Tracking Kernel Rate of Change

Pat Riehecky

CentOS Dojo - Feb 2022 - FERMILAB-SLIDES-22-002-SCD

4 Feb 2022

Disclaimers

Pat Riehecky is **not** speaking as official spokesperson for:

- Fermi Research Alliance
- Fermi National Accelerator Laboratory
- US Department of Energy

Neither the United States nor the United States Department of Energy, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any data, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

This is not an endorsement of any technology, software, service, or organization.

This presentation is an act of *research*, rather than a prescription of future practices.

The Road Ahead in this Presentation

Once again, I'll be largely reading the slides to you...

Alas, I still couldn't think of a better way to ensure a uniform understanding of the topics ahead, simplify translation for non-english speakers, and provide a way to use the slide deck as quick reference to extract relevant content for folks.

External kernel modules and CentOS Stream

A lot of what follows is an attempt to answer this question: How do I get a kernel module that doesn't ship with CentOS Stream working on CentOS Stream? What interfaces can I use? How do I figure out what is changing? What resources exist?

To answer this we will need to explore the kernel interfaces, how things differ inside the EL ecosystem, and what this means for your specific kmod.

Mainline Linux Driver Interface Stability

Excerpts from The Linux Kernel Driver Interface:

- Please realize that this article describes the **in kernel interfaces**, not the kernel to userspace interfaces. (emphasis added)
- …Linux does not have a binary kernel interface, nor does it have a stable kernel interface.
- …kernel developers find bugs in current interfaces, or figure out a better way to do things. If they do that, they then fix the current interfaces to work better. When they do so, function names may change, structures may grow or shrink, and function parameters may be reworked. If this happens, **all of the instances** of where this interface is used **within the kernel** are fixed up at the same time, ensuring that everything continues to work properly. (emphasis added)
- Releasing a binary driver for every different kernel version for every distribution is a nightmare, and trying to keep up with an ever changing kernel interface is also a rough job.
- What you want is a stable running driver, and you get that only if your driver is in the main kernel tree.

RHEL Kernel Driver Interface Stability

Excerpts from Red Hat's What is Kernel Application Binary Interface (kABI)? :

- The Kernel Application Binary Interface (kABI) is a set of in-kernel symbols used by drivers and other kernel modules. **Each major and minor** Red Hat Enterprise Linux release has a set of in-kernel symbols… (emphasis added)
- Once a symbol has been introduced into the kernel Application Binary Interface (kABI) for a particular major release of Red Hat Enterprise Linux it will not be removed, nor will its meaning be changed during that kernel **major release**'s complete life cycle. (emphasis added)
- …it can happen that a symbol in a kABI ~~whitelist~~ [stable list] needs to be changed
- Red Hat generally endeavors not to make changes even to non-stablelist kABI symbols in regular security and errata updates but there is no guarantee that they will be compatible within a same minor release.
- When non-stablelist kABI symbols are changed in kernel updates, it is not considered as kABI breakage.

RHEL Kernel Driver Interface Stability vs userspace

The userspace elements that consume kernel functionality should “just work” and “not break”. Those guidelines come from the mainline kernel and are inherited in the RHEL one.

For folks thinking about the userspace binary compatibility guide, that explicitly excludes the kernel.

Link to presentation on that topic:
Thinking About Binary Compatibility and CentOS Stream

The RHEL kernel has a few “stable” symbols and a lot of “mutable” symbols. I prefer the term “mutable” rather than “non-stable” as the symbols only change for cause, rather than constantly. For example, 601 symbols (~ 2.9%) haven’t changed in the RHEL 8.5 GA kernel since the RHEL 6.0 GA kernel – only 81 of them are on the stable list – and the stable list isn’t intended to cross major versions.

How does the Stream Kernel relate to RHEL releases?

At minor releases within the RHEL product lifecycle (8.x, 9.x) the RHEL kernel is updated with changes published in the CentOS Stream kernel since the last minor release (x-1).

For a while after the publication of a minor release, the two kernels are in sync. Eventually the Stream Kernel runs ahead of RHEL. Later, RHEL catches up.

What is the CentOS Stream Kernel?

The CentOS Stream kernel is analogous to the “RHEL kernel” + δ (delta)

Ok, what then is δ? Here δ is just a placeholder for “changes to the source code from the RHEL kernel *today*”.

Depending on a number of factors, δ will be zero sometimes.

For example, when RHEL 8.5 released on Nov 9, 2021 with the 4.18.0-348 kernel, CentOS Stream 8 had been running it for 19 days (Oct 20, 2021). Until Nov 15, 2021, δ remained zero. It went back to zero again on Nov 17, 2021 when the two distributions once again ran the same kernel. Then on Dec 21, 2021 they fell out of sync, only to get back into sync the very next day.

From Nov 9, 2021 to January 9, 2022 the RHEL kernel and the Stream kernel were in sync for 58 of 61 days. Or put another way, on those 58 days δ = 0.

