Faults in Linux 3.x

TAPASWENI PATHAK

Tapasweni Pathak has completed her Bachelor's in computer science from IGDTUW (previously IGIT), Delhi. She has worked with Qualcomm Inc., SAP, and now is with Mapbox. She contributes to open source projects like Linux Kernel, OWASP, Debian Sources, X.org and ABI's Syster Org's projects. She is interested in studying more about decentralization of networks and operating systems as her research interest.
tapaswenipathak@gmail.com

Prior studies have used tools to find bugs in the Linux kernel versions 1 and 2. In this article, I share the results for faults in 3.x versions. This study is a continuation of the work by Chou et al. [1] for versions 1.0 to 2.4.1 and Palix et al. [2a–c] for 2.6 versions. I explain the types of bugs studied, trends for these bugs over newer versions, and how the reports were generated across the different Linux kernel versions.

In 2001, Chou et al. used static analysis tools run over each kernel version to get the results, and the number of common faults found was very high. By 2011, Linux kernel was in its third decade. Palix et al. found that the number of common faults decreased from the previous study results, implying better code quality in 2.6.x, but it was still very high. On February 8, 2015, Linux kernel version 3.19 was released. Patches are regularly submitted for faults found using checkpatch [3], Sparse [4], Coccinelle [5] and Smatch [6]. The number of lines of code in the Linux kernel also crossed 15M at this time. I wanted to follow the path of the previous studies and research how many bugs were in 3.x versions.

Methodology

Palix et al. used the open source tools Coccinelle [5], to automatically find faults in source code, and Herodotos [7, 2c], to run Coccinelle for each fault type and to track the faults across multiple versions of the Linux kernel. Coccinelle and Herodotos are available on the open access archive HAL [7, 2b]. Coccinelle is a tool for pattern matching and text transformation. To study the bed of faults it is necessary to understand the history behind them. When were they first released? When did they die if they did? Did they move after they were first introduced? Following the methodology deployed in the 2011 study, I used Coccinelle to automatically find problematic programming patterns in Linux kernels, and Herodotos to correlate these fault reports between different versions of the Linux kernel. The data about faults in this article were compared with the last study performed and helped to improve the reports generated for the study done on 3.x versions. As an example, there were cases where false positives previously reported moved around in different places in the code file.

Emac's org mode (orgmode.org), a text file format, was used to categorize the reports as bug or false positive. With this it was easier to move between different versions of the Linux kernel for the same report and study the history and reason behind a given bug type classification. This manual process was performed to make sure that none of the false positives generated were marked as bugs. I cloned all Linux kernel versions from 3.0 to 3.19 and considered the function stack, calls, and all possible inputs, outputs, Linux kernel standards, stack size etc. to categorize these reports as bugs or false positives. I also submitted patches for the bugs that were present in the then-current Linux kernel version.

In a few cases, I was not able to categorize the reports as bugs or false positives. In these cases, I used UNKNOWN/IGNORED.

The tools and marked reports generated were publicly made available in a GitHub repository [8].

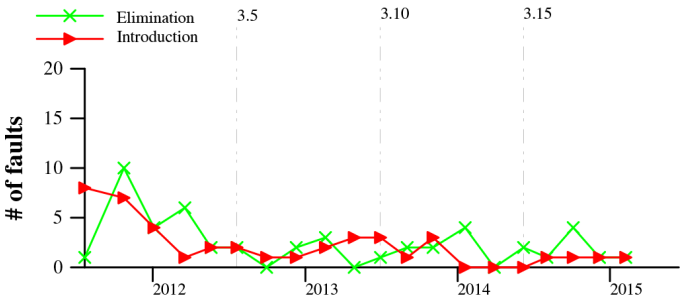

Figure 1: Birth and death of IsNull

After analyzing the reports, Nicolas Palix generated figures to highlight the rise and fall of the number of bugs in the different versions. I used org2sql to update the database of records for the Linux kernel versions 3.xx. org2sql tries to import all the faults and needs of at least two parameters: the prefix of files to drop (/fast_scratch/linuxes/) and the new.org file to import. The output is on stdout, which I then directed to an SQL file, which later was used with psql. All the figures were generated using these data and scripts [9].

