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

Background –contd.

- **This research**

  automatically get errors And concentrate on the errors themselves

  - **advantages**
    - fair comparison cross 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:**
  err_rate =(inspected+ projected) errors/notes.
  .

mythology (checker and corresponding bugs)

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.

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 (For example, the Null checker notes every call to kmalloc or other routines that can return NULL).

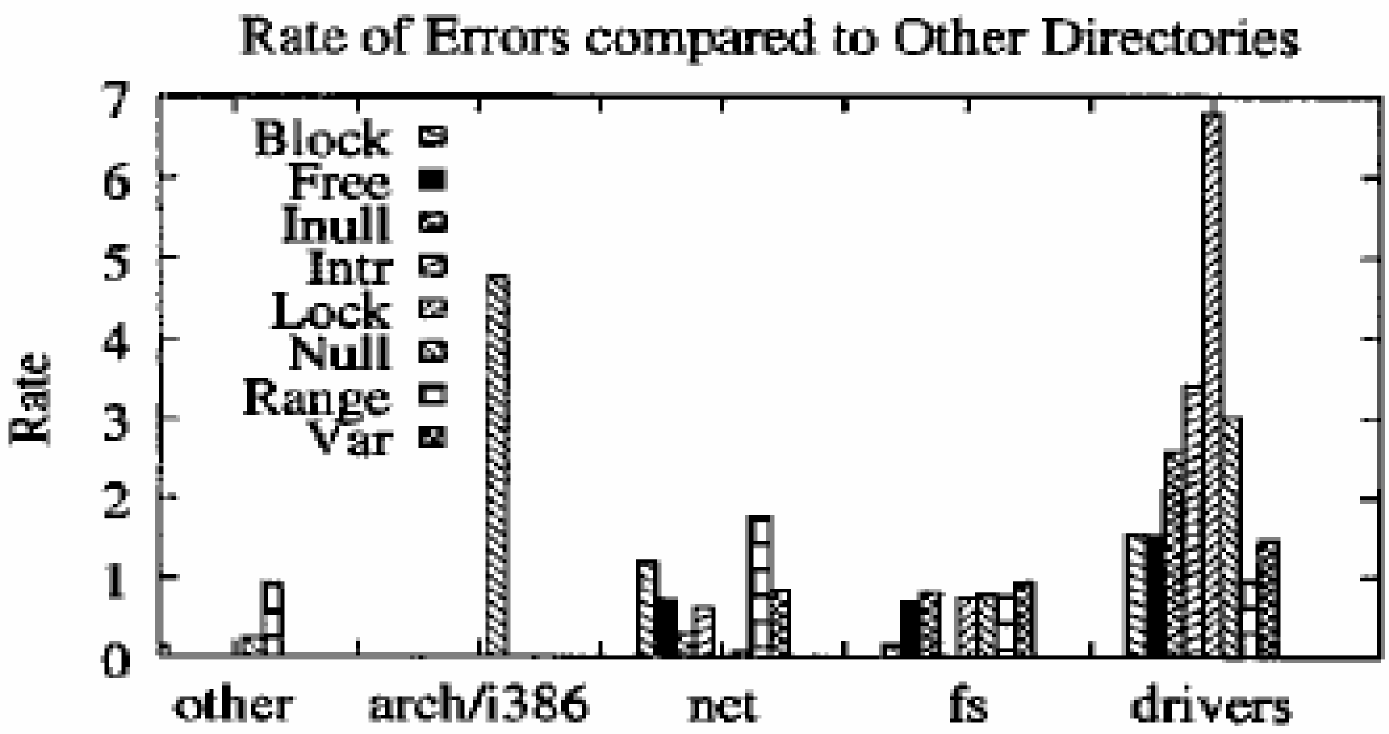
- Relative **error rate:**
  err_rate =(inspected+ projected) errors/notes.
  .

mythology (**caveat**)

- 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.