What is the CentOS Stream Kernel?

RHEL 8.4 released the 4.18.0-305 kernel on May 18, 2021. On June 8, 2021, the CentOS Stream 8 kernel gained a δ that would persist until Nov 9, 2021.

Or to put another way - on Nov 8, 2021 (the day before 8.5 released) δ was very much **not** zero – and it became non-zero on June 8, 2021 in a way that persisted until Nov.

Proving two things are not equal is very different from being able to say how they diverge at a specific time (δ) – or even measuring how that difference is expressed (ε epsilon) or what the impact might be on your workloads.

Thankfully we are working with source code rather than calculus, so we are nearly done with our ε-δ proofs (to measure the instantaneous rate of change).... nearly

What is the CentOS Stream Kernel?

Trying to define the CentOS Stream Kernel in a more formal (mathematical) language:

The `LIMIT` of the RHEL kernel as Time approaches the end of Full Support is the CentOS Stream kernel.

For the less mathematically inclined: During Full Support, the RHEL kernel is, at regular intervals, trying to be the CentOS Stream kernel. **Each major and minor** Red Hat Enterprise Linux release has a set of in-kernel symbols they try to maintain, so the two match most closely at minor release updates.

How do I understand the differences?

What is the ε (difference expressed in behavior change) between 4.18.0-305.el8 and 4.18.0-310.el8? Do we measure it in lines changed, features added, function calls changed, bugs fixed, user experience, or something else?

The patches added to the kernel are δ. I care that it exists; I care when it is non-zero **and** makes ε non-zero. If δ is not zero, but ε is zero (however ε is quantified), I'm not sure I care that something in the source changed.

RHEL uses modversion CRC checks stored in the RPM to expose the ABI honored by the stable list. So that is what we'll use to generate our δ. As for ε, that is about how your module interfaces with the kernel (and your workflows).

`modversion` is not a great way to do this

To be clear, I'm not aware of a better way to do this. That is **not** the same as saying that tracking the CRC changes provides a definitive picture of what is changing.

In my experience, folks care about how changes impact them. For example, if you're not using the `mpt3sas` driver changes there are not relevant to you. This is also true if your card doesn't have the silicon being targeted by this change. Counting raw checks on CRC results is not workload specific and not necessarily behavior specific.

Additionally, these checks are limited in how much introspection they perform (none). This is a good, if sad, thing. To get the introspection I'd want the kernel would need to instrument itself to instrument itself to instrument itself…..

The `modinfo srcversion` is handy, but most useful for mainline kernels or kpatches, so I'm going to ignore it here. It is neat, but not targeted by our current processes.

A Case Study: OpenAFS, 2.6.32-279.el6, ext4, on i686 (oversimplified)

RHEL6.4 backported a code change in readdir() cookies. As a result the 64bit inodes of ext4 were now represented by 64 bit integers (as they should be) on i686 systems (oh…).

The OpenAFS kmod filesystem cache uses the inode to manipulate its elements. This bypasses a bunch of pointless directory traversal and makes cache poisoning **a lot** harder.

But, when the OpenAFS cache says "here are all my known inodes (32bit) of cached elements" and the kernel wants 64bit inodes… Things work less well…

The `modversion` checksums did not find this. Why? The relevant code returns a `struct` of `pointers` to the results, rather than the results - the size of the `struct` didn't change, the size of the `pointer` didn't change – neither did the quantity of `pointers`. The value of the element pointed-to did change. These checksums, by definition, cannot find this behavior.

This type of thing is a common pattern in the kernel. Rather than copy the results from memory into something else so we can return it, it performs a *zero-copy operation* and just says "I got what you wanted; it is over there." This is a lot faster than a copy and return!

Example: Feature Flags

Consider the following kernel interface:

```
ssize_t copy_file_range(int fd_in, loff_t *off_in, int fd_out,
                        loff_t *off_out, size_t len, unsigned int flags);
```

When support for a new flag is added, does the CRC of the call/return change? No it does not. The whole point of the `flags` parameter is to avoid changing the interface when new behavior is added.

This pattern makes the development and maintenance of the interface easier. Today, the only supported argument to `flags` on `copy_file_range` is `0`. That will probably change at some point. Detecting that is not something we can do from the interface level.

Having a `flags` argument is considered a Best Practice in the kernel. Admittedly, this example is a function exposed to userspace, but the idea is clear – and you might want to use `copy_file_range` in your kmod.

What then does all this setup mean?

Performing an analysis of the CRC differences will get us some sort of δ. But it won't find things like the `readdir` change or support for new `flags`. So it is really one type of minimum δ, with no way to tell if it is also the maximum δ. In the end it is just a defensible δ.

But what you want is ε – the expressed differences. Is a 32bit readdir() cookie on ext4 for i686 a difference you care about or could even see? How about input validation on an Intel Ethernet E810? What about the `flags` on an XFS filesystem? These are real world δ examples. But if they aren't expressed in your workflow, they aren't relevant to your ε.

