AUTOTESTCON 99, 30 August to 2 September 1999, San Antonio, Texas.

A DIMENSIONALITY MODEL APPROACH TO TESTING AND IMPROVING SOFTWARE ROBUSTNESS

Jiantao Pan
Electrical and Computer Engineering Department & Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, PA 15213
412-268-4264
jpan@cmu.edu

Philip Koopman
Electrical and Computer Engineering Department & Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, PA 15213
412-268-5225
koopman@cmu.edu

Daniel Siewiorek
Computer Science Department & Institute for Complex Engineered Systems
Carnegie Mellon University, Pittsburgh, PA 15213
412-268-5228
dps@cs.cmu.edu

Abstract - Software robustness problems may hinder the use of Commercial Off-The-Shelf (COTS) software modules and legacy software modules in mission-critical and safety-critical applications. This research focuses on hardening COTS and legacy software modules against robustness failures triggered by exceptional inputs. An automated approach is presented that is capable of identifying the triggers of the robustness failures. A fault model – the Dimensionality Model – is used to guide analysis. An experiment is described which demonstrates the feasibility of automating the process of analyzing failure causes and hardening against certain data types in POSIX function calls, for example, `NULL` pointer values and scalar data types such as `INT` and `FLOAT`. The final goal of this research is to provide users a tool to harden COTS and legacy software modules automatically.

I. INTRODUCTION

The robustness of a software component is a measure of how it functions in the presence of exceptional inputs or stressful environmental conditions [10]. Software robustness is gaining more and more significance among application developers. The reasons are three-fold:

First, our lives are becoming more "computerized". Traditional analog devices are being replaced by their cheaper digital counterparts. The use of micro-controllers is growing in automobiles, airplanes, weapons, medical devices, consumer products, etc. New services based on computerized facilities are also emerging. Hence, more and more aspects of our life are dependent on software.

Second, in order to cut development cost and time, application developers are being pressured to use Commercial Off-The-Shelf (COTS) software modules or legacy software components to assemble applications [1][2]. Often, COTS software components are optimized for cost or performance, and have not been specifically designed to operate in mission-critical or application-critical systems. COTS components may function correctly under normal conditions, but they may crash, hang or exhibit other robustness failures when exceptional or unspecified conditions occur.

Third, robustness may not have been a design priority in COTS software. With the shortening of software product cycles and shrinking of profit margins, development cost is becoming a dominant concern along with time-to-market. High performance and new functionality, as opposed to robustness, are often given first priority.

A particular source of robustness problems is exceptional inputs. As many as two-thirds of system crashes might be caused by exception-handling failures. Decades ago, the Apollo 11 Lunar Lander computer rebooted three times due to exceptional operating conditions, nearly causing an aborted mission. More recently, the maiden flight of the Ariane 5 rocket failed, with an estimated loss of $500 million, due to an improperly handled exception in the software of the dual-redundant on-board control computer [3].

There is every indication that exception handling will continue to be a problem in critical applications, and may well become a serious problem in everyday computing as well. To make matters worse, the trend to using COTS software may mean that a lack of source code or detailed specifications will make improving robustness of systems even more challenging than it has been in the past with custom-written software.

The goal of the Ballista project (http://www.ices.cmu.edu/ballista) is automatic hardening of COTS and legacy software modules against exceptional inputs that cause robustness problems. There are three major steps toward achieving this goal: automated robustness testing, automated failure analysis, and automated protection code generation.

Automated robustness testing has already been accomplished [2][9]. Additionally, a Dimensionality Model [11] has shown that more than 80% of the robustness failures found in the 15 POSIX compliant operating systems we have tested are caused an exceptional value on only a single parameter.

In this paper we introduce an analysis method guided by the Dimensionality Model that can pinpoint the triggers for robustness failures automatically. Additionally, we show that automated protection code generation is feasible for at least some exceptional parameter values. Experiments have proven successful in analyzing and hardening against `NULL` pointer values and exceptional scalar values for integers and floating-point numbers.

