An Empirical Study of Operating Systems Errors

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{acc, junfeng, bchelf, shallem, engler}@cs.stanford.edu

Abstract
We present a study of operating system errors found by automatic, static, compiler analysis applied to the Linux and OpenBSD kernels. Our approach differs from previous studies that consider errors found by manual inspection of logs, testing, and surveys because static analysis is applied uniformly to the entire kernel source, though our approach necessarily considers a less comprehensive variety of errors than previous studies. In addition, automation allows us to track errors over multiple versions of the kernel source to estimate how long errors remain in the system before they are fixed.

We found that device drivers have error rates up to three to seven times higher than the rest of the kernel. We found that the largest quartile of functions have error rates two to six times higher than the smallest quartile. We found that the newest quartile of files have error rates up to twice that of the oldest quartile, which provides evidence that code "hardens" over time. Finally, we found that bugs remain in the Linux kernel an average of 1.8 years before being fixed.

1 Introduction

This paper examines features of operating system errors found automatically by compiler extensions. We attempt to address questions like: Do drivers account for most errors? How are bugs distributed? How long do bugs last? Do bugs cluster? How do different operating system kernels compare in terms of code quality?

We derive initial answers to these questions by examining bugs in 21 snapshots of Linux spanning seven years. We cross check these results against a recent OpenBSD snapshot. The bugs that we examine were found in previous work, which used compiler extensions to automatically find violations of system-specific rules in kernel code [8]. These bugs fall into several categories including: not releasing acquired locks, calling blocking operations with interrupts disabled, using freed memory, and dereferencing potentially null pointers.

Basing our analysis on compiler-found errors has two nice properties. First, the compiler applies a given extension uniformly across the entire kernel. This even-handed error slice allows us to do a mostly "apples-to-apples" comparison across different parts of the kernel. Likewise, we can compare two different kernels by running the same checks over both. These comparisons would be difficult to make with manual error reports because they tend to overrepresent errors where skilled developers happened to look or where bugs happened to be triggered most often. Second, automatic analysis lets us easily track errors over many versions, making it possible to apply the same analysis to trends over time.

The scope of errors used in this study, though, is limited to those found by our automatic tools. These bugs are mostly straightforward source-level errors. We do not directly track problems with performance, high-level design, user space programs, or other facets of a complete system. Whether or not our conclusions will apply to these types of issues is an open question.

The paper revolves around five central questions:

1. Where are the errors? Section 3 compares the different subsections of the kernel and shows that driver code has error rates three to seven times higher for certain types of errors than code in the rest of the kernel.
2. How are bugs distributed? Section 4 shows that the error distribution is readily matched to a logarithmic series distribution whose properties could yield some insight into how bugs are generated.
3. How long do bugs live? Section 5 calculates information about bug lifetimes across all 21 kernel snapshots and shows that the average bug lifetime for certain types of bugs is about 1.8 years.
4. How do bugs cluster? We would expect that if a function, file, or directory has one error, it is more likely that it has others. Section 6 shows that clustering tends to occur most heavily where programmer ignorance of interface or system rules combines with copy-and-paste. For the most heavily clustered error type, less than 10% of the files that were checked contained all of the errors.
5. How do operating system kernels compare? Section 7 shows that OpenBSD has a higher error rate than Linux on each of the four checkers we used to compare them. OpenBSD's error rates range from 1.2 to six times higher.

The paper is laid out as follows. Section 2 describes the kernels we check and how we gather data from them. Section 3 examines where bugs are. Section 4 discusses the distribution of error counts and matches it to a theoretical distribution. Section 5 addresses how long bugs live. Section 6 describes how bugs cluster. Section 7 compares OpenBSD and Linux. Finally, Section 8 summarizes related work.

2 Methodology

This section discusses the versions of Linux that we use for our study and the system that we use to gather our results.

2.1 Where the data comes from

Our data comes from 21 different snapshots of the Linux kernel spanning seven years. We use Linux for several reasons. First, the source code is freely available. Without this feature, a compiler-driven study could not work. Release snapshots dating back to the early nineties are readily accessible, allowing us to look for trends in time and allowing others to get these same releases to check our results. Second, Linux is widely used. As a result, relative to other systems, its code has been heavily tested, meaning that many of the bugs that are easy to find have already been removed. Finally, many programmers have developed Linux code. In aggregate, this effect should reduce the degree to which our results are skewed because of individual idiosyncrasies.

Structurally, the Linux kernel is split into 7 main sub-directories: `kernel` (main kernel), `mm` (memory management), `ipc` (inter-process communication), `arch` (architecture specific code), `net` (networking code), `fs` (file system code), and `drivers` (device drivers). Figure 1 shows the size of the code that we check across time. The size is measured in millions of lines of code (LOC), including newlines and comments. Each of the 21 different releases that we check are marked with a point. The graph ignores all parts of the kernel specific to architectures other than x86.

The graph shows several interesting features:

- The checked snapshots have grown by a factor of roughly 16 (from 105K lines at version 1.0 to 1.6 million lines in version 2.4.1).
- The bulk of the code we check comes from the drivers. At the extreme ends of the graph, versions 1.0 and 2.4.1, driver code accounts for about 70% of the code size; in the middle of the graph, this percentage drops to slightly over 50%.
- In the two years between 2.3.0 and 2.4.1 the size of the OS almost doubles, growing as much as it did in the previous 5 years. Most of this growth comes from drivers. Secondary contributors are the file systems and network code.

2.2 Measurements

Most of the graphs in this paper are built upon four different measurements. The first three are computed directly from the code, while the last is calculated from the other metrics:

Inspected errors: these were errors we manually reviewed.

Projected errors: these were unreviewed errors found by low false positive checkers.

Notes: these count the number of times a check was applied. If there are no notes there can be no errors.

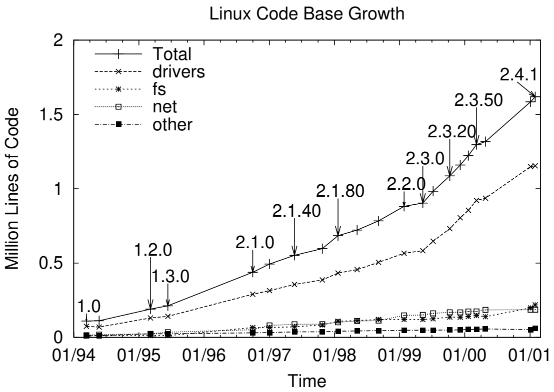

Figure 1: The size of the Linux tree that we check over time. Versions 1.1.13, 2.1.{20,60,100,120}, 2.3.{10, 30, 40}, 2.3.99-pre6, and 2.4.0 have a "+" mark but are not labelled. Most of the growth comes from drivers; secondary contributors are the file system and network code. The growth of the rest of the kernel is significantly smaller. The growth rate changes at 2.3.0 where the rate of new driver code increases.

Relative error rate: this metric is the number of errors, either inspected or projected, divided by the number of notes for that error type: $err\_rate = errors/notes$. For example, if one kernel has one error and ten notes, its average error rate will be $1/10 = 10\%$. We use this to normalize results when comparing different code bases or checkers.

2.3 Gathering the Errors

Our errors were found by the twelve system-specific checkers listed in Table 1. These come from previous work on the `xgcc` compiler [8]. Whereas this past work demonstrated the effectiveness of system-specific static analysis, it was relatively unreflective about how and why the errors arose. This paper takes the approach as a given and focuses solely on the errors.

To get the inspected errors, we manually examined the error logs produced by the checkers for a small number of kernel versions and determined which reports were bugs and which were false positives. These selected error logs were annotated with this information and propagated to all other versions. The propagation process used the inspected error logs for one kernel version to automatically annotate any errors that also appear in other, uninspected error logs. For example, for the `Null` checker, we manually inspected the errors for Linux 2.4.1. Each error report was annotated, and then the annotated results were propagated backwards through each version back to 1.0. If a bug in 2.4.1 was also reported for an earlier version, these versions automatically got the bug annotation. We did this back propagation for all bugs found in the 2.4.1 kernel. In addition to inspecting error logs ourselves, we distributed them to system implementors for external confirmation.

To get the projected errors, we ran checkers with low false positive rates over all Linux versions and treated their unexamined results as errors. We primarily use three low false positive checkers in this paper: `Var`,

Check	Nbugs	Rule checked
`Block`	206 + 87	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.
`Null`	124 + 267	Check potentially NULL pointers returned from routines.
`Var`	33 + 69	Do not allocate large stack variables ($> 1K$) on the fixed-size kernel stack.
`Inull`	69	Do not make inconsistent assumptions about whether a pointer is `NULL`.
`Range`	54	Always check bounds of array indices and loop bounds derived from user data.
`Lock`	26	Release acquired locks; do not double-acquire locks.
`Intr`	27	Restore disabled interrupts.
`Free`	17	Do not use freed memory.
`Float`	10 + 15	Do not use floating point in the kernel.
`Real`	10 + 1	Do not leak memory by updating pointers with potentially `NULL` realloc return values.
`Param`	7	Do not dereference user pointers.
`Size`	3	Allocate enough memory to hold the type for which you are allocating.

Table 1: The twelve checkers used in this paper. If the checker has few false positives, we report the number of bugs as $inspected + projected$. In total there are 1025 bugs. The top three are the primary projected checkers: we assume all potential errors reported by these checkers are real bugs. The middle set of checkers are used throughout the paper, but we only count manually inspected errors from 2.4.1 as real bugs. The bottom set of checkers are used only occasionally throughout the paper.

