An Empirical Study of Operating Systems Errors

Jing Huang
Background

- **previous research**
  manual inspection of logs, testing, and surveys because static analysis is applied uniformly to the entire kernel source

- **This research**
  automatic, static, compiler analysis applied to the Linux and OpenBSD kernels
  less comprehensive variety of errors
Background –contd.

- **previous research** (static analysis)
  primarily focus on the machinery and methods used to find the errors
  - **advantages:**
    can survey more comprehensive variety of errors
  - **disadvantages:**
    over-represent errors where skilled developers happened to look or where bugs happened to be triggered most often
This research automatically get errors and concentrate on the errors themselves.

- **Advantages**
  - fair comparison across different parts of the kernel (the compiler applies a given extension uniformly across the entire kernel)
  - easily track errors over many versions making it possible to apply the same analysis to trends over time.

- **Disadvantages:**
  - types and content of errors are limited to those found by our automatic tools.
error scope

- **Considered**
  straightforward source-level errors

- **Unconsidered**
  facets of a complete system other than source-level errors
  - performance
  - high-level design
  - user space programs
five central questions

- Where are the errors?
- How are bugs distributed?
- How long do bugs live?
- How do bugs cluster?
- How do operating system kernels compare?
mythology (Research data source)

- from 21 different snapshots of the Linux kernel spanning seven years (from v1.0—v2.4.1).
- from different parts of Linux kernel
  - kernel (main kernel)
  - mm (memory management)
  - ipc (inter-process communication)
  - arch (architecture specific code)
  - net (networking code)
  - fs (filesystem code)
  - drivers (device drivers)
mythology (Gathering the Errors)

- **Inspected errors**: manually examined the error logs produced by the checkers (annotated and propagated from one version to another)

- **Projected errors**: unexamined results occurred by ran checkers with low false positive rates over all Linux versions (Vat, Block, and Null)

- **Notes**: add by 1 for a specific checker whenever an extension encounters an event that (For example, the Null checker notes every call to kmalloc or other routines that can return NUL).

- **Relative error rate**: 
  \[ \text{err_rate} = (\text{inspected} + \text{projected}) \text{ errors/notes}. \]
mythology (checker and corresponding bugs)

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

- **Inspected errors**: manually examined the error logs produced by the checkers (annotated and propagated from one version to another)

- **Projected errors**: unexamined results occurred by running checkers with low false positive rates over all Linux versions (Vat, Block, and Null)

- **Notes**: add by 1 for a specific checker whenever an extension encounters an event (For example, the Null checker notes every call to kmalloc or other routines that can return NULL).

- **Relative error rate**:
  \[
  \text{err\_rate} = (\text{inspected} + \text{projected}) \text{ errors/notes}.
  \]
whether this set of bugs is representative
  - reason: error only come from automatic compiler analysis
  - compensation ways:
    - using results from a collection of checkers that find a variety of different types of errors
    - comparing our results with those of manually conducted studies

bugs has been treated equally
  - compensation ways:
    - find patterns only in important bugs

poor quality code can masquerade as good code
  - reason: it does not happen to contain the errors for which we check
  - compensation ways:
    - Examine bugs across time
    - Present distributions
    - Aggregate samples

checks could misrepresent code quality
  - Reason: they are biased toward low-level bookkeeping operations, ignoring the quality of code
Analysis and answer

- Where Are The Bugs?
Analysis and answer –contd.

**Answer:**

Driver has the highest error rate and absolute number of bugs

- the error rate in driver code is almost three times greater than the rest of the kernel.
- Drivers account for over 90% of the Block, Free, and Intr bugs, and over 70% of the Lock, Null, and Var bugs.

**Possible Reasons:**

- make mistakes using OS interfaces they do not fully understand
- Only a few test sites may have a given device so that most drivers are not as heavily tested as the rest of the kernel
How are bugs distributed?

A common pattern always emerges from summary of the errors sorted by the number of errors found per file. A few files have several errors in them, and a much longer tail of files have just one or two errors. This phenomena can be described by the log series 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.
Analysis and answer –contd.

- Sub-conclusion
  - the log series gives a distinctly better fit if we omit the Block checker..
  - for the Block checker, the Yule distribution fit better than the log series distribution.
Analysis and answer — contd.

- How are bugs distributed?

![Graph showing the distribution of bugs across files and error counts.](image)
Analysis and answer – contd.

- How long do bugs live?

