Clustering the Kernel

Alexandre Lissy
Mandriva S.A.
alissy@mandriva.com

Jean Parpaillon
Mandriva S.A.
jparpaillon@mandriva.com

Patrick Martineau
University François Rabelais Tours, Laboratory of Computer Science (EA 2101)
Team Scheduling and Control (ERL CNRS 6305)
patrick.martineau@univ-tours.fr

Abstract

Model-checking techniques are limited in the number of states that can be handled, even with new optimizations to increase capacity. To be able to apply these techniques on very large code base such as the Linux Kernel, we propose to slice the problem into parts that are manageable for model-checking. A first step toward this goal is to study the current topology of internal dependencies in the kernel.

1 Introduction

As a general goal of "applying model-checking techniques to the Linux Kernel", we studied the literature around this topic in [9]. One major conclusion is that despite the use of model-checking *under the hood* in some tools (such as Coccinelle [4]), direct application of existing model-checking tools and algorithms seems impossible because the whole kernel is too big, i.e. contains too much state.

To circumvent this limitation, we propose to work at the source code level instead of working on the model-checking algorithm itself. By breaking down the code base into smaller chunks, the number of states that must be analyzed is reduced. Before working on the slicing itself, we propose to study the current internal dependency topology of the kernel.

In the remainder of this paper, we will first make a brief presentation in section 2 of the topic discussed in [9]; then in section 3 we describe how we built the graph representing the kernel. Section 4 will focused on the analysis of what is inside the graph and what it means for us, before we conclude.

2 Model-Checking and Kernel

We are interested in answering the question: "is it possible to directly apply model-checking to the Linux Kernel code base". A literature survey for kernel-related verification, including but not limited to Linux, that might be linked to model-checking, revealed at least two major projects:

- SLAM, initiated by Microsoft
- Coccinelle

A third major project to be cited is the effort conducted by Engler et al.

2.1 A first step: compiler extensions to check rules

Using compiler extensions to check system rules has been the first major attempt to check system code. This has been performed as a fork of the GCC compiler with a matching language called METAL, which allows definition of a state machine and patterns to match in source code. Thanks to this tool, called `xgcc` [6], a first major empirical study of errors in kernel source code has been performed [5] against Linux, OpenBSD and the FLASH embedded system. Results showed that device drivers were the main hot point in term of bugs.

Those results were corroborated ten years later by the Coccinelle team as part of a similar updated survey [12] using their own tools. Newer kernels showed a real improvement since the first study.

2.2 Verification at Microsoft: SLAM

With the introduction of boolean programs [2], work started on verification of code at Microsoft. The goal was to verify correct usage of interfaces, i.e. APIs [3]. Boolean programs are an abstraction of source code where all variables are substituted with booleans: this allows for state evaluation to take place. This is code using a CEGAR (Counter-Example Guided Abstraction Refinement) loop, allowing to determine feasible paths in the source code. It allows, *in fine*, to find code-paths that are not good and hence bugs.

Results are good enough (in term of bug finding and false positive) that the tool has been included in the Windows 7 DDK as the Static Driver Verifier [1].

2.3 Coccinelle, tracking Linux bugs

The Coccinelle tool was designed to apply massive changes to APIs, defined as “collateral evolutions” [8, 10] and has been rapidly used to hunt bugs in the code. In order to apply API evolutions, the tool must work at a higher level than only source code: it is a “semantic patch” tool, i.e. instead of replacing a line by another, the code is manipulated at the level of adding or removing a parameter to a function. This is internally done by transforming the patch into a temporal logic formula, which is matched against the source code: model-checking.

Once the tool could match source code, its developers thought of using it to find bugs: the Semantic Patch Language is used to describe what to change and how; with a couple of new operators introduced, it can be used to find bugs. So naturally Coccinelle has been used to track them. It is now being used for discovering protocols, i.e. how an API should be used [7], and to track bug life cycles [11].

3 Kernel graph

The goal is to be able to study dependencies between “modules” in the source code of the kernel. We consider those modules to be the `.o` files, produced during the compilation. This assumption is done because:

- it is easy to extract symbols usages from object files using elf parsing (`readelf` for example)
- it can be matched to C source files

The directed graph is then defined as:

- Nodes are the objects files that have been analyzed
- Edges are symbols used/exported by object files
- Edges are directed.

