

# Investigating Rare-Event Failure Tolerance: Reductions in Future Uncertainty

J. Voas, F. Charron  
Reliable Software Technologies  
21515 Ridgetop Circle  
Sterling, VA 20166  
{jmvoas,fhchar}@RSTcorp.com

K. Miller  
Dept. of Computer Science  
U. of Illinois at Springfield  
Springfield, IL 62794  
miller@uis.edu

## Abstract

*At the 1995 Computer Assurance (COMPASS) conference, Voas and Miller presented a technique for assessing the failure tolerance of a program when the program was executing in unlikely modes (with respect to the expected operational profile). In that paper, several preliminary algorithms were presented for inverting operational profiles to more easily distinguish the unlikely modes of operation from the likely modes. This paper refines the original algorithms. This paper then demonstrates the new algorithms being used in conjunction with a failure tolerance assessment technique on two small programs.*

## 1 Introduction

This paper describes the implementation of a technique that assesses the failure tolerance of the “rare” region of an input space. The *rare* region is the partition of the input space that contains inputs that are highly unlikely to be selected for execution under typical testing and operation.<sup>1</sup> Here, *rare* does not necessarily imply a small subset of the input space, i.e., it is possible by our definition that the rarest portion of the input space is actually a larger subset than is represented by the more frequently chosen inputs.

When we observe a program executing on inputs that were selected from the rare region of the input space, we will gain knowledge of how *robust* (or “failure tolerant”) the software will be during unusual operational events. When we observe a lack of robustness during our analysis, this suggests that the software may not perform acceptably if unusual events occur during

---

<sup>1</sup>At this time, we will not concern ourselves with what probability thresholds constitute highly unlikely, and simply consider it as a conceptual notion that can be set by the user.

operation. Likewise, if we repeatedly observe robust behavior, this suggests that the software is sufficiently hardened against anomalous events. This suggestion is not a guarantee of future failure-tolerant behavior, but it is an indicator of that desired behavior.

Our notion of software failure tolerance is different from the traditional meaning of fault-tolerance in electrical and computer engineering [2]. Traditionally, fault-tolerance refers to building subsystems from redundant components that are placed in parallel. For example, the space shuttle has one main computer and two back-up computers. The back-up computers are expected to replace the main computer if the main computer fails. Similarly, n-version programming is a programming paradigm that advocates executing multiple versions (that were independently designed/written and implement the same software function) in parallel and then taking a vote of the different versions to determine the most frequent output. The general intuition behind n-version programming is that the most frequent output value observed must be the correct result; this is, of course, not always true [1].

Our meaning of failure tolerance contrasts with the traditional view. For us, software is deemed as failure-tolerant if and only if:

1. The program is able to compute an *acceptable* result even if the program itself suffers from incorrect logic, *and*
2. The program, whether correct or incorrect, is able to compute an *acceptable* result even if the program itself receives *corrupted* incoming data during execution.

The key to this definition is the semantics for “acceptable.” The criteria for acceptance can include characteristics such as correct behavior and safe behavior.

## 2 The Rare-event Failure tolerance Assessment Technique

Our rare-event failure tolerance technique is based on a fault-injection process termed “extended propagation analysis” (EPA) [3]. EPA injects program state corruptions into a program that is being executed with respect to some test or operational profile, and studies whether these “artificially created” program state corruptions propagate, i.e., cause the program output to also be corrupted in various ways. The results provided by EPA are dependent on how the fault-injection is implemented, and the distribution of inputs employed when the software is executed. Because the main contributions of this paper are: (1) the algorithms for making distribution inversion more practical, and (2) the results from a small experiment, we will not redescribe EPA, and will instead refer the reader to [3].

“Inverse operational distributions” derived from operational distributions are not exact mathematical inverse functions, however they capture the intuitive property of elements that occur frequently in the original operational distribution,  $Q$ , occur infrequently in the inverse operational distribution,  $\bar{Q}$ . An *input distribution* for a piece of software is the probability density function that assigns to each possible test case  $i$  a probability that  $i$  will be selected on a randomly selected execution of the software in a certain environment. If a particular  $Q(i)$  equals 0, then  $i$  is not a member of the input space as we previously defined it. The “inverted” input distribution, call it  $\bar{Q}$ , assigns a large probability to elements that had a small probability in  $Q$ , and assigns a small probability to elements that had a large probability in  $Q$ . To use our approach, you will need a description of  $Q$ ; the more accurate this is, the more accurate  $\bar{Q}$  will be. (If a description of  $Q$  is unavailable, then our approach cannot be employed.)