II. BACKGROUND

Historically, limited effort has been devoted to understanding software robustness. Research in software robustness focuses on testing methods and fault-injection techniques. Both approaches are combined in the work presented here.

Black-box testing [4] assumes only inputs and outputs are accessible for the unit under test. There is no need to know the code structure and execution paths. In addition to tests for validating the module's functional correctness, a significant portion are "dirty tests", which consist of combinations of valid and invalid inputs, in order to stimulate abnormal behaviors in the software module.

The AETG system [6] uses a combinatorial testing method. Based on programmer experience that faults caused by interactions of parameter values are relatively rare, AETG uses the minimum number of test cases to cover test for parameters singly, in pairs, and in small tuples.

Fault-injection is another way to elicit software robustness problems. Faults and exceptional values are injected into the module under test. A more robust software module can withstand more and longer "attacks" before breaking down or gracefully degrading. Because in most cases the testing domain is infinite, randomly generated values are used to probe the module. Examples include CRASHME [14], CMU-Crashme [13], and Fuzz [7]. The randomness of these approaches and their dependence on concurrent execution of many tests makes any robustness failures that are found difficult to reproduce and isolate.

The Xept [8] project at Bell Labs has developed an instrumentation tool to intercept library function calls to link in error detection and error recovery code. It provides a language to write specifications for interception and handling code for functions, and an object-code instrumentation tool, called Xmangler, to link application code with error detection and error recovery code. Although not focused on automated code generation technique, Xept provides a convenient framework and proves that a software wrapper can be used to intercept library function calls in object code.

III. TOWARD AUTOMATIC ROBUSTNESS HARDENING

Our goal is to automate the process of testing and hardening COTS and legacy software modules against robustness failures triggered by exceptional inputs. The process works as follows:

1. Test the software module using normal and exceptional inputs.

2. Analyze the test results; identify the corresponding parameter(s), and exceptional value(s) that trigger robustness failures.

3. Generate protection wrapper code to guard against the exceptional values that caused robustness failures.

4. Link the protection wrapper code to the module being hardened.

While Bell Labs has developed methods [8] for the linking stage, technological advances are required in the first three steps. The second step is important to bridge the gap between the testing results and protection code generation. Once the exceptional parameters have been identified, protection code can be generated to check for these parameter values.

Automated Robustness Testing

A full-scale, web-based client-server testing engine [9] has been developed for automated robustness testing at the API level. It has the ability to test 233 POSIX system calls specified in standard POSIX.1b. [5] Test cases are predefined in the template file. User programs can also be tested, provided that the parameter data types are recognized by the server. Users can define their own data types and test cases to expand the testing capability by writing a template for each new data type. Test instances are generated using a combinatorial method. The CRASH scale [2] is used to measure the severity of the robustness failures. The testing method and the CRASH scale metric are described in the remainder of this section.

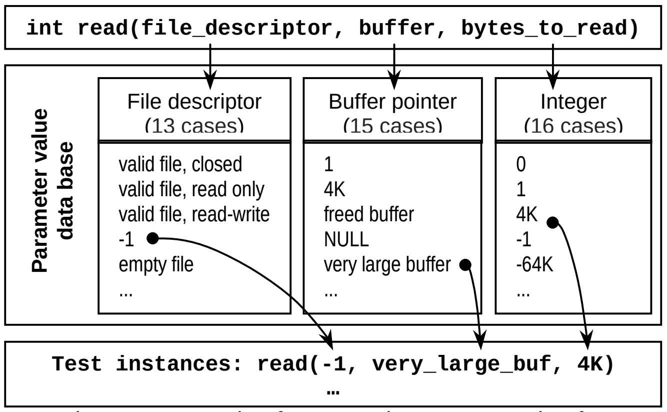
The Combinatorial Testing Method

A combinatorial testing methodology is used to generate tests. As an example, testing of the POSIX function call `read` is shown in Figure 1. The system call `read` accepts three parameters, `fd* file_des`, `char *buffer`, and `int size`. The testing harness maintains a test data template file containing all the test cases defined for each data type. During testing, the harness will generate all possible combinations of the parameters from the template file. In this example, there are a total of 3120 possible combinations of the test values for the three parameters, so 3120 actual tests can be executed to exercise this function. A single combination is tested at a time, with necessary setup and tear down of the environment performed individually for each test.