Edge directions are defined with the source being the object file (A) that is uses a symbol and the target being the object file (B) that exports this symbol: this symbolize a dependency: $A \rightarrow B$.

The analyzed object files are limited to the architecture that the code has been built for. To date, all the processing has only been done on an AMD64 system. Also, since we limit ourselves to build the kernel using the `defconfig` and `allyesconfig` configurations, we are sensitive to changes of those configurations. The first one will only build with a subset of modules that fits for the system, while the second will enable much more code, nearly everything.

Code used to extract all the information is available at `git://git.mandriva.com/users/alissy/callgraph.git`, and is written in Python using SQLAlchemy, PyLibELF and Tulip modules.

4 Graph analysis

In this section we explain what is measured on the graph, and why. Then we present, explain and try to interpret those results.

Linux versions considered were 3.0 through 3.4, covering a time span of nearly one year.[1]

4.1 Measures

We will look at occurrence of edges in the graph: they are labeled with the symbols corresponding to the relation, so it allows us to see how much a symbol is used. From this we can derive which symbols are the most important in term of usage.

---

[1] 3.0 released July 22nd 2011; while 3.4 released May 20th 2012.

We will see how dense the graph is. Graph density is defined as follows, for a given graph $G = (E, N)$:

$$d_G = \frac{|E|}{|N| \times (|N| - 1)}$$

We will check the average path length inside the graph, that is how "far" apart two nodes are. It is computed directly thanks to the `Tulip`[2] library, which states:

> Returns the average path length of a graph, that is the sum of the shortest distances for all pair of distinct nodes in that graph divided by the number of those pairs. For a pair of non connected nodes, the shorted distance is set to 0.

We will have a look at the degrees of the graph, in and out. As a reminder, in-degree of a node is the number of exported symbols used, and out-degree is the number of imported used ones. By used, it means we can have duplicates: a symbol is used by several other ones.

To have a better view of the dependency, we propose to produce "heatmap" of the kernel dependency, with subdirectory granularity, available in section 4.7.

4.2 Graph size

Before we present the specific measures, we can already have a look at the general graph size: number of nodes, number of edges. The raw values are available in Figure 1. Variations are presented in Figure 2 and are computed using the previous one, considering the first version as a basis. Each version is compared to its first ancestor, e.g. `v3.1` against `v3.0`. In the second table, positive values indicate increases, while negative values indicate decreases.

We also measured the size of the code base using SLOCCount (v2.26) to compare the size evolution of the graph and the related code base. This information is presented in Figure 3. The evolution is computed the same way than previously explained.

A first observation we can make is that, looking at Figures 3 and 2, while the size of the code base evolution is similar between `defconfig` and `allyesconfig`, and raw numbers shows that the difference is very small, it seems not to be correlated with the evolution of nodes nor edges.

	Nodes	
Version	`defconfig`	`allyesconfig`
v3.0	1836	9593
v3.1	1842	9764
v3.2	1861	9897
v3.3	1874	10044
v3.4	1871	10172

	Edges	
Version	`defconfig`	`allyesconfig`
v3.0	51700	321463
v3.1	52390	332865
v3.2	53005	337717
v3.3	53418	344314
v3.4	53646	349271

Figure 1: Number of nodes and edges

	Nodes	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+0.33%	+1.78%
v3.2	+1.03%	+1.36%
v3.3	+0.70%	+1.49%
v3.4	−0.16%	+1.27%

	Edges	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+1.33%	+3.55%
v3.2	+1.17%	+1.46%
v3.3	+0.78%	+1.95%
v3.4	+0.43%	+1.44%

Figure 2: Variations in nodes and edges

	SLOCCount	
Version	`defconfig`	`allyesconfig`
v3.0	9614824	9612505
v3.1	9704743	9702470
v3.2	9862036	9860466
v3.3	9977312	9976172
v3.4	10120350	10119606

	Evolution	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+0.94%	+0.94%
v3.2	+1.62%	+1.63%
v3.3	+1.17%	+1.17%
v3.4	+1.43%	+1.44%

Figure 3: Code base size evolution

---

[2] http://tulip.labri.fr

4.3 Measure: Symbols occurrences

Symbols occurrences are computed simply by counting how many times a symbol is used, i.e. how many edges with this symbol exists in the graph. Raw values for kernel v3.0 are available in Figure 4; values for other versions (v3.1 to v3.4) are not provided but they are close. The table is limited to top 10. In the most used edges, throughout the versions, we can derive three categories:

- String manipulations, with `printk()` being one of the most used symbols
- Memory management, for example functions `kfree()`, `kmalloc_caches()`, `kmem_cache_alloc_trace()` and `__kmalloc()`
- Locking primitives case, including `mutex_lock()` (in the `defconfig`), `mutex_lock_nested()` (in the `allyesconfig`) and `mutex_unlock()`

Looking at the 50 most used symbols, there is not much change in the categories involved: strings see more symbols used (`sprintf()`, `str*()`); memory management sees its space extended with for example `memset()`, `memcpy()`, `_copy_from_user()` and `_copy_to_user()`; locking primitives are also in the top 50.

Extending our view from top 10 to the top 50, however, shows a difference when looking at the results on `allyesconfig`: strings functions are less present in the hall of fame, and driver-related symbols appear: `drv_get_drvdata()`, `drv_set_drvdata()`. Also, workqueues symbols (especially `__init_work()`) appear in the top 50 when looking at `allyesconfig` but not in `defconfig`.

4.4 Measure: Graph density

The raw values are available in Figures 5 and 6.

Figure 5 shows that the density for `allyesconfig` is slightly lower than that of `defconfig`, which is not a surprise considering the definition of both. A fact that is not that obvious, however, is that in `defconfig`, on the set of versions studied, density is quite stable; while in `allyesconfig` we can see that it is constantly dropping, even though the decrease is slow.

Symbol	Occurrences
`defconfig`	
_raw_spin_lock	782
_cond_resched	806
__kmalloc	846
current_task	864
mutex_lock	912
mutex_unlock	936
kmem_cache_alloc_trace	1254
kmalloc_caches	1270
printk	1658
kfree	1706
`allyesconfig`	
mutex_lock_nested	4614
mutex_unlock	4898
__kmalloc	5156
__stack_chk_fail	6258
kmem_cache_alloc_trace	6922
kmalloc_caches	6950
kfree	10152
printk	11336
__gcov_init	19014
__gcov_merge_add	19014

Figure 4: Symbols occurrences in the graph, kernel v3.0

	Density	
Version	`defconfig`	`allyesconfig`
v3.0	0.015346	0.003494
v3.1	0.015449	0.003492
v3.2	0.015313	0.003448
v3.3	0.015219	0.003413
v3.4	0.015333	0.003376

Figure 5: Graph density

To have a better understanding we looked more closely at the density inside kernel v3.0. For each subdirectory containing source code, we compare density between `defconfig` and `allyesconfig`; the values are available in Figure 6. Some directories are highly impacted: `crypto`, `drivers`, `fs`, `net`, `security`, `sound` while some other are not, or only slightly: `arch`, `block`, `init`, `ipc`, `kernel`, `lib`, `mm`.

4.5 Measure: Average path length

The raw values are available in Figure 7. A more detailed per-subdirectory overview on kernel v3.0 to v3.4 is available in Figure 8 for the `defconfig` build and in Figure 9 for the `allyesconfig` build.

In Figure 7, we observe that in both `defconfig` and

Linux v3.0	Density	
Subdir	`defconfig`	`allyesconfig`
arch	0.039320	0.035418
block	0.268398	0.281667
crypto	0.241935	0.073537
drivers	0.021583	0.002376
fs	0.063002	0.018673
init	0.291667	0.291667
ipc	0.712121	0.719697
kernel	0.122087	0.126854
lib	0.019572	0.016000
mm	0.309949	0.299454
net	0.060322	0.015070
security	0.288762	0.103541
sound	0.173263	0.024607

Figure 6: Graph density per subdirectory, v3.0

	Average Path Length	
Version	`defconfig`	`allyesconfig`
v3.0	2.410770	2.013618
v3.1	2.413076	2.013085
v3.2	2.415017	2.013867
v3.3	2.417895	2.014709
v3.4	2.429603	2.014612

Figure 7: Graph average path length

`allyesconfig` the average path length slowly increases, at least between versions 3.0 and 3.4; moreover, there is an order of magnitude of difference in the increase between both build configurations: the increment for `defconfig` is around 0.002 (although going from 3.3 to 3.4 shows an increment of 0.012), while in `allyesconfig` it is around 0.0006 with two majors points: going from 3.0 to 3.1, we have a decrease of about 0.0005 and from 3.3 to 3.4 it also decreases but only 0.00009.