|                        | High failure tolerance | Low failure tolerance |
|------------------------|------------------------|-----------------------|
| High Likelihood Inputs | <b>Little</b>          | <b>Great</b>          |
| Low Likelihood Inputs  | <b>Little</b>          | <b>Little</b>         |

Table 1: **The likelihood of software failure occurring given low and high probability inputs and different levels of failure tolerance**

There is an interesting relationship between software testing’s ability to produce failures (when faults exist),  $Q$ ,  $\bar{Q}$ , and failure-tolerance, as shown in Table 1. Note that when failure-tolerance is high, software testing has little ability to produce failures, even if faults

exist. Also note that inputs, which might have great fault-revealing ability, are of little help during testing if they have little or no chance of being selected. So during testing the only hope of detecting problems is to have software that has reduced failure tolerance with respect to the high probability inputs, because those are traditionally the ones selected.

We begin with a review of the inversion algorithm published at Compass’95 [4]; throughout this paper, this algorithm will be referred to as **Original**;

### Algorithm 1:

1. Let  $N$  be the number of different legal inputs. Let  $M = 1/N$ , the mean of the probability density distribution over  $N$  possible inputs.
2. For each element  $i$ , let  $g'(i) = 2 * M - Q(i)$ .
3. Find the minimum  $g'(i), m$ . If  $m \geq 0$ ,  $g'(i)$  is  $\bar{Q}$ . Otherwise, proceed to step 4.
4. When  $m < 0$ , let  $g''(i) = (g'(i) + \mathbf{abs}(m))/(1 + N * (\mathbf{abs}(m)))$ . Then  $g''$  is  $\bar{Q}$ .

### 2.1 Bounds

For almost all operational distributions, lower and upper bounds must be specified if we are to make the inverted distribution well-defined. **Original** failed to require this, causing it to result in poor inverted approximations of the operational profile.

The summation or integration of all the density values for a discrete or continuous distribution must equal 1. Therefore, bounds must be included in order to avoid violating this definition. In some distributions, the range is bounded by the parameters (i.e. Equilibrally, Uniform). In distributions where the range is not bounded, bounds may be provided by the user or determined by shifting the mean by some specified number of standard deviations ( $\mu \pm 3\sigma$ ). Often, the lower bound is already defined to be 0 as part of the definition of the distribution.

### 2.2 Using Original

In this effort, **Original** was implemented first to determine how problematic it actually is. The algorithm was applied to invert the Exponential( $\mu = 1$ ) distribution. In order to invert the Exponential(1) distribution, the algorithm was modified slightly to extend to

continuous distributions. The step-by-step customization of **Original** to the Exponential(1) distribution is described below.

1. For **Original**, we define a value  $M$  that represents the mean of the Equilikely distribution over the specified range of inputs. Since we are now interested in extending the algorithm to a continuous range  $[a, b]$ , we will begin by redefining  $M$  so that it represents the mean of the Uniform distribution over the specified range. Let  $M = 1/(b - a)$ .

For the Exponential(1) example, the distribution definition already specifies the lower bound at 0. We specify the upper bound by setting it 6 standard deviations away from the mean ( $\mu + 6\sigma = 7$ ). Since we have now specified a range  $[0, 7]$ , we can calculate the value of  $M = 1/7$ .

2. Instead of running through each element as described in this step for the discrete case, we simply define a new continuous function

$$g'(x) = 2M - f(x), \quad a \leq x \leq b$$

where  $f(x)$  is the probability density function (p.d.f.) of the continuous distribution.

For a general Exponential distribution,  $g'(x)$  takes the form of

$$g'(x) = \frac{2}{b - a} - \frac{1}{\mu} e^{-x/\mu}, \quad a \leq x \leq b.$$

In particular, we have selected  $\mu = 1$ ,  $a = 0$  and  $b = 7$ , yielding the following p.d.f.

$$g'(x) = \frac{2}{7} - e^{-x}, \quad 0 \leq x \leq 7.$$

3. We need to find  $m$ , the minimum of  $g'(x)$ . In general, the value of  $x$  such that  $g'(x)$  is minimum is the same as the value of  $x$  such that  $f(x)$  is a maximum. In other words, the minimum value of  $g'(x)$  can be obtained by finding the value which yields the maximum of the original probability density function.