To achieve scalability, the test cases are defined per data type rather than for each function. New functions can be easily tested if the data types have already been defined. To test all 233 POSIX calls, only 20 data types are needed.

The CRASH Scale Metric

In general we categorize the test responses into passes and fails. When an appropriate error code is returned, the test case is considered a “Pass”. Failures are categorized by a scale called “CRASH”, which stands for Catastrophic, Restart, Abort, Silent and Hindering. Catastrophic failures refer to failures that can cause the whole system to stop functioning, requiring rebooting the machine. Restart failures mean that the process hangs, requiring intervention such as killing the test task. Restart failures are detected by a watchdog timer process. Abort failures refer to the abnormal termination of the tester process (*i.e.*, core dumps in UNIX operating systems). Silent failures are


int read(file_descriptor, buffer, bytes_to_read)
Parameter value data base
File descriptor (13 cases)
valid file, closed
valid file, read only
valid file, read-write
-1
empty file
...
Buffer pointer (15 cases)
1
4K
freed buffer
NULL
very large buffer
...
Integer (16 cases)
0
1
4K
-1
-64K
...
Test instances: read(-1, very_large_buf, 4K)
…
Figure 1. Example of parametric test generation for POSIX function read

false successes, which means that a success is returned when actually an error code should have been. Hindering failures mean returning incorrect error codes. In this study Restart and Abort failures are the initial targets.

Automated Failure Analysis

We seek an automated approach for analyzing the patterns of test responses to determine which parameter values cause failures. Ideally, the analyzer algorithm should also provide guidance to the testing engine. 1.1 million data points gathered from Ballista robustness testing on 15 POSIX API implementations[1] in UNIX operating systems have been analyzed. Based on this analysis, a Dimensionality Model method [11] has been developed to guide the code generation process. The remainder of the section describes the Failure Analysis process.

The Dimensionality Model [11]

The idea of dimensionality is illustrated by two definitions.

- Parameter dimensionality: Consider a software module `f`, taking a list of arguments `(x1, x2, ...)`. The parameter dimensionality is defined as the number of arguments taken by the software module.

For example, the POSIX function `read(file_des, buffer, bytes_to_read)` takes three parameters, so its parameter dimensionality is three.

There is also a special case that a function accepts no parameters at all, which is beyond the scope of this model.

- Robustness failure dimensionality: Given a particular set of parameter values that cause a robustness failure, the number of the parameters that actually contributes to the failure is defined as the robustness failure dimensionality.

[1] Includes QNX 4.22, QNX 4.24, FreeBSD 2.2.5, NetBSD 1.3, SunOS 4.1.3, Digital Unix 3.2, Digital Unix 4.0, SunOS 5.5, IRIX 5.3, HP-UX B.10.20, IRIX 6.2, LINUX 2.0.18, LynxOS 2.4.0, HP-UX A.09.05, AIX 4.1.

For example, suppose that `f(x1, x2, x3)` fails when `x1=NULL`, regardless of the values of `x2` and `x3` (both normal and exceptional). In this case the `NULL` value of parameter `x1` is the only contributing factor to the failures. So all the failures where `x1=NULL` would be 1-dimensional failures.

