Faults in Linux: Ten years later

A case for reproducible scientific results

Nicolas Palix et. al

ASPLOS 2011

The story begins in 2001...

Chou et al.: An empirical study of operating system bugs [CYC+01]

- Static analysis of bug evolution in Linux versions 1.0 - 2.4.1
- Often condensed to the most important finding: "Drivers are the one major source of bugs in operating systems", which becomes the scientific fundament for a huge body of OS research:
    - Mike Swift: Nooks [SABL06], Microdrivers [GRB+08], Carbon [KRS09]
    - Tanenbaum: Minix 3 [HBG+06]
    - UNSW: Dingo [RCKH09] + Termite [RCK+09]
    - Gun Sirer: Reference Validation [WRW+08]
    - TUD, UNSW and more: user-level drivers [LCFD+05], DDE
    - UKA: DD/OS [LUSG04]
    - Microsoft: Singularity + Signed Drivers [LH10]

The story begins in 2001...

Chou et al.: An empirical study of operating system bugs [CYC+01]

- Static analysis of bug evolution in Linux versions 1.0 - 2.4.1
- Often condensed to the most important finding: "Drivers are the one major source of bugs in operating systems", which becomes the scientific fundament for a huge body of OS research:
  - Mike Swift: Nooks [SABL06], Microdrivers [GRB+08], Carbon [KRS09]
  - Tanenbaum: Minix 3 [HBG+06]
  - UNSW: Dingo [RCKH09] + Termite [RCK+09]
  - Gun Sirer: Reference Validation [WRW+08]
  - TUD, UNSW and more: user-level drivers [LCFD+05], DDE
  - UKA: DD/OS [LUSG04]
  - Microsoft: Singularity + Signed Drivers [LH10]

But it's now been 10 years. Have things changed?

Block

To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.

```
// A) Call schedule() with interrupts disabled
asm volatile ("cli");
schedule();
asm volatile ("sti");

// B) Call blocking function with lock held
//    (BlockLock)
DEFINE_SPINLOCK(l);
unsigned long flags;
spin_lock_irqsave(&l, flags);
..
void *foo = kmalloc(some_size, GFP_KERNEL);
```

NULL / Free

Check potentially NULL pointers returned from routines.

```
my_data_struct *foo =
      kmalloc(10 * sizeof(*foo), GFP_KERNEL);
foo->some_element = 23;
```

Do not use freed memory

```
free(foo);
foo->some_element = 23;
```

Var

Do not allocate large stack variables (>1K) on the fixed-size kernel stack.

```
void some_function()
{
   char array[1 << 12];
   char array2[MY_MACRO(x,y)]; // not found
   ...
}
```

Inull

Do not make inconsistent assumptions about whether a pointer is NULL.

```
void foo(char *bar)
{
   if (!bar) { // IsNull
      printk("Error: %s\n", *bar);
   } else {
      printk("Success: %s\n", *bar);
      if (!bar) { // NullRef
         panic();
      }
   }
}
```

LockIntr

Release acquired locks; do not double-acquire locks. Restore disabled interrupts.

```
void foo() {
   DEFINE_SPINLOCK(l1); DEFINE_SPINLOCK(l2);
   unsigned long flags1, flags2;

   spin_lock_irqsave(&l1, flags1);
   spin_lock_irqsave(&l2, flags2);
   // double acquire:
   spin_lock_irqsave(&l1, flags1);
   ..
   spin_unlock_irqrestore(&l2, flags2);
   // unrestored interrupts for l1/flags1
   // + unreleased lock l1
}
```

Range

Always check bounds of array indices and loop bounds derived from user data.

```
int index = -1;
int n = copy_from_user(&index, userptr,
                       sizeof(index));
if (!n) {
  kernel_data[index] = 0x0815;
}
```