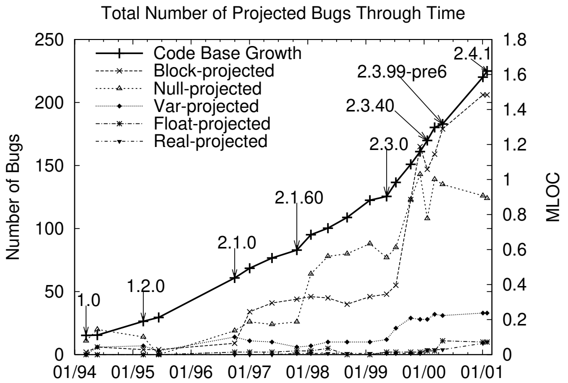

Figure 2: The absolute number of projected errors in this study. We believe 1000 is a conservative estimate of the number of unique bugs we have. The errors found by the three projected checkers are usually a function of code size, though the block checker has an unusual dip from version 2.1.60 to 2.3.0. The number of projected errors goes down at 2.3.40 for `Block` and `Null` because about 30 `Block` errors and 40 `Null` errors were fixed in that version.

`Block`, and `Null`. The `Var` checker produces almost no false positives, `Block` less than three percent, and `Null` less than ten percent. While the projected results have more noise, they are fairly representative of the inspected results.

Raw error counts alone cannot answer questions relating to error rates, which require some notion of the number of times a programmer has correctly obeyed a given restriction. Thus, we also use notes, which are emitted whenever an extension encounters an event that it checks. For example, the `Null` checker notes every call to `kmalloc` or other routines that can return `NULL`; the `Block` checker the number of critical sections it encounters, the `Free` checker the number of deallocation calls it sees, etc. Notes are the number of places a programmer could make a mistake relevant to a given check. Thus, for a given checker, dividing the number of errors by the number of notes gives the relative error rate.

Figure 2 graphs all the projected errors we use. We have approximately 1000 unique bugs in total, counting both projected and inspected errors. There are several features to note about the graph:

- The number of errors for the unsupervised checkers generally rises over time, especially after the release of version 2.3.0.
- The `Block` checker accounts for an unexpectedly large number of the errors. Many developers seem unaware of the restriction that it checks.
- The `Null` checker also accounts for a large number of errors. This is caused by careless slips, ignorance of exactly which functions might return `NULL`, and the ubiquitous use of `NULL` pointers to indicate special cases.

2.4 Scaling

A key feature of our experimental infrastructure is that it is almost completely automatic. The main manual parts are actually writing checkers and, for inspected bugs, auditing their output for a single run. Running a checker over all versions of Linux requires typing a single command. These results are then automatically entered in a database and cross-correlated with previous runs. A common pattern is inspecting errors from the most recent release and then having the system automatically calculate over all releases how long each error lasts, where it dies, how many checks were done, and the relative error rate. Further, with the exception of some axis labeling, all the graphs in this paper are generated from scripts. Thus, adding new results and even new checkers or operating systems requires very little work.

2.5 Caveats

There are several caveats to keep in mind with our results. First, while we have approximately a thousand errors, they were all found through automatic compiler analysis. It is unknown whether this set of bugs is representative of all errors. We attempt to compensate for this by (1) using results from a collection of checkers that find a variety of different types of errors and (2) comparing our results with those of manually conducted studies (§ 8).

The second caveat is that we treat bugs equally. This paper shows patterns in all bugs. An interesting improvement would be to find patterns only in important bugs. Potential future work could use more sophisticated ranking algorithms (as with Intrinsa [11]) or supplement static results with dynamic traces.

The third caveat is that we only check along very narrow axes. A potential problem is that poor quality code can masquerade as good code if it does not happen to contain the errors for which we check. We try to correct for this problem by examining bugs across time, presenting distributions, and aggregating samples. One argument against the possibility of extreme bias is that bad programmers will be consistently bad. They are not likely to produce perfectly error-free code on one axis while busily adding other types of errors. The clustering results in Section 6 provide some empirical evidence for this intuition.

A final, related, caveat is that our checks could misrepresent code quality because they are biased toward low-level bookkeeping operations. Ideally they could count the number of times an operation was eliminated, along with how often it was done correctly (as the notes do). The result of this low-level focus is that good code may fare poorly under our metrics. As a concrete example, consider several thousand lines of code structured so that it only performs two potentially failing allocations but misses a check on one. On the other hand, consider another several thousand lines of code that perform the same operation, but have 100 allocation operations that can fail, 90 of which are checked. By our metrics, the first code would have a 50% error rate, the second a 10% error rate, even though the former had an arguably better structure.

3 Where Are The Bugs?

Given the set of errors we found using the methodology of the previous section, we want to answer the following questions: Where are the errors? Do drivers actually account for most of the bugs? Can we identify certain types of functions that have higher error rates?

3.1 Drivers

Figure 3 gives a breakdown of the absolute count of inspected bugs for Linux 2.4.1. At first glance, our intuitions are confirmed: the vast majority of bugs are in drivers. This effect is especially dramatic for the `Block` and `Null` checkers. While not always as striking, this trend holds across all checkers. Drivers account for over 90% of the `Block`, `Free`, and `Intr` bugs, and over 70% of the `Lock`, `Null`, and `Var` bugs.

Since drivers account for the majority of the code (over 70% in this release), they should also have the most bugs. However, this effect is even more pronounced when we correct for code size. Figure 4 does so by plotting the ratio of the relative error rate for drivers versus the rest of the kernel using the formula:

$$err\_rate_{drivers}/err\_rate_{non-drivers}$$

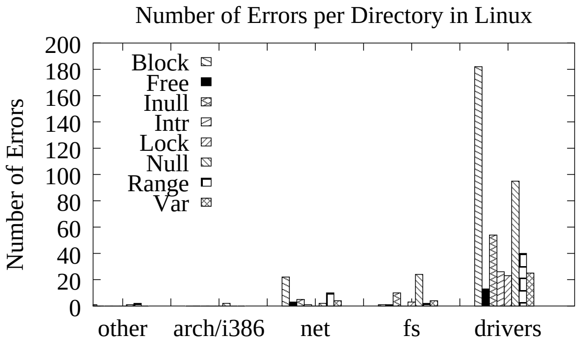

Figure 3: This graph gives the total number of bugs for each checker across each main sub-directory in Linux 2.4.1. We combine the `kernel`, `mm`, and `ipc` sub-directories because they had very few bugs. Most errors are in the driver directory, which is unsurprising since it accounts for the most code. Currently we only compile `arch/i386`. The `Float`, `Param`, `Real`, and `Size` checkers are not shown.

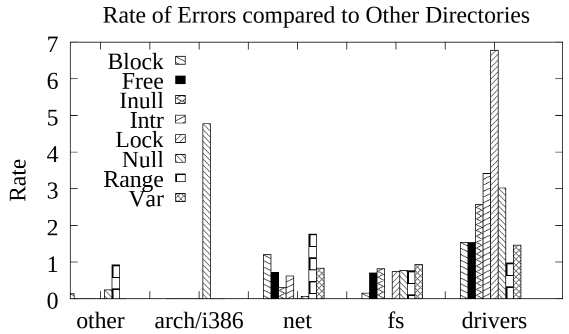

Figure 4: This graph shows drivers have an error rate up to 7 times higher than the rest of the kernel. The `arch/i386` directory has a high error rate for the `Null` checker because we found 3 identical errors in `arch/i386`, and `arch/i386` has relatively few notes.

If drivers have a relative error rate ($err\_rate_{drivers}$) identical to the rest of kernel, the above ratio will be one. If they have a lower rate, the ratio will be less than one. The actual ratio, though, is far greater than one. For four of our checkers, the error rate in driver code is almost three times greater than the rest of the kernel. The `Lock` checker is the most extreme case: the error rate for drivers is almost *seven times* higher than the error rate for the rest of the kernel.

The only checker that has a disproportionate number of bugs in a different part of the kernel is the `Null` checker. We found three identical errors in `arch/i386`, and, since there were so few notes in the `arch/i386` directory, the error rate was relatively high.

These graphs show that driver code is the most buggy, both in terms of absolute number of bugs (as we would suspect from its size) and in terms of error rate. There are a few possible explanations for these results, two of which we list here. First, drivers in Linux and other systems are developed by a wide range of programmers who tend to be more familiar with the device

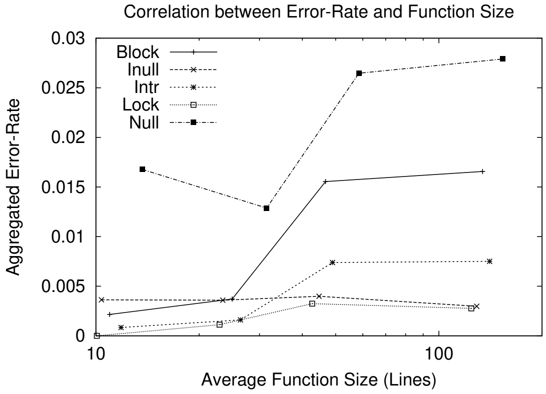

Figure 5: This graph shows the correlation between function sizes and error rates. It is drawn by sorting the functions that have notes by size, dividing them equally into four buckets, and computing the aggregated error rate per bucket for each checker. For all of the checkers except `Inull`, large functions are correlated with higher error rates.