Since the major difference between both consists of more drivers (not only the `drivers` subdirectory), it is trivial to assume that the reason is inside those. We propose to have a closer look at this in the table available in Figure 9, in Section 4.5.1. To have a better understanding of the "big" increase between 3.3 and 3.4 in `defconfig`, we will have a look at the details in Figure 8 in Section 4.5.2.

4.5.1 Detailed Average Path Length, kernel 3.0 to 3.4, `defconfig`

In Figure 8, we can notice:

- The `arch` subdirectory is nearly constantly decreasing, apart from the 3.3 to 3.4 evolution which shows a slight increase.
- The `block` subdirectory shows a constant average path length, with a step between 3.2 and 3.3, going from 1.80 to 2.17, before and after it is strictly the same values.
- The `crypto` and `drivers` subdirectories are evolving together, especially with 3.2 showing a light decrease on both, while they increase the rest of the time.
- For `fs` we can observe a constant decrease, although kernel 3.4 shows a noticeable increase. This is probably to be linked with the number of commits concerning `cifs` (54), `xfs` (57), `ext4` (59), `nfsd` (61), `proc` (64), `btrfs` (118), and `nfs` (181).
- While `init` is slowly increasing, `ipc` and `mm` are much more stable.
- The `lib` subdirectory shows a decrease, dropping from 1.15 to 0.96.
- Security-related subdirectory, `security`, is having a rough time, alternating between increase, decrease and stability (3.2 and 3.4 shows the same values
- The `net` subdirectory also shows an alternating behavior.
- Main part of the kernel, in the `kernel` subdirectory, is decreasing in a quite stable way, dropping from 2.0607 to 2.0417 over the studied versions.
- The `sound` part is also slowly decreasing over versions.

So, generally speaking, some parts of the kernel are "shrinking", i.e. each sub-part is getting closer to its neighbors: this is when average path length decreases. Some other parts are in expansion, with a good example being the `drivers` part. Finally, the global increase between 3.3 and 3.4 observed in the previous table can be explained by the changes in `crypto`, `drivers`, `fs` and `net`.

	Average Path Length – `defconfig`				
Subdir	v3.0	v3.1	v3.2	v3.3	v3.4
arch	2.140192	2.132726	2.118681	2.090058	2.096498
block	1.809524	1.809524	1.809524	2.169355	2.169355
crypto	1.790323	1.817204	1.802151	1.881048	1.989919
drivers	2.910024	2.911436	2.897755	2.919029	3.008717
fs	2.570742	2.532482	2.525733	2.491692	2.680926
init	1.805556	1.833333	1.833333	1.944444	1.944444
ipc	1.363636	1.348485	1.348485	1.348485	1.348485
kernel	2.060689	2.067626	2.059235	2.049458	2.041655
lib	1.155375	1.140758	1.125184	1.002295	0.964706
mm	1.914116	1.914116	1.917551	1.911837	1.921633
net	2.432165	2.359474	2.475096	2.350066	2.490345
security	2.613087	2.563300	2.832659	2.834008	2.832659
sound	2.191919	2.191287	2.181980	2.181420	2.181420

Figure 8: Graph average path length per subdirectories, v3.0 to v3.4, `defconfig`

	Average Path Length – `allyesconfig`				
Subdir	v3.0	v3.1	v3.2	v3.3	v3.4
arch	2.079778	2.073589	1.959125	1.969219	2.192387
block	1.930000	1.907692	1.907692	2.278049	2.278049
crypto	1.897335	1.897335	1.921848	1.906866	1.920599
drivers	3.005424	3.011469	3.028173	3.009538	3.000608
fs	3.037271	3.044064	3.048455	3.016527	3.007562
init	1.861111	1.861111	1.861111	1.861111	1.861111
ipc	1.363636	1.348485	1.348485	1.348485	1.348485
kernel	1.915090	1.914885	1.914908	1.914107	1.914962
lib	1.026194	0.956759	0.953852	1.248667	1.254550
mm	1.934973	1.938251	1.942359	1.958333	1.960813
net	2.855261	2.843975	2.854679	2.840436	2.836698
security	3.212210	3.227209	3.187363	3.187960	3.177200
sound	2.546933	2.495413	2.533996	2.530829	2.530232