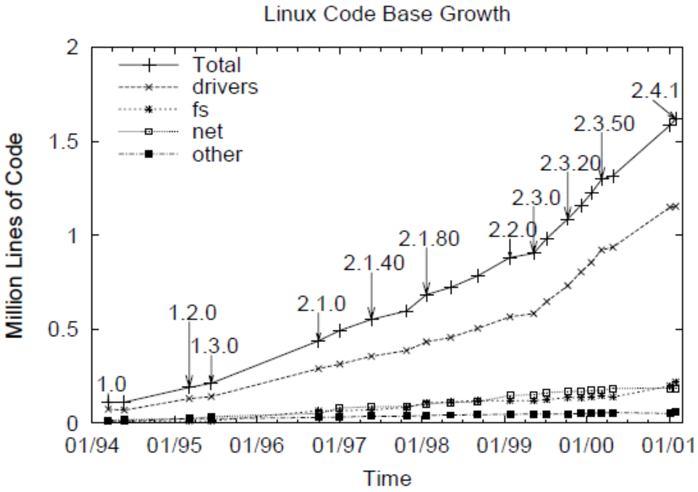
Size

Allocate enough memory to hold the type for which you are allocating.

```
typedef int       myData;
typedef long long yourData;

yourData *ptr = kmalloc(sizeof(myData));
```

Lines of Code



Lines of Code

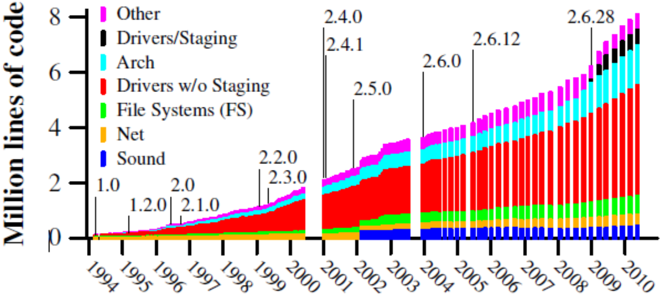


Figure 1. Linux directory sizes (in MLOC)

Fault candidates (notes) over time

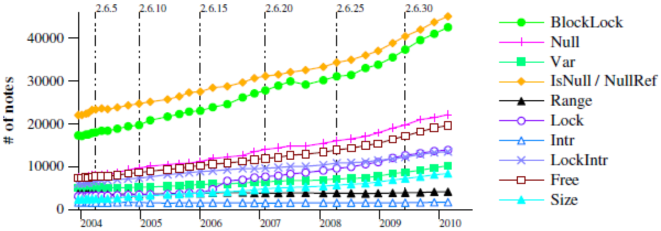
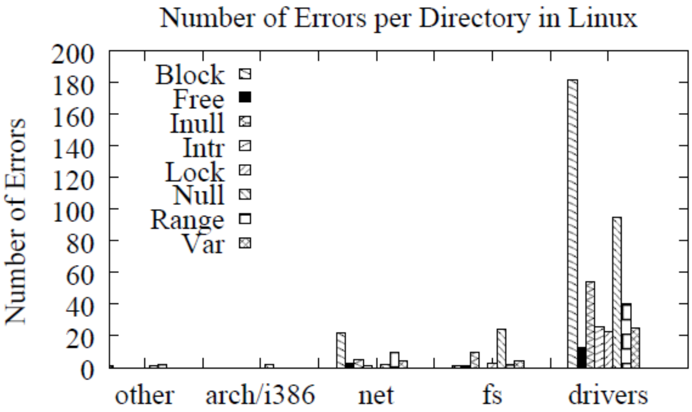
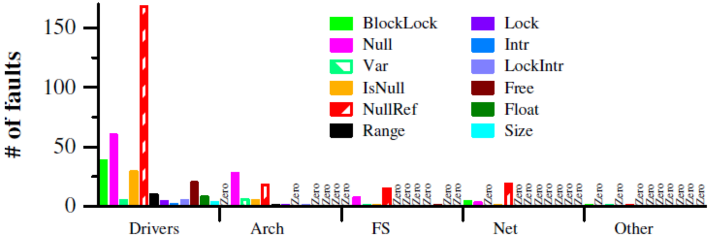


