An Approach for Analyzing the Robustness of Windows NT Software*

Anup K. Ghosh, Viren Shah, & Matt Schmid
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250, Sterling, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
email: {aghosh,vshah,mschmid}@rstcorp.com
http://www.rstcorp.com

Abstract

Today, the vast majority of software executing on defense systems is untrusted commercial off-the-shelf software such as Microsoft Windows software. Vulnerabilities in this software may be exploited to gain unauthorized access to security-critical systems. A number of studies have analyzed the robustness of software that run on Unix systems. The results of these studies have pointed to vulnerabilities in the software that could be potentially exploited into security intrusions. This paper describes a new study aimed at analyzing the robustness of software running on Windows NT systems. This relatively new operating system has not yet been thoroughly analyzed by the security research community using the tools and techniques applied to Unix system software. The goal of the research is to identify robustness gaps in the application software and operating system software that potentially could be exploited for violations of security. Contributions in this paper include a grammar-based input generator, a taxonomy of failure conditions, and experimental results from robustness testing of software running on the NT platform.

1 Introduction

The specter of malicious computer users, organized crime, or hostile nations waging information warfare against the United States is a growing threat—enough to concern the upper echelons of the U.S. government [4, 10]. Because the threat is real, the U.S. must urgently prepare for information warfare attacks. Current security analysis tools attempt to assess network-level vulnerabilities for a given site [2, 3, 9, 5]. These tools do not provide an assessment of an organization's vulnerability to novel threats against vulnerable software.

Recognizing that 90% of military systems use commercial architectures [10], the problem of untrusted software becomes of critical importance to those concerned with information warfare. Application-level vulnerabilities have particular significance in the area of information warfare. While some information warfare campaigns might be waged through frontal assaults on a network firewall, more insidious campaigns are those that wage war from within—via applications that are currently executing on commanders' desktops. A dramatic example of the concern in the U.S. Department of Defense about the vulnerability of defense systems is illustrated by a memo the U.S. Air Force issued. The memo stated that all "push-pull" technology, such as those found in current versions of PointCast, Marimba, Netscape Communicator 4.0, and Microsoft Internet Explorer 4.0, are to be disabled from Air Force network installations. The memo says: "Currently, these technologies introduce security risks and impact data throughput on our networks than cannot be tolerated." [1]. The risk is that push-pull technology can be used to automatically schedule downloads of untrusted executables onto sensitive systems. Provided that Web service is provided to internal users, firewalls are currently ineffective in preventing malicious executables from downloading, installing, and executing using push-pull technology.

Software applications with built-in vulnerabilities (whether intentionally placed or not) may allow a malicious misuse of system resources contrary to security policy. Most software security vulnerabilities result from two factors: program bugs and malicious misuse. Technologies and methodologies for analyzing software in order to discover these vulnerabilities

*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 REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

(and potential avenues for exploitation) have focused on software running on Unix systems. However, with the ever growing market share of Windows NT systems in mission-critical applications, the vulnerability of Windows NT software becomes imperative to assess. In addition, source-code–based analysis techniques often used on Unix systems are not feasible for commercial off-the-shelf (COTS) software running on Windows systems.

In this paper, an approach for analyzing the robustness of Windows NT software is described. The approach, in principle, can be applied to almost any software executing on the Windows NT desktop including application software, user utilities, COM components, shared libraries, and system calls. The goal of this research is to determine what gaps in Windows NT software exist, if any, to classify the nature of these robustness gaps, and to determine how they may impact the security of Windows NT systems. An architecture for a tool that analyzes the robustness of Windows NT software is presented along with results from analyzing GNU console applications that run on Windows NT systems.

2 Prior art

Two research projects have independently defined the prior art in assessing system software robustness: Fuzz [8] 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 [8]. In 1990, the group published a study of the reliability of standard Unix utility programs [7]. 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%.

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.