For the Exponential distribution, the maximum of the p.d.f.  $f(x)$  always occurs at  $x = 0$ . So, the minimum of  $g'(x)$  is

$$g'(0) = \frac{2}{b - a} - \frac{1}{\mu}.$$

For the Exponential(1) example,  $m$  evaluates to  $-5/7$ . Since  $m < 0$ , we proceed to Step 4.

4. For this step, instead of running through each element as described for the discrete case, again we simply define a new continuous function  $g''(x)$ :

$$g''(x) = \frac{g'(x) + abs(m)}{\int_a^b (g'(x) + abs(m)) dx}, \quad a \leq x \leq b.$$

The integration term in the denominator serves to normalize the function so that integration of the function  $g''(x)$  over range  $[a, b]$  is 1.

For the Exponential distribution, if  $m < 0$  the general form of the  $g''(x)$  function is

$$g''(x) = \frac{\frac{1}{\mu} - \frac{1}{\mu} e^{-x/\mu}}{\int_a^b \left( \frac{1}{\mu} - \frac{1}{\mu} e^{-x/\mu} \right) dx}, \quad a \leq x \leq b.$$

Solving the integration term in the denominator, this simplifies to

$$g''(x) = \frac{\frac{1}{\mu} - \frac{1}{\mu} e^{-x/\mu}}{\frac{b}{\mu} - \frac{a}{\mu} + e^{-b/\mu} - e^{-a/\mu}}, \quad a \leq x \leq b.$$

For the Exponential(1) example, we make the substitutions  $\mu = 1$ ,  $a = 0$ , and  $b = 7$ , and obtain

$$g''(x) = \frac{1 - e^{-x}}{6 + e^{-7}}, \quad 0 \leq x \leq 7$$

A new probability density function has now been derived based on the original Exponential(1) distribution, by following the steps in **Original**. The resulting cumulative probability density function (c.d.f.),  $G''(x)$  can be obtained as well:

$$G''(x) = \frac{x + e^{-x} - 1}{6 + e^{-7}}, \quad 0 \leq x \leq 7.$$

It can be verified that the cumulative density function  $G''(x)$  evaluates to 1 at  $x = 7$ .

We are interested in generating a random number from this new distribution we derived from the original Exponential(1) distribution. First, let  $u$  be a randomly generated number from the Uniform(0,1) distribution. A random number generated from this new distribution with c.d.f.  $G''(x)$  is the value of  $x$  that satisfies:

$$\frac{x + e^{-x} - 1}{6 + e^{-7}} = u$$

or

$$\frac{x + e^{-x} - 1}{6 + e^{-7}} - u = 0$$

The root of this equation can be approximated numerically. The value of the root is returned as the randomly generated value in the range  $[0, 7]$  from the inverted Exponential(1) distribution, where inversion was the result of applying **Original**.

## 2.3 A New General Inverting Algorithm

So a new algorithm was developed to produce an “inverse” that is more intuitive than the “flattened” effect we observe when **Original** is applied to the exponential. This new algorithm is designed to reflect the shape of the original distribution, so that the rarest possible event(s) in the original distribution will occur with the highest level of frequency enjoyed by the most common occurring event(s) in the original distribution. Such an algorithm would transform the characteristic form of the Normal curve from “bell-shaped” to “bowl-shaped”, for instance.

General algorithms for reverse-ordering the frequencies with which values are randomly generated are presented here. The fundamental paradigm exploited here is that we take the original values given by the distribution and simply exchange which inputs are mapped to each of these values. The first algorithm applies to discrete random variables and the second one applies to continuous random variables. Short-cut algorithms that perform the reverse-ordering more efficiently for special classes of probability distributions are discussed later.

For inverting a probability distribution with p.d.f.  $f(x)$  for a *discrete* variable, the following general algorithm should be used:

### Algorithm 2:

1. For each value of  $x$  in the defined range  $[a, b]$ , calculate the probability density function value  $f(x)$ .
2. Let  $N = b - a + 1$ . Order the values of  $x$  from 1 to  $N$  according to their respective  $f(x)$  values. Fill  $N$ -sized vector  $\mathbf{O}$  with values in range  $[a, b]$  such that

$$f(O_1) \geq f(O_2) \geq \dots \geq f(O_{N-1}) \geq f(O_N)$$

where each  $O_i$  value is unique.