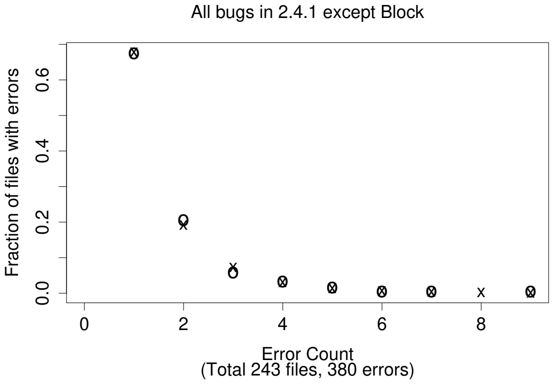

Figure 6: This graph shows a histogram of the number of files with a given number of bugs for all checkers except for `Block`. The O's are the actual data points; the X's are the theoretical log series distribution that best fits the data. The theoretical distribution has a parameter θ. For this data set, the maximum likelihood value is $\theta = 0.567$, which gives the density formula for the distribution: $Pr[X = k] = \frac{1.20 \times 0.567^k}{k}$.

rather than the OS the driver is embedded in. These developers are more likely to make mistakes using OS interfaces they do not fully understand. Second, most drivers are not as heavily tested as the rest of the kernel. Only a few sites may have a given device, whereas all sites run the kernel proper.

3.2 Large Functions

Figure 5 shows that as functions grow bigger, error rates increase for most checkers. For the `Null` checker, the largest quartile of functions had an average error rate almost twice as high as the smallest quartile, and for the `Block` checker the error rate was about six times higher for larger functions. Function size is often used as a measure of code complexity, so these results confirm our intuition that more complex code is more error-prone. Some of our most memorable experiences examining error reports were in large, highly complex functions with contorted control flow. The higher error rate for large functions makes a case for decomposition into smaller, more understandable functions.

4 How are bugs distributed?

When we report the errors found by checkers, we also provide a summary of the errors sorted by the number of errors found per file. A common pattern always emerges from these summaries: a few files have several errors in them, and a much longer tail of files have just one or two errors. In this section we consider the bugs in 2.4.1 and show that this phenomena can be described by the log series distribution [12]. Fitting a theoretical distribution is useful because it (1) compactly describes the basic characteristics of the error data we have, (2) makes quantitative, testable predictions, and (3) allows us to derive a theoretical metric of kernel-wide error clustering behavior (§ 6). The log series distribution implies these high-level properties:

1. The mode (most common value) of the number of errors per unit (i.e. files or chunks of attempts) is
1.
2. The probability of seeing a unit with E errors is a monotonically decreasing function of E. Informally, "In few files, many bugs; in many files, a few bugs."
3. The distribution is completely determined by a single parameter, θ, which can be estimated directly from the data.

4.1 The Data

Figure 6 shows the distribution of errors in files using a histogram-like representation. The "O" points are the actual data points, and the "X" points are the theoretical distribution, described in the next section. Note that over 60% of the files contain only one error, about 20% contain two, and the distribution rapidly drops off for files with three or more errors. Also note that while there are files with a large number of errors, they get sparser as the number of errors increases (the "O" points are sparser in the tail of the distribution). This implies that there will likely be several files each with a unique (and large) number of errors.

4.2 Fitting a Distribution

To fit a distribution to the graph, we start with a set of distributions to test. Each distribution has one or more parameters that change the shape of the curve. We estimate these parameters using the method of maximum likelihood, a well-established statistical technique for such estimations [12, 22]. This technique finds the value of the parameters that is most likely to give rise to the observed data. Once we have the parameters, we can determine how well the data fits the distribution using the χ^2 (Chi-squared) goodness-of-fit test [22]. The specific details of these calculations will be postponed to the next subsection. After performing the statistical analysis, we discovered that the data is best fit by

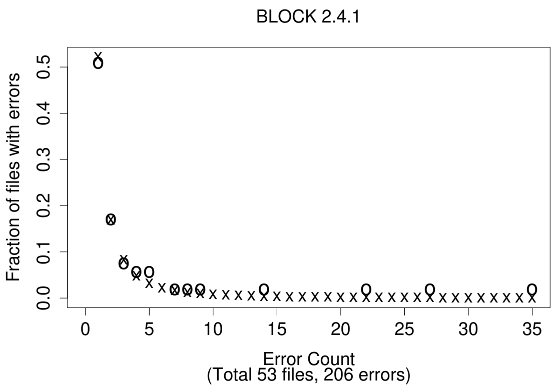

Figure 7: This graph shows a histogram of the number of files with a given number of bugs for the `Block` checker. The O's are the actual data points; the X's are the theoretical Yule distribution that best fits the data. The Yule distribution has parameter ρ which has a maximum likelihood value of $\rho = 1.09$ for this data.

a logarithmic series distribution if we omit the `Block` checker.

It is not possible to prove that a data set is drawn from a particular distribution; the data set could always be an anomaly. However, it is possible to show that a data set is *more likely* to come from a particular distribution than another one. Table 2 shows the other distributions we tried, their maximum likelihood parameters for the non-`Block` errors, and the χ^2 p-value obtained. The p-value can be roughly interpreted as the probability of seeing the data obtained if it actually came from the theoretical distribution. Standard statistical convention requires a p-value of 5% or lower before rejecting a distribution (sometimes a more stringent value of 2% is used). Only the Yule, geometric, and log series distributions are not rejected by the χ^2 test: these distributions have p-values of 23%, 24%, and 79%, respectively. We chose the log series distribution because it provided the better fit and because it is relatively easy to analyze and describe.

The `Block` checker's errors do not fit the log series distribution as well as the results from other checkers. The main cause is that too many files have a large number of `Block` errors; there is simply more clustering (§ 6) than the log series distribution predicts. We discuss the distribution of `Block` checker bugs later in this section.

4.3 The Logarithmic Series Distribution

This subsection describes the statistical methods we used for fitting and testing a distribution for the data. A logarithmic series distribution gives the probability of seeing any value k as:

$$Pr[x = k] = \frac{\alpha\theta^k}{k} \quad (1)$$

where $k > 0$ is the number of bugs, θ is the parameter for the curve ($0 < \theta < 1$), and α is a normalization constant chosen such that the probabilities will add up to 1. Once we have determined θ, we have a curve. For the bugs in 2.4.1 except for the `Block` checker, we calculated a maximum likelihood value of $\theta = 0.567$

Distribution	Parameter(s)	χ^2 p-value
Binomial	$n = 7$, $p = 0.234$	$< 2.2 \times 10^{-16}$
Geometric	$p = 0.639$	0.24
Log series	$\theta = 0.567$	0.79
Poisson	$\lambda = 0.564$	1.4×10^{-8}
Yule	$\rho = 2.57$	0.23
Zipf	$\rho = 1.46$	7.3×10^{-3}

Table 2: The distributions we attempted to fit to all of the errors we found in 2.4.1 except for `Block`. The geometric and Poisson distributions were shifted to make the first value 1 (we only model the distribution of files with at least one error). The parameter values are the maximum likelihood values (some were derived numerically). In the statistical literature a p-value of 0.05 or lower is required to "reject" a distribution. Sometimes a more stringent value of 0.02 is used. By either of these criteria, we cannot reject the log series, geometric, or Yule distributions; however, the log series gives a distinctly better fit.

	Bin 1	Bin 2	Bin 3	Bin 4
Observed	164	50	14	15
Expected	164.7	46.7	17.6	14.1

Table 3: χ^2 calculation for all errors in 2.4.1 except for `Block` bugs, using 4 bins.

using the techniques found in Johnson and Kotz [12]. Visually, the distribution appears to be a good fit for the data, but a statistical test is more precise than the human eye. If this distribution passes the χ^2 goodness-of-fit test, then we have some assurance that the data could really have come from that distribution. To apply the test, the data is partitioned into a small number of bins such that the number of errors in each bin is as equal as possible. For example, with three bins: all files with 1 error are typically in bin 1, then all files with 2 errors in bin 2, and all other files are in bin 3. These bins are picked to put a reasonable number of points in each bin (the χ^2 test usually requires that each bin has at least 5 expected errors). The χ^2 value is calculated as:

$$\chi^2 = \sum_{i=1}^{b} \frac{(O_i - E_i)^2}{E_i} \quad (2)$$

where b is the number of bins, and O_i, E_i are the number of errors observed and predicted for bin i respectively. Once this χ^2 value is calculated the probability of seeing such a data set can be looked up in a table of χ^2 values, with the "degrees of freedom" parameter equal to $b - 1$.

Table 3 shows the observed and expected number of errors in each bin, when testing the logarithmic series distribution with parameter $\theta = 0.567$. For this θ we obtained $\chi^2 = 1.05$, which corresponds to a p-value of 79%. This means that about 79% of the time a random sample that actually came from a log series distribution would be as different from the theoretical distribution as our bug sample is.

4.4 Some Properties of the Distribution

We give a useful approximation for the maximum likelihood θ [12]. Let x be the average number of errors per file, for files with errors (this implies $x >= 1$). If

$x < 25$ (our data typically has $1 < x < 2$), then θ can be approximated as:

$$\theta \approx 1 - \frac{1}{1 + [(\frac{5}{3} - \frac{1}{16}\ log\ x)(x-1) + 2]\ log\ x} \quad (3)$$

Once we have calculated θ, we can use it in various formulas that follow from the distribution. For example, we can predict the total number of files with errors of a given type, given only the number of files that contain exactly one bug (n_1):

$$F = -\frac{n_1}{\theta} log(1-\theta) \quad (4)$$

We can also predict the total number of errors found:

$$E = \frac{n_1}{1-\theta} \quad (5)$$

After calculating θ for all non-`Block` bugs in 2.4.1, we used our data set to check these approximations. Given that $n_1 = 164$ files contained one bug, the formulas predicted that there would be $F = 242$ total files with bugs, and $E = 378$ bugs in total. The actual number of files with errors was 243, and the total number of bugs was 380. Note that since we derived θ from the data, this is not a rigorous evaluation, but it does give a feel for how well the "best" curve from this distribution fits our data set.