The causes of crashes were investigated by Fuzz researchers analyzing source code provided by the commercial vendors in addition to the source code available through freely distributed software. Errors programmers made include pointer/array errors, using dangerous input functions, errors in signed characters, and checking for the end of file when reading input. For example, incrementing the pointer past the end of an array is a common error made by many programmers. Also, the use of dangerous input functions such as the `gets()` C function can result in program crashes. More insidious manipulation of dangerous input functions can permit "stack smashing" attacks that allow the execution of arbitrary program code embedded in user input. Another example of a programmer error is assuming that the end-of-file character will always immediately follow a newline character. User input may not necessarily follow this format. Though the Fuzz study did not investigate the vulnerability of programs to buffer overrun attacks, some of the gaps in robustness as measured by the Fuzz study may be exploitable in this manner for security violations.

Finally, it is worth mentioning that in addition to Unix utilities, the Fuzz research studied the robustness of Unix network services, X-Window applications and X-Window servers.

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 com-

binations 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 [6]. 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 Hindering (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 some-times 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 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 Analyzing the robustness of Windows NT software

The objective of the tool described here is to develop an environment in which Windows NT software can be tested for robustness. The types of software to be analyzed in this environment include:

- console applications,
- GUI applications,
- network servers,
- Dynamically Linked Libraries (DLLs),
- system functions, and
- OLE/COM/DCOM components.

The Random and Intelligent Data Design Library Environment (RIDDLE) is an environment that was created for testing the robustness of COTS software on Windows NT systems. Because RIDDLE is used for analysis of COTS software, no access to source code is assumed. The environment uses black-box testing techniques to generate input for the application being tested. Because the goal of the testing is to assess the *robustness* of the software being tested, the input generated can be characterized as "anomalous". That is, the input generated by RIDDLE in most cases falls outside of the normal operational profile for the software being tested.

Figure 1 shows the architecture implemented by RIDDLE for assessing robustness of Windows NT software. A component is tested using the input generation components of RIDDLE. The functions in the input generation library support testing with random input, intelligent input generation using the input grammar of the component, and generation of malicious input. In addition to random testing, generating input intelligently using the input grammar of the component permits stress testing of more of the software's functions from a black-box perspective. Combining random input, malicious input, and boundary value conditions with the legal grammar of the program, the component can be tested more thoroughly than with simple random black-box testing.

RIDDLE does not use an oracle for its analysis. That is, RIDDLE will not reveal whether the component executed correctly. Rather, RIDDLE uses