Figure 3. Notes through time per kind

Faults per subdirectory



Faults per subdirectory



(a) Number of faults broken down per directory and category

Faults per subdirectory

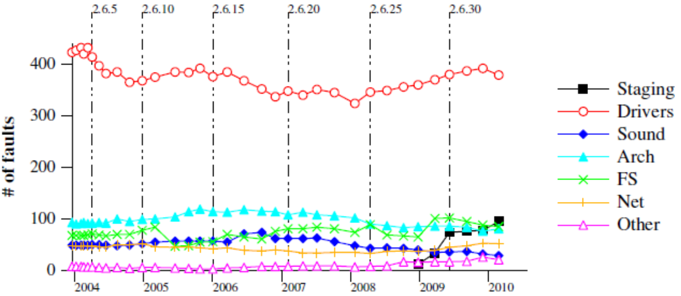
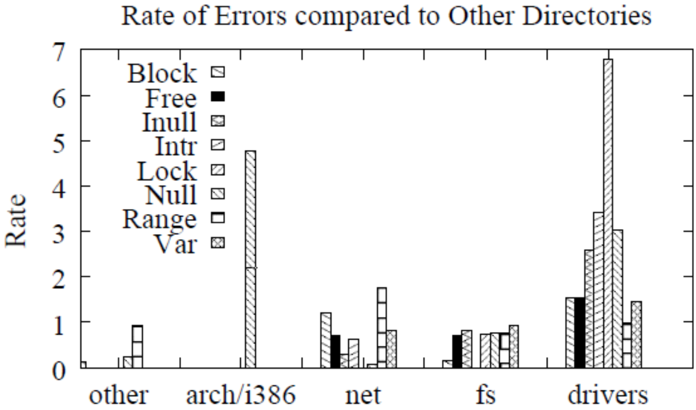
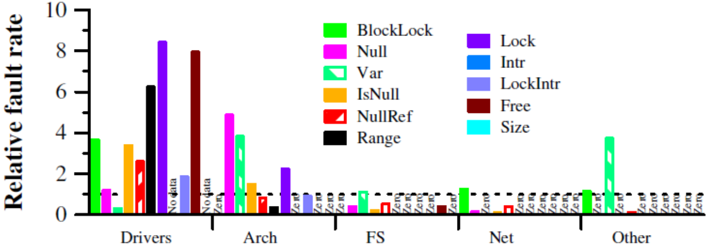


Figure 9. Faults per directory

Fault rate per subdirectory



Fault rate per subdirectory



(b) Rate of faults compared to all other directories

Fault rate per subdirectory

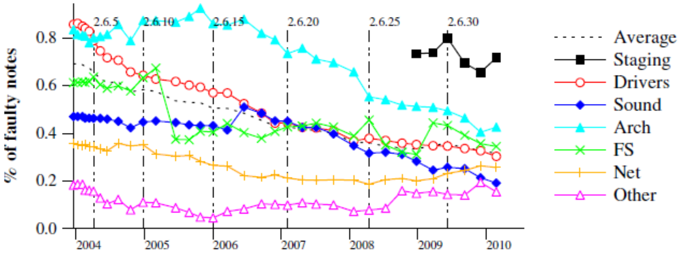
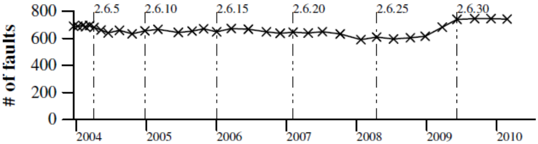
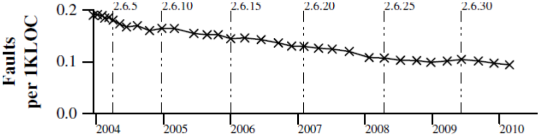


Figure 10. Fault rate per directory

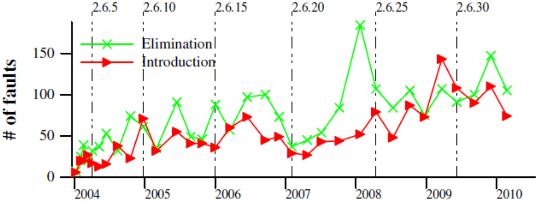
Faults over time (total)