Calculating these differences doesn't mean *nothing*, but neither does it tell you what will be different when *you* are running this kernel in *your environment* with *your code*.

Methodology #1 (finally)

The RHEL 8.4⇨8.5 kernel transition provides a good reference case for what sorts of changes happen, how they happen, and the parity between them.

1) Download all the `kernel-core` rpms from CentOS 8 and Stream 8 relevant to the time frame
2) Run `rpm -qp --provides $rpm |grep kernel\( | sort -u > $rpm.syms`
3) Comparisons, tool of your choice

That was a lot of background, for “compare these two lists of things”, but the core concepts are really “code changes” (δ) may not equal “behavior changes” (ε) in your environment. And some code changes just aren’t traceable this way – and never will be.

Package List #1

EL 8.4 kernels:

kernel-core-4.18.0-305.el8.x86_64.rpm

kernel-core-4.18.0-305.3.1.el8.x86_64.rpm

kernel-core-4.18.0-305.7.1.el8.x86_64.rpm

kernel-core-4.18.0-305.10.1.el8.x86_64.rpm

kernel-core-4.18.0-305.12.1.el8.x86_64.rpm

kernel-core-4.18.0-305.17.1.el8.x86_64.rpm

kernel-core-4.18.0-305.19.1.el8.x86_64.rpm

kernel-core-4.18.0-305.25.1.el8.x86_64.rpm

kernel-core-4.18.0-348.el8.x86_64.rpm

kernel-abi-stablelists-4.18.0-348.el8.noarch.rpm

CentOS Stream 8 kernels:

kernel-core-4.18.0-305.el8.x86_64.rpm

kernel-core-4.18.0-305.3.1.el8.x86_64.rpm

kernel-core-4.18.0-310.el8.x86_64.rpm

kernel-core-4.18.0-315.el8.x86_64.rpm

kernel-core-4.18.0-326.el8.x86_64.rpm

kernel-core-4.18.0-331.el8.x86_64.rpm

kernel-core-4.18.0-338.el8.x86_64.rpm

kernel-core-4.18.0-348.el8.x86_64.rpm

Datapoints (abi sizes)

```
$ wc -l kabi-rhel84/kabi_stablelist_x86_64

725 kabi-rhel84/kabi_stablelist_x86_64


$ wc -l kabi-rhel85/kabi_stablelist_x86_64

725 kabi-rhel85/kabi_stablelist_x86_64


$ diff kabi-rhel84/kabi_stablelist_x86_64 kabi-rhel85/kabi_stablelist_x86_64 |
wc -l

0
```

Line 1 of these files reads : `[rhel8_x86_64_stablelist]`,

leaving 724 stable symbols listed

Datapoints (abi sizes)

```
$ rpm -qp --provides kernel-core-4.18.0-305.el8.x86_64.rpm |grep kernel\( |
sort -u |wc -l

19849

$ rpm -qp --provides kernel-core-4.18.0-348.el8.x86_64.rpm |grep kernel\( |
sort -u |wc -l

20210
```

This means we have 724 stable symbols, 19,849 total symbols in the 305.el8 kernel and 20,210 total symbols in the 348.el8 kernel on x86_64.

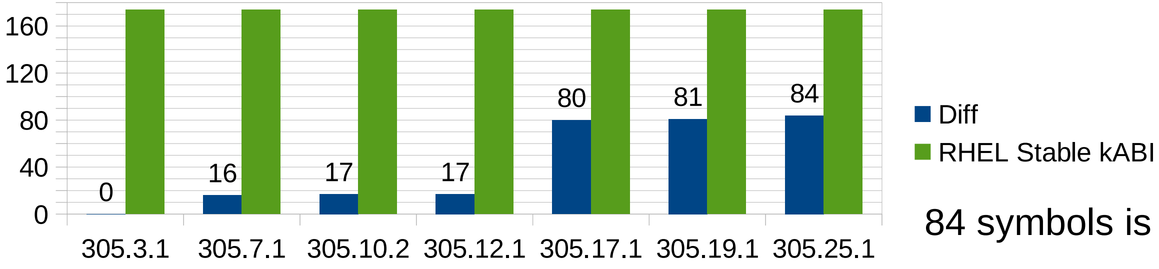
The stable symbols are ~3.5% of the total kernel symbols in RHEL 8 on x86_64.