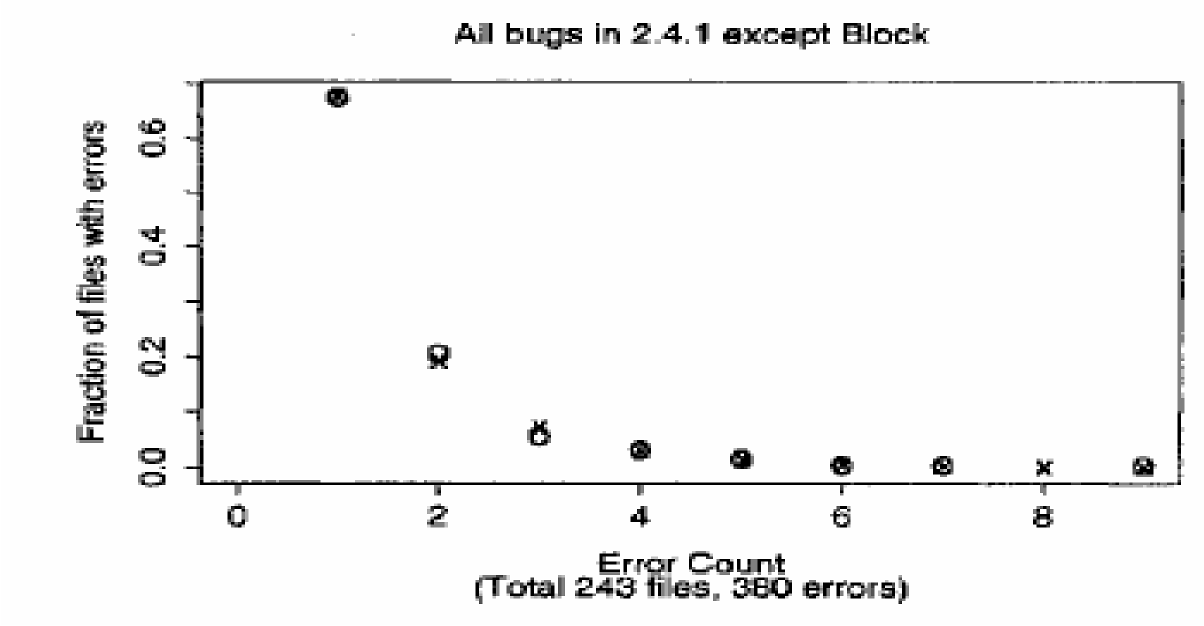
- **Possible Reasons:**
  - make mistakes using OS interfaces they do not fully understand
  - Only a few test sites may have a given device so that most drivers are not as heavily tested as the rest of the kernel

Analysis and answer –contd.

- **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.

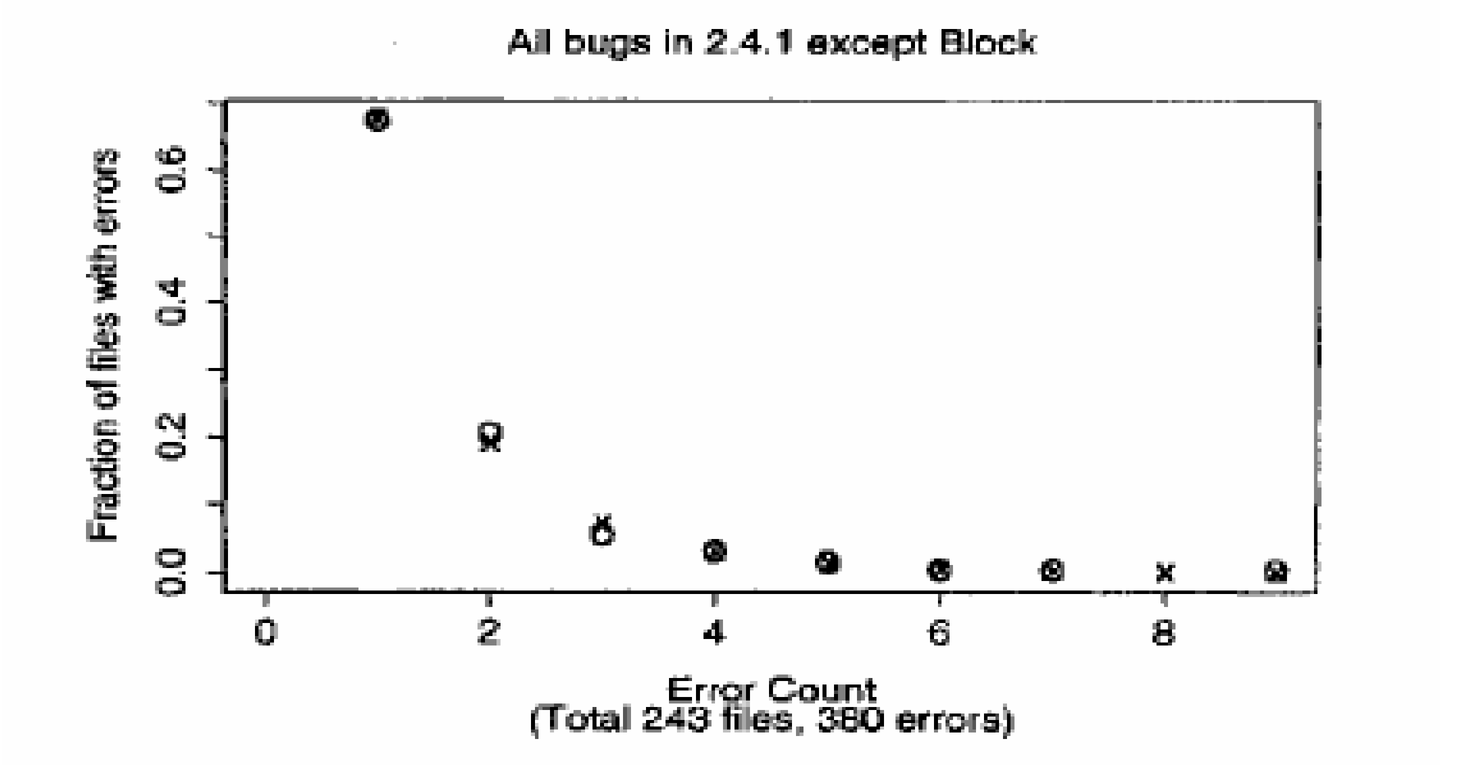
Analysis and answer –contd.