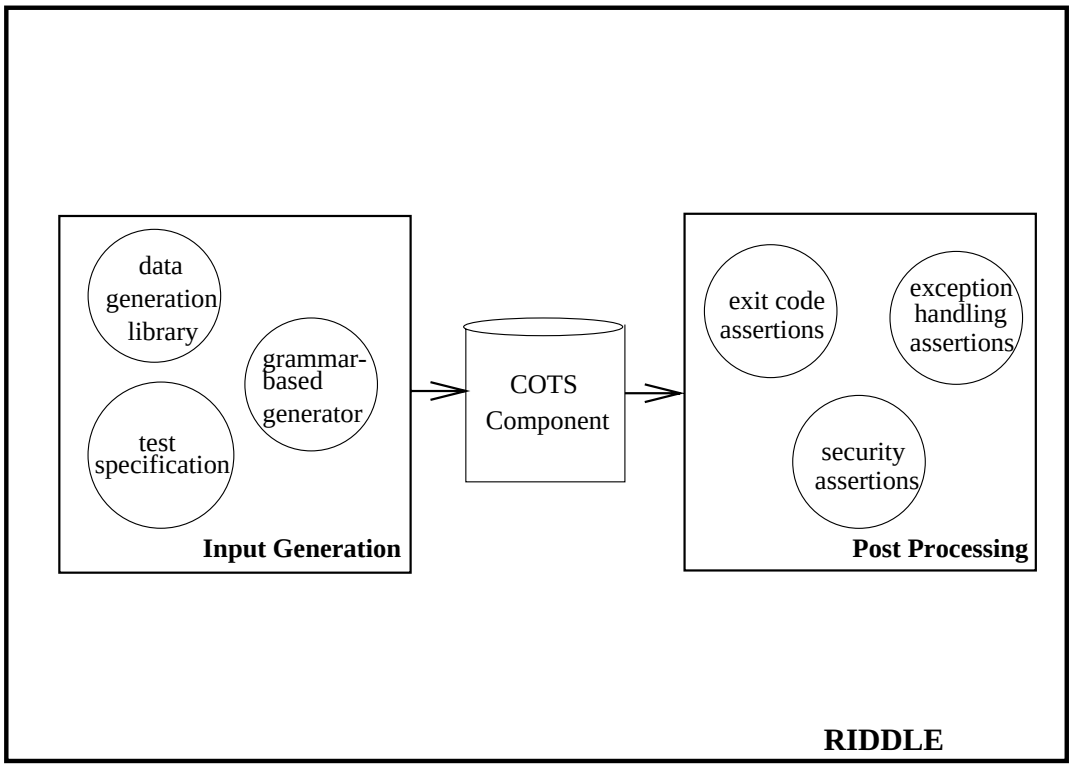

Figure 1: The RIDDLE architecture. RIDDLE is an environment that supports robustness testing of COTS software components. Once the testing specification for a component is decided, grammar-based input generation creates syntactically correct but anomalous input for the component under test. The RIDDLE environment controls the execution of the component under test to send inputs and capture the resulting output from the component. The use of assertions in post-processing can reveal robustness gaps in the software.

assertions of incorrect exit codes, unhandled exceptions, hung processes, insecure behavior (such as wiping files), or system crashes to determine whether a component is robust to anomalous input. The software component under test can be as fundamental as a system function in the NT operating system or as complex as a desktop application with a graphical user interface (GUI). RIDDLE provides the set of input generation functions to drive these components given an interface specification. RIDDLE currently supports testing of console applications (such as DOS utilities) and network servers. Development is on-going for creating test harnesses for OS functions, Windows events, and COM components.

4 Grammar-based robustness testing

The ability to generate intelligent input is essential for the type of stress testing necessary for robustness assessment. The simplest form of testing involves generating random streams of data that are used by the program being tested; this was done in the Fuzz project. While random input generation can test the ability of a program to handle non-conforming input, it typically will not exercise much of a program's functionality. Testing applications with syntactically correct data will result in more thorough testing of that application than testing with purely random data. For example, many applications that take command line arguments will immediately terminate if they do not receive the correct number of parameters, or if they receive an invalid flag. In this situation, random testing will not test any further into the program than this initial check.

On the other hand, syntactically correct arguments (or input parameters) will result in more of the application being tested. In order to exercise more of a program's functionality and to test more of the function's response to anomalous input, RIDDLE employs a grammar-based input generation component. With the creation of a grammar-based input generation component, RIDDLE can test software with syntactically correct data that contains unexpected, anoma-

lous input. The anomalous input itself will be generated through function calls to the data generation library.

Figure 2 shows the architecture of the grammar-based input generation component of RIDDLE. RIDDLE takes a grammar specification as input, and produces random, yet syntactically correct, strings of data by employing functions from the data generation library.

The data generation library can be used to generate data with a variety of levels of intelligence. When testing an application that takes a file name as a parameter, the data generation library can be used to produce a number of substitutions for this parameter. In the case of a file name, the data generation library can produce the name of an existing file, a valid file name that doesn't exist, an invalid file name, the name of a file with specific permissions set, an extremely long file name, or otherwise. Each of these possibilities results in a different test case that may exercise the application being tested in a new way.

The grammar specification is defined in two parts. The first part is a definition of the grammar written in a format similar to Backus-Naur Form (BNF). The second part is a file that contains definitions of all of the tokens used in the grammar. RIDDLE begins by parsing the grammar definition and checking that it is syntactically correct (see Figure 2). Next, RIDDLE begins the process of generating data based on the grammar that it has read. The data that RIDDLE generates relies on the terminal definitions that have been supplied in the token definition file and the functions called from the data generation library.

For each program being tested, the grammar definition must be created. The definition declares the format for the input that is syntactically correct for exercising the program under test. Each production, or rule, consists of a left-hand side and a right-hand side, separated by a colon. The left-hand side identifies a single non-terminal. The right-hand side of the production identifies a set of non-terminals and tokens that the non-terminal on the left-hand side can reduce to. A single non-terminal could have a number of choices of reductions. In this case, each reduction is separated by the symbol "|". Tokens are productions that reduce only to a single terminal (they are therefore only one step away from being terminals themselves). In RIDDLE's case, terminals always reduce to a string. The terminal that a token reduces to is specified separately from the grammar definition in the token definition file.