Datapoints (RHEL 8.4 consistency to 8.4 launch kernel)

This is a count of differences from the 8.4 GA kernel to the last 8.4 before 8.5:

```
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.3.1.el8_4.x86_64.sym |wc -l
0
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.7.1.el8_4.x86_64.sym |wc -l
16
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.10.2.el8_4.x86_64.sym |wc -l
17
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.12.1.el8_4.x86_64.sym |wc -l
17
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.17.1.el8_4.x86_64.sym |wc -l
80
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.19.1.el8_4.x86_64.sym |wc -l
81
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-305.25.1.el8_4.x86_64.sym |wc -l
84
```

Remember, there are 19,849 total symbols in the 8.4 kernel



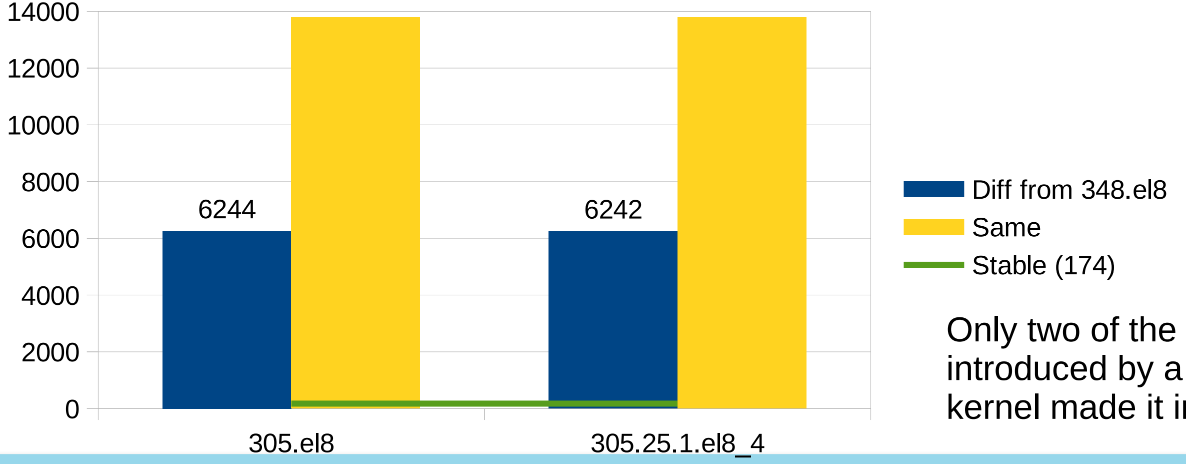
84 symbols is ~0.4%

Datapoints (RHEL 8.4 GA kernel and final kernel to RHEL 8.5 kernel)

This is a count of differences and similarities from the 8.4 GA kernel and the last 8.4 kernel against the 8.5 kernel:

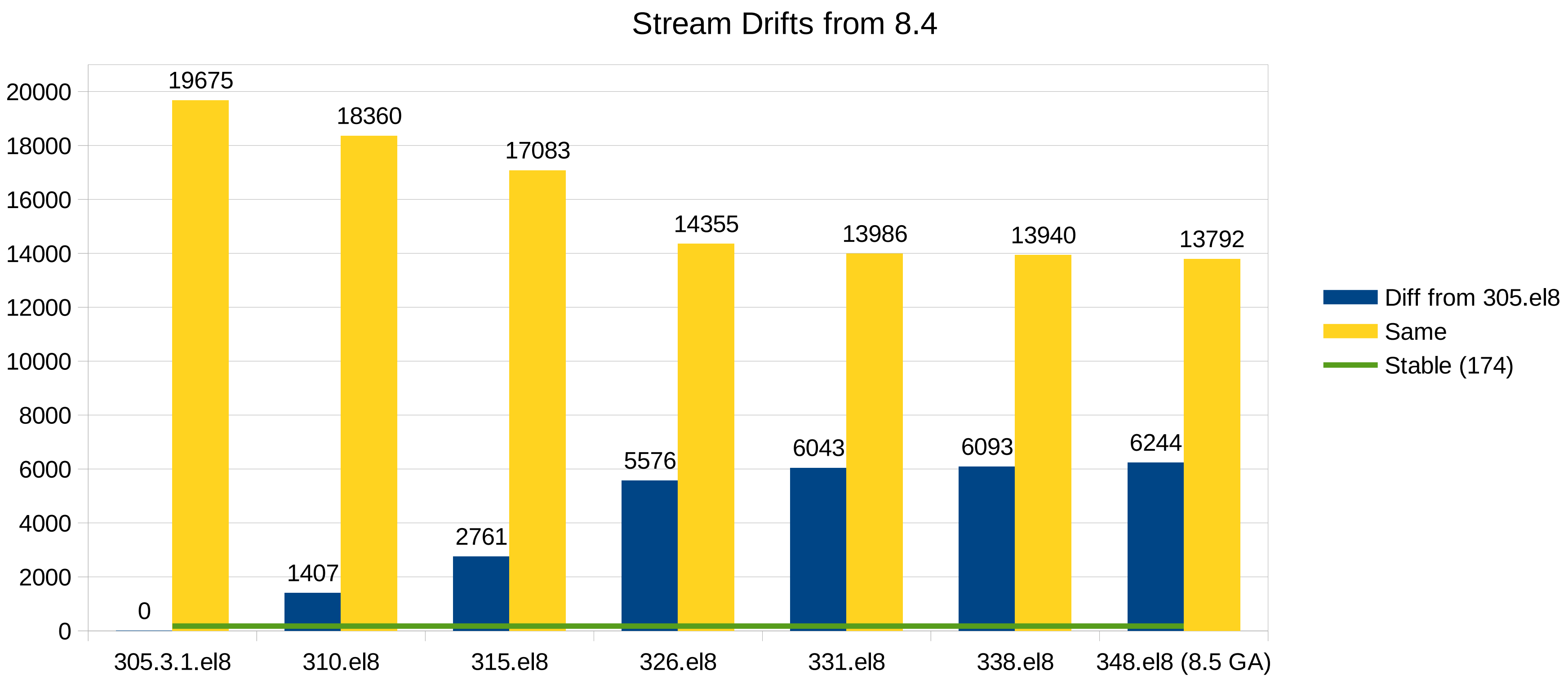
```
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.el8.x86_64.sym 4.18.0-348.el8.x86_64.sym |wc -l
6244
[riehecky@leibniz el8]$ comm -1 -3 4.18.0-305.25.1.el8_4.x86_64.sym 4.18.0-348.el8.x86_64.sym |wc -l
6242
[riehecky@leibniz el8]$ comm -1 -2 4.18.0-305.el8.x86_64.sym 4.18.0-348.el8.x86_64.sym |wc -l
13966
[riehecky@leibniz el8]$ comm -1 -2 4.18.0-305.25.1.el8_4.x86_64.sym 4.18.0-348.el8.x86_64.sym |wc -l
13968
```

