at the general reliability of a variety of application programs running on Windows NT. We hope that this study inspires the production of more robust code. We first discuss the results from the previous section then provide some editorial discussion.

The tests of random valid keyboard and mouse events provide the best sense of the relative reliability of application programs. These tests simulated only random keystrokes and random mouse movements and clicks. Since these events could easily be caused by a user, they are of immediate concern. The results of these tests show that many commonly-used desktop applications are not as reliable as one might hope.

The tests that produced the greatest failure rates are the random Win32 message tests. In the normal course of events, these messages are produced by the kernel and sent to an application program. It is unlikely (though not impossible) that the kernel would sent messages with invalid values. Still, these tests are interesting for two reasons. First, they demonstrate the vulnerability of this interface. Any application program can send messages to any other application program. There is nothing in the Win32 interface that provides any type of protection. Modern operation systems should provide more durable firewalls. Second, these results point to a need for more discipline in software design. Major interfaces between application software components and between the application and the operating system should contain thorough checks of return values and result parameters. Our inspection of crashes and the diagnosis of the emacs source code shows the blind de-referencing of a pointer to be dangerous. A simple action, such as protecting the de-reference with an exception handler (by using the Windows NT Structured Exception Handling facility, for example), could make a qualitative improvement in reliability.

As a side note, many of those applications that did detect the error did not provide the programmer with reasonable or pleasant choices. These applications did not follow with an opportunity to save pending changes made to the current document or other open files. Doing a best-effort save of the current work (in a new copy of the user file) might give the user some hope of recovering lost work. Also, none of the applications that we tested saved the user from seeing a dialog pertaining to the cause of the crash that contained the memory address of the instruction that caused the fault, along with a hexadecimal memory dump. To the average application user, this dialog is cryptic and mysterious.

Our results also lead to observations about current software testing methodology. While random testing is far from elegant, it does bring to the surface application errors, as evidenced by the 274 crashes encountered during the study. While some of the bugs that produced these crashes may have been low priority for the software makers due to the extreme situations in which they occur, a simple approach to help find bugs should certainly not be overlooked.

The lack of general access to application source code prevented us from making a more detailed report of the causes of program failures. GNU Emacs was the only application for which we were able to diagnose the cause of the crash. In our 1990 and 1995 studies, we were given reasonable access to application source code by the almost all the UNIX vendors. As a result, we provided *bug fixes*, in addition to our bug reports. Today's software market makes this access to application source code more difficult. In some extreme cases (as with database systems, not tested in this study), even the act of reporting bugs or performance data is forbidden by the licence agreements [1] (and the vendors aggressively pursue this restriction).

Will the results presented in this paper make a difference? Many of the bugs found in our 1990 UNIX study were still present in 1995. Our 1995 study found that applications based on open source had better reliability than those of the commercial vendors. Following that study, we noted a subsequent overall improvement in software reliability (by our measure). As long as vendors and, more importantly, purchasers value features over reliability, our hope for more reliable applications remains muted.

Opportunity for more analysis remains in this project. Our goals include

1. Full testing of the applications on Windows 2000: This goal is not hard to achieve, and we anticipate having the full results shortly.

2. Explanation of the random Win32 message results: We were surprised that the `PostMessage` and `SendMessage` results differed. This difference may be caused by the synchronous vs. asynchronous nature of `PostMessage` and `SendMessage`, or the priority difference between these two types of messages (or other reasons that we have not identified). We are currently exploring the reasons for this difference.

3. Explanation of the Windows NT 4.0 vs. Windows 2000 results: Given that we test identical versions of the applications on Windows NT 4.0 and Windows 2000, our initial guess was that the results would be identical. The differences could be due to several reasons, include timing, size of the screen, or system dependent DLLs. We are currently exploring the reasons for this difference.

6 Related Work

Random testing has been used for many years. In some ways, it is looked upon as primitive by the testing community. In his book on software testing[7], Meyers says that randomly generated input test cases are "at best, an inefficient and ad hoc approach to testing". While the type of testing that we use may be *ad hoc*, we do seem to be able to find bugs in real programs. Our view is that random testing is one tool (and an easy one to use) in a larger software testing toolkit.