There are also explicit formulas for the mean number of errors per file (μ) and the variance (σ^2):

$$\mu = \frac{\alpha\theta}{1-\theta} \quad (6)$$

$$\sigma^2 = \mu(\frac{1}{1-\theta} - \mu) \quad (7)$$

In Section 6 we use these equations to derive a theoretical measure of kernel-wide clustering.

The log series distribution not only fit the data for all the bugs that we found, but also fit for each individual checker's bugs separated from the others (with different θ's). However, for the `Block` checker, the Yule distribution [12] fit better than the log series distribution. We omit the details of the Yule distribution, but at a high level it is similar to the log series distribution in that it is a monotonically decreasing distribution with a single parameter. Qualitatively, the primary difference is that it has a longer tail: there is more probability of seeing large numbers of bugs in a file, which is in accord with our finding that the `Block` bugs exhibit significantly more clustering (§ 6). Figure 7 shows the distribution of `Block` bugs in 2.4.1 with the maximum likelihood Yule distribution, which had a χ^2 p-value of 99%.

4.5 Discussion

One potential pitfall with our method is that we rely on files as appropriate units for aggregating error counts. Relying on files is appealing because programmers who introduce errors also group related code into files, making them a natural unit for such aggregation. The disadvantage of using files is that they are not equal in terms of size, complexity, and especially number of chances for making an error that our checkers can find. As a result, it is possible that the distribution of file sizes or notes is influencing the distribution of bugs that we observe.

To counter this possibility of bias, we also computed the distribution of bugs over equal-sized chunks of notes. The distribution of errors over these chunks was essentially equivalent both qualitatively and with respect to fitting the theoretical distribution. To form these chunks, we order all of the notes by the full pathname of the file in which they occur. Then, we take consecutive chunks of N notes and count the number of errors among those notes. The distribution of chunks with at least one error are then plotted as described above. The choice of N is somewhat arbitrary, but one natural choice is the average number of notes per file (excluding the `Var`, `Float`, and `Inull` checkers, whose notes are somewhat misleading due to the nature of the checks), which is about 35. We tried many possible values for N, and for most of them the log series distribution fit, both qualitatively and also by passing the χ^2 test (for non-`Block` 2.4.1 errors, the p-value was 78% for chunks of size 35, almost the same as it was for files).

Clearly, more data needs to be collected on different types of systems before general conclusions can be drawn about the distribution of bugs in all systems. However, from our initial results analyzing the distribution of bugs in Linux, we believe that there is significant evidence that recognizable patterns do exist.

5 How long do bugs live?

The last two sections looked at how bugs were distributed through space. This section looks at their distribution through time by examining bug lifetimes. A lifetime spans the time a bug is introduced (born) to the time it is eliminated (killed). If we sample a system at time t, the lifetime of the bugs in the system at t are controlled by birth rate, which is determined by how many of the current bugs were born at each point in the past, and death rate, which is symmetrically determined by how many of the current bugs will die at each point in the future. In aggregate, these control how old the bugs in the system are, and how many, on average, will be killed in a given time span. We show four views of this data:

1. The lifetime of all bugs in this study. The lifetime of bugs is an indication of the effectiveness of the testing and inspection process for a system. In an ideal situation, all bugs would be fixed instantly, and bug lifetimes would be zero. In general, shorter bug lifetimes are better.
2. A back-projection of the bugs alive in the most recent release (2.4.1) showing when they were born, the birth rate, and the percentage of the bugs in 2.4.1 present in each of the past releases used in this study.
3. A magnification of all bug births and deaths for the unsupervised checkers across all releases that shows birth and death rates across many different points in time as well as the number of bugs shared by each release.
4. An estimation of bug lifetimes. This calculation is difficult because many bugs we examine are still alive. The problem is analogous to measuring the lifetime of patients in a medical study: patients enter at different times (as with our bugs) and typically some number are still alive when the study ends. We can use the Kaplan-Meier (KM) estimator [7] to estimate bug lifetime within some confidence range.

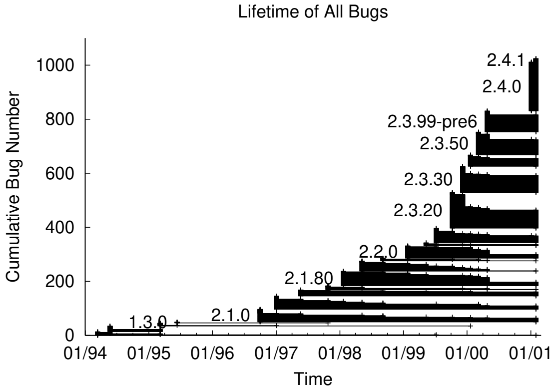

Figure 8: This graph shows the lifetimes of all bugs. Each horizontal line represents a bug's lifetime, sorted first by birth date, then by death date. Bugs found in 2.4.1 appear to 'die' at 2.4.1 but are really censored.

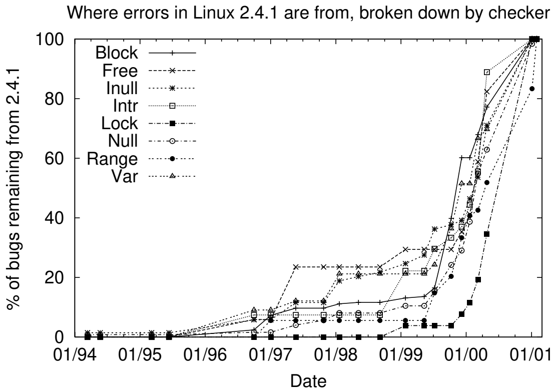

Figure 9: This graph shows the origin of 2.4.1 bugs for each bug type by calculating the percentage of bugs that came from before a given date.

5.1 A bug's life

Figure 8 gives the raw lifetime data of all bugs in this study. Each bug has its own unique horizontal line representing its lifetime. The left endpoint denotes when the bug was introduced into the kernel (born). The right denotes when the bug was fixed (died). Bugs that are still alive in the last release have an artificially truncated right endpoint, since we cannot follow them into the future. This truncation is called "censoring" in the statistical literature [7] and must be handled when explicitly calculating bug lifetimes (we return to this point at the end of the section). Left and right endpoints are truncated to version releases, since this is the granularity at which we sample the kernel. We sort the data within a release by the lifetime of the bugs so that the shortest-lived bugs are at the top of the band and the longest at the bottom. This gives a distinct "lip" to each version, which corresponds to the bugs that we only detected in that version. Notice that some bugs were detected in just one version but others lasted many years and through many versions.

5.2 Magnification: birth rates of 2.4.1 bugs

We magnify the data in Figure 8 by taking all bugs alive in 2.4.1 and following them back in time ("back propagation") to see when they were introduced. Figure 9 shows this information by plotting the percentage of 2.4.1 bugs alive at each release broken down by checker (some checkers are omitted from this graph to simplify it). The back propagation lets us compare the age of bugs in 2.4.1 across many checkers. While back propagation only shows "half" of a bug's life, it gives a feel for the shape of the birth distribution. The distributions for all of the checkers have a similar shape: a sharp dropoff followed by a somewhat longer tail.

5.3 Magnification: births and deaths through time

The previous figure focused on known, inspected bugs for a single release. For a more complete picture we would like to answer: (1) what are the birth and death rates? (2) how do these rates fluctuate over time? To answer these questions we use three of our projected checkers, `Block`, `Null`, and `Var`, to obtain errors for all releases. Because these checkers give few or no false positives, they allow us to automatically extract an accurate picture of the actual number of errors, as well as when they were born and when they died.

Figure 10 shows both the bug birth and death rates across all kernel releases. The left side of each curve's peak corresponds to the back-propagation birth-rate curve of Figure 9, while the right side shows the mortality rate (by propagating bugs forward to later releases to see how long before they are fixed).

The primary feature of the graph is that the curves have a sharp dropoff in both directions, which is similar to the back-propagation shape of 2.4.1 errors. The rapid falloff going backward in time (to the left) from any peak means that typically around 40%-60% of the bugs in a given version are introduced during the previous year; the rest are carried over from code older than a year.

For example, consider the peak on the curve labeled 2.3.20. The peak appears at version 2.3.20 because it is the only version containing all of 2.3.20's bugs. For each version both forwards and backwards we plot the number of 2.3.20's bugs that still appear in that version. The graph decreases forward in time (to the right) as more and more bugs die and backwards in time as we consider releases before a bug was introduced. The graph shows a sharp dropoff in both directions; approximately 125 errors in 2.3.20 were carried over from 2.3.0, while about 110 errors survived until 2.4.1. A similar peak appears for other versions, though usually they are not quite as sharp.