This is ~31.5% of the kernel symbols changing.



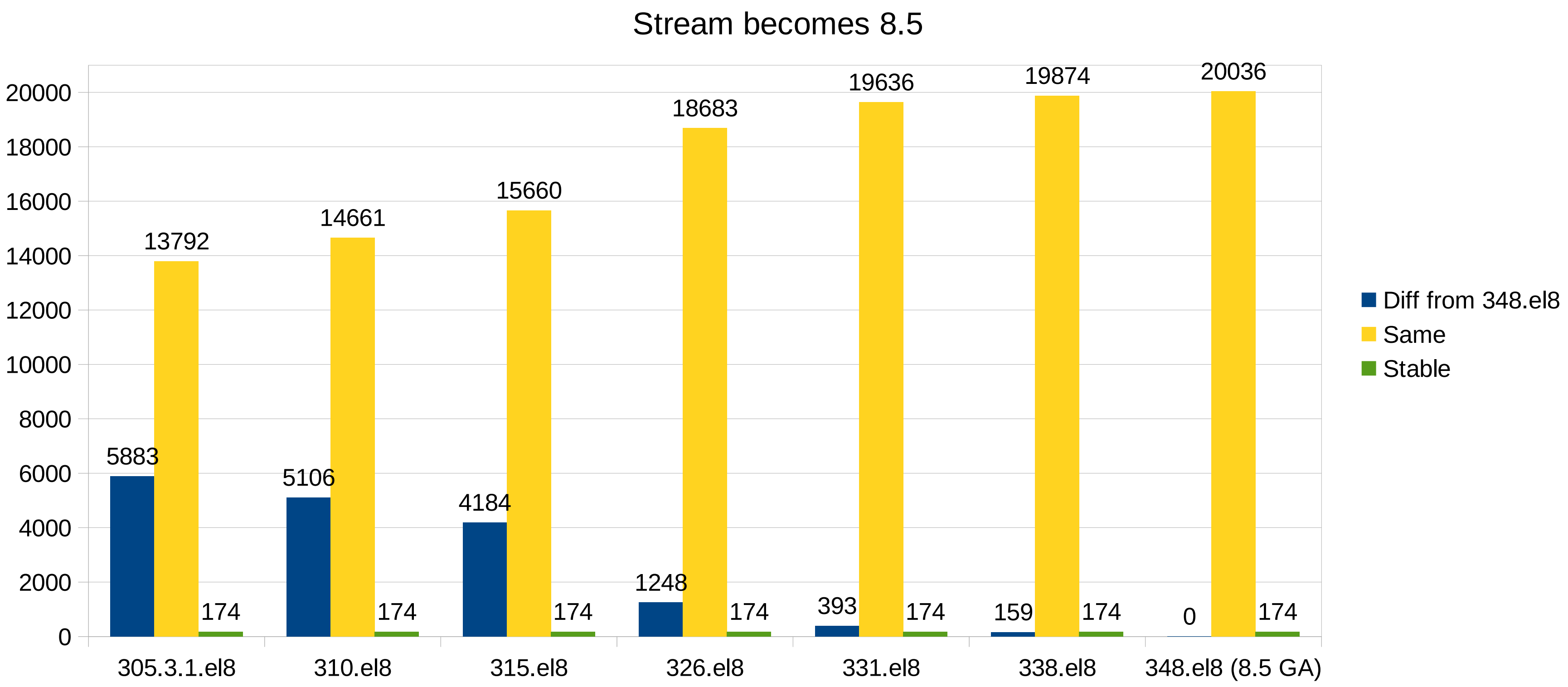
Only two of the symbols introduced by a later 8.4 kernel made it into 8.5

Datapoints (RHEL 8.4 GA against CentOS Stream)



Fermilab

Datapoints (RHEL 8.5 GA against CentOS Stream)



Fermilab

Initial Conclusions (if any)

Other than the obvious, “some kernel symbols change when the kernel changes”, I’m unclear how to draw anything useful from these numbers.