- **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..



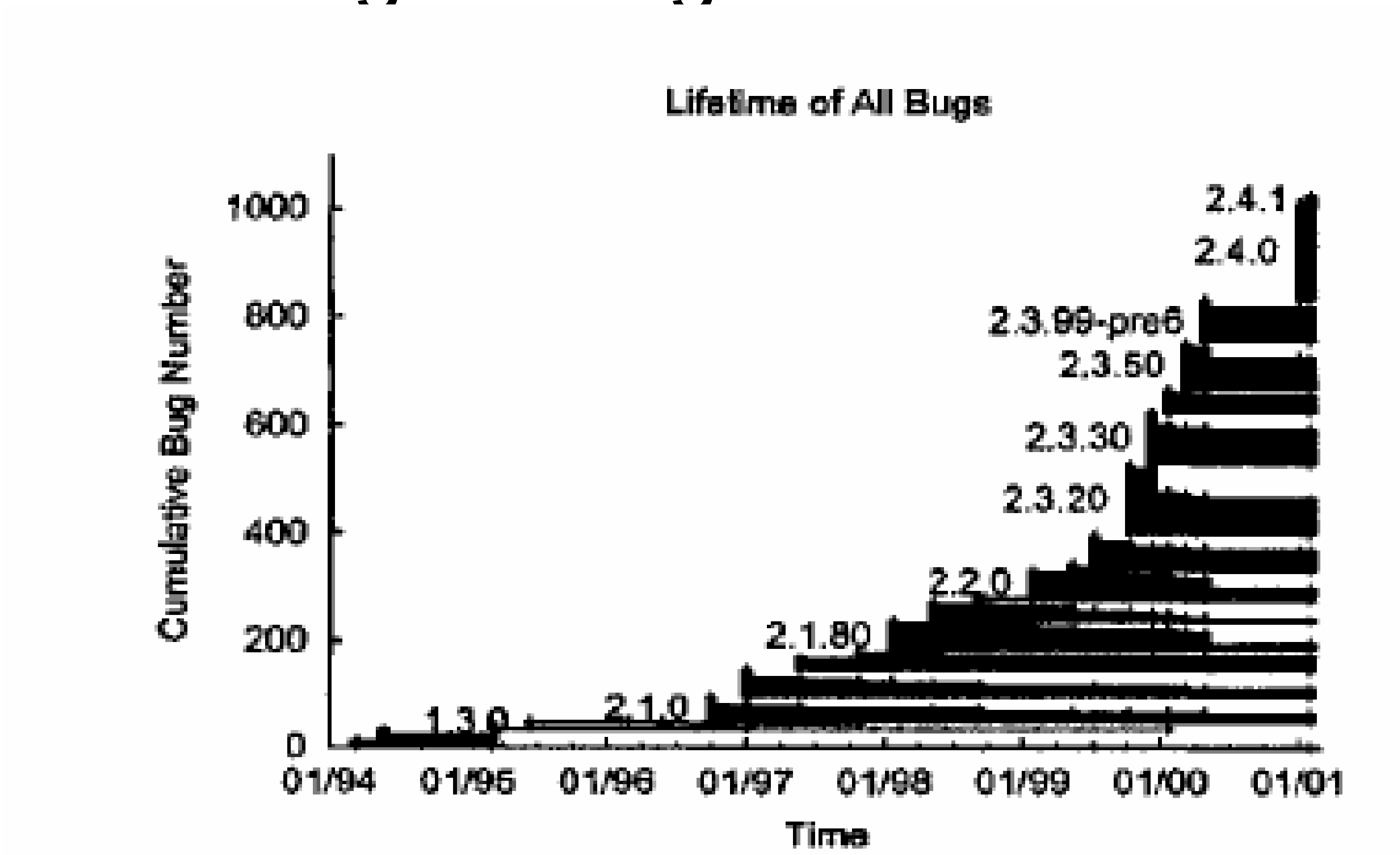
Analysis and answer –contd.

- **How are bugs distributed?**



Analysis and answer –contd.

- How long do bugs live?



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

Analysis and answer –contd.

- **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

Analysis and answer –contd.

- **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 ncorporate 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

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)

Analysis and answer –contd.

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

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

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

Analysis and answer –contd.

- How do operating system kernels compare?

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

Checker	Percentage			Bugs		Notes	
	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

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

conclusion

- **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?