The Dimensionality of Failures – A Fault Model for Characterizing Software Robustness

Jiantao Pan
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jpan@cmu.edu

Abstract

Software robustness is a major concern in using Commercial Off-The-Shelf (COTS) software components in mission-critical applications. Exceptional inputs are a major cause of robustness problems, but it can be difficult to properly guard against them. In this paper we introduce the Dimensionality Model – a model for characterizing the triggers for software robustness failures. The model is based on pin-pointing how many function call parameters are responsible for a failure. The study reveals that approximately 82% of robustness failures found in operating systems we have tested are attributable to 1-dimensional values, namely, failures caused by exceptional values for a single parameter. This result suggests that screening based on a 1-dimensional assumption might be useful in hardening COTS software modules against robustness failures triggered by exceptional input values.

1. Introduction

Writing robust software [3] is costly. The problem is aggravated by increasingly severe constraints such as tight budget, rapid time-to-market and short product life cycle. Using Commercial Off-The-Shelf (COTS) software modules or legacy software components to assemble applications might be a good way to cut cost; but the COTS and legacy modules may not have been written to handle exceptional conditions well. Usually COTS software is designed to function correctly with normal inputs under normal situations, but may crash, hang, or exhibit other non-robust behaviors under exceptional inputs or in abnormal execution environments. If such software modules are used in mission-critical or safety-critical applications, they may cause losses or even entire mission failure, such as happened with Ariane Flight 501 [6]. This creates a dilemma: while on one hand we want to use COTS software modules because of cost issues, on the other hand non-robustness of the COTS software modules might put mission success at risk.

If candidate software modules can be tested, analyzed and hardened automatically to improve robustness, that might promote the use of COTS and legacy software in an even broader range of applications including mission-critical and safety-critical areas. Such a capability might also make it cheaper and more efficient to improve the robustness of newly written software components. But, there are no existing characterizations of the types of values that must be hardened against for this approach to work.

This paper introduces a fault model, called the Dimensionality Model, to characterize Application Programming Interface (API) level software robustness failures. Using test results gathered by a combinatorial testing method, this model helped explain the patterns formed by parameter(s) contributing to observed failures. Preliminary results in the POSIX [5] API found that 82% of the robustness failures identified were caused by a single parameter.

Section 2 discusses the Ballista testing method for data gathering. Section 3 introduces the Dimensionality Model, while Section 4 gives the preliminary results based on analyzing the test data gathered from 15 POSIX API implementations in UNIX operating systems. Conclusions can be found in Section 5.

2. Testing methodology

This work is built upon the prior work of the Ballista project [1]. The Ballista project has implemented an API-level automated testing tool for COTS software modules. The test cases are specified per data type, with identical test cases applied to every function that takes any particular set of data types.

A combinatorial testing method is used to generate the test cases for each module. For example, the system call read() accepts three parameters: `fd* file_des, char`

`*buffer`, `int size`, and each of these three parameters has a set of tests associated with it. During testing, a test harness generates all the possible combinations of predefined test cases, drawing upon tests for each parameter data type. Each combination is tested individually, and robustness failure results (pass, crash, or hang) are logged into a file.

3. The Dimensionality Model

In this section we take steps to introduce the Dimensionality Model. We introduce two important definitions that summarize the Dimensionality Model.

- 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.

The definition stems from the dimensionality of the function when drawing the pattern of test response in space. For example, `f(x1)` has a parameter dimensionality of 1, but `g(x1, x2)` is 2-dimensional. Function `read(file_des, buffer, bytes)` takes three parameters, so its parameter dimensionality is three.

- 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. For example, suppose that `f(x1, x2, x3)` fails when `x1=NULL`, regardless of the values of `x2` and `x3` (normal or 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)`, 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.

Failure dimensionality information is crucial for writing efficient protection code. After testing a function, a list of robustness failures is generated, many of which may be the result of several low-dimensionality failure values left unguarded. Using the Dimensionality Model, we may categorize the failures into different groups. For example, a `NULL file pointer`, when not checked, can lead to many 1-dimensional failures. We can choose to write code to check for 1-dimensional values like this first to improve run-time efficiency, or even just harden 1-dimensional failures if resources or capabilities are too limited to address multi-dimensional failure modes.

4. Low dimensionality is common

Although perhaps not articulated as such, the idea of dimensionality has existed for a long time. In the testing domain (*e.g.*, [2]), failures caused by interactions of multiple parameters are assumed to be rare, so that test case generation is optimized to cover 1 or 2 parameters. The prevalence of low dimensionality is assumed to be true, according to programmer experience. The question of whether that assumption is justified remains open until now, mainly because there are not enough data studies.