An early paper on random testing was published by Duran and Ntafos[3]. In that study, test inputs are chosen at random from a predefined set of test cases. The authors found that random testing fared well when compared to the standard partition testing practice. They were able to track down subtle bugs easily that would otherwise be hard to discover using traditional techniques. They found random testing to be a cost effective testing strategy for many programs, and identified random testing as a mechanism by which to obtain reliability estimates. Our technique is both more primitive and easier to use than the type of random testing used by Duran and Ntafos; we cannot use programmer knowledge to direct the tests, but do not require the construction of test cases.

Two papers have been published by Ghosh *et al* on random black-box testing of applications running on Windows NT[5,6]. These studies are extensions of our earlier 1990 and 1995 Fuzz studies[8,9]. In the NT studies, the authors tested several standard command-line utilities. The Windows NT utilities fared much better their UNIX counterparts, scoring less than 1% failure rate. This study is interesting, but since they only tested a few applications (attrib, chkdsk, comp, expand, fc, find, help, label, and replace) and most commonly used Windows applications are based on graphic interfaces, we felt a need for more extensive testing.

Random testing has also been used to test the UNIX system call interface. The "crashme" utility[2] effectively exercises this interface, and is actively used in Linux kernel developments.

Source Code

Note that the source and binary code for the fuzz tools for Windows NT is available from our Web page at: `ftp://grilled.cs.wisc.edu//fuzz`.

Acknowledgments

We thank Susan Hazlett for her help with running the initial fuzz tests on Windows NT, and John Gardner Jr. for helping with the initial evaluation of the Fuzz NT tool. We also thank Philip Roth for his careful reading of drafts of this paper. Microsoft helped us in this study by providing a pre-release version of Windows 2000.

References

[1] M. Carey, D. DeWitt, and J. Naughton, "The 007 Benchmark", *1993 ACM SIGMOD International Conference on Management of Data*, May 26-28, 1993, Washington, D.C. pp. 12-21.

[2] G.J. Carrette, "CRASHME: Random Input Testing", 1996, `http://people.delphi.com/gjc/crashme.html`.

[3] J. W. Duran and S.C. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering* **SE-10**, 4, July 1984, pp. 438-444.

[4] S. Garfinkel and G. Spafford, **Practical UNIX & Internet Security**, O'Reilly & Associates, 1996.

[5] A. Ghosh, V. Shah, and M. Schmid, "Testing the Robustness of Windows NT Software", *1998 International Symposium on Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, November 1998.

[6] A. Ghosh, V. Shah, and M. Schmid, "An Approach for Analyzing the Robustness of Windows NT Software", 2*1st National Information Systems Security Conference*, Crystal City, VA, October 1998.

[7] G. Meyers, **The Art of Software Testing**, Wiley Publishing, New York, 1979.

[8] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl, “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, University of Wisconsin-Madison, 1995. `ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisted.ps`.
Appears (in German translation) as “Empirische Studie zur Zuverlasskeit von UNIX-Utilities: Nichts dazu Gerlernt”, *iX*, September 1995.

[9] B. P. Miller, L. Fredriksen, B. So, “An Empirical Study of the Reliability of UNIX Utilities”, *Communications of the ACM* ***33****,* 12, December 1990, pp. 32-44.
`ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisted.ps`
Also appears in German translation as “Fatale Fehlerträchtigkeit: Eine Empirische Studie zur Zuverlassigkeit von UNIX-Utilities”, *iX* (March 1991).

[10] C. Petzold, **Programming Windows**, 5th ed., Microsoft Press, Redmond, WA, 1999.

[11] J. Richter, **Advanced Windows**, 3rd ed., Microsoft Press, Redmond, WA, 1997.

[12] D. Solomon, **Inside Windows NT**, 2nd ed., Microsoft Press, Redmond, WA, 1998.

[13] J. A. Whittaker and A. Jorgensen, “Why Software Fails”, *Technical Report*, Florida Institute of Technology, 1999, `http://se.fit.edu/papers/SwFails.pdf`.