3. Define a second  $N$ -sized vector,  $\mathbf{P}$ , that serves as a mapping from each value of  $x$  to its corresponding order number. The relationship between  $\mathbf{O}$  and  $\mathbf{P}$  is

$$O_i = x \Leftrightarrow P_x = i, a \leq x \leq b, 1 \leq i \leq N.$$

4. Generate a random number  $r$  in the range  $[a, b]$  from the original distribution.

5. Return the value corresponding to the reverse-order position in  $\mathbf{O}$ . Using the vectors defined above, let  $i = N - P_r + 1$ . Return the value  $O_i$  as a randomly generated value from the inverted distribution.

A problem with specifying a bounded range  $[a, b]$  is that the original probability density function used to generate random number  $r$  in Step 4, integrated over  $[a, b]$  does not truly evaluate to 1, but some value close to 1. In order to be more statistically correct, the p.d.f. of the original distribution used to generate the random number should be normalized by the integral over  $[a, b]$ . However, if the range is chosen such that the difference between the integral and 1 is insignificant enough, this number can be randomly generated from the original distribution over the unbounded range and any values generated beyond the range  $[a, b]$  can be thrown out.

To generate random numbers from an inverted *continuous* distribution, we propose the following algorithm:

### Algorithm 3:

1. Break up the range  $[a, b]$  into  $N$  subintervals, with equal interval length  $\delta$ . Create an  $N$ -sized vector,  $\mathbf{F}$ , in such a way that

$$F_i = \frac{f(a + (i - 1)\delta) + f(a + i\delta)}{2}, 1 \leq i \leq N,$$

where  $f(x)$  is the probability density function of the continuous distribution being inverted. The right-hand side of the equation is an estimate of the integral of  $f(x)$  between  $a + (i - 1)\delta$  and  $a + i\delta$

2. Order values in  $\mathbf{F}$  from 1 to  $N$ . Fill  $N$ -sized vector  $\mathbf{O}$  with values in range  $[1, N]$  such that

$$F_{O_1} \geq F_{O_2} \geq \dots \geq F_{O_{N-1}} \geq F_{O_N}$$

where each  $O_i$  value is unique.

3. Define a second  $N$ -sized vector,  $\mathbf{P}$ , that serves as a mapping from each index into  $\mathbf{F}$  to its corresponding order number. The relationship between  $\mathbf{O}$  and  $\mathbf{P}$  is

$$O_i = j \Leftrightarrow P_j = i, 1 \leq i \leq N, 1 \leq j \leq N.$$

4. Generate a random number  $r$  in the range  $[a, b]$  from the original distribution. Define  $i$ :

$$i = \lfloor (r - a) / \delta \rfloor + 1,$$

where the  $i$ -th subinterval is the one containing value  $r$ . Calculate the lower bound on the  $i$ -th subinterval,  $a_i$ :

$$a_i = a + (i - 1)\delta$$

Finally, calculate the difference between  $r$  and  $a_i$ :

$$\Delta = r - a_i$$

5. Find the index  $j$  in  $\mathbf{O}$  that is the reverse-order position of index  $i$  in  $\mathbf{F}$ :

$$j = N - P_i + 1.$$

Let  $k = O_j$ .

6. The value to return is contained in the  $k$ -th subinterval. Calculate the lower bound of the  $k$ -th subinterval,  $a_k$ ,

$$a_k = a + (k - 1)\delta$$

Shift the  $a_k$  value by  $\Delta$  to obtain the value within the  $k$ -th subinterval to return. Return  $a_k + \Delta$  as a randomly generated value from the inverted distribution.

## 2.4 Simplified Inverting Algorithms for Special Cases

These two new algorithms for inverting discrete and continuous distributions are not necessary for reverse-ordering distributions with special characteristics. In general, any discrete or continuous distribution that is a strictly increasing or decreasing function over the defined range can be easily reverse-ordered by substituting the following algorithm:

### Algorithm 4:

1. Generate a random number  $r$  in the range  $[a, b]$  from the original distribution.
2. The distribution inverter can simply return the value:

$$x = a + b - r$$

An example of a distribution for which this algorithm can be applied is the Exponential( $\mu$ ) distribution. The form of the probability density function is such that the maximum function value is always at the lower bound of the range, which is 0. The p.d.f. is a strictly decreasing function, since as  $x$  increases,  $f(x)$  decreases exponentially.