The non-centered version of the graph encodes several interesting properties. Connecting the peaks gives a curve showing the absolute number of errors over time. The graph allows one to estimate:

1. How many bugs two versions share: take the peak for version A, and follow its curve either forwards or backwards to version B's position on the x axis. The height of A's curve at that point is the number of bugs that A and B have in common.
2. The number of bugs introduced and fixed between two versions: take the peak for version A, and follow its curve to version B. The number fixed is A's peak minus the height of its curve at version

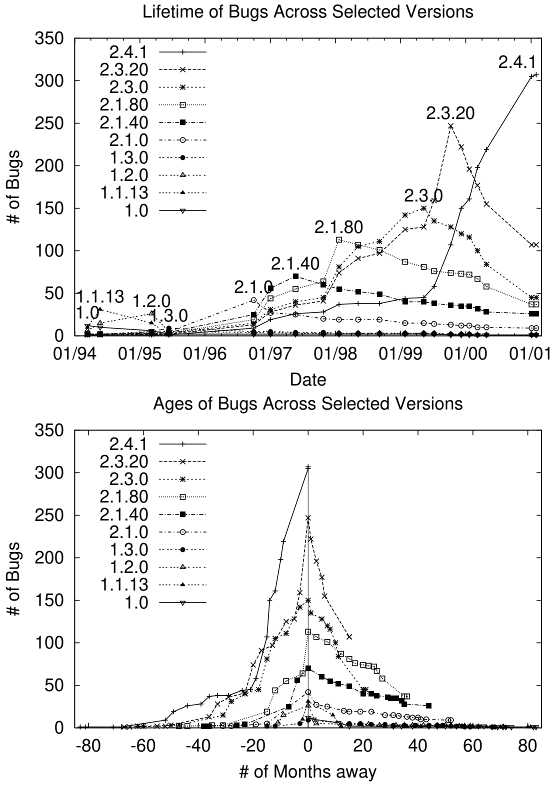

Figure 10: The upper graph shows the forward and backward lifetimes of bugs in low false positive checkers (`Block`, `Null` and `Var`) for selected major release snapshots. Symmetrically, all snapshots have sharp dropoffs both forward and backwards in time, giving further evidence that most bugs die quickly. The lower graph shows this symmetry by centering all lifetime distributions around the same point.

B. The number of bugs introduced between A and B is the height of B's peak minus A's value at B.

3. For a given version, the distribution of how old its bugs are: take the peak for version A and consider all the peaks before it in time. Follow all of their curves to version A; the relative distances between these curves indicate how many errors are from each version in the past.

5.4 Calculating average bug lifetime

Some of the most natural questions to ask about aggregated data are what its average and median values are. Extracting these from the type of data we have has three main problems: quantization error, censoring, and interference. First, the granularity of the versions we check limits our precision. Most of the versions are separated by about four months, but the gap ranges from about one month (between 2.4.0 and 2.4.1) to about one year (between 1.1.13 and 1.2.0). Another effect of this quantization is that we will completely miss bugs whose lifespan falls between the versions we check, which tends to make the average lifetime we calculate artificially long. We deal with quantization by assuming that the birth date (i.e., the first version in which

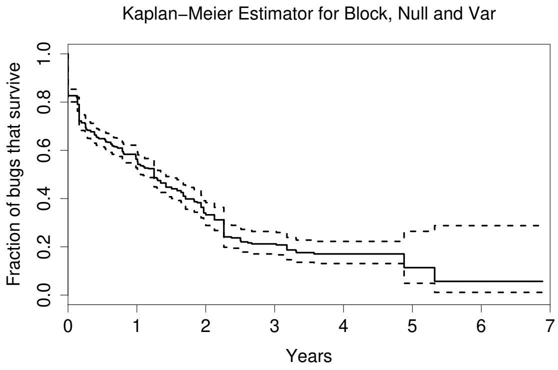

Figure 11: A Kaplan-Meier survival curve that shows the probability of a bug living longer than a given number of years. The dotted curves show the 95% confidence band. This estimator was computed using only bugs from low false positive checkers (`Block`, `Null`, and `Var`).

the bug appears) and death date (i.e., the last version in which the bug appears) are exact, which underestimates the lifetime of the bugs we find. We also ignore the bugs we don't find because their lifetimes fall between versions. Because these two effects have bias in different directions, they will partially cancel each other out.

Second, as we mentioned above, we have no exact death data for many bugs since they are still alive at 2.4.1 (i.e., right censoring). Naively averaging their observed lifespan (treating truncation as death) will significantly underestimate real lifetime. Fortunately, this is a well-studied problem with a standard statistical solution: the Kaplan-Meier (KM) estimator [7]. The Kaplan-Meier estimator gives a maximum-likelihood distribution of bug lifetimes by taking bugs censored at age X (i.e. their age at 2.4.1) to mean "lives at least as long as X." Thus, the Kaplan-Meier estimator takes both censored and uncensored data into account. For a meaningful estimate, however, some uncensored data must be available to serve as the basis for extrapolation. We use the bugs whose real birth and real death were observed in the unsupervised runs (i.e. `Block`, `Var`, and `Null`) described in the previous subsection.

Another problem with calculating lifetimes is our own interference: by providing bug logs we may have shortened overall bug lifetimes. In previous work [8] we found several hundred errors in 2.3.99 and released them to kernel developers. We cannot be sure exactly how many errors were fixed because of our error reporting, but we do have evidence that our efforts did shorten bug lifetime. In Figure 8, there is a noticeable vertical "edge" at 2.3.99, but not at most other versions. This edge corresponds to a significant number of bugs having their last appearance at 2.3.99. Specifically, 95 bugs died at 2.3.99, while only 14-76 died in each previous 2.3.x release, and only 4 died at 2.4.0. We can finesse this problem by treating bugs that die at 2.3.99 as right censored; doing this results in a longer lifetime (average 2.5 years, median 1.7 years if we censor at 2.3.99, as

Checker	Died	Censored	Mean (yr)	Median (yr)
`Block`	87	206	2.52 ± 0.15	(1.93, 2.26, -)
`Null`	267	124	1.27 ± 0.10	(0.64, 0.98, 1.01)
`Var`	69	33	1.43 ± 0.23	(0.26, 0.29, 0.79)
All	423	363	1.85 ± 0.13	(1.11, 1.25, 1.42)

Table 4: Average bug lifetimes predicted by the Kaplan-Meier estimator. **Died** is the number of bugs for which we observed the entire lifetime, while **Censored** is how many were still present in 2.4.1 and are therefore right-censored. The **Mean** is presented with the standard error (the standard deviation of random samples of this size from the true mean). The **Median** is presented as a triple (LB, M, UB) where LB is the lower 95% confidence level, M is the median, and UB is the upper 95% confidence level. An "-" means the statistic is not derivable from the data we have.

opposed to average 1.8 years, median 1.25 years if we censor at 2.4.1). However, this estimate will err on the long side since an unknown number of those 95 bugs may have been removed without our interference. We do not consider the problem of interference further in the rest of this section.

Finally, any calculation of bug lifetimes should take into account the nature and purpose of development during the period measured. Traditionally the odd releases (1.3.x, 2.1.x, 2.3.x) are development versions that incorporate new features and fix bugs, whereas the even versions (1.2.x, 2.2.x, 2.4.x) are more stable release versions, with most changes being bug fixes (though there are exceptions). Development on odd versions proceeds in parallel with the stabilization of even versions. We have chosen to model mostly the development path, picking up many minor releases of odd versions but only the major releases of even versions (except for 2.4.1). This choice also allows us to linearly order the versions in time, which we could not have done if we also examined the minor releases of even versions.

Given a set of data points representing bug lifetimes, the Kaplan-Meier (KM) estimator derives a maximum-likelihood *survivor function*, which is defined as follows. Let X be a random variable representing the lifetime of a bug. The survivor function $F_X(t)$ gives the probability that a bug lives at least as long as t:

$$F_X(t) = Pr[X >= t] = \prod_{i=0}^{t} (1 - \frac{d_i}{r_i}) \qquad (8)$$

In this function, d_i is the number of bugs that die at time i, and r_i is the number of bugs still alive at time i (including censored bugs).

Figure 11 shows the KM survival curve for all of the low false positive checkers combined. The two dotted curves surrounding the curve show the 95% confidence band. Notice that this band grows rapidly as the bug lifetime increases. This increase is due to the small number of bugs in our data that have extremely long lifetimes. For each possible bug lifetime up to the maximum lifetime observed, the KM curve gives us the probability that a random bug will live that long. For example, this curve shows us that there is approximately a 50-60% chance that a bug will last for over one year, but that percentage drops to 30-40% for lifetimes over two years. Notice that the upper-left hand corner of the curve does not begin at 1.0 because of our sampling granularity. There are a significant number of errors that only show up in a single version, and we count these as having a lifetime of zero.

The graph can also be used to estimate the maximum error birth rate that can be tolerated without increasing the total number of bugs. If the birth rate exceeds the death rate then kernel will tend to accrue errors over time, and if the converse holds then errors will gradually be purged. Assuming KM is a good approximation of the errors we did not manually inspect, then the figure can also be used as a crude metric for how well the system is tested and audited: a sharp downward slope indicates a rapid rate of bug fixing; a more horizontal slope indicates that fewer bugs are being removed in a given time, with the implication that bug lifetimes will be longer.

We were surprised to find that errors tend to live quite a long time before being extinguished. The average bug lifetime we calculated is around 1.8 years, with the median around 1.25 years. Table 4 breaks this number down by checker. Note the much shorter median lifetime for `Var` compared with `Block` and `Null`. This perhaps indicates the relative importance of these errors to kernel developers, or could be due to the fact that given one `Var` bug it is easy to find all other bugs that use the same type using a string search. On the other hand, the relatively long average lifetime for `Var` may mean that once the easy errors were removed, the rest were more difficult to find, perhaps because they didn't cause many stack overflows in practice.

Average Age. If we compute the average age of bugs in any one particular version, it is always relatively small. For example in 2.4.1 the average age of a bug is about 1 year. How does this mesh with our estimate of 1.8 years for the average bug lifetime? A simple analogy with human lifetimes clarifies the situation. Suppose an average person has a life expectancy of 77 years. If at any time we take a sample of the ages of people and find the mean, it will always be far less than 77 years. Every sample point in a given version is right-censored, so we will always get a much shorter average age than the average lifetime.