The following simple grammar is equivalent to the regular expression $(a)^*b$, or any number of a's followed by a single b (*i.e.*, b, ab, aab, etc).

Grammar Definition

```
START:     B
     |     A    START ;
```

Token Definitions

```
     A:     "a"
     B:     "b"
```

RIDDLE always begins with the first production rule that is given. All possible syntactically correct sentences must begin from this production. In this example, the production that defines the non-terminal START is the starting production. When reducing the non-terminal START, the data generator has a choice of picking either the production rule that results in token B (in the example above), or the production rule that results in the token A followed by the non-terminal START. The data generation component uses a repeatable random number generator to choose between the production choices that it faces. Additionally, RIDDLE provides a means of weighting the choice of production rules. This probability is specified by adding a weight to the end of a production. The START production could be written as:

```
START:     B        1
     |     A START  3 ;
```

This grammar definition indicates that the chance of non-terminal START reducing to A START is 3 times more likely then it reducing to B. Or in other words, there is a 75% chance that it will reduce to A START and a 25% chance that it will reduce to B.

When the grammar-based data generation component is called upon to produce syntactically correct data it begins with the starting production. It then chooses productions at random (taking into account the probabilities that were added) until all of the non-terminals have been reduced to tokens. For example, the grammar above might produce the string of tokens $AAAAB$. The final step of the process is to reduce the tokens to their values. In our example, A reduces to the string "a" and B reduces to string "b", giving us the string "aaaab". The data that is produced will be syntactically correct with respect to the grammar that was used to create it.

RIDDLE allows the user to specify that a token reduces to either a string literal, or a function that returns a string literal. In the previous example, token B reduced to the string literal "b". The user could specify that token B reduce to the function