(a) Faults

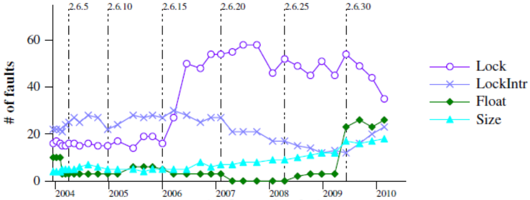


(b) Fault density (Faults per 1KLOC)

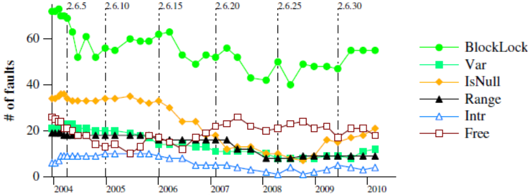


(c) Introduction and elimination of faults

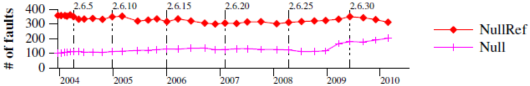
Faults over time (by type)



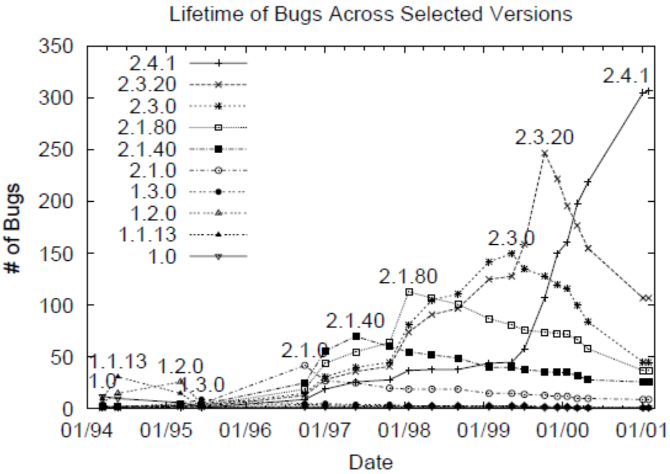
(a) Increasing faults



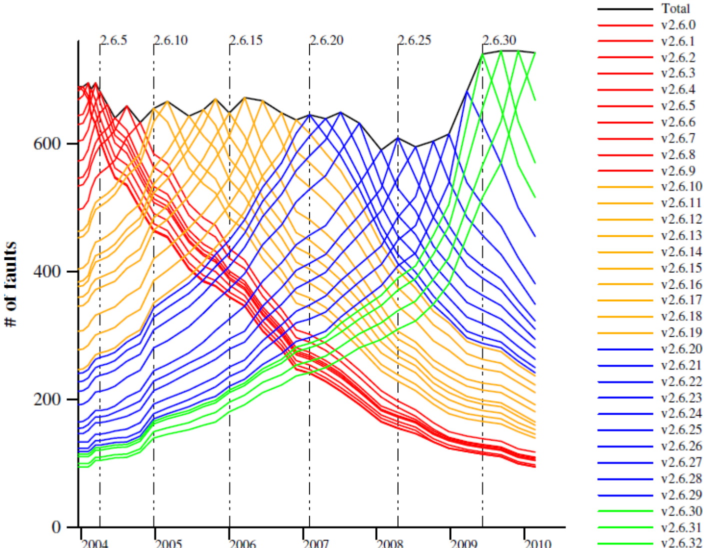
(b) Decreasing faults



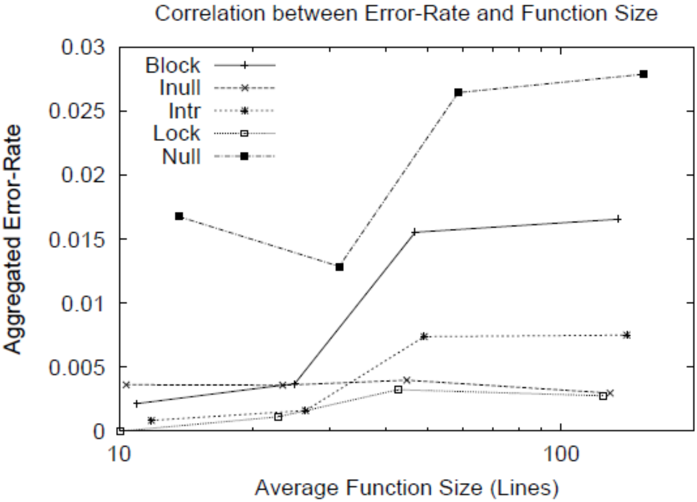
Lifetime of a fault



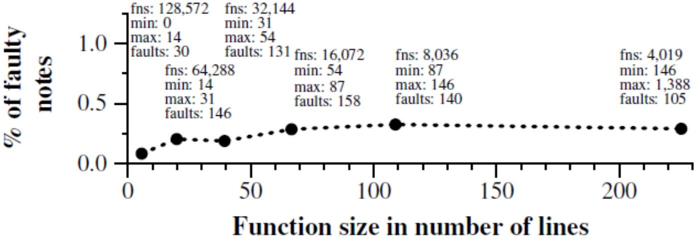
Lifetime of a fault



Function size vs. fault rate



Function size vs. fault rate



(b) Fault rate by function size in 2.6.33

Conclusion

- Drivers are not the single-most important source of faults anymore.
  - Claim: all the research into driver safety has paid off.
  - Counter-claim: adding shiny new CPU architectures is now more attractive to would-be kernel programmers and reviewing new arch code is much harder anyway. (Plus: Chou in 2001 only looked at x86 code).
- Static analysis has come a long way and is pretty helpful.
- SA fails for state-of-the-art faults, e.g., data races and deadlocks (the authors only use heuristics to prevent DL).

Crying for help

...Because Chou et al.s fault finding tool and checkers were not released, and their results were released on a local web site but are no longer available, it is impossible to exactly reproduce their results on recent versions of the Linux kernel...

In laboratory sciences there is a notion of experimental protocol, giving all of the information required to reproduce an experiment...

...Chou et al. focus only on x86 code, finding that 70% of the Linux 2.4.1 code is devoted to drivers. Nevertheless, we do not know which drivers, file systems, etc. were included...

...Results from Chou et al.s checkers were available at a web site interface to a database, but Chou has informed us that this database is no longer available. Thus, it is not possible to determine the precise reasons for the observed differences...