Code Hardening. A widely held systems belief is that as code ages, it becomes less buggy, presumably because testing and inspection will gradually weed out more errors. In order to test this hypothesis, we calculated the average number of bugs in each file as a function of file age. We ordered all files in 2.4.1 that have notes by their ages, equally divided them into four buckets and computed the aggregated error rate per bucket for each checker. Figure 12 shows that older files tend to have lower error rates. For all of the checkers shown, the newest quartile of files have an average error rate about twice as high as the oldest quartile of files.

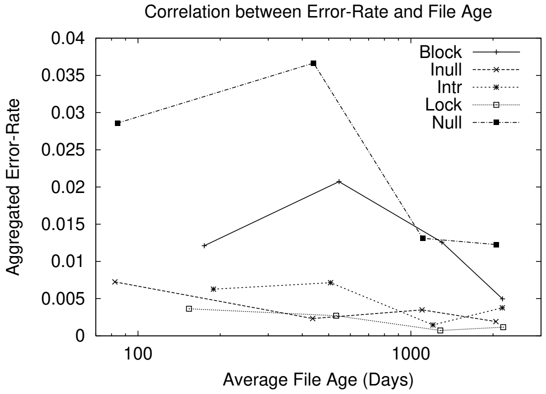

Figure 12: Older files tend to have lower error rates.

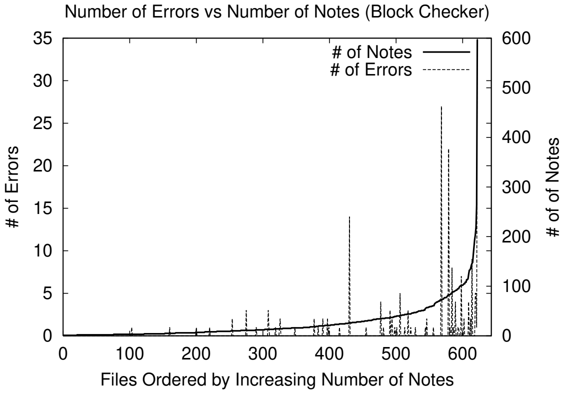

Figure 13: A view of the `Block` errors found in the entire Linux 2.4.1 kernel. Files are numbered by the number of notes and placed in sorted order on the x-axis. For random coin tosses one would expect the number of errors to be a relatively stable fraction of the number of notes. The absolute number of notes in each file is plotted on the right y-axis, while the absolute number of errors in each file is plotted on the left y-axis.

6 How do bugs cluster?

Many errors are not independent: if a type of bug appears once in a function, file, or directory, another bug is much more likely to appear within that unit than in another random unit. Two reasons for dependence are incompetence and ignorance. As programmer competence degrades, error rates increase. Similarly, if a programmer is ignorant of system restrictions, he/she will tend to make more errors than a programmer that is not. Thus, for large systems with many programmers, dependent errors will cause error "clustering," where parts of the OS have much higher error rates than the global error rate. This section:

1. Graphically shows clustering in Linux, both in raw form and compared to the clustering that would be measured from random events.
2. Presents and applies an intuitive measurement of clustering for subsystems. This measurement can be used to rank subsystems for auditing: subsystems with strong clustering on one rule will reasonably have clustering on others and be the most profitable candidates for auditing.
3. Presents and applies a global measure of clustering that can be used to compare the system-wide clustering of different types of errors. If a particular type of error exhibits system-wide clustering, then programmers are probably ignorant of the particular rule or interface involved. This implies that the rule or interface should be abolished or programmers should be educated to prevent future errors.

6.1 Views of clustering

Figures 13 shows a graphical demonstration of clustering for the `Block` checker. We sort the files by the number of notes and then plot the number of errors in each file. For a random distribution, we would expect the number of errors to be a relatively stable fraction of the number of notes with few deviations far above the expected curve. Instead, we see several unlikely spikes and a relatively weak correlation with the number of notes, which together indicate clustering.

Another way to look at clustering is to consider the skew caused by clustered bugs. We use "skew" to describe a situation where a small portion of the population (e.g., files, directories, and notes) accounts for a large portion of the interesting cases. Figure 14 shows how clustering can be viewed as skew in a distribution by considering the `Block` checker. On the x-axis we place the files ordered by number of bugs, with the files with the most bugs on the far left. On the y-axis we plot how many bugs remain in files to the right of the point. For the `Block` checker (the far left curve), approximately 80% of the total errors are accounted for by 50% of the files that contain errors (point A). (Note that all errors from the `Block` checker occur in 8% of all files, so 4% of the total files account for 50% of the errors from the `Block` checker.)

Of course, even if we threw bugs into the source at random, there would be some "clustering" since the errors would sometimes fall together even though they are really independently placed. Thus, for comparison, we plot the results of a random experiment run 1,000 times over the distribution of notes and plot the maximum and minimum results (the two far right curves). Two things are immediately apparent: the random experiment predicts that 60%-100%+ more files will have errors than actually do, and, unlike the observed results, 80% of the errors would be accounted for by between 75% (point B) and 80% (point C) of the files.

6.2 Clustering in Subsystems

In this subsection we calculate approximately how likely an observed cluster is due to chance using *Chernoff bounds* for independent Poisson trials [19]. This metric is useful for ranking subsystems for auditing, but it does not provide a whole-kernel metric; we present a global metric in the next section.

Suppose we model each call to a function that can return a NULL pointer as a coin flip, and with some probability P_{bug} the caller forgets to check the return value along some path that later uses that pointer. Tak-

DIR	Notes	NBugs	Expect	P
Block Checker				
net/atm	152	22	1.94	3.1×10^{-15}
drivers/i2o	692	35	8.81	2.6×10^{-10}
drivers/isdn/hisax	956	41	12.17	7.9×10^{-10}
drivers/usb/serial	94	9	1.20	3.2×10^{-5}
drivers/isdn/icn	86	8	1.10	1.2×10^{-4}
drivers/net/pcmcia	240	12	3.06	5.7×10^{-4}
drivers/net/wan/lmc	146	9	1.86	8.7×10^{-4}
drivers/isdn/act2000	24	3	0.31	1.6×10^{-2}
drivers/atm	317	6	4.04	6.6×10^{-1}
Null Checker				
fs/udf	88	11	1.97	5.0×10^{-5}
drivers/net/tokenring	22	6	0.49	7.5×10^{-5}
drivers/char/drm	69	9	1.54	2.2×10^{-4}
drivers/net/pcmcia	39	6	0.87	1.6×10^{-3}
drivers/pcmcia	28	5	0.63	2.4×10^{-3}
drivers/scsi	253	13	5.65	3.1×10^{-2}
drivers/char/rio	19	3	0.42	3.7×10^{-2}
drivers/telephony	7	2	0.16	3.9×10^{-2}
drivers/ide	56	5	1.25	4.2×10^{-2}

Table 5: The number of notes and bugs found in Linux 2.4.1 for the top 9 most clustered leaf directories. **Expect** is the expected number of errors that we would find if errors were distributed as coin flips, i.e. **Expect** $=$ **Notes** $* P_{bug}$, where P_{bug} is the total number of bugs found divided by the total number of notes for the entire kernel for that checker. There were 206 errors / 16176 notes for `Block`, and 124 errors / 5550 notes for `Null`. We use the Chernoff bound for independent Poisson trials to calculate **P**, an upper bound on the probability that the number of bugs would exceed the expected number by as much as we observed. As expected, errors for `Block` (a rule that programmers seem relatively ignorant of) cluster significantly more than `Null` errors.

ing P_{bug} to be the measured average rate of such errors over the entire Linux kernel, sometimes a single function or directory will have no errors, sometimes one error, and sometimes many.

To test this model, we compute the likelihood that randomly distributed errors cluster in the same way as our observations for each directory in the Linux 2.4.1 distribution. Table 5 lists leaf directories with significantly more bugs than the expected number for the `Block` and `Null` checkers. We apply the Chernoff bound for Poisson trials to bound the probability P that a directory has many more bugs N than the expected number of errors E:

$$P = Pr[N > (1+\delta)E] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{E} \qquad (9)$$

Here N is the number of bugs in the directory, E the expected number of errors for the directory, and δ is the fraction by which the actual number of bugs exceeds the expected number. If clustering did not exist, then the probability P for each directory would be expected to have reasonably large values (close to 1), especially since the number of directories (samples) is relatively small. However, the extremely low bounds are strong evidence that the errors are not randomly distributed and that clustering exists. At the most extreme, the likelihood of seeing the results found in the `net/atm` directory is less than 3.1 out of 1,000,000,000,000,000! This clustering turned out to be caused primarily by one file that called a blocking function 22 times with a spin lock held. Another example was a single file in the `drivers/i2o` directory. Within this file, 34 of the errors were caused by cut-and-paste: one of the errors was copied in 10 places and another in 24!

6.3 A Global Clustering Metric

While equation 9 gives a simple way to rank clustering in subsystems by how much they deviate from a coin flip process, it does not provide a single summary metric for how much clustering exists in the entire kernel. The reason is that the bound depends on the global error rate being different from the error rate in particular subsystems; when applied to the whole kernel this difference in error rates disappears and the bound becomes meaningless. For example, it cannot be used to compare how errors from different checkers cluster across the entire kernel.

We propose a simple metric that provides a global quantitative measure of clustering. To provide some intuition for our metric, consider the possible arrangements of four errors in four files, shown in Table 6. The value of this metric can be seen from this example: given a system with a number of known errors, it provides a quantitative way of describing how closely the errors are grouped together.

Intuitively, the first row has the least clustering, and the last row has the most. A natural way to formalize this intuition is to note that clustering is an aggregate measure of how far the distribution of errors deviates from the average for each file.