Studied Fault Types and Their State

Inconsistent Assumptions about NULL

Dereferencing a pointer is undefined if the pointer is null. This fault type comes in two flavors: IsNull and NullRef. An IsNull fault is where a NULL test is done preceding a dereference, and a NullRef fault is where a NULL test is done following a dereference. The former is always an error, while the latter may be an error or may simply indicate overly cautious code, if the pointer can never be NULL.

Both fault types consistently decreased between versions 3.0 to 3.19. Figure 1 shows that the introduction of the IsNull bug type moved close to zero with Linux version 3.19 from the highest point with Linux version 3.0.

208 NullRef faults (Figure 2) were reported in total in Linux 3.19, and 112 of them were introduced in 3.0 or later.

As an example, a bug in Linux 3.15 occurred where a null check was done after referencing it inside the file drivers/staging/media/rtl2832u_sdr/rtl2832_sdr.c, line 992, in the function `rtl2832_sdr_start_streaming` for the *s* variable.

```
dev_dbg(&s->udev->dev, "%s:\n", __func__);
if (!s->udev)
```

An interesting false positive (FP) was found in Linux-3.11 inside the file net/nfc/llcp_core.c, lines 724 (null test) and 761 (nullref), in the function `nfc_llcp_tx_work()`, if `llcp_sock` is checked for null with one more condition (&&):

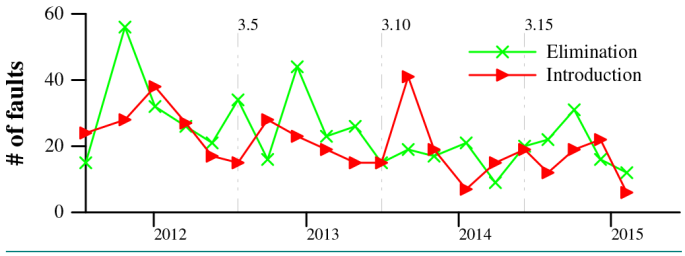

Figure 2: Birth and death of NullRef

```
if (llcp_sock == NULL && nfc_llcp_ptype(skb) == LLCP_PDU_I)
        .....
else if (llcp_sock && !llcp_sock->remote_ready)
        .....
```

Then inside the `else`, `llcp->sock` is dereferenced using

```
skb_queue_tail(&llcp_sock->tx_pending_queue, copy_skb);
```

The code is only a problem if `llcp_sock` is null and if `ptype == LLCP_PDU_I`. But `ptype` is defined as `u8 ptype = nfc_llcp_ptype(skb)`. And up at the top of the sequence of ifs there is another case for where `llcp_sock == NULL && nfc_llcp_ptype(skb) == LLCP_PDU_I`.

Disabling but Not Reenabling Interrupts

This includes interrupts that are turned off but not turned on again, using the function `spin_lock_irqsave`. `spin_lock_irqsave` is used to save the interrupt state before acquiring the spin lock. This is because spin lock disables the interrupt, when the lock is taken in interrupt context, and reenables it while unlocking or when using `local_irq_disable` and `local_irq_save`. The interrupt state is saved so that it can reinstate the interrupts again.

Locking but Not Unlocking and Double Locking

Double locking is a bug. This check looks for cases where a lock is taken but not released, that is, where an unlock is missing. In a few cases, interrupts are disabled at the same time that a lock is taken. Figure 3 shows that for the LockIntr bug type, the introduction rate reached its peak during 2013. With the introduction of Linux 3.9, the LockIntr rate fell to zero, implying there were no new LockIntr bugs that were produced with this release.