Like discussions about ε, these numbers don’t mean anything until you add a use case. There are 6244 symbol changes between RHEL 8.4 and RHEL 8.5 – each one means something. Glossing over them with a chart discards their value and any complexity in adapting to those changes.

We’re looking at the “kABI” because we want to build/load modules against it, so let’s look at some actual kmods out in the wild. These kmods do something, so they at least represent a use case – even if they are not specific to the actual workflow folks are using.

Methodology #2

We can track which kmods break and with which versions using dnf `repoclosure`.

The kmod packages I've pulled down for testing appear to target the 8.5 kernel or are the latest available.

Selection criteria are a little less critical here as we're looking for places where the packages don't work. A slight variety of build targets can help identify these locations.

Package List #2 – kmod SIG CentOS Stream 8

kmod-3w-9xxx-4.19-7.el8s.x86_64.rpm

kmod-3w-sas-4.19-5.el8s.x86_64.rpm

kmod-arcmsr-4.19-7.el8s.x86_64.rpm

kmod-exfat-5.16-1.el8s.x86_64.rpm

kmod-isci-5.0-5.el8s.x86_64.rpm

kmod-iwlegacy-5.12-8.el8s.x86_64.rpm

kmod-jme-5.6-4.el8s.x86_64.rpm

kmod-megaraid_mbox-4.18-7.el8s.x86_64.rpm

kmod-mvsas-4.18-7.el8s.x86_64.rpm

kmod-ntfs3-5.15-4.el8s.x86_64.rpm

kmod-pata_amd-5.13-5.el8s.x86_64.rpm

kmod-pata_atiixp-5.13-5.el8s.x86_64.rpm

kmod-pata_jmicron-4.18-5.el8s.x86_64.rpm

kmod-pata_via-4.19-5.el8s.x86_64.rpm

kmod-sata_nv-5.4-5.el8s.x86_64.rpm

kmod-sata_sil24-5.4-5.el8s.x86_64.rpm

kmod-sata_sil-5.4-5.el8s.x86_64.rpm

kmod-sata_sis-4.18-6.el8s.x86_64.rpm

kmod-sata_uli-4.18-5.el8s.x86_64.rpm

kmod-sata_via-5.4-5.el8s.x86_64.rpm

kmod-skge-5.7-7.el8s.x86_64.rpm

kmod-sky2-5.8-7.el8s.x86_64.rpm

kmod-wireguard-1.0.20211208-1.el8s.x86_64.rpm

kmod-xt_time-5.9-5.el8s.x86_64.rpm

Package List #2 – ELRepo EL8 kmods

kmod-3c59x-0.0-4.el8_4.elrepo.x86_64.rpm

kmod-3w-xxxx-2.26.02.003-5.el8_4.elrepo.x86_64.rpm

kmod-aacraid-1.2.1-6.el8_5.elrepo.x86_64.rpm