![Graph showing the lifetime of all bugs with cumulative bug numbers over time.](image-url)
Analysis and answer –contd.

- **A Bug’s life**
  - A bug was born when it was introduced into the kernel and was died when the bug was fixed.
  - Bugs that are still alive in the last release have an artificially truncated right endpoint.
Calculating average bug lifetime

Four main problems:
- 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 to about one year
- Miss bugs whose lifespan falls between the versions we check
Calculating average bug lifetime

Four main problems (con’t)
- we have no exact death data for many bugs
  - they are still alive at 2.4.1 (i.e., right censoring).
- Our own interference
- Take into account the nature and purpose of development
  - Traditionally the odd releases (1.3.x, 2.1.x, 2.3.x) are development versions that incorporate new features and fix bugs
  - the even versions (1.2.x, 2.2.x, 2.4.x) are more stable release versions, with most changes being bug fixes
Analysis and answer – contd.

- Average bug lifetimes predicted by the Kaplan-Meier estimator

<table>
<thead>
<tr>
<th>Checker</th>
<th>Died</th>
<th>Censored</th>
<th>Mean (yr)</th>
<th>Median (yr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>87</td>
<td>206</td>
<td>2.52 ± 0.15</td>
<td>(1.93, 2.26, -)</td>
</tr>
<tr>
<td>Null</td>
<td>267</td>
<td>124</td>
<td>1.27 ± 0.10</td>
<td>(0.64, 0.98, 1.01)</td>
</tr>
<tr>
<td>Var</td>
<td>69</td>
<td>33</td>
<td>1.43 ± 0.23</td>
<td>(0.26, 0.29, 0.79)</td>
</tr>
<tr>
<td>All</td>
<td>423</td>
<td>363</td>
<td>1.85 ± 0.13</td>
<td>(1.11, 1.25, 1.42)</td>
</tr>
</tbody>
</table>
Analysis and answer – contd.

- Maximum likelihood **survivor function**
  - $X$ be a random variable representing the lifetime of a bug
  - $d_i$ is the number of bugs that die at time $i$
  - $r_i$ is the number of bugs still alive at time $i$

\[
F_X(t) = Pr[X \geq t] = \prod_{i=0}^{t} \left(1 - \frac{d_i}{r_i}\right)
\]
Analysis and answer – contd.

- How do bugs cluster?
  - Reasons:
    - dependent errors will cause error clustering
    - programmer competence degrades
      poor programmers are more likely to produce many errors in a single place
    - a programmer is ignorant of system restrictions
    - cut-and-paste is more likely to contain clusters of errors
How do operating system kernels compare? 

compare Linux (2.4.1) and OpenBSD (2.8) releases using four checkers: Intr, Free, Null, and Param.

<table>
<thead>
<tr>
<th>Checker</th>
<th>Percentage Linux</th>
<th>Percentage OpenBSD</th>
<th>Ratio</th>
<th>Bugs Linux</th>
<th>Bugs OpenBSD</th>
<th>Notes Linux</th>
<th>Notes OpenBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>1.786%</td>
<td>2.148%</td>
<td>1.203</td>
<td>120</td>
<td>27</td>
<td>6718</td>
<td>1257</td>
</tr>
<tr>
<td>Intr</td>
<td>0.465%</td>
<td>0.617%</td>
<td>1.328</td>
<td>27</td>
<td>22</td>
<td>5810</td>
<td>3566</td>
</tr>
<tr>
<td>Free</td>
<td>0.297%</td>
<td>0.596%</td>
<td>2.006</td>
<td>14</td>
<td>13</td>
<td>4716</td>
<td>2183</td>
</tr>
<tr>
<td>Param</td>
<td>0.183%</td>
<td>1.094%</td>
<td>5.964</td>
<td>9</td>
<td>18</td>
<td>4905</td>
<td>1645</td>
</tr>
</tbody>
</table>
Analysis and answer –contd.

- **Sub-conclusion for Cross-Validation**
  
  For these checkers, OpenBSD is always worse than Linux, ranging from about 20% worse to almost a factor of six.

- **Potential shortcomings**
  
  - the comparison based on a limited number of checkers
  - the checkers only examine low-level operations, and thus give no direct measurement of design quality
the relative error rate of drivers is far higher than that of other kernel code
errors cluster roughly a factor of two more tightly than from a random distribution
bugs last an average of about 1.8 years
errors more objectively than manual inspection could hope to
Questions?