I even found a few interesting FPs where I plan to improve the semantic patch in Linux 3.5 inside the file kernel/workqueue.c at line 1013, in the function `__queue_work()`:

```
 spin_lock_irqsave(&last_gcwq->lock, flags);
 worker = find_worker_executing_work(last_gcwq, work);
if (worker && worker->current_cwq->wq == wq
gcwq = last_gcwq;
  else {
```

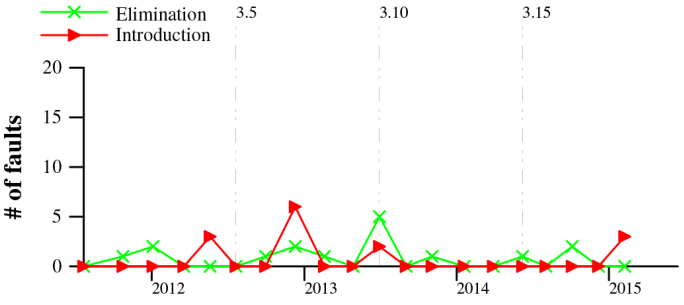

Figure 3: Birth and death of LockIntr

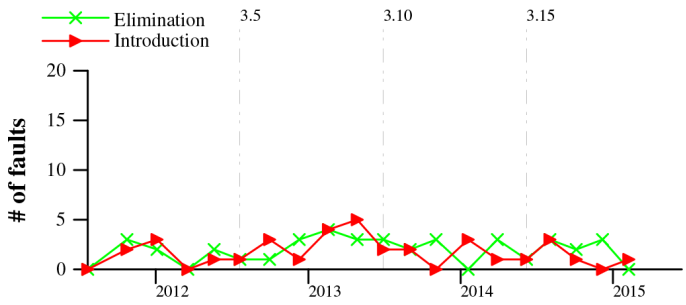

Figure 4: Birth and death of Lock

```
/* meh... not running there, queue here */
spin_unlock_irqrestore(&last_gcwq->lock, flags);
spin_lock_irqsave(&gcwq->lock, flags);
```

In the case where the unlock seems to be missing, there is the code `gcwq = last_gcwq`. In the case where the unlock is present, it is followed by the code `spin_lock_irqsave(&gcwq->lock, flags)`. That is, the whole set of nested `if`s terminates with the need to unlock `&gcwq->lock`. This lock is unlocked later. And in the case where `worker && worker->current_cwq->wq == wq`, it is the case that `gcwq = last_gcwq`, so the subsequent unlock of `&gcwq->lock` will unlock the `last_gcwq` lock because they are the same.

Calling Blocking Function with Interrupts Disabled or Spinlock Held

Blocking with interrupts disabled or a spinlock held can lead to deadlock. Basic memory allocation functions, such as the kernel function `kmalloc`, often take as their argument the constant `GFP_KERNEL` when `kmalloc` is allowed to block until a page becomes available. Thus, a function that contains a call with `GFP_KERNEL` as an argument may block.

However, blocking with interrupts turned off is not necessarily a fault, and indeed core Linux scheduling functions, such as `interruptible_sleep_on`, call "schedule" with their interrupts turned off. This issue was taken into account when checking for false positives.

This fault type checked for locks around possibly blocking functions.

Figure 4 shows that the birth and death of the Lock bug type had a fall. The slight increase in the introduction with Linux 3.19 is explained below.

In Linux 3.19, in the file drivers/staging/emxx_udc/emxx_udc.c at line 2797, inside the function `nbu2ss_ep_queue()`, `GFP_KERNEL` is used when calling `dma_alloc_coherent`. `GFP_KERNEL` was replaced with `GFP_ATOMIC` with a patch, as the latter will fail if the heap doesn't have enough free pages but will not sleep and hence avoids deadlock.

Wrong Use of `krealloc`

This fault type checked for a wrong use of `krealloc`. `krealloc` reallocates memory, while the contents of the memory remain unchanged. If `krealloc()`' returns NULL, it doesn't free the original pointer, which was pointing to the memory allocated. So any code of the form `foo = krealloc(foo, ...);` is certainly a bug. `krealloc` should use a temporary pointer for allocations and check the temporary pointer returned against NULL too.

For `krealloc` type reports, all reports were bugs and none were FPs in the case for 3.x versions. The most recent was in Linux-3.16 in the file drivers/pinctrl/sunxi/pinctrl-sunxi.c at line 740:

```
pctl->functions = krealloc(pctl->functions,
        pctl->nfunctions * sizeof(*pctl->functions),
        GFP_KERNEL);
```

If reallocation fails, `krealloc` will return NULL to `pctl->functions` without freeing the memory previously pointed to by `pctl->functions`.