kmod-aic7xxx-7.0-1.el8_5.elrepo.x86_64.rpm

kmod-ath5k-0.0-6.el8_5.elrepo.x86_64.rpm

kmod-be2net-12.0.0.0-8.el8_5.elrepo.x86_64.rpm

kmod-cxgb3-1.1.5-6.el8_5.elrepo.x86_64.rpm

kmod-e100-3.5.24-5.el8_4.elrepo.x86_64.rpm

kmod-ecryptfs-0.0-4.el8_5.elrepo.x86_64.rpm

kmod-forcedeth-0.0-6.el8_4.elrepo.x86_64.rpm

kmod-ftsteutates-20190927-5.el8_4.elrepo.x86_64.rpm

kmod-hfsplus-0.1-2.el8_5.elrepo.x86_64.rpm

kmod-hpsa-3.4.20-6.el8_5.elrepo.x86_64.rpm

kmod-ib_qib-1.11-2.el8_5.elrepo.x86_64.rpm

kmod-lru_cache-0.0-3.el8_5.elrepo.x86_64.rpm

kmod-mbgclock-4.2.10-5.el8_5.elrepo.x86_64.rpm

kmod-mlx4-4.0-6.el8_5.elrepo.x86_64.rpm

kmod-mpt3sas-37.101.00.00-1.el8_5.elrepo.x86_64.rpm

kmod-mptfc-3.04.20-6.el8_5.elrepo.x86_64.rpm

kmod-mptsas-3.04.20-6.el8_5.elrepo.x86_64.rpm

kmod-mptspi-3.04.20-6.el8_5.elrepo.x86_64.rpm

kmod-mvsas-0.8.16-5.el8_5.elrepo.x86_64.rpm

kmod-qla2xxx-10.02.00.106-1.el8_5.elrepo.x86_64.rpm

kmod-r8168-8.048.03-2.el8_4.elrepo.x86_64.rpm

kmod-reiserfs-0.1-6.el8_5.elrepo.x86_64.rpm

kmod-rtl8187-0.0-5.el8_5.elrepo.x86_64.rpm

kmod-sata_mv-1.28-1.el8_5.elrepo.x86_64.rpm

kmod-usbip-0.0-9.el8_5.elrepo.x86_64.rpm

kmod-v4l2loopback-0.12.5-3.el8_5.elrepo.x86_64.rpm

Package List #2 – nvidia, OpenZFS and Lustre EL8 kmods

kmod-nvidia-418.226.00-4.18.0-348-418.226.00-3.el8.x86_64.rpm

kmod-nvidia-450.156.00-4.18.0-348-450.156.00-3.el8.x86_64.rpm

kmod-nvidia-460.106.00-4.18.0-348-460.106.00-3.el8.x86_64.rpm

kmod-nvidia-470.82.01-4.18.0-348-470.82.01-3.el8.x86_64.rpm

kmod-nvidia-495.29.05-4.18.0-348-495.29.05-3.el8.x86_64.rpm

kmod-zfs-2.0.6-1.el8.x86_64.rpm

kmod-lustre-client-2.12.8_6_g5457c37-1.el8.x86_64.rpm

The ELRepo nvidia drivers are great too; this is mostly for variety of packagers and methods (more on that later).

Package List #2 – locally built EL8 kmods

kmod-a3818-1.6.7-1.el8.x86_64.rpm

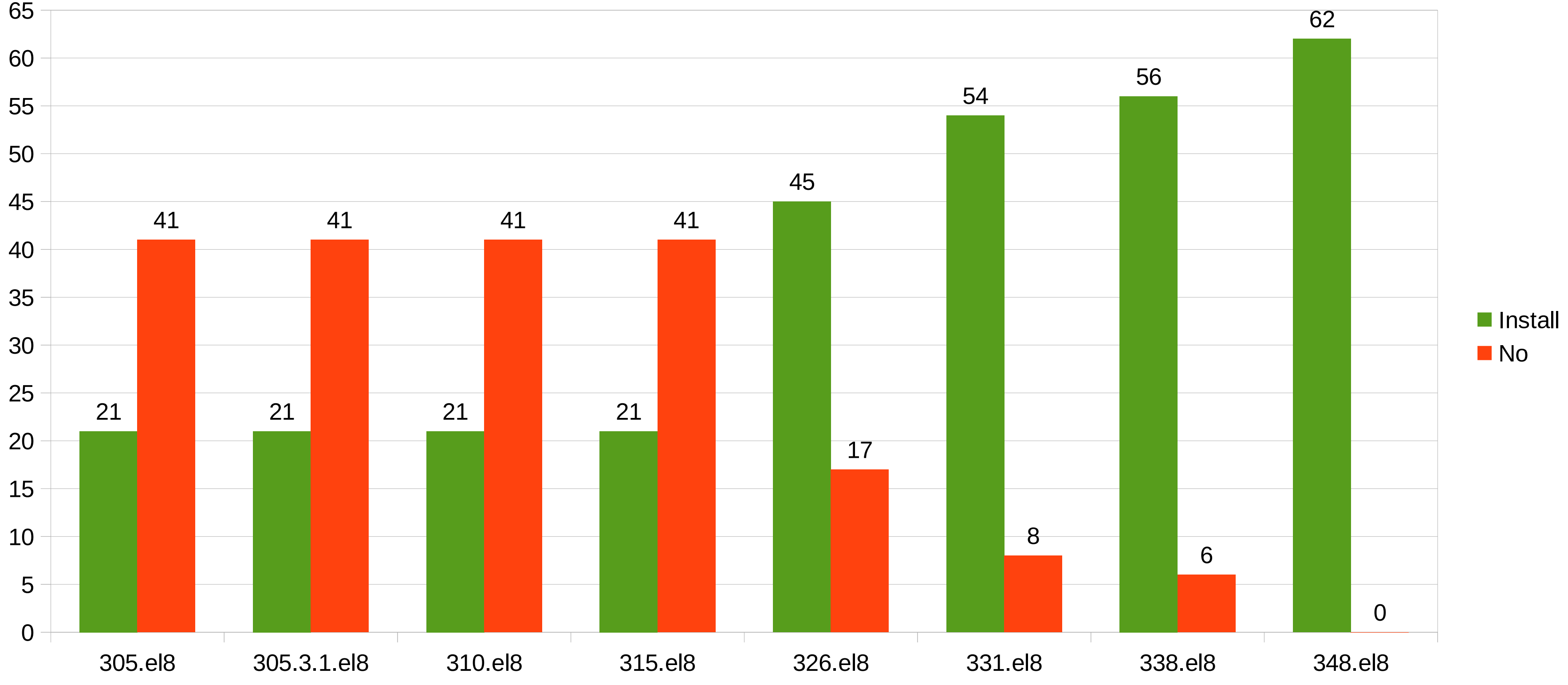
kmod-openafs-1.8.8.1-1.4.18.0_348.el8.x86_64

These kmod packages are not currently available directly to the public, but can be built up from the vendor sources without much trouble.