It is obvious that the failure dimensionality can not exceed the parameter dimensionality. In the example of `read(file_des, buffer, bytes_to_read)`, an invalid `file_des` may trigger 1-dimensional failures, if the function does not check to prevent invalid `file_des` values. It is also possible that if `bytes_to_read` is greater than `buffer (length)`, we can expect 2-dimensional failures, since both of the parameter values contribute to the failures. Note that it is possible for a specific failure to belong to both a high- and low-dimensionality failure set. In such cases, we can count that failure as having the lowest possible dimensionality. In other words, for our measurements the lower dimensionality characteristic prevails.

Experimental results in POSIX API testing show that 1-dimensional failures are the most common failures encountered, accounting for more than 80% of the overall failure rate. This means that if we only choose to protect 1-dimensional failures, we can lower the overall system robustness testing failure rate by 80%.

Automatic Identification of Fault Dimensionality

Robustness failure dimensionality indicates the number of concurrent triggers required to activate a particular robustness failure. If we could automate the process of determining robustness failure dimensionality, we would be able to know the exceptional parameter values that are responsible for observed failures. This is an important step toward automatic hardening.

While intuitively parameter dimensionality is the number of parameters accepted by the function, robustness failure dimensionality is not immediately obvious for functions with parameter dimensionality higher than one. However, the dimensionality can be revealed by the pattern of the robustness responses observed during testing. As an example, Figure 2 shows the robustness response pattern of the function `fprintf(FP, STR)` in HP-UX B.10.20. This function prints the string pointed to by `STR` to the file pointed to by `FP`. Each number along the axes represents an index to an actual test value. For example, number 9 on axis `FP` represents test case `NEG_VALUE` (an integer value of negative one). A circle means the test fails at the combination of the parameters at the point, exhibiting a robustness failure. A dot means the test passes.

The test responses form patterns. In the column `FP(index)=9`, all the tests fail regardless of the value of `STR`. We can not be 100% sure that all this column of robustness failures are 1-dimensional, caused by an exceptional value of `FP=NEG_VALUE`, because we did not test all possible values that `STR` can have (and this is not possible). However, we can still conclude that the failures in the column of `FP=NEG_VALUE` is 1-dimensional, since it is unlikely that all eight values coincidentally would give the same behavior, given that `STR` has both good and exceptional values. Furthermore, we know that if we write protection code to check this single exceptional input of `FP=NEG_VALUE`, we can effectively guard against this column of exceptional values, including the other infinite possible `STR` values not covered in the testing space. The other four points at (2,5), (2,6), (5,5), (5,6) are 2-dimensional failures. To protect against them, both parameters need to be checked. In general, for n-dimensional failures, n parameters must be checked to protect against them.

More generally, 1-dimensional failures will form a line in a 2-D graph, as shown in Figure 2. They will form a plane in a 3-D graph. 2-dimensional failures will form single or clustered circles in a 2-D graph, and lines in a 3-D graph.

This process of identifying dimensionality information from combinatorial testing response patterns can be effectively automated. If test cases are properly defined, the Analyzer does not need any specific knowledge of the semantic information of the function, the function name under test, data types or the test case values.

Automated Protection Code Generation

For integer, floating-point data types and `NULL` pointers, value checking is sufficient to detect exceptional values. Therefore, the process of protection code generation for these data types are is straightforward. After the dimensionality information is revealed and the exceptional values found, value comparison statements against these exceptional values are generated and plugged into a skeleton wrapper program. Any call to the target function to be protected is redirected to the wrapper. This call redirection can be achieved using a tool such as Xmangler [8].

IV. EXPERIMENTAL RESULTS

In the experiment, we have implemented 1-dimensional failure hardening for integer, floating-point, and `NULL` pointer data values.