Interrupts Turned Off but Not Turned On Again

Calling the `local_irq_save` function disables interrupts on the current processor and saves current interrupt state as flags (passed to this function). `local_irq_restore` function enables interrupts and restores state using the flags. In early versions of Linux, locks and interrupts were managed separately: typically interrupts were disabled and reenabled using `cli` and `sti`, respectively, while locks were managed using operations on spinlocks or semaphores. This fault type checked for the case where interrupts were turned off using the functions `local_irq_save` or `save_and_cli` but were not turned on again.

Figure 5 shows that in Linux kernel 3.17, this bug type had new introductions as well as eliminations. By Linux 3.19 both introduction and elimination reached zero.

I found a total of four bugs of this type. One was in Linux 3.17 in the file arch/mips/kvm/tlb.c at line number 206, inside method `kvm_mips_host_tlb_write()`:

```
local_irq_save(flags);
```

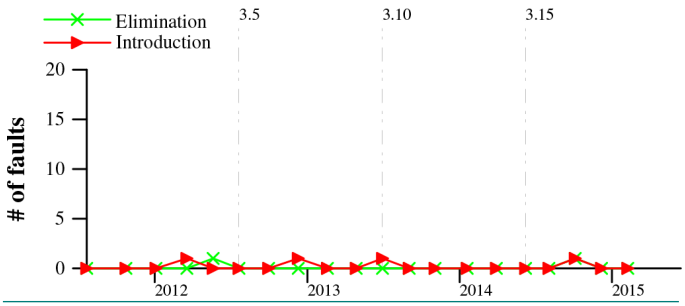

Figure 5: Birth and death of interrupt-related bugs

The interrupt state is saved so that it should reinstate the interrupts again, but, in this case, after the above call to `local_irq_save()`, there is no call to `local_irq_restore()`.

Using Freed Memory

`kfree` frees previously allocated memory. Using freed memory can cause the kernel to crash, can lead to a write-what-where condition, and can have consequences like corruption of valid data and the execution of arbitrary code. I checked for cases where there was a use after `kfree` and after a function that directly or indirectly calls `kfree`. The false positives were mostly when the variable freed was accessed only after a null check.

There were a lot of cases where `goto` was being used immediately after the `kfree`, which doesn't allow the statement to execute when using the freed memory. There were also many cases where an immediate return was done after `kfree`, and thus the statement where a variable accessed after `kfree` was not executed. There were cases where a check on the variable just freed (inside an `if`) was being done, hence avoiding a buggy situation.

Allocating Large Arrays on the Stack

All the local variables in the function are allocated on the stack. If too much memory is allocated on the stack, the kernel might run out of stack memory because the Linux kernel stack has a fixed size.

This fault type checked for instances where large arrays were allocated on the stack. I considered an array to be large if it contained more than 1023 bytes. Anything below that was marked as a false positive, and anything greater than that was considered a bug.

No new bugs of this bug type were introduced with Linux kernel version > 3.11. I found one FP of type var in Linux 3.11 in the file drivers/staging/lustre/lnet/klnds/socklnd/socklnd_cb.c at line 1034:

```
static char ksocknal_slop_buffer[4096];
```

In this case it was a global static variable, only visible in one function and not declared on the stack, so this was an FP.

Using Value Taken from User as Array Bounds and Loop Index without Check

Values taken from userspace should be checked for limits before using these values. A value could be huge, or it could be negative if the type of the field is not unsigned. `copy_from_user` is used to copy a block of data from userspace to kernel space. It then returns the number of bytes that could not be copied. On success, this will be zero. If some data could not be copied, this function will pad the copied data to the requested size using zero bytes. `get_user` is used to get a simple variable from userspace. This macro copies a single simple variable from userspace to kernel space.

This fault type checked for the case where unchecked values were obtained from the user level through `copy_from_user` and `copy_from_user` may be used as an array index or loop bound.

The Linux kernel from versions 3.0 to 3.19 had very few instances of introduction of this bug type. The Linux kernel 3.19 had no new user-value bug type introductions.

