Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers

Leonardo Passos *
University of Waterloo, Canada
lpassos@gsd.uwaterloo.ca

Jesús Padilla
University of Waterloo, Canada
jpadillagaeta@gsd.uwaterloo.ca

Thorsten Berger
University of Waterloo, Canada
tberger@gsd.uwaterloo.ca

Sven Apel †
University of Passau, Germany
apel@uni-passau.de

Krzysztof Czarnecki
University of Waterloo, Canada
kczarnec@gsd.uwaterloo.ca

Marco Tulio Valente
Federal University of Minas Gerais, Brazil
mtov@dcc.ufmg.br

Abstract

Feature code is often scattered across wide parts of the code base. But, scattering is not necessarily bad if used with care—in fact, systems with highly scattered features have evolved successfully over years. Among others, feature scattering allows developers to circumvent limitations in programming languages and system design. Still, little is known about the characteristics governing scattering, which factors influence it, and practical limits in the evolution of large and long-lived systems.

We address this issue with a longitudinal case study of feature scattering in the Linux kernel. We quantitatively and qualitatively analyze almost eight years of its development history, focusing on scattering of device-driver features. Among others, we show that, while scattered features are regularly added, their proportion is lower than non-scattered ones, indicating that the kernel architecture allows most features to be integrated in a modular manner. The median scattering degree of features is constant and low, but the scattering-degree distribution is heavily skewed. Thus, using the arithmetic mean is not a reliable threshold to monitor the evolution of feature scattering. When investigating influencing factors, we find that platform-driver features are 2.5 times more likely to be scattered across architectural (subsystem) boundaries when compared to non-platform ones. Their use illustrates a maintenance-performance trade-off in creating architectures as for Linux kernel device drivers.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.2.8 [*Software Engineering*]: Metrics; D.3.4 [*Programming Languages*]: Processors—Preprocessors

Keywords Pre-processor, Linux kernel, Feature, Scattering

* Funded by CAPES, grant BEX 0459-10-0.

† Funded by DFG, grants AP 206/4, AP 206/5, and AP 206/6.

MODULARITY'15, March 16–19, 2015/Fort Collins, CO, USA.

http://dx.doi.org/10.1145/

1. Introduction

Scattering of feature code is commonly stated as an undesirable situation [15, 28, 29, 42]. Scattered features are not implemented in a modular way, but are rather spread over the code base, possibly across multiple subsystems. The intermingling of scattered features with different implementation parts can lead to ripple effects, while requiring developers to be kept in constant sync, hindering parallel development. Thus, scattered features may significantly increase the maintenance effort of a system [3, 40]. Yet, feature scattering is common in practice [30, 31, 39].

Feature scattering allows developers to overcome design limitations when extending a system in unforeseen ways [40], or when circumventing modularity limitations of programming languages, which impose a dominant decomposition [2, 43, 44]. In other cases, the cost of modularizing features might be initially prohibitive or simply too difficult to be handled in practice [26]. In contrast, feature scattering requires little upfront investment [3], although maintenance costs may rise as the system evolves.

In all cases, many long-lived and large software systems have shown that it is possible to achieve continuous evolution, while accepting some extent of feature scattering. Examples span different domains, including operating systems, database management systems, text editors, and others [30, 38, 39].

Surprisingly, there are no empirical studies investigating feature scattering in the evolution of large and long-lived software systems. Such studies would be key in creating a widely accepted set of practices to govern feature scattering and may eventually contribute to the formulation of a general scattering theory, which could serve as a guide to practitioners—for instance, in identifying implementation decay [37], assessing the maintainability of a system [17], identifying scattering patterns [18], or setting practical scattering thresholds [39].

To contribute to a deeper understanding of feature scattering and its evolution, we conduct a longitudinal case study of one of the largest and long-living software systems in existence today: the Linux kernel. Its features manifest in terms of configuration options that users select when generating customized kernel images. Our analysis investigates feature scattering in almost eight years of the kernel's 20 years of evolution. In these eight years, the kernel growth has been steady, growing from 4,752 to 13,165 features. Of these, a large portion is said to be scattered [38].