We selected `NULL` pointer in our experimental study on an intuitive basis. `NULL` pointers are one of the most commonly encountered of exceptional inputs and are easily overlooked. While processing the data gathered in POSIX API testing, the extraordinarily high failure rate of `NULL` drew immediate attention. We estimate that the overall test result will be 10%

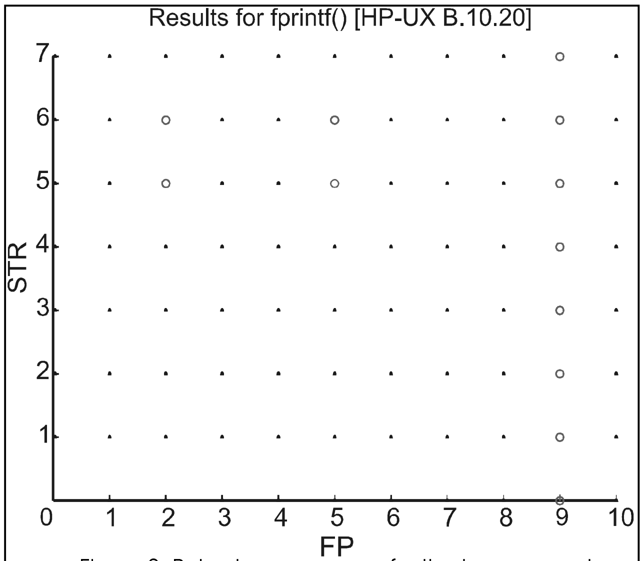


Figure 2. Robustness response for the two parameter function fprintf from HP-UX

```
/* Here is a sample user c program that hidden NULL
value can cause robustness failures.*/
#include <stdio.h>

void main(){
  FILE *fp;     /* file pointer*/
  char buf;
  fp = fopen ("datafile", "r");
  printf("Reading data file\n\n");
  buf= fgetc(fp);
}
```

(a) A typical user program

```
Reading data file

Segmentation fault (core dumped)
```

(b) The output of the program when datafile does not exist

Figure 3. Example user program with hidden robustness failures

better if the POSIX implementations add `NULL` pointer checking. To be specific, 82.5% of tests involving a `NULL` file pointers and 46.0% of tests involving `NULL` buffer pointers cause robustness problems.

As an example of how easily `NULL` pointers can arise, Figure 3(a) shows a user program module `myread.c`. It opens a file and tries to read a character from the file. Since the program does not check for the existence of the file and necessary access permissions, there are potential robustness problems if the access rights are violated. For example, if the file does not exist, the user will see a core dump after getting the first output message, as shown in Figure 3(b).

The failure depicted in Figure 3 happens because the validity of the file pointer is not properly checked by POSIX function `fgetc()`. To protect the program against this failure, function `fgetc()` is tested by the Ballista robustness testing suite. The robustness failures related to file pointer are found in the test result file, shown in Table 1. In Table 1, the first column shows the testing results, considering only Abort failures (core dumps). The second column shows the return value if the test passes, or –1 if the test fails. The third column shows the combinations of the parameter values of each test case. In this example, only one parameter is accepted, which is `fpxx`. `fpxx` is the equivalent name for data type file pointer, or `FILE *`. The test value of fpxx is composed of three orthogonal attributes called "dials" (because one can imagine spinning various dials to create combinations of such attributes): `Existence`, `Access mode`, and `Access permissions`.

We then perform a dimensionality analysis on the failure data file. It is obvious that all the three failures are 1-dimensional, caused by invalid parameter value `fpxx_NOTEXIST.` In

Result	**Return**	**Parameters**
Pass	9	fpxx_EXIST fpxx_APPEND fpxx_NOPERMISSIONS
Pass	9	fpxx_CLOSED fpxx_READ fpxx_NOPERMISSIONS
Abort	-1	fpxx_NOTEXIST fpxx_WRITE fpxx_NOPERMISSIONS
Pass	0	fpxx_EXIST fpxx_READ fpxx_NOPERMISSIONS
Pass	9	fpxx_CLOSED fpxx_APPEND fpxx_NOPERMISSIONS
Pass	9	fpxx_CLOSED fpxx_WRITE fpxx_NOPERMISSIONS
Abort	-1	fpxx_NOTEXIST fpxx_APPEND fpxx_NOPERMISSIONS
Pass	9	fpxx_EXIST fpxx_WRITE fpxx_NOPERMISSIONS
Abort	-1	fpxx_NOTEXIST fpxx_READ fpxx_NOPERMISSIONS