With the help of the Ballista robustness testing service, we have gathered 1.1 million data points from the testing of 15 POSIX compliant operating systems from 10 vendors. The overall failure rate of an operating system is the average of all the POSIX functions tested on that operating system. The results are normalized by the number of tests within each function, and then averaged across all the functions tested. These results (potentially weighted by execution profile) can be interpreted as the

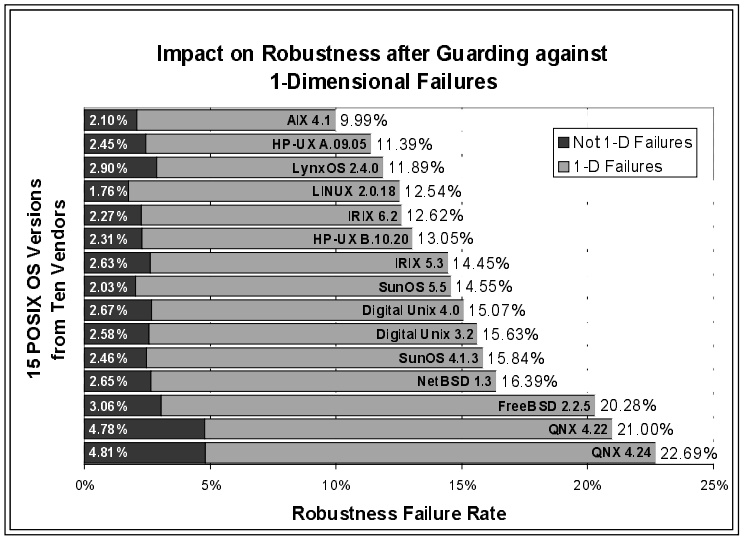

Figure 1. Impact of 1-Dimensional Failures

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

relative likelihood of encountering a robustness failure when passing an exceptional data value to POSIX API functions on that platform. In Figure 1, the robustness failure rate ranges between 9.99% for IBM AIX and 22.69% for QNX 4.24. The variation is substantial, although they are all implementations of the same interface specification. For some applications, those levels of robustness might not be acceptable. In certain cases, it might therefore be desirable to automatically harden those POSIX calls against robustness failures, if they are to be used in mission-critical and safety-critical situations, which is our future research goal.

In our experiment, we focused on the "likelihood" that a robustness failure is 1-dimensional. In Figure 1, the light portion of the bars indicates this likelihood. In other words, if we have successfully protected against 1-dimensional failures, all the scores of the operating system will be significantly improved, from 15.16% failure rate on average down to merely 2.76%, cutting down the failures to 18% of the original failure rate if we simply check for 1-dimensional robustness failures.

Figure 2 gives another view of the same information, stretching the system failure rate to 100% scale. The graph indicates that although the variation of the failure rate is significant from one operating system to another, the relative portion of failures that are 1-dimensional is approximately a constant. Specifically, on average 81.75% of the system failure rate is attributed to 1-dimensional failures, with a standard deviation of 3.24%. This provides evidence that 1-dimensional failures, the simplest form of robustness failures in the Dimensionality Model and presumably the easiest to protect against, are uniformly prevalent across all the operating systems we have tested.

5. Conclusions

In this paper we have presented the Dimensionality of Failures, a new model for characterizing software robstness failures. We discussed how the model is related to combinatorial testing, and showed that one-dimensional failures are prevalent across a wide range of POSIX operating system implementations.

Based on these results, we speculate that robustness hardening can be done with the help of the Dimensionality Model. The dimensionality information can be automatically distilled given combinatorial testing results. The correct sequence of analyzing and protecting against robustness failures should start with low dimensionality failures, in order to be cost-effective.

The Dimensionality Model is intended to be general, so that it can be extended to characterize other kinds of software defects, or other characteristics in a parameter space. Extending its applicability is an area of future work.

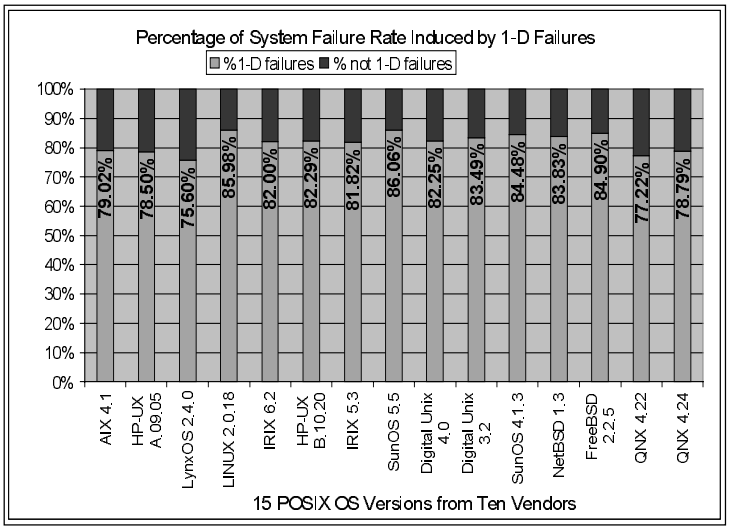

Figure 2. Percentage of Failure Rate Due to 1-Dimensional Failures

6. Acknowledgements

This research was sponsored by DARPA contract DABT63-96-C-0064 (the Ballista project) and ONR contract N00014-96-1-0202. Thanks to my advisor Philip Koopman and co-advisor Daniel Siewiorek for their helpful advice on this paper and my research.

7. References

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

[2] Cohen, D., S. Dalal, M. Friedman & 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.

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

[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] Lions, J. (Chair), *Ariane 5 Flight 501 Failure*, European Space Agency, Paris, July 19, 1996. *http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html*, Accessed Dec. 4, 1997.