A Chou, J Yang, B Chelf, S Hallem, and D Engler.
An empirical study of operating system bugs.
In *SOSP 01 ACM Symposium on Operating System Principles*, pages 78–81. ACM Press, 2001.

Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha.
The design and implementation of microdrivers.
ACM SIGARCH Computer Architecture News, 36(1):168–178, 2008.

Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum.
MINIX 3: a highly reliable, self-repairing operating system.
SIGOPS Oper Syst Rev, 40(3):80–89, 2006.

Asim Kadav, Matthew J Renzelmann, and Michael M Swift.
Tolerating hardware device failures in software.
Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles SOSP 09, page 59, 2009.

Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser.
User-Level Device Drivers: Achieved Performance.
Journal of Computer Science and Technology, 20(5):654–664, 2005.

J Larus and G Hunt.
The Singularity system.
Communications of the ACM, 53(8):72–79, 2010.

Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz.
Unmodified device driver reuse and improved system dependability via virtual machines.
In *Symposium A Quarterly Journal In Modern Foreign Literatures*, number December, pages 17–30. USENIX Association, 2004.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser.
Automatic device driver synthesis with termite.
Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles SOSP 09, page 73, 2009.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser.
Dingo : Taming Device Drivers.
Analysis, pages 275–288, 2009.

Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy.
Recovering device drivers.
ACM Transactions on Computer Systems, 24(4):333–360, 2006.

Dan Williams, Patrick Reynolds, Kevin Walsh, Emin G, and Fred B Schneider.
Device Driver Safety Through a Reference Validation Mechanism.
Symposium A Quarterly Journal In Modern Foreign Literatures, pages 241–254, 2008.