- https://www.caen.it/products/a3818/

- https://www.openafs.org/release/openafs-1.8.8.1.html

Results #2 – kmod repoclosures, do the test packages install with kernel



Reviewing the data from the test kmods

This data along with the earlier charts can be used to focus in on a few specifics:

- 21 packages made the transition from 8.4 to 8.5 without any need for modification or rebuilding.
- When kmod relevant symbols change, they appear to stay with the new configuration
  - `kernel(napi_enable)` and `kernel(kmalloc_caches)` for example change one time
- The `326.el8` kernel is where most of the kmod relevant symbols moved from "8.4 compatibility" to "8.5 compatibility"
  - `kmod-usbip` has the most missing symbols at this point – 12, all usb related
  - Most packages complain about `kernel(napi_enable)` changing
- nvidia is largely bypassing the kABI and focusing on Modularity Streams.
- nvidia and OpenAFS are the only packages listed on the `338.el8` kernel
- The `331.el8` kernel has only two non-nvidia/OpenAFS packages listed (`mlx4` and `usbip`)
- Largest impact kernels: `326.el8` (diff 1248 syms from 8.5) and `331.el8` (diff 393 syms from 8.5)
- Unique symbols consumed: 2112 (~10.5%)
- Changed symbols consumed: 288 (~1.4%)
- Stable symbols consumed: 566 (~2.8%)
- Unchanged symbols consumed: 1258 (~6.2%)

Conclusions - General

The approach from nvidia is closest to what the upstream kernel recommends (without getting the code into the tree). With their existing setup, they could start building against the Stream kernel today without any impact on their existing users. They are doing the right masking in `modules.yaml`. Anyone know who we should talk to over there about getting that in place? Building against the Stream kernel would guarantee their EL users a kmod on GA day and extend their reach into Stream.

The kABI users had really two Stream kernels of major interest – the 326.el8 and 331.el8 kernels. The two most-consumed symbols by my test packages are `kernel(kmalloc_caches)` and `kernel(napi_enable)`. The network subsystem has the most-consumed external symbols, with the usb subsystem coming in second.

I checked the RPMs that required a rebuild between the 305.el8, 326.el8, and 331.el8 kernels. I found a few that required multiple rebuilds between 8.4 GA and 8.5 GA: kmod-ath5k (3), kmod-ib_qib (2), kmod-iwlegacy (3), kmod-lustre-client (2), kmod-reiserfs (2), kmod-rtl8187 (3), kmod-usbip (3), kmod-wireguard(3).

Most of the test kmods recompiled without any changes on the various kernels.

Conclusions – OpenAFS and nvidia

Taking OpenAFS apart a bit further, on the public symbol list the only symbol to change is `kernel(kmalloc_caches)`. But as we've already noted, OpenAFS can get caught by things we can't clearly introspect. So the value in this data is unclear.

Taking the nvidia modules apart, they look to be using symbols that changed in the `326` kernel. The major bits are around the DRM subsystem and again the `kernel(kmalloc_caches)` symbol.

Conclusions – Recommendations for module builders

- Need a clear, well documented place for a tool to “listen” for new kernel package builds
  - I’m hopeful, when this exists, it will be added to the KMOD SIG documentation or the build system documentation.
  - I believe this is tied up with the GitForge migrations and FedMsg plans.
- Running an automatic `repoclosure` against the new kernel will kick up obvious problems

Three modules had the “worst case” under my test list (of 62 modules) and required three rebuilds over the time between 8.4 GA and 8.5 GA. This is a period of 175 days. Over that same period 288 symbols (~0.02%) relevant to my test kmods changed vs the total of 6244 symbols (~31.5%) that changed between RHEL 8.4 GA and RHEL 8.5 GA.

I’m deliberately not quantifying how much work is required to adjust to these changes. In some cases, a recompile with the same code will “just work”, in others, development effort could be required. In the examples here, none of the effort porting to the next Stream kernel is “Stream Only” because all the relevant symbol changes appear within the RHEL 8.5 GA kernel and came from the upstream kernel interfaces. If your module needs to run on a mainline kernel, you’re already doing the development work required here.

Notice of Production

Work supported by the Fermi National Accelerator Laboratory, managed and operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

Questions?