Due to the sheer size of the Linux kernel, we scope our analysis to features in the *driver* subsystem, which we identified as the largest and fastest growing kernel subsystem (see Sec. 3). We analyze

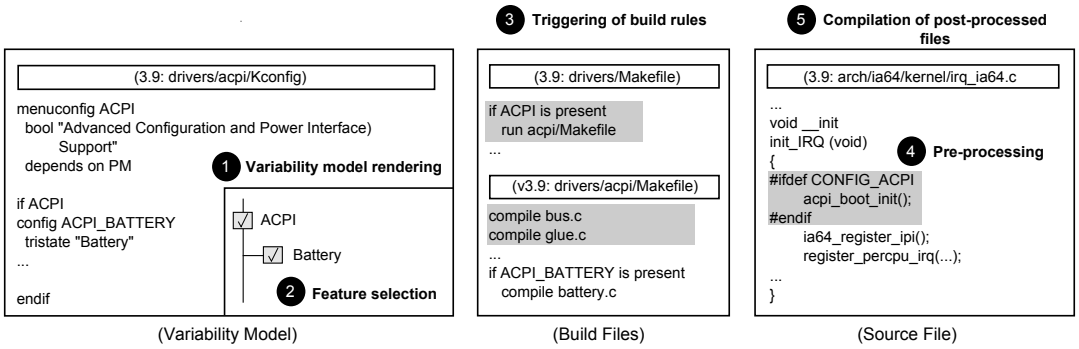


Figure 1: Binding of different kernel artifacts

scattering of driver features within and across the device-driver subsystem. Our study is guided by the following four research questions:

RQ 1) How does the growth of scattered features differ from non-scattered ones? We analyze the relative and absolute growths of scattered and non-scattered driver features, comparing the evolution of the two kinds—for instance, to understand whether the proportion of scattered features is increasing, decreasing, or stable over time.

RQ 2) How does the growth of locally scattered features differ from globally scattered ones? We analyze the relative and absolute growths of driver features that are (i) scattered within the *driver* subsystem only (local scattering), and (ii) of those that are scattered across at least another subsystem (global scattering). We compare both growth rates and aim at understanding how scattering is related to the kernel's architecture.

RQ 3) How does the extent of feature code scattering evolve over time? We analyze the extent (degree) of the scattering of feature code, aiming at understanding the underlying distribution and possible thresholds. We also want to assess how the scattering degree relates to local and global scattering.

RQ 4) What are possible factors influencing scattering of feature code? We formulate and test hypotheses about factors influencing scattering. To this end, we collect and classify 170 features (10 % of all scattered driver features), identifying possible characteristics affecting where a feature is scattered across (local versus global scattering) or that lead to higher scattering degrees.

Our contributions comprise:

- A *large dataset covering almost eight years of the evolution of feature code scattering* extracted from the Linux kernel repository. The dataset can be used as a replication package, a benchmark for tools, and for further analyses.
- *Descriptive statistics* aimed at understanding the state-of-practice of feature scattering in the Linux kernel.
- An inspection and classification of 170 scattered driver features, from which we test hypotheses to verify possible factors influencing where a feature is scattered across or its scattering degree.
- A discussion and explanation of the underlying results, with insights to guide future investigations.
- An *online appendix* [1] with further details on our dataset, scripts to analyze the data, and additional statistics.

2. Background

This section discusses how the Linux kernel explicitly represents its features, in addition to contextualizing the kernel evolution. We also set terminology and definitions for the remainder of the paper.

2.1 Feature Representation

Features in the Linux kernel are explicitly declared in a variability model written in the Kconfig language [6, 27]. These features are referenced in build rules of Linux's Makefiles and in C pre-processor directives, controlling the compilation of entire source files or fragments therein, respectively. Henceforth, we refer to such fragments as *extensions*. As summarized in Table 1, the kernel's code base comprises mostly C implementation and header files.

To illustrate how the variability model, Makefiles, and C files bind together, consider the *Advanced Configuration and Power Interface* (ACPI) driver. ACPI is an industry standard to manage power consumption of hardware devices [22]. Figure 1 illustrates the steps involved in configuring the ACPI feature, with excerpts of each file type. First, an interactive configurator renders the Linux kernel variability model (step 1). From the rendered model, users select features of interest (step 2). Once the user is done with selecting features, the build process triggers build rules (illustrated as gray boxes) that conditionally compile specific source files matching the feature selection (step 3). In our example, when feature `ACPI` is selected, the build process enters the `acpi` directory and executes its Makefile, which triggers the compilation of `bus.c` and `glue.c`. Note that compiling `battery.c` further requires selecting `ACPI_BATTERY`. Prior to a file compilation, the build process invokes the C pre-processor (step 4) to resolve all macro references and conditional pre-processor directives —`#if`, `#ifdef`, `#ifndef`, `#elif` (henceforth, generically called *ifdefs*). In our example, if a user builds the kernel for the IA64 CPU, selecting `ACPI`[1] causes the C pre-processor to include a call to `acpi_boot_init` (shown in gray) inside the implementation of `init_IRQ` in `irq_ia64.c`. After pre-processing, the resulting source files are compiled into

[1] By convention, feature macros in the Linux kernel are prefixed with `CONFIG_` to distinguish them from other macros.