Table 1. Test result file for `fgetc()`

```
name FILE* fpxx;

......

  READ{
      fileMode = ’r’;
    }
  WRITE{
      fileMode = ’w’;
    }
  APPEND{
      fileMode = ’a’;
    }
  EXIST, CLOSED{
      _theVariable = fopen(filename,&fileMode);
    }
  NOTEXIST{
      _theVariable=NULL;
    }
   PERMISSIONS{
      chmod(filename,_PERMISSIONS_INT);
    }
......
```

Figure 4. Template file for file pointer

the Ballista test suite, this is the denotation for a non-existent file, equivalent to a `NULL` file pointer. The analyzer comes to this conclusion by sorting the testing results by parameter value `fpxx_NOTEXIST` and observing that a `NULL` value always results in a test failure.

The actual code segment in the template file that generates the test cases for `fpxx` is shown in Figure 4. For the case of `fpxx_NOTEXIST`, the value is equivalent to a `NULL` file pointer. Based on this knowledge, we can generate protection code guarding against this value. In this case, a checking statement will be sufficient to intercept the NULL file pointer causing the failure. The generated header file for `fgetc` is shown in Figure 5.

The call h_`fgetc`() is the hardened version of `fgetc()` with embedded NULL file pointer checking. Using the `Xmangler` tool, we will be able to redirect call instances from `fgetc()`to `h_fgetc()` without user intervention. In this experiment we do not yet have the `Xmangler` tool [8] available, so we manually change all instances of `fgetc()` to `h_fgetc()` in the

```
/*Common Include Files*/
......
/*User defined Include Files*/
......

void Check1Dparam1(FILE* param1){
......

       FILE* _theVariable;
       char fileMode;
       int fd;
......

       /*NOTEXIST*/
       _theVariable=NULL;
       if(param1 == _theVariable){
              puts("Dangerous 1-Dimensional parameter value detected\n");
              exit(DEFAULT_RVAL);
       }
}/*Check1Dparam1*/

int h_fgetc(FILE* param1){
       Check1Dparam1(param1);
       return fgetc( param1);
}
#endif /*__HARDENED_FGETC__*/
```

Figure 5. Generated protection file for fgetc()

user source file to finish the last linking phase.

To verify the effectiveness of the generated protection file for `fgetc()`, we first tested it again using the Ballista testing suite. As expected, the above three robustness failures are replaced by successes with error return code 99. Second, we compile this header file together with the user function shown in Figure 3(a). As shown in Figure 6, the exceptional inputs are captured. The function returns with a warning message.

In the experiment, the hardened version of `fgetc()` simply turns Abort failures into a default error return code. To achieve more flexibility, we can provide more options before exiting. For example, for resource related problems, retrying the task sometimes can fix the problem. Process migration can keep a long-running task from aborting when facing resource contention. Garbage collection or disk de-fragmentation can also be launched at some point if a problem is related to memory or disk resources. For the parameter values that are not in the testing database, or are ambiguous, checkpoint-rollback procedure can be called to save and restore system state.

```
Reading data file

Dangerous 1-Dimensional parameter value detected
```

Figure 6. Output of the user program after hardening

- Optimize protection code efficiency to reduce run-time cost.

V. FUTURE WORK

Although the experimental results to date are successful and promising, the following aspects should be considered to make the process more practical for hardening generic user software.

- Scale the hardening capability to encompass more data types.

Simple data types are probably easier to protect than complex parameters such as data structures. The challenge is to protect these complex data types while avoiding false alarms.