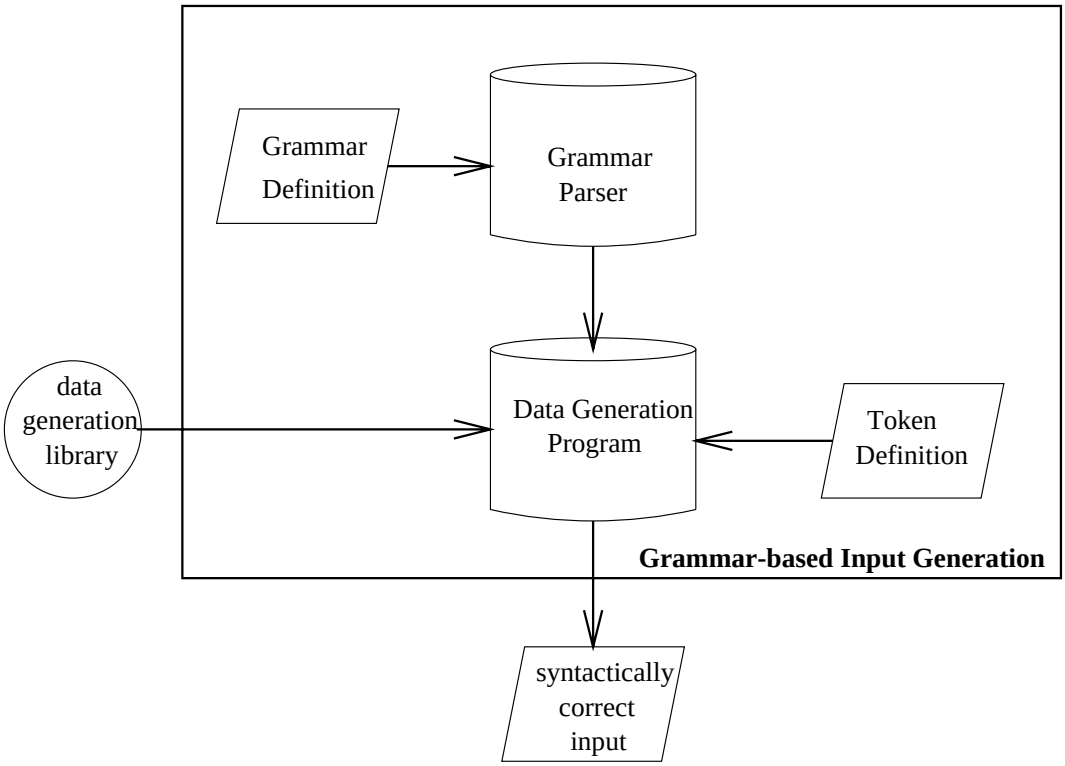

Figure 2: Grammar-based input generation. This component of the RIDDLE architecture permits intelligent black-box testing using anomalous input. The data generation program uses the grammar of a program to generate syntactically correct, albeit anomalous input, for the program under test. Core input generation functions are called from the data generation library of RIDDLE.

call `RandomFileName()`. This function could return a string that would be used in place of the token B (*e.g.*, the choices of file names that were mentioned earlier). This is how RIDDLE is able to generate data that will serve as anomalous program input. The functions that can be used in the reduction of a token make up the data generation library. This library will contain functions that reduce to numerous user-specified strings that can be used for testing purposes.

RIDDLE is designed to test two types of applications: those that take input from the command line, and those that take streams of data as input. The former class of applications includes many commonly used operating system utilities. Examples of such Unix utilities include the `cp`, `ls`, `man`, and `ps` commands. Windows NT examples include `mode`, `tree`, `subst`, and `format`. Applications that rely on streams of data include Web servers (`httpd`), `ftp` servers, `ftp` clients, `lpr`, and `grep`.

A simplified example of testing the UNIX utility `cp` can be demonstrated. This grammar definition only accounts for a small subset of the command's functionality, but it is useful for illustrative purposes.

Grammar Definition

```
START:     SP     LOW_F     START
     |     SP     LOW_P     START
     |     SP     FILE_NAME   FILE_NAME ;
```

Token Definition

```
LOW_F:     "-f"
LOW_P:     "-p"
FILE_NAME: GenerateFileName()
SP:         " "
```

This grammar specification will provide for the production of an input string that consists of any number of -f's and -p's followed by two strings produced by the `GenerateFileName()` function. The input strings that are produced could then be used in test cases. The following are some test cases that RIDDLE might produce:

```
cp -p oaisud aoisudf
cp -p -f -p -p <existing file>
```

```
   <existing directory>
cp -f -p <open file>
   <extremely long buffer of characters>
```

These test cases are syntactically correct usages of the `cp` command, combined with anomalous data. If the anomalous data that is supplied by the data generation library results in undesirable application failure, then a weakness in the robustness of the application has been detected.

5 Experimental analysis

The RIDDLE architectures shown in Figure 1 and Figure 2 have been implemented in a prototype tool for experimentation. RIDDLE has been used to test GNU Win32 software. The GNU software was selected partially based on the results from the Fuzz analysis [8]. In the Fuzz analysis, the GNU software fared the best in robustness testing compared to other commercial implementations of similar utilities. Because GNU tools are written and maintained by world-class programmers, this software should serve as the baseline of how robust good software can be. The study, however, is not comprehensive. Further research will investigate other software that runs on the NT platform. The GNU software tested are: `ls`, `rm`, `mkdir`, `rmdir`, `tcsh`, `cat`, `cp`, and `chmod`.

RIDDLE tests each program using only its known external interface. For each different test run, RIDDLE can be customized to use a different number of iterations, argument lengths, and argument content. Using the grammar-based input generation, RIDDLE can also intelligently test the program using anomalous, but syntactically correct, input data.

For each program tested, an input set was generated. Each input set was then varied along with the two independent variables (length and content) in order to get variation with the 3-tuple input set (parameter-set, length, content). The length variable was set to seven different values representing the number of bytes in input: 0, 10, 100, 512, 1024, 4096, and 8192. The content was varied according to: English alphabet characters, printable ASCII characters, and the full ASCII character set.