Table 1: Distribution of file types (averages over the whole evolution)

File type	Average (%)
C implementation file	43
C header file	39
Assembly	4
Other	14

object code (step 5) and eventually linked into the kernel binary image or a *loadable kernel module* (LKM).

LKMs are dynamically loaded at runtime—either upon user request, as a dependency of another LKM, or when the operating system identifies a hotplugged device, for which it must load the supporting device-driver module (if any). Not all driver features can become an LKM, except those having the feature type "tristate" in the variability model (e.g., `ACPI_BATTERY`). Three possible values can be selected for a tristate feature: `y` (compile into kernel image), `m` (compile as LKM), or `n` (absent). Features that do not result in LKMs are either Boolean (e.g., `ACPI`), with values `y` (present) or `n` (absent), or value-type features, such as integer, string, and hex (not shown in the example).

The Linux kernel is a highly configurable software system [12, 32, 38, 41], meaning that users can derive customized variants (kernel images) by selecting particular features of interest. The variability of the kernel is either resolved at build-time, by preprocessing *ifdefs* and static linking, or at runtime (e.g., when loading/unloading LKMs).[2]

2.2 Kernel Evolution

The Linux kernel evolves continuously. Fig. 2 summarizes its growth by source lines of code (SLOC) and number of features. As Figure 2a shows, the code base has increased by $159\,\%$ since the first release recorded in the kernel's git repository (v2.6.12, June 2005),[3] with a growth of $2.6 \pm 1.5\,\%$ between two consecutive stable releases. The short-hand $2.6 \pm 1.5\,\%$ denotes an arithmetic mean of $2.6\,\%$ with standard deviation of $1.5\,\%$. In the remainder of this paper, the mean (or average) should always be understood as the arithmetic mean. The kernel's feature set, shown in Fig. 2b, displays a similar trend, and strongly correlates with SLOC growth (Pearson product-moment correlation $r = 0.996$).[4] Since v2.6.12, it increased by $177\,\%$, growing $2.8 \pm 1.4\,\%$ between stable releases. The latest kernel release in our analysis (v3.9, April 2013) contains over 13,000 features implemented in more than 33,000 C files, which amount to over 10 million SLOC. These C files contain over 34,000 *ifdefs* that explicitly refer to at least one feature in the variability model.

2.3 Feature Scattering

We consider a feature scattered when it is not implemented in a modularized way, but rather distributed over multiple extensions in the code base. We trace these extensions by identifying *ifdefs* that reference the corresponding feature. Thus, our measurement of scattering is based on the declaration of features in the variability model and their syntactic reference in code—both as defined by the original developers. This notion of feature scattering captures the number of potential places that a developer may change upon changing a feature of interest [39].

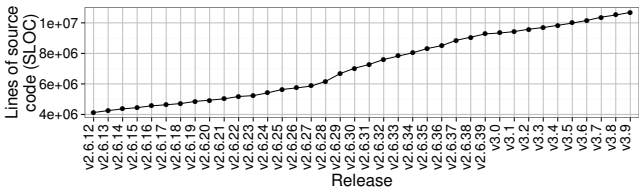
3. Methodology

This section describes our methodology of collecting the evolution data of feature code scattering and of the respective analyses.

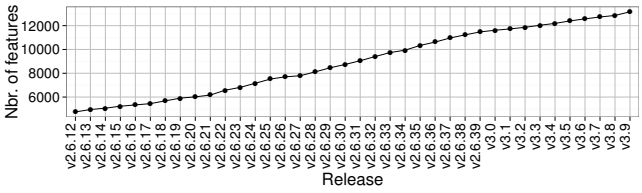
3.1 Scoping

In our study, we concentrate on driver features, that is, features defined in the *driver* subsystem of the kernel. This decision relies on existing work stating that the Linux kernel evolution is mainly driven by the evolution of its device drivers [16, 20, 24, 32, 38], and on our own analyses (explained shortly).

Setting the scope to features in the *driver* subsystem requires us to distinguish them from features of other subsystems. Next, we explain how we perform such distinction.