Another simplified algorithm may be substituted for the new general inverting algorithms if the distribution form is known to be symmetric about a single extreme point (maximum or minimum). For this special case, the following method provides a more efficient way to reverse-order the original distribution:

### Algorithm 5:

1. Generate a random number  $r$  in range  $[a, b]$  from the original distribution.
2. Define  $x^*$  to be the value of  $x$  for which the value of  $f(x)$  is its maximum or minimum. The distribution inverter returns

$$x = a + x^* - r, \text{ if } r < x^*$$

or

$$x = x^* + b - r, \text{ if } r \geq x^*.$$

An example of a symmetric distribution for which this algorithm can be applied is the Normal distribution. By definition of the standard Normal distribution,  $x^* = 0$ . As a rule, for a Gauss distribution (general form of the Normal), this algorithm may be used with  $x^* = \mu$ . Note that this shortcut algorithm for reverse-ordering may not be applied to the Lognormal distribution, because it is not symmetric.

## 2.5 Inverting Multidimensional Distributions

The idea of reverse-ordering single-dimensional distributions can be extended to multi-dimensional distributions. A general algorithm could be developed that will invert any multi-variable distribution using the reverse-ordering technique. Such an algorithm would involve the discretization of the hypersurface defined by the p.d.f., and then ordering the discretized components according to density. This approach is not practical. Furthermore, the random generation of correlated random variables usually is handled using the multivariate Normal distribution.

The algorithm for inverting a multivariate Normal distribution follows. The multivariate Normal

distribution has the symmetric property that simplifies the reverse-ordering process. The algorithm describes the steps involved in generating a random vector  $\mathbf{x} = (x_1, x_2, \dots, x_N)^T$ . Bounds must be supplied to the ranges for each  $x_i$ , similar to the single-dimensional distributions. Specify lower bound vector  $\mathbf{a} = (a_1, a_2, \dots, a_N)^T$  and upper bound vector  $\mathbf{b} = (b_1, b_2, \dots, b_N)^T$ , where  $a_i \leq x_i \leq b_i$ , for  $i = 1$  to  $N$ . To invert the  $N$ -dimensional multivariate Normal distribution with mean vector  $(\mu_1, \mu_2, \dots, \mu_N)^T$ , apply the algorithm below.

**Algorithm 6:**

1. Define the maximum value of the p.d.f.  $f(\mathbf{x})$ , which occurs at vector  $\mathbf{x}^* = (\mu_1, \mu_2, \dots, \mu_N)^T$ .
2. Generate random vector  $\mathbf{r} = (r_1, r_2, \dots, r_N)^T$  from the original multivariate Normal distribution.
3. For  $i = 1$  to  $N$ , calculate:

$$x_i = a_i + x_i^* - r, \text{ if } r < x_i^*$$

or

$$x_i = x_i^* + b_i - r, \text{ if } r \geq x_i^*.$$

Return the vector  $\mathbf{x} = (x_1, x_2, \dots, x_N)^T$  as the random vector generated from the inverted multivariate normal distribution.

### 3 Experimental Results

After developing the formulae in Section 2, we implemented them. We then inserted this inversion utility into the random test case generation utility in our EPA tool, and ran two experiments. Our two examples were supplied by NASA: an aircraft yaw controller, and an aircraft automatic landing system.

#### 3.1 Experiment 1: Yaw

Our first experiment was a yaw controller. *Yaw* refers to the motion about the z-axis, which is the axis perpendicular to the plane of the aircraft. Yaw control is accomplished through the adjustment of the aircraft’s rudder. We analyzed the yaw damping algorithms that are implemented in the **yaw.c** module; this

code reduces yaw oscillations by computing rudder position adjustments. **yaw.c** has six components: **YAW-DAMP**, **WOUT**, **FOLAG**, **lim**, **FCAS**, and **LINTERP**. This code was supplied by NASA-Langley; it was generated with a CASE tool and is the yaw control component of an avionics flight control system. In the set of inputs accepted by the **yaw** program, it was found that the values supplied to the variable **YAWDAMP\_STATE** in the **main()** subroutine are approximately described by a Gauss distribution, with parameters  $\mu = 0$  and  $\sigma = 2/3$ .