For the eight different programs chosen, a total of 16 parameter sets were created. The independent variables for each set were varied giving 21 unique test runs (7 length values * 3 content values) per parameter sets. Thus, 336 test runs (21 * 16) were created. Each test run consisted of a 100 iterations that varied the random content according to the random generators. Thus a total of 33,600 different test cases were run on the software.

After running the experiments, the resulting data was post-processed by indexing the data according to various fields such as (application, content) and (application, length). The results from the post-processing were then analyzed to detect any pattern present in the error codes that resulted from the testing.

Figure 3 shows the taxonomy of failures used for analyzing the results of the robustness testing. The classification of failures resulting from the experiment were divided into executions that terminated and those that did not. Processes that did not terminate were caught by a timeout in RIDDLE and classified as hung processes. Executions that terminated were classified as those that terminated abnormally, and those that terminated normally. Of the latter, these are further sub-divided into applications that reported an error condition on exit, and those that did not.

5.1 Summary of results

As stated earlier, the results from the experiments were analyzed for failure classification. This analysis consisted of examining and classifying the exit codes and errors resulting from the application execution. The purpose of the analysis was to find significant patterns relating to the appearance of invalid error conditions.

From the results obtained, 23.41% of the test runs ended with the applications exiting abnormally with system error conditions, 1.55% of the applications hung (the timeout was 15 seconds), and 11.51% of the applications exited silently with no errors. In the latter condition, the applications accepted anomalous input without either crashing or without signaling an error. From a purely robustness standpoint, this condition may be desirable to permit continued operation (particularly in applications where high availability is essential). However, the lack of an error code may indicate a latent error condition that may manifest itself otherwise such as in flawed outputs that are trusted. The rest of the test runs ended with the applications exiting normally after signaling an error condition (through the use of exit codes).

Correlating different control variables, the analysis shows a relationship between the length of the input and the abnormal termination of the applications. The analysis found that as the length of the generated input parameters increased, the probability of the application terminating abnormally increased as well. This effect reached a plateau at length 4096. Thus, there was not a significant difference between the results achieved with a 4 kilobyte string and those with an 8 kilobyte input. On the other hand, none of