Figure 9: Graph average path length per subdirectories, v3.0 to v3.4, `allyesconfig`

4.5.2 Detailed Average Path Length, kernel 3.0 to 3.4, `allyesconfig`

Figure 9, shows that the behavior for `arch` is not exactly the same as in `defconfig` build configuration. `crypto` increases slightly. Meanwhile `drivers` starts higher than in `defconfig`, increases slightly in 3.1, and then decreases in v3.3 to finish at a similar level than in `defconfig`. The `fs` subdirectory, however, shows an inverse behavior in `allyesconfig` than in `defconfig`, with higher average path length, increasing from roughly 2.5 to 3.0.

The `init` directory remains stable, with similar values than in `defconfig` configuration. Similarly `ipc` remains unchanged, as does `kernel`: the values are nearly constant along versions for `allyesconfig`, and shows a light difference (1.91 versus 2.05) from `defconfig`. The `lib` subdirectory also shows an inverse behavior in `allyesconfig`. The `mm` subdirectory shows a slight increase, while in `defconfig` there was a slight decrease.

The alternating behavior observed for `net` is confirmed with values ranging from 2.83 to 2.85. In `allyesconfig`, the `security` subdirectory shows a constant decrease and an important delta, ranging from 3.21 to 3.17 while it was between 2.56 and 2.83 for `defconfig`. Finally, the `sound` subdirectory is quite stable around 2.53 with a light decrease at 2.49 for kernel v3.1, exposing a similar behavior than in `defconfig` build configuration, the only difference being the values: around 2.18.

Major differences are `fs`, `security`, `sound` and `lib`. One could have expected that `drivers` showed a much more bigger difference.

4.6 Measure: Degrees

The raw values, from an aggregated subdirectory point of view are available in Figure 10 for kernel v3.0 and 11 for kernel v3.4. Those values are another point of view of the heatmap available, for example, in Figure 12. As a reminder, in-degree of a node in the graph we use maps to *exported* symbols, i.e. they are used by other nodes.

A first look at values shows that:

- Between successive versions of the kernel, there is an increase in degrees, both in and out. This is consistent with the expansion we already exposed.
- The top three consuming subdirectories are `drivers`, `net` and `fs`, in that order for `defconfig` and `drivers`, `fs` and `net` for `allyesconfig`.
- The top five consumed subdirectories are `kernel`, `drivers`, `net`, `fs` and `mm` in `defconfig`. The `allyesconfig` mode shows the same results, apart from `mm` being replaced by `lib`.

Those results can be generalized to versions from 3.0 to 3.4, even though we can notice a decrease in out-degree for kernel 3.4 in `defconfig` build configuration for the `arch` subdirectory. A closer look at this specific subdirectory shows that:

- In-degree is constantly growing, meaning more and more symbols exported.

Linux v3.0	Degrees in	
Subdir	`defconfig`	`allyesconfig`
arch	4540	16112
block	541	1621
crypto	258	907
drivers	8803	87986
fs	6097	28262
init	85	135
ipc	103	104
kernel	12876	92789
lib	4006	26397
mm	5418	24879
net	6504	29760
security	721	1403
sound	1681	11054

	Degrees out	
Subdir	`defconfig`	`allyesconfig`
arch	3489	6831
block	602	906
crypto	497	1322
drivers	17081	191946
fs	7751	42208
init	316	357
ipc	354	431
kernel	4298	6904
lib	396	1001
mm	1721	2798
net	10668	37958
security	1252	3176
sound	3155	25304

Figure 10: Graph degrees per subdirectory, v3.0

Linux v3.4	Degrees in	
Subdir	`defconfig`	`allyesconfig`
arch	4983	19822
block	633	1816
crypto	258	1087
drivers	9106	94828
fs	6224	30152
init	85	138
ipc	105	106
kernel	13336	101674
lib	4010	29268
mm	5551	25978
net	6704	31429
security	747	1580
sound	1863	11337

	Degrees out	
Subdir	`defconfig`	`allyesconfig`
arch	3421	7594
block	733	1192
crypto	515	1382
drivers	17696	207607
fs	8080	46375
init	325	374
ipc	360	464
kernel	4599	7767
lib	399	1149
mm	1776	3105
net	10960	41552
security	1271	3888
sound	3430	26482

Figure 11: Graph degrees per subdirectory, v3.4

- Out-degree is increasing-decreasing: 3489 for v3.0, 3421 for v3.1, 3608 for v3.2, 3343 for v3.3 and finally 3421 for v3.4.