Consider a system with N units (files, for example). Let $E = \{e_1, e_2, ...e_N\}$ denote the number of errors

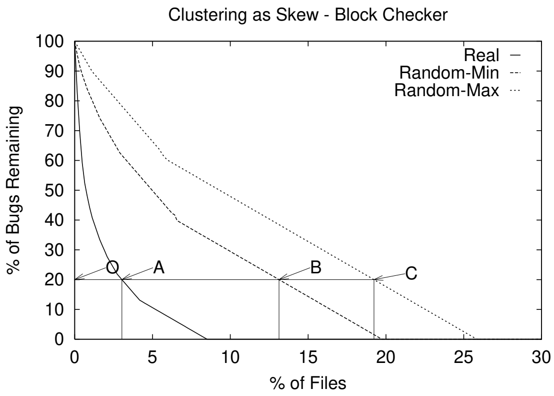

Figure 14: A plot of clustering as skew: 50% of the files account for 80% of the errors. The maximum and the minimum are shown by the curves `Random-Max` and `Random-Min`. The curve for `Real` represents the observed results. If the bugs were thrown into files randomly, we would expect that the curve for `Real` would fall somewhere between `Random-Max` and `Random-Min`.

Arrangement				**Clustering**
e1	e2	e3	e4	c
x	x	x	x	0
xx	x	x		0.5
xx	xx			1
xxx	x			1.5
xxxx				3

Table 6: The possible arrangements of four errors in four files and the clustering value assigned to the arrangement by the clustering metric. Permutations of these arrangements result in identical amounts of clustering and are not shown.

found in each unit. Let $\mu = \frac{1}{N}\sum_{i=1}^{N} e_i$ be the average number of errors per unit. Then our clustering metric c is defined as:

$$c = \frac{\sum_{i=1}^{N}(e_i - \mu)^2}{\sum_{i=1}^{N} e_i} \tag{10}$$

The clustering formula is very similar to the formula for the variance of the e_i's, except that the denominator is the total number of errors, not the total number of units. The problem with directly using a measure such as standard deviation or variance as the metric for clustering is that the number of units is rather arbitrary. Using either of those metrics, it would be difficult to compare two different clustering values if their number of units were different. However, notice that the variance, σ^2 is related to c by:

$$c = \frac{\sigma^2}{\mu} \tag{11}$$

Thus our clustering metric is simply the ratio of the variance to the mean. In the statistical literature, this is called the dispersion index. It is often used in biological applications to measure the clustering tendency of populations, and it has also been used in modeling the burstiness of network traffic [16]. The main feature of c is that it predicts if the distribution is clustered as much as random ($c = 1$), less clustering than random ($c < 1$), or more clustered than random ($c > 1$), where "random" is taken to mean a Poisson distribution. If $c = 0$, then the distribution is uniform and there is no clustering at all.

Table 7 shows the value of c for Linux version 2.4.1 broken down by checker (for now, only the value of c matters; $c_{theoretical}$ will be explained in the next section). The clustering metric is computed twice: once by using chunks of notes as the units (as described in § 4) and once by using files as the units. Using chunks is a more accurate measure of clustering because each chunk has an equal number of notes, and each note represents an opportunity for an error. In contrast, using files ignores the fact that different files will contain a different number of notes. In the remainder of this section we focus on the clustering results for chunks.

The most striking thing about the clustering data is that, with the exception of `Block`, all of the checkers show clustering less than random. For the `Param`, and `Size` checkers, this is easy to explain: there are simply too few bugs to show much clustering. For chunks, the `Free`, `Intr`, and `Lock` checkers show clustering near uniform (i.e. almost no clustering at all). We hypothesize that this is due to the level of understanding of the programmers. For example, anyone who calls `cli` to disable interrupts should know the meaning of disabling interrupts and therefore should know not to leave them disabled. By using part of this interface, the programmer demonstrates knowledge of what the interface does and knowledge of how to use it. As a result, the bugs that we find are mostly isolated mistakes.

The `Inull` and `Null` checkers show higher clustering values, though they still cluster less than random. For the `Null` checker, some programmers might not have known all of the functions that can return `NULL`. For the `Inull` checker, we believe the clustering was largely caused by cut-and-paste of incorrect code.

Clustering values greater than one potentially demonstrate that the programmer did not understand what they were doing in the places with all the bugs. Only the `Block` checker falls into this category. It is very likely that a programmer does not realize that he cannot call a blocking function when interrupts are disabled or a spin lock is held. Or perhaps he knows this rule, but fails to know every single function that can block. In either case, the programmer's understanding of the interface is incorrect, and he is therefore more likely to make the same mistake repeatedly.

It is important to note that the clustering values are calculated without including the chunks that have no bugs. In other words, these numbers only represent the clustering among chunks that have bugs. If we were to include all chunks, these numbers would be higher, so we use them as an underestimate of the clustering. We ignore the zero-bug chunks so that we can compare these numbers to the theoretical distribution that we calculated in Section 4.

We can relate the above clustering metric to the distribution of errors in the following way. As we have seen, by fitting a logarithmic series curve to the data, we obtain a value for θ that in turn determines a theoretical approximation for the variance σ^2 (Equation 7). Dividing the approximation by μ derives an approximation for c:

$$c_{theoretical} = \frac{\sigma^2}{\mu} = \frac{\mu(\frac{1}{1-\theta} - \mu)}{\mu} = \frac{1-\alpha\theta}{1-\theta} \tag{12}$$

where $\alpha = -[log(1-\theta)]^{-1}$. Using the log series distribution, we can not only derive good approximations for the mean and standard deviation, we can also use the value of θ to estimate the clustering of that data. Table 7 shows some values of $c_{theoretical}$ for comparison with the actual clustering values. The closer the distribution of errors follows the log series distribution, the more likely the theoretical clustering will be close to the actual clustering. For example, for all of the checkers except for `Block` (the distribution shown in Figure 6) the actual clustering value by file is 0.766, while the theoretical value is 0.755. Since the theoretical clustering value in this case is derived from θ, which was in turn derived from the data, this cannot be used to judge the predictive value of Equation 12; however it does give a sense for how well the "best" estimate of the clustering according to the log series distribution matches the actual value.

6.4 Discussion

So far we have not discussed how clustering comes about. One simple hypothesis is that programmers have different abilities and that poor programmers are more likely to produce many errors in a single place. However, from our experience, this is probably only the second most important cause of clustering. The most important cause is probably ignorance: many programmers appear to be ignorant of the relevant system rules, and they produce highly concentrated clusters of errors without even being aware of it. In addition to ignorance, the prevalence of cut-and-paste error clustering among different device drivers and versions suggests that programmers believe that "working" code is correct code. Unfortunately, if the copied code is incorrect, or it is placed into a context it was not intended for, the assumption of goodness is violated. Finally, some code is simply not executed as often, making it less well tested and therefore more likely to contain clusters of errors.

7 Initial Cross-Validation with OpenBSD

If Linux is simply a "bad" system, then studying its errors would not be particularly useful. To provide an initial examination of this possibility, we compare recent Linux (2.4.1) and OpenBSD (2.8) releases using four checkers: `Intr`, `Free`, `Null`, and `Param`. We used older versions of these checkers than was used in the rest of this paper, but we controlled for checker variation by using identical checker versions for both kernels. The only difference between them was the text file used to specify the names of the routines to check (e.g., what routines disable interrupts, free memory, return potentially NULL pointers, or manipulate user pointers). As such, these checkers provide a roughly uniform comparison in that they will be vulnerable to the same types of false positives and miss the same kinds of errors.

OpenBSD had far a factor of 2-4 fewer checked locations (notes) but a higher error rate for the four checkers we compared. The closest result was for the `Null` checker. Here OpenBSD had an error rate of 2.148% (1 error in 50 possibly failing calls), which was roughly 20% worse than Linux's rate of 1.786%. The `Intr` checker was next (.617% versus .465%). The `Free` checker had an error rate two times worse than Linux (one call site out of 200 was incorrect) whereas the `Param` checker, arguably the most important, was almost a factor of six worse.

In fact, these numbers may be biased towards understating the difference in the kernels. The OpenBSD errors were all hand verified by an OpenBSD implementor (to the point that many were fixed and checked into the main kernel tree during this diagnosis). In contrast, the bugs found in Linux were diagnosed by us and are, therefore, more likely to be over-reported.

This being said, the numbers in Table 8 do not give a full picture of code quality. Two generic problems are that (1) they are only for a limited number of checkers and (2), as discussed in Section 2, the checkers only examine low-level operations, and thus give no direct measurement of design quality. More specific to the actual measurements, part of OpenBSD's high error rate comes from a very small number of files that see little to no use on most sites. This skew was especially true for the errors found by the `Param` checker that mostly resided in the "System 4" compatibility layer, which sees limited use. The checkers found significantly fewer errors in the rest of the kernel.

	Chunks		**Files**	
Checker	c	c_{theor}	c	c_{theor}
`Block`	3.26	2.51	10.92	6.03
`Free`	0.0551	0.0651	0.220	0.145
`INull`	0.778	0.595	0.224	0.238
`Intr`	0.217	0.259	0.0356	0.0394
`Lock`	0.0705	0.0879	0.0369	0.0411
`Null`	0.770	1.51	0.394	0.420
`Param`	0.167	0.644	0.457	0.495
`Realloc`	-	-	0.533	0.912
`Size`	0.167	0.644	0.167	0.644
All ex. `Block`	0.385	0.455	0.766	0.755
All	1.61	0.958	5.80	1.75

Table 7: This table shows the clustering values, c, for bugs in 2.4.1 computed by chunks (35 notes / chunk) and by file. Values $c < 1$ imply a more uniform distribution, $c = 1$ a random distribution (frequencies follow a Poisson distribution), and $c > 1$ indicate more clustering than random. The values of c_{theor} are computed from equation 12 using the maximum likelihood value of θ for each checker. Considering chunks, `Block` bugs cluster more than any other type, with `Null` a distant second. The `Float`, `Range`, and `Var` checkers were omitted because their notes do not always correspond well to the number of times the rule was checked.