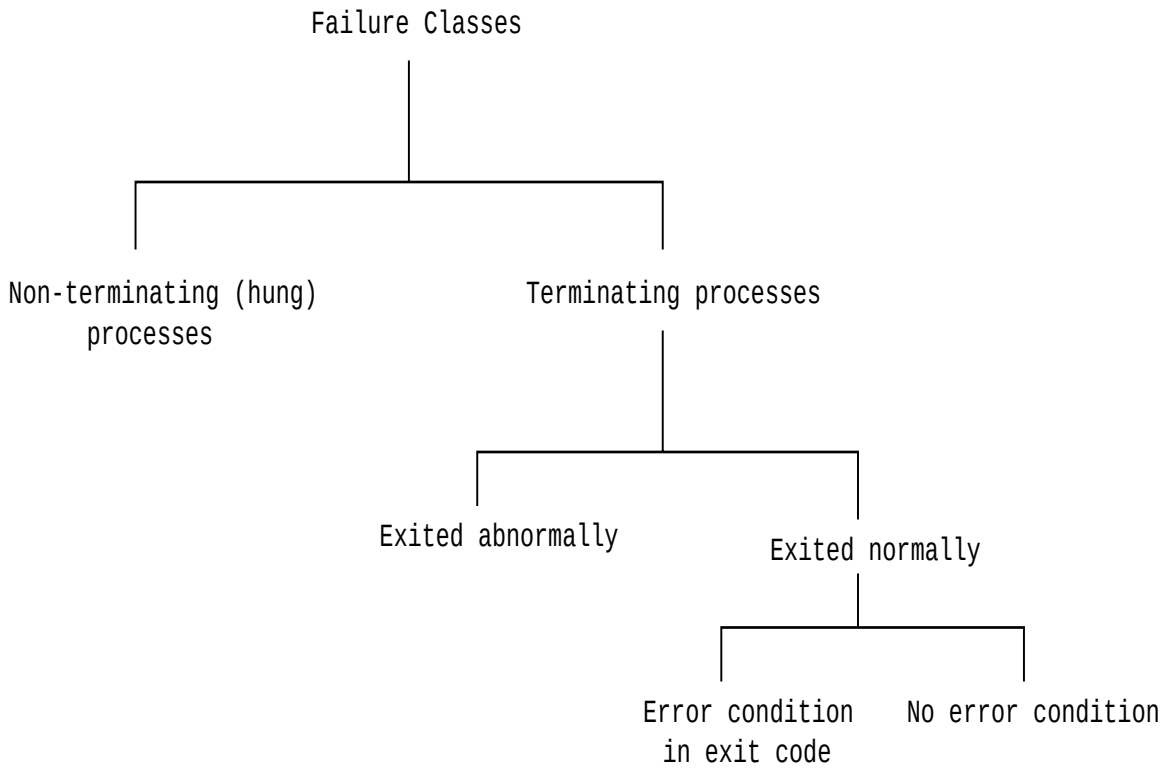

Figure 3: Taxonomy of failures. This figure shows a taxonomy of failures developed from analyzing the outputs of the robustness testing of software running on the Windows NT platform. Processes that terminate abnormally create system error conditions. Processes that exit normally exit either with an error code or fail silently.

the tested applications terminated abnormally when the parameter length was 0, 10 or 100 bytes.

A similar, though not identical, trend was noticed in regards to the content of the generated parameters. None of the applications terminated abnormally when the generated inputs consisted of only the alphabet (a-z, A-Z). The abnormal terminations were divided almost equally between parameters generated using only printable ASCII, and those using the full character set, with the printable ASCII strings getting a slightly higher (though not significant) rate of abnormal terminations.

6 Conclusions

This paper describes an environment and approach to testing Windows NT software for robustness to unexpected, anomalous input. The importance of robustness testing was established by prior research on Unix-based systems using Fuzz and Ballista testing environments. To our knowledge, RIDDLE is the first research tool to be applied outside of the operating system vendor to testing the robustness of NT systems.

The grammar-based input generation component gives RIDDLE the unique capability to generate intelligent (*i.e.*, syntactically correct) input that is anomalous just the same. This capability gives the ability to black-box test software in a manner that can stress test and exercise its functions more fully than random black-box testing techniques. The objective of this research is to develop a robustness testing environment for Windows NT systems to identify potential robustness gaps in software that runs on Windows NT systems.

To date, RIDDLE has been applied to the GNU Win 32 utilities. A taxonomy of failures was developed that allowed classification of the failure conditions that resulted from the robustness testing. The analysis showed that nearly one-quarter of the test runs ended with abnormal terminations with system error conditions. The conclusions that can be drawn from this result are that even this well-maintained software written by software experts around the world is not handling anomalous inputs completely robustly. Another significant result from the analysis showed that the applications fared worse when the input length increased

up to 4 kilobytes in length.

Future research will involve exploring robustness gaps to determine the potential to be exploited into security intrusions. In addition, more analysis using the grammar-based input generation component, more analysis of other NT software including network servers, Windows graphical interfaces, operating system functions, dynamically linked libraries, and COM components. Further development in RIDDLE will develop a menu-driven interface for RIDDLE for enabling easy set-up of experiments for testing NT software.

References

[1] Edupage Editors. Air force thinks push-pull technology too risky. *RISKS Digest*, 19(57), January 25 1998.

[2] D. Farmer and E.H. Spafford. The COPS security checker system. In *USENIX Conference Proceedings*, pages 165–170, Anaheim, CA, Summer 1990.

[3] D. Farmer and W. Venema. Improving the security of your site by breaking into it. Available by ftp from `ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking.101.Z`, 1993.

[4] Louis Freeh. Domestic law enforcement and electronic civil defense. In *Proceedings of the 5th InfoWarCon*, September 1996. Presentation, September 6.

[5] C. Klaus. Internet security scanner. Available by ftp from `ftp://ftp.iss.net/pub/iss`, 1995.

[6] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks.

[7] 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.

[8] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[9] D.R. Safford, D.L. Schales, and D.K. Hess. The TAMU security package: An ongoing response to Internet intruders in an academic environment. In *Proceedings of the Fourth Usenix UNIX Security Symposium*, pages 91–118, Santa Clara, CA, October 1993.

[10] 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.