- Implement random sampling to increase dimensionality analysis confidence.

Because the Failure Analyzer bases its conclusion purely on the test response pattern, validation testing by random sampling before reaching a conclusion will increase prediction accuracy and avoid at least some false alarms.

- Adopt the Xmangler tool introduced in [8].

Using this tool, we will be able to protect COTS modules at the object code level. No source code is required to harden a module.

- Add on user-defined hooks to direct exceptional input value handling to user functions.

VI. CONCLUSIONS

The objective of this research is to explore the possibility of generating robustness failure protection code automatically. We have been successful in achieving limited results for integers, floating point numbers, and `NULL` pointer values. This proves that automatic protection code generation for COTS software is feasible. It remains to be seen to what extent it can be generalized to other data types.

A template-based protection code generation methodology directed by the Dimensionality Model is discussed in detail. Protection code generation includes four phases: detection, diagnosis, protection code generation and linking. This paper puts emphasis on the automation of diagnosis and protection code generation phases. The Dimensionality Model is used in the diagnosis phase to pinpoint the trigger for a failure. The result is utilized by the code generator to effectively generate protection code.

The cost and development time of software could be significantly reduced if there were a widely used component industry. [1] Automatic robustness hardening might enable more people to use commercial software modules for mission critical applications and safety critical applications, and to reuse legacy software modules for new and existing applications. Although there are many factors that must be addressed in using a COTS software component approach, automated robustness hardening may be one technique that helps developers reduce design costs and improve time to market while producing a robust system.

ACKNOWLEDGEMENT

This research was sponsored by DARPA ITO under contract DABT 63-96-C-0064.

REFERENCES

[1] Voas, J., *Certifying Off-the-Shelf Software Components,* IEEE Computers, June 1998, page 53-59

[2] Kropp, N., Koopman, P. & Siewiorek, D., *Automated Robustness Testing of Off-the-Shelf Software Components*, FTCS, Munich, Germany, June 23-25, 1998.

[3] Lions, J., *Ariane 5 Flight 501 Failure*, European Space Agency, Paris, July 19, 1996. *http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html*, Accessed May 1, 1999.

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

[5] *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 1: Realtime Extension [C Language]* (IEEE Std 1003.1b-1993), IEEE Computer Society, 1994.

[6] Cohen, D., S. Dalal, M. Fredman & G. Patton, *The AETG System: an approach to testing based on combinatorial design,* IEEE Trans. on Software Engr., 23(7), July 1997, pp. 437-444.

[7] Miller, B., *et al.*, *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*, Computer Science Technical Report 1268, Univ. of Wisconsin-Madison, 1995.

[8] Vo, K. P., Wang, Y. M., Chung, P. E., & Huang, Y., *Xept: A Software Instrumentation Method for Exception Handling,* Proc. Int. Symp. on Software Reliability Engineering (ISSRE), Nov. 1997.

[9] DeVale, J., Koopman, P., Guterndorf, D., *The Ballista Software Robustness Testing Service*, to appear in Testing Computer Software Conference (TCS) 1999, June 1999.

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

[11] Pan, Jiantao. *The Dimensionality of Failures – A Fault Model for Characterizing Software Robustness.* To appear in Proceedings of the 29th International Symposium on Fault Tolerant Computing (FTCS-29), Madison, WI, June 1999.

[12] Feibus, Mike. Intel's new orange bug-fix strategy. PC WEEK Online, May 04, 1999. http://www.zdnet.com.au/pcweek/opinion/0526/26feibus.html accessed May 1, 1999.

[13] Mukherjee, A., Siewiorek, D., "Measuring Software Dependability by Robustness Benchmarking," CMU Technical Report CS-94-148, 1994

[14] Carrette, G., "CRASHME: Random input testing", (no formal publication available) http://people.delphi.com/gjc/crashme.html accessed May 12, 1998.