8 Related Work

Numerous projects have used static analysis to find errors [1, 4, 11, 25]. While these indirectly contrast different code bases, they primarily focus on the machinery and methods used to find the errors. In contrast, we assume some way of automatically getting errors and concentrate on the errors themselves.

System reliability studies have focused on: (1) in-

	Percentage			Bugs		Notes	
Checker	**Linux**	**OpenBSD**	**Ratio**	**Linux**	**OpenBSD**	**Linux**	**OpenBSD**
`Null`	1.786%	2.148%	1.203	120	27	6718	1257
`Intr`	0.465%	0.617%	1.328	27	22	5810	3566
`Free`	0.297%	0.596%	2.006	14	13	4716	2183
`Param`	0.183%	1.094%	5.964	9	18	4905	1645

Table 8: Comparison of the most recent shipping versions of Linux (2.4.1) and OpenBSD (2.8) on four checkers. For these checkers, OpenBSD is always worse than Linux, ranging from about 20% worse to almost a factor of six.

spection of error logs, (2) analysis of system behavior under fault injection, and (3) testing. We consider each below.

While there have been many performance studies of operating systems [5, 21, 20], there have been relatively few that look at code quality from within the OS community. Most defect studies come from the software engineering or fault-tolerant fields, and almost all are based on data gathered from post-mortem inspection of error logs or "defect reports," typically for high availability systems.

These studies are largely complementary to our work. Their focus naturally leads to different questions than those we consider, such as (1) what causes faults, (2) what their effects are, (3) if they can be predicted, and (4) how well the system (mis)handled them. Further, their error populations and ours have different implications. They have two main advantages: (1) they only contain realistically exploitable bugs (we treat all bugs equally) and (2) their end-to-end checks can find higher level or deeper errors than our checkers. However, they also have limitations that we do not. The most important is that they are restricted to errors that were detected and diagnosed with testing or field use. Because their errors are biased towards modules and paths that workloads happened to exercise, they can give a potentially misleading view of error properties such as bug distributions (one of their main focuses). In contrast, we do not suffer these biases since our checkers can detect all errors of a certain class on all paths, regardless of whether a particular workload triggered them. We consider a representative sample of these studies below.

Gray surveyed outages in Tandem systems between 1985 and 1990, using manually gathered bug reports to classify the causes of outages [10]. In a subsequent study, Lee and Iyer [15] looked at 200 memory dumps of field software failures in the Tandem GUARDIAN 90 operating system collected over 1 year. They focused on the effectiveness of fault detection and recovery, and classifying errors by type (uninitialized variables, race conditions).

Sullivant and Chillarege [23] examined MVS operating system failures, classifying error causes and manifestations. They randomly sampled 250 reports (out of a population of 3000) gathered over a five year period. Their main focus was on measuring errors caused by memory corruption versus "everything else." They found that the former generated the highest number of reported system crashes. They also measured how often and why bug fixes introduced other bugs. They did similar study for databases [24].

More recently, Xu et al. [26] used reboot logs to measure the dependability of a 503 node Windows NT cluster over 4 months (and 2,127 reboots). They classified the causes of failures (hardware, software), time to recover, and availability measurements. An interesting result is that reboots occur in bursts, which is similar to our finding that errors cluster in source code.

While the studies above largely ignore the questions we address in this paper, the following two are closer.

Fenton and Ohlsson [9] examined faults in two consecutive versions of a telecommunication switching system. They found strong support for the "Pareto principle," the hypothesis that a small number of modules accounts for a large fraction of the faults, which is essentially what we call skew. Specifically, they found that 20% of the modules account for 60% of the faults discovered during testing. In contrast, for Linux 2.4.1, about 11% of the files accounted for all of the errors we found with automatic checkers. While our numbers are different (keep in mind we take files instead of modules as our unit), the basic point of the principle seems to be supported by our results. They also found that an even smaller portion of the modules (10%) account for almost all of the operational failures.

Basili and Perricone [3] report on a manually conducted study of satellite planning software consisting of 90k LOC spread across 370 modules (functions). They found that reusing modules for a new purpose decreased initial development cost, but more effort was required to correct errors in them. Thus there was a tradeoff between the cost of initial development time and the cost of adapting modules to a new specification.

A related area are fault injection studies [2, 14], which dynamically insert bugs into the system to see how it crashes or survives [6]. These focus mostly on robustness in the face of artificial errors, whereas we are interested more in the features of actual errors.

Another approach is explicit testing, such as the "fuzz" studies that compare how a set of systems utilities behaved in the face of random inputs [17, 18].

In terms of examining bugs found with automatic techniques, the closest work compared to us is a study by Koopman et al. that used randomized testing to measure the effectiveness of error handling code on 13 different POSIX implementations [13]. They measured both the types of errors that resulted (silent, machine crash, caught, caught but misreported) as well as the overlap of errors ("diversity") of the different operating systems. Their study, like ours, has a measure of objectivity in that it applies its tests uniformly across a set of code rather than being biased by what locations programmers have decided to examine. Their study has the advantage that it spans many operating systems.

However, they focus only on error handling, rather than general rules. They look at a limited portion of the OS (e.g., not device drivers, or much of the auxiliary code). Finally, their study does not address error behavior over time.

9 Conclusion

This paper uses roughly 1000 unique, automatically detected operating system errors to test and spot patterns in kernel code such as the relative error rate of drivers as compared to other kernel code (up to a factor of ten worse), how errors cluster (roughly a factor of two more tightly than from a random distribution) and how long bugs last (an average of about 1.8 years). We gathered data from seven years of Linux releases. We countered checker specific artifacts by using twelve automatic checkers, which found errors more objectively than manual inspection could hope to. We view these as promising, but initial results. We hope that other researchers will find the results of this study useful for understanding the nature of errors in systems code.

10 Acknowledgements

Peter Glynn and Peter Salzman suggested discrete distributions, maximum likelihood techniques, the χ^2 test, and the Kaplan Meier estimator. The consulting staff at Stanford's Depart of Statistics were also helpful in guiding our statistical analysis. Diane Tang's critical reading greatly improved the presentation, clarity, and structure of this paper. We thank Ken Ashcraft and Evan Parker for contributing the `Range` and `Inull` checkers and inspecting the results. We thank the readers of `linux-kernel` for their generous feedback and support. Alan Cox verified and fixed many of the errors we reported to `linux-kernel`. Costa Sapuntzakis inspected error logs for OpenBSD and committed fixes to the CVS repository. Linus Torvalds inspected a near-final draft and provided helpful comments on clustering. We thank our shepherd, Mike Jones, for his careful reading and valuable feedback. This work was supported by NSF award 0086160 and by DARPA contract MDA904-98-C-A933.

References

[1] A. Aiken, M. Fahndrich, and Z. Su. Detecting Races in Relay Ladder Logic Programs. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 1998.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation – A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[3] V. Basili and B. Perricone. Software Errors and Complexity: an Empirical Investigation. *Communications of the Association for Computing Machinery*, 27(1):42–52, 1984.

[4] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing systems*, pages 131–152, Spring 1996.

[5] B. Chen, Y. Endo, K. Chan, D. Mazires, A. Dias, M. Smith, and M. Seltzer. The Measured Performance of Personal Computer Operating Systems. *ACM Transactions on Computer Systems(TOCS)*, February 1996.

[6] R. Chillarege and N. Bowen. Understanding Large System Failures - A Fault Injection Experiment. In *The 19th International Symposium on Fault Tolerant Computing*, June 1989.

[7] D.R. Cox and D. Oakes. *Analysis of Survival Data*. Chapman and Hall, London, UK, 1984.

[8] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[9] N. E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[10] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Software Engineering*, 39(4), October 1990.

[11] Intrinsa. A Technical Introduction to PREfix/Enterprise. Technical report, Intrinsa Corporation, 1998.

[12] N.L. Johnson and S. Kotz. *Discrete Distributions*. John Wiley & Sons, New York, NY, 1969.

[13] Philip J. Koopman Jr., John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. Comparing Operating Systems Using Robustness Benchmarks. In *Symposium on Reliable Distributed Systems*, pages 72–79, 1997.

[14] A. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Software Engineering*, 44(2), February 1995.

[15] I. Lee, R. Iyer, and F. Symptoms. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In *International Symposium on Fault-Tolerant Computing (FTCS)*, 1993.

[16] W.E. Leland, M.S. Taqq, W. Willinger, and D.V. Wilson. On the Self-Similar Nature of Ethernet Traffic. In *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[17] B.P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[18] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CS-TR-1995-1268, University of Wisconsin, 1995.

[19] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.

[20] S.E. Perl and R.L. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *Operating Systems Design and Implementation*, pages 169–183, 1996.

[21] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Symposium on Operating Systems Principles*, pages 285–298, 1995.

[22] S.D. Silvey. *Statistical Inference*. Chapman and Hall, London, UK, 1975.

[23] M. Sullivan and R. Chillarege. Software Defects and Their Impact on System 118 Availability – A Study of Field Failures in Operating Systems. In *21st International Symposium on Fault Tolerant Computing*, June 1991.

[24] M. Sullivan and R. Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *22nd International Symposium on Fault-Tolerant Computing*, July 1992.

[25] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, February 2000.

[26] Z. Xu, R. Kalbarczyk, and Iyer. Networked Windows NT System Filed Failure Data Analysis. In *Proc. of Pacific Rim International Symposium on Dependable Computing*, 1999.