I found one `copy_from_user` type bug in Linux 3.12, in the file fs/btrfs/ioctl.c at line number 2736, inside the function `btrfs_ioctl_file_extent_same()`; `copy_from_user` is done using the `same` structure. `same->logical_offset` is then assigned to `off`, and `same->length` is assigned to `len`. The `len` variable is then checked for the maximum value it can have; if it exceeds that, it is assigned the maximum it can take. But later, the loop uses `same->dest_count` and not `len`.

I found one bug of type `get_user` in Linux 3.14 in fs/btrfs/ioctl.c at line number 2759, inside the function `btrfs_ioctl_file_extent_same()`'. No checks were done on count, and later it was used as an array index.

Wrong Assumption about Size of Object Being Allocated Memory

There were a total of 25 bugs relating to size type, all in <= 3.9 versions of the Linux kernel. A very simple way to identify this bug was in Linux 3.5 in the file drivers/net/wireless/mwifiex/ie.c at line 166. The two structures mwifiex_ie_list and mwifies_ie are different, which makes this usage buggy.

Using Floating Point Values

When a userspace process uses floating-point instructions, the kernel catches a trap for a floating-point instruction and then initiates the transition from integer to floating-point mode. This varies by architecture. In a kernel space process, the kernel cannot trap itself to support floating point. This is supported by manually saving and restoring the floating-point registers, among other chores. Saving and restoring floating point register state also makes floating-point operations slower than integer operations. People have always been advised not to use floating-point operations in the kernel.

Type	Reports	Bugs	Unknown
Kfree	304	138	1
	180 + 124	111 + 27	1 + 0
isNull	152	122	0
	108 + 44	80 + 42	
NullRef	1813	1578	24
	1313 + 500	1169 + 409	23 + 1
LockIntr	252	103	2
	186 + 66	89 + 14	2 + 0
Intr	35	23	0
	25 + 10	19 + 4	
Krealloc	25	21	0
	14 + 11	10 + 11	
Lock	674	230	4
	454 + 220	198 + 32	4 + 0
var	66	36	0
	51 + 15	35 + 1	
copy_from_user	5	5	0
	4 + 1	4 + 1	
get_user	25	19	1
	24 + 1	18 + 1	1 + 0
Float	549	46	0
	532 + 17	46 + 0	
size	214	52	0
	149 + 65	27 + 25	

Table 1: Number of bugs in Linux 2.6.x and 3.x versions. The first number in each row shows the total bugs for both 2.6.x and 3.x, and the pairs of numbers following are for 2.6.x first and 3.x second.