(a) Kernel growth in SLOC



(b) Kernel growth in number of features

Figure 2: Kernel growth evolution

3.2 Identifying Driver Features

To distinguish driver features from features of other subsystems, we first slice the kernel according to its constituent subsystems. According to Corbet et al. [9], there are seven major subsystems in the kernel: *arch* (architecture dependent code), *core* (scheduler, IPC, memory management, etc), *driver* (device drivers), *firmware* (firmware required by some device drivers), *fs* (file system), *net* (network stack), and *misc* (miscellaneous).

Greg Kroah-Hartman, the main developer of the Linux kernel stable branch,[5] provides a mapping between files in the code base of the Linux kernel and the subsystems reported by Corbet et al.[6] We take Hartman's mapping to be expert knowledge, reusing it without modifications.

Once we apply Hartman's mapping to identify the subsystem of a given file, we consider the subsystem of a feature's declaring Kconfig file as the feature's subsystem. Some features in the *driver* subsystem, although very few ($0.65 \pm 0.46\,\%$), are also declared in other subsystem(s) (e.g., inside *core*). As we cannot decide which subsystem should these features be mapped to, we exclude such driver features from analysis.

Once we distinguished the unique features in each kernel subsystem, we were able to confirm that the Linux kernel is actually driven by the evolution of driver features. As Fig. 3 shows, the *driver* subsystem is not only the largest in number of features, but also the fastest growing.

3.3 Data Collection

The data-collection procedure follows the process shown in Fig. 4. With a cloned repository of the Linux-kernel source code in place, we query the kernel's source management system (git) to list all

[2] Other kinds of runtime variability also exist, such as changing the attributes of a device driver through the sysfs virtual filesystem. See [10, 46] for further details.

[3] `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`

[4] Since our sample is a snapshot of the kernel population, covering all releases from v2.6.12 to v3.9, the correlation coefficient is necessarily significant.

[5] `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS`

[6] `https://raw.github.com/gregkh/kernel-history/master/scripts/genstat.pl`

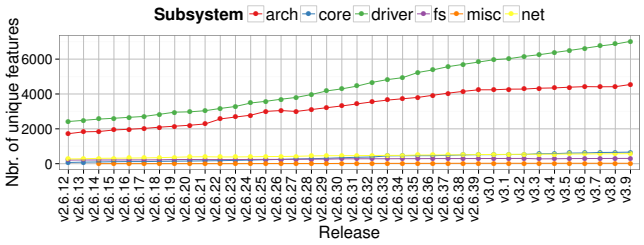


Figure 3: Feature distribution per subsystem

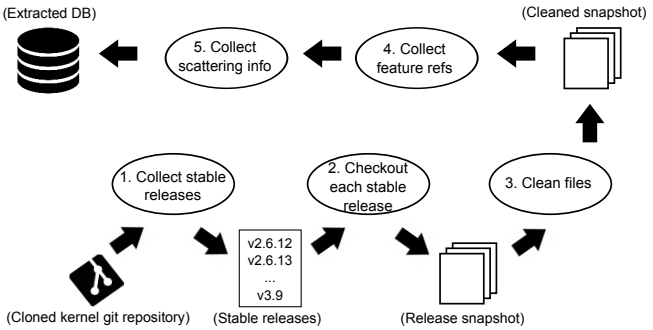


Figure 4: Data extraction process

release tags. From the listing, we filter the stable-release identifiers (step 1).[7] We then check out each stable release (step 2), setting the repository to a particular release snapshot. In the checked-out release, we list all C implementation and header files therein and clean them by removing empty lines and comments, and by transforming multilines[8] into single ones (step 3). We also eliminate strings in the source code as a means to facilitate pattern matching when mining feature references across the code (step 4). Finally, we collect metadata of each identified reference (step 5), including the name of the file in which a feature is referred, the line in which the reference occurs, and the associated *ifdef* pre-processor directive. All feature references and their associated metadata are then stored in a relational database. For any given feature reference, there exists an associated file record in the database, which in turn links to a kernel subsystem in a given stable release.

All steps in our process are fully automated and are currently supported by extensions made to a toolset [36] we developed previously. For further details on the collected data, the database schema, pointers to download our dataset, and associated scripts, we refer to our online appendix [1].

3.4 Data Analysis

To answer our research questions, we issue SQL queries through the R statistical environment, which we connect to our database [8, 21]. Then, we plot the results and perform different statistical analyses according to each research question (we defer details to Sec. 4).