An experiment was conducted by randomly generating one suite of 100 test cases, where the Gauss(0, 2/3) distribution was used for the **YAWDAMP\_STATE** variable values, and another suite of 100 test cases where the inverted Gauss(0, 2/3) distribution was used for the **YAWDAMP\_STATE** variable values. The **yaw** program was instrumented using the *PiSCES Safety Net* tool, by defining a hazard condition in the **yaw.c** module. This tool implements the EPA algorithm described in [3]. The **yaw** program was then executed using the original Gauss(0, 2/3) test suite. The results from analysis performed during these runs are shown in Table 2. The total failure tolerance of the **yaw** program was estimated to be 0.435. Further investigation of the results revealed that the **lim()** procedure in the **yaw.c** module was responsible for limiting the failure tolerance of the entire program.

The **yaw** program was then executed using the inverted Gauss(0, 2/3) test suite, and the results from analysis performed during these runs are shown in Table 2. The rare-event failure tolerance estimated for the **yaw** program dropped to 0, which was again limited by the score assessed in the **lim()** procedure. If the results from instrumented lines of source code within the **lim()** procedure are examined in both test cases, we discover that for line 286, the inverted Gauss test suite resulted in a rare-event failure tolerance score of 0, while the original Gauss test suite resulted in a score of 0.714.

The differences in the scores resulting from running the two different test suites provide a good example of the benefits that can be realized by using the distribution inverter. The score of 0 returned from running the **yaw** program on the inverted test suite does not mean that this test suite has revealed safety faults in the code; however, it does call attention to places in the code which may cause hazard conditions that were not otherwise as obvious by using the expected operational distribution.

| source code   | failure tolerance<br>Gauss(0, 2/3) | failure tolerance<br>inverted<br>Gauss(0, 2/3) |
|---------------|------------------------------------|--|
| gauss_set.sfr | 0.434783                           | 0  |
| main.c        | 1                                  | 1  |
| GetDouble     | 1                                  | 1  |
| GetBool       | 1                                  | 1  |
| main          | 1                                  | 1  |
| yaw.c         | 0.434783                           | 0  |
| YAWDAMP       | 0.72                               | 0.45   |
| WOUT          | 1                                  | 1  |
| lim           | <b>0.434783</b>                    | <b>0</b>                                       |
| Line 280:     | 0.77                               | 0.51   |
| Line 282:     | 0.434783                           | 0.5306   |
| Line 286:     | <b>0.714286</b>                    | <b>0</b>                                       |
| Line 288:     | 0.454545                           | 0.5294   |
| Line 292:     | 0.981818                           | 1  |
| FCAS          | 1                                  | 1  |
| LINTERP       | 1                                  | 1  |

Table 2: Selected results for the execution of the yaw program.

### 3.2 Experiment 2: Auto-pilot

The **b737** program is a more complex piece of C source code, providing the auto-pilot controls for a B737 airplane. This code was supplied to us by NASA-Langley; it was generated with a CASE tool and has the functionality of a portion of an avionics flight control system. Given data for the inputs into the **b737** program, we were able to select distributions to approximate the expected usage for several of these input variables. These selected distributions were then inverted using the new inverting algorithms described earlier. One test suite of 100 input sets was generated using the selected operational distributions, and another test suite of 100 input sets was generated using the inverted distributions.

*PiSCES Safety Net* was applied to the program executing each of the two test suites, using identical hazard conditions specifications. Selected results from the failure tolerance analysis are displayed in Table 3 below. The original distribution test suite resulted in a high failure tolerance score of 0.95, contrasted with the low score of 0.5 resulting from the inverted test suite. Comparing the functions and line scores for the two sets of analysis results, we discover that at several points in the code, the inverted test suite reveals significantly lower failure tolerance scores than the original test suite. These low scores serve as markers for potential problem areas in the code, should the program

| source code | failure tolerance<br>original | failure tolerance<br>inverted |
|-------------|-------------------------------|-------------------------------|
| b737.sfr    | 0.95                          | 0.5                           |
| b737ansi.c  | 0.95                          | 0.5                           |
| LIMITER     | 0.96                          | 0.54                          |
| Line 117:   | 1                             | 1                             |
| Line 121:   | 1                             | 1                             |
| Line 125:   | <b>0.96</b>                   | <b>0.54</b>                   |
| Line 130:   | 1                             | 1                             |
| LONGAP      | 1                             | 0.5                           |
| Line 1813:  | 1                             | 1                             |
| Line 1820:  | 1                             | 1                             |
| Line 1830:  | 1                             | 1                             |
| Line 1834:  | 1                             | 1                             |
| Line 1845:  | 1                             | 1                             |
| Line 1851:  | <b>1</b>                      | <b>0.5</b>                    |
| Line 1856:  | 1                             | 1                             |
| Line 1857:  | 1                             | 1                             |
| Line 1859:  | 1                             | 1                             |
| Line 1860:  | 1                             | 1                             |
| CAS_ADJ     | 0.95                          | 0.6                           |
| Line 2247:  | 0.95                          | 0.6                           |
| Line 2248:  | <b>0.98</b>                   | <b>0.64</b>                   |
| mainansi.c  | 1                             | 1                             |