4.7 Measure: Heatmaps

Heatmaps are generated from the previously presented dependency graph. It allows to more easily visualize how things are organized:

- First, we merge together nodes at a defined depth (in term of subdirectories), while keeping edges as they were originally: hence, we get the same dependencies but with a bigger granularity, more human-readable
- Then, we process all the newly-created nodes, and we count the number of edges between each pairs of nodes
- Finally, to be able to compare between versions of the kernel, we normalize things

A first look at two heatmaps, Figures 12 and 13 which are Linux v3.0 kernel's root, respectively in `defconfig` and `allyesconfig` builds. Note that color scale maximums differ at 0.16 and 0.3; we can see the same results as those presented in Section 4.4.

A closer look at the `drivers` subdirectory is available in Figure 14. A first observation is that dependencies are mainly contained inside each subdirectory: there is a thin line with variable value, but nearly always maximum, that runs for each subdirectories' intersection with itself. We can also note that there are three other lines, yet lighter: `base`, `pci` and `usb`. Those directories contains generic stuff for all drivers, or `PCI/USB` stack, hence it is normal that they are being used by a lot of other sub-directories. Other versions of the kernel (e.g. v3.4 in Figure 15) shows nearly the same behavior, only the range of values changes.

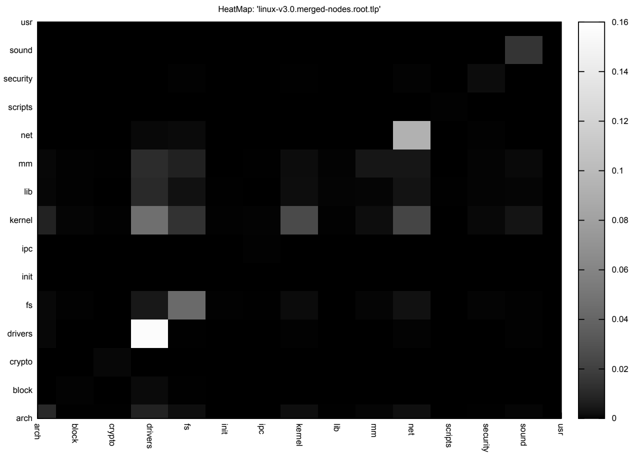

Figure 12: HeapMap of Linux v3.0, `defconfig`

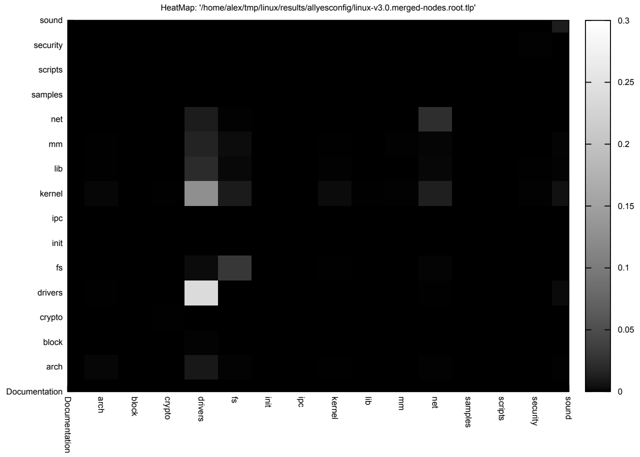

Figure 13: HeapMap of Linux v3.0, `allyesconfig`

Another kind of “heatmap” could be created, that we could call *binary heatmap*, in which we do not count the number of edges between two nodes, but only the existence of an edge between those nodes: it is an adjacency matrix.

5 Conclusion

Those few metrics allows us to have a better look at the kernel: a first interesting fact, easily readable in the heatmaps and not that surprising, is that dependencies are rather located in spots. Drivers depend mainly on stuff that concern the specific driver, a bit on generic stacks such as `base`, `pci` or `usb`, plus a couple of generic libraries in the kernel such as `mm` for memory management, `lib` for things like strings handling and `kernel` for basic stuff.