In our analysis, we measure the scattering degree (*SD*) of a driver feature *ft* in terms of its scattering degree at each implementation and header C file f in a set of target subsystems S, i.e.:

$$SD(ft, S) = \sum_{s \in S} \sum_{f \in s} SDF(ft, f) \quad (1)$$

where $SDF(ft, f)$ is the number of *ifdefs* (`#if`, `#ifdef`, `#ifndef`, `#elif`) in f referring to *ft*. This is an alternative, yet equivalent, definition to how other researchers measure scattering [39].

The *SD* metric falls under the umbrella of absolute metrics that count the number of source code entities relating to a given feature. In contrast, relative metrics assess feature-scattering relative to the code size of extensions [13]. Existing research [14] comparing absolute metrics with relative ones shows that the former correlate better with defects, which justifies our choice for the *SD* metric.

We identify a feature as *scattered* when its *SD*-value is at least 2. A single extension ($SD = 1$) does not qualify a feature to be scattered, as it has no spread in the source code. In our previous example (see Fig. 1), `ACPI` is a scattered feature; in addition to the reference in the *ifdef* inside `init_IRQ`, it also has 110 corresponding *ifdefs* elsewhere.

4. Results

This section reports the results of our respective research questions, which aim at understanding the evolution of scattered versus non-scattered features (*RQ1*), of scattering within and across subsystem boundaries (*RQ2*), of scattering degrees (*RQ3*), and possible causes of the observed scattering and scattering degrees (*RQ4*).

4.1 Scattered versus Non-Scattered Features

RQ 1) How does the growth of scattered features differ from non-scattered ones?

To answer this question, we plot the proportion of scattered driver features in each kernel release, along with their absolute number. In both cases, we compare the growth rate of scattered driver features with the evolution of non-scattered ones. Figure 5 displays both plots, with summary statistics provided in the Tables 2 and 3. When applying Eq. 1 to identify scattered features, we take S as the union of all subsystems in the Linux kernel.

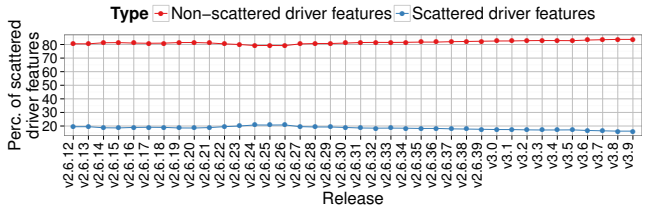
On average, $18 \pm 1.2\,\%$ of driver features are scattered in any given release, with a maximum of $21\,\%$ and a minimum of $16\,\%$. The average proportion is stable over time, although a decreasing trend starts from release v2.6.26. In absolute terms, the number of scattered driver features grows by $2.5 \pm 2.4\,\%$ between each pair of consecutive stable releases. Since the first release under analysis (v2.6.12), the number of scattered driver features has grown by $142\,\%$, as given by the *Diff* statistic.[9] In release v3.9, the kernel has over 1,000 scattered driver features. The latter, however, grows almost six times slower when compared to non-scattered driver features, as given by the ratio of their regression line slope coefficients. Moreover, the absolute growth of scattered driver features is not monotonic, with three small periods of decrease: v2.6.13–v2.6.14, v2.6.26–v2.6.27, and v3.5–v3.6.

The collected data indicate that the kernel architecture allows most driver features to be incorporated without causing any scattering. Some driver features, however, do not fit well into this architectural model and are thus scattered across the source code. Moreover, the proportion of scattered driver features is nearly constant, which may indicate that it is an evolution parameter actively controlled throughout the kernel evolution.

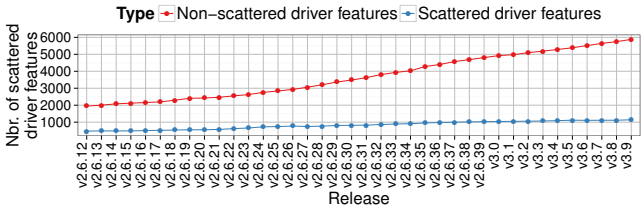
[7] Unstable releases are suffixed with *-rc* (e.g., v2.6.32-rc1), whereas stable ones are not.

[8] Multilines end with the '\' character. They spread many physical lines, but are interpreted as a single one by the C compiler.

[9] The percentage difference (*Diff*) of two non-percentage values x_2 and x_1 is $100 \times (x_2 - x_1)/x_1$. If x_2 and x_1 are percentages, the *Diff*-value is simply $x_2 - x_1$. When calculating *Diff* for a given metric (e.g., number of scattered driver features), we take x_2 to be the metric value at the last inspected kernel release (v3.9), whereas x_1 is the metric value for the first release (v2.6.12).