Table 3: Selected results for the execution of the b737 program.

encounter unexpected input values.

Failure tolerance scores of 1.0 for Line 1851 of function **LONGAP** and 0.95 for the entire **CAS\_ADJ** function, as returned from the analysis performed using the original distributions, would not cause much concern regarding those parts of the program. However, the rare-event failure tolerance scores resulting from the inverted distribution test run, 0.5 for Line 1851 and 0.6 for **CAS\_ADJ**, indicate that some additional failure-tolerant mechanisms may be needed for those places in the code.

As in the **yaw** experiment, the **b737** experiment demonstrates that running programs with inputs drawn from the rare-event space can often reveal low failure tolerance scores which would not be revealed when running those same programs with inputs drawn from the common event space. These low tolerance scores indicate places in the code which require closer inspection. If any of the portions of the program identified by a low failure tolerance score contains a fault, then it is very likely that this fault will result in a hazardous state if the program is given an unexpected input value.

## 4 Conclusions

EPA provides a *systems engineering* perspective, making it somewhat unique among software metrics. EPA assesses the “goodness” of the code with respect to how it will behave within the complete system, not in isolation. Today’s mechanical systems often contain embedded software, hence it is prudent to analyze the software with respect to the machine, not simply the software with respect to itself. In an embedded environment, software makes controlling decisions based on the information received from physical devices. It is vital that these parts be reliable enough to ensure accuracy. Reliable physical parts may be prohibitively expensive—one way to circumvent this cost might be to replace high quality parts with redundant cheaper parts. Another option for improving the tolerance caused by input data imprecisions involves inserting software mechanisms to “smooth out” the inaccuracies of the sensor data to a level that does not promote harmful outputs. This naturally has a performance cost.

This paper has taken failure tolerance measurement one step further by considering rare operating modes. But our approach is not without limitations. When you invert a profile to sample the “rarer” test cases, their “rareness” is achieved solely because of the infrequency with which they are sampled during operational use, and not because they are necessarily a small subset of  $\Delta$ . It is possible that the “rarer” test cases represent a substantial portion of  $\Delta$ . If there are more rare inputs than common inputs, then you will need to either sample often amongst them or be more selective when setting the threshold concerning what is or is not rare. Also, the value of this approach is limited by the quality of the description of the operational profile. Recognize also that there are operational distributions for which inversion is useless: a uniform distribution will invert back to itself. Distributions with large variance will be more amenable to this technique, but such distributions, once inverted, will exhibit larger rare input spaces. And larger spaces require more sampling to achieve confidence.

To date, all experimental results using this approach are small in scope and must be viewed as anecdotal. We observed that when the distributions change radically, the tolerance of a particular program to corruptions changes dramatically for some code regions, and little change in tolerance is observed for other regions. This is about what we expected. We wish we better understood what these results say, if anything, about the true reliability of software when operational profiles change. We do not have enough insight into that

problem at this time.

## Acknowledgements

This research was partially funded by US Air Force Contract F30602-95-C-0158 and DARPA Contract F30602-95-C-0282 monitored by Rome Laboratory. 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, ROME LABORATORY, OR THE U.S. GOVERNMENT.

## References

- [1] J. KNIGHT AND N.G. LEVESON. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.
- [2] J. J. MARCINIAK. *Encyclopedia of Software Engineering*. Wiley, 1994.
- [3] J. VOAS AND K. MILLER. Dynamic Testability Analysis for Assessing Fault Tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.
- [4] J. VOAS AND K. MILLER. Examining Software Quality (Fault-tolerance) Using Unlikely Inputs: Turning the Test Distribution Up-side Down. In *Proc. of Eighth Annual Conference on Computer Assurance*, pages 3–11, National Institute of Standards and Technology, Gaithersburg, MD, June 1995.