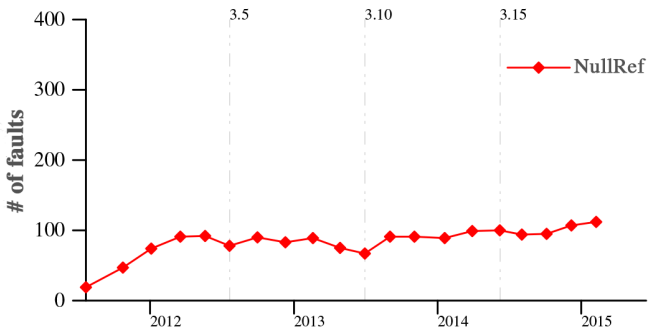

Figure 8: Count of bugs of NullRef type

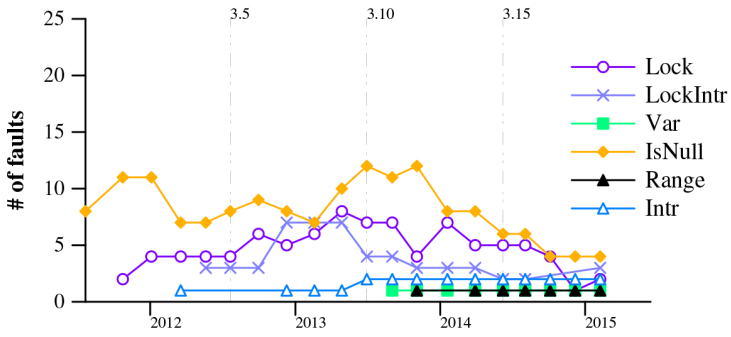

Figure 6: Overall birth and death of six fault types

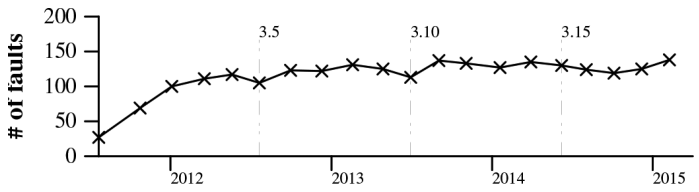

Figure 7 : Faults introduced with version 3.0 and after

This fault type checked for floating-point usages in kernel code. There was only one report for this case. Most false positives occur when the computation can be simplified at compile time to an integer. The checker only reports a floating-point constant that is not a subterm of an arithmetic operation involving another constant, and hence may end up in the compiled kernel code. Examples of false positive occurred where values like 1.6 * 1000 * 10 were being used.

The Overall Results

The total number of reports generated using the tools described in the Methodology section were 4114. This number constitutes the results generated by both 2.6.x and 3.x correlated reports. There were 1074 reports belonging only to 3.x, out of which 567 were bugs and 506 were false positives. We marked one as unknown. This table breaks down the numbers per each fault type. The second line in each cell breaks down the total into the numbers from 2.x and 3.x versions.

Overall Birth and Death of Faults

This graph indicates the number of each type of bug introduced in the 3.x versions and the number of bugs introduced and removed in each version.

All of the six fault types have decreased over the period of 2012 to 2015, with the greatest decrease being for the IsNull bug type. The Intr bug type, which was once zero, increased with the 3.10 version but has remained flat up to Linux 3.19. Figure 6 also suggests that these bug types did not reach zero until 2015.

Faults Introduced in 3.0 or After

Figure 7 shows the overall number of all the faults. The slope of the faults increases with newer Linux kernel versions, with NullRef being the highest (see Figure 8).

Future Work

Julia Lawall, Nicolas Palix, and I plan to study these fault types for Linux kernel 4.x.

Acknowledgements

Thanks to Julia Lawall, Inria Senior Research Scientist, and Nicolas Palix, Assistant Professor at University Grenoble Alpes, for working alongside me on this and for reviewing the article.

References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, pp. 73–88: http://www.stanford.edu/~engler/.

[2a] N. Palix, G. Thomas, S. Saha, C. Muller, J. Lawall, "Faults in Linux 2.6," *ACM Transactions on Computer Systems*, vol. 32, no. 2 (June 2014): https://arxiv.org/pdf/1407.4346v1.pdf.

[2b] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller, *Faults in Linux: Ten Years Later* (Dec. 2010): http://faultlinux.lip6.fr/.

[2c] N. Palix, S. Saha, G. Thomas, C. Calvès, J. Lawall, and G. Muller, *Faults in Linux: Ten Years Later*, Research Report RR-7357, INRIA (July 2010): http://hal.inria.fr/inria-00509256.

[3] Checkpatch: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/checkpatch.pl.

[4] Sparse: https://sparse.wiki.kernel.org/.

[5] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and Automating Collateral Evolutions in Linux Device Drivers," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08)*, pp. 247–260: https://www.cse.unsw.edu.au/~cs9242/08/exam/paper2.pdf.

[6] The Kernel Janitors, "Smatch, the Source Matcher," 2010: http://smatch.sourceforge.net.

[7] N. Palix, J. Lawall, and G. Muller, "Tracking Code Patterns over Multiple Software Versions with Herodotos," in *Proceedings of the ACM International Conference on Aspect-Oriented Software Development (AOSD '10)*, pp. 169–180: DOI:http://dx.doi.org/10.1145/1739230.1739250.

[8] GitHub, "Coccinelle/Faults in Linux: Experimental Bed to Study Linux Faults": https://github.com/coccinelle/faults-in-Linux.

[9] Scripts to generate graphs for Linux kernel fault study: https://github.com/npalix/linux-study-figures.