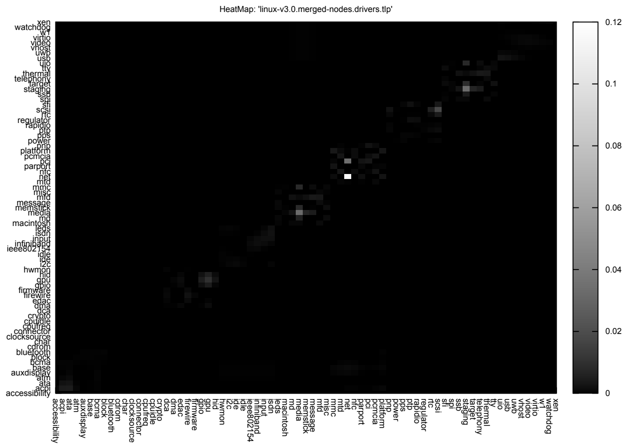

Figure 14: HeapMap of Linux v3.0, `drivers` subdirectory

A second result, which is also not surprising, is that the kernel is expanding, not only in term of code base size (this fact is well known), but we can see it also from a density and average path length: over time, at least within the time span studied, density is decreasing and average path length is increasing. This, only from a global point of view: the details in average path length and density shows that each subsystem evolves in a specific way.

There are some limitations to the present overview of the kernel. First, even if we extract all kind of supported symbols (by the terms of `libelf`) and store them with the correct type in the database, we do not (yet) make use of this kind of meta-data: is the symbol a function, a variable? When trying to cluster the kernel, this might become a good point.

Another limitation, due to the current implementation, is that we do not reproduce the “full tree” in the graph, we only assign nodes a label with the full path: this could ease a more generic and deeper analysis, especially interesting for `drivers` or `fs` subdirectories.

The current study only covers the kernel over one year, ranging from 3.0 to 3.4: it is clearly not enough to draw very generic conclusions. An attempt has been made to run the current symbols extraction over kernel ranging from 2.6.20 to 3.4; it has not been possible due to time constraints: building old kernel with recent GCC

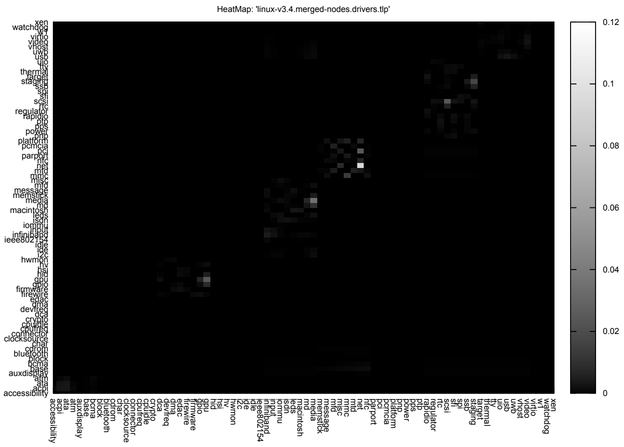

Figure 15: HeapMap of Linux v3.4, `drivers` subdirectory

seems not to be trivial, and symbols extraction for so many kernel would have required much more time than available. This is, however, an issue that must be addressed to be able to confirm the current observation over a wider sample of kernel.

A new measure that could enhance this study is clustering coefficient: we have not been able to perform this one due to time constraints. Applying the same analysis to other (big) code bases, such as Mozilla (Firefox) or LibreOffice, would be interesting.

References

[1] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *International Conference on Computer Aided Verification*, 2010.

[2] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[3] Thomas Ball and Sriram K. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[4] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009.

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. pages 73–88, 2001.

[6] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[7] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 43–52, Estoril, Portugal, July 2009.

[8] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, March 2005.

[9] Alexandre Lissy, Stéphane Laurière, and Patrick Martineau. Verifications around the linux kernel. In *Ottawa Linux Symposium (OLS 2011)*, Ottawa, Canada, June 2011.

[10] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, April 2006.

[11] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with herodotos. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 169–180, New York, NY, USA, 2010. ACM.

[12] Nicolas Palix, Suman Saha, Gaël Thomas, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. Research Report RR-7357, INRIA, 08 2010.