(a) Relative growth



(b) Absolute growth

Figure 5: Growth of (non-)scattered driver features

Table 2: Summary statistics of (non-)scattered driver features (relative)

Type	Min	Max	Avg	Diff	Slope
Scat.	16.23 %	20.79 %	18.44 ± 1.2 %	-3.22 %	-0.08
Non-scat.	79.21 %	83.77 %	81.56 ± 1.2 %	3.22 %	0.08

Table 3: Summary statistics of (non-)scattered driver features (absolute)

Type	Min	Max	Avg	Diff	Slope
Scat.	471	1,140	819.45 ± 228.88	142.04 %	20.4
Non-scat.	1,949	5,880	3,702.87 ± 1,280.49	201.69 %	114.36

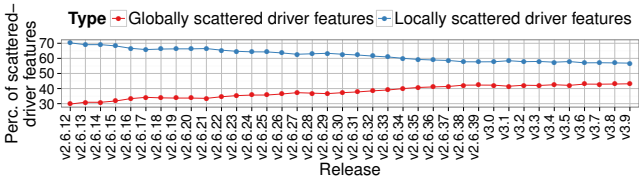
4.2 Local versus Global Scattering

With the next research question, we investigate to what extent the scattering of driver features is local and to what extent it is global. A globally scattered driver feature has at least one associated *ifdef* in an implementation or header C file that is not in the *driver* subsystem. In the case of a locally scattered driver feature, referring *ifdefs* occur only in source files in the *driver* subsystem. Ideally, most scattering should be local, contributing to internal cohesion, while decreasing coupling.

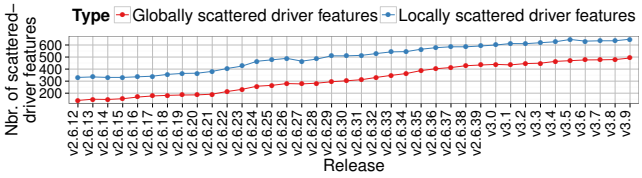
RQ 2) How does the growth of locally scattered features differ from globally scattered ones?

The growth of locally scattered driver features varies along the Linux kernel evolution. Nonetheless, it dominates the growth of globally scattered driver features, both proportionally and in absolute numbers. Figure 6 shows the corresponding plots, with summary statistics provided in Tables 4 and 5.

In release v2.6.12, the proportion of locally scattered driver features is 70 %—the highest across all releases. Immediately after, the proportion follows a steady decrease, which stabilizes around 57 % from v2.6.38 onwards. In the latest release (v3.9), the percentage of locally scattered features is 56.8 % (648 absolute). The stabilization of local scattering causes a stabilization of globally scattered driver features at 43 %. The latter, however, was preceded by an increasing trend. In absolute terms, the number of globally scattered driver features grows at a faster rate than locally scattered ones, as given by their corresponding slope coefficients. Consequently, their relative difference decreases over time, resulting in the funnel shape of Fig. 6a.



(a) Relative growth



(b) Absolute growth

Figure 6: Growth of locally and globally scattered driver features

Table 4: Summary statistics of locally and globally scattered driver features (relative)

Type	Min	Max	Avg	Diff	Slope
Local	56.84 %	70.28 %	62.13 ± 4.12 %	-13.43 %	-0.36
Global	29.72 %	43.16 %	37.87 ± 4.12 %	13.43 %	0.36

Table 5: Summary statistics of locally and globally scattered driver features (absolute)

Type	Min	Max	Avg	Diff	Slope
Local	331	648	500.08 ± 110.89	95.77 %	9.82
Global	140	492	319.37 ± 118.53	251.43 %	10.58

The relative and absolute dominance of local scattering contributes to internal cohesion within the *driver* subsystem. We conjecture that it eases maintenance, as local scattering requires less synchronization across subsystems. Nonetheless, it is interesting to see that the gap between the proportions of locally and globally scattered features has consistently decreased, with a growing proportion of globally scattered driver features. Consequently, there is an increasing dependency from other subsystems to driver features. Although the latter may indicate an evolution decay, it does not seem to hinder the Linux kernel growth. As we showed in Sec. 2.2, the

Table 6: Relative and absolute growth of scattered outlier features

Type	Min	Max	Avg	Diff	Slope
Relative	1.09 %	3.76 %	2.25 ± 0.71%	2.2 %	0.05
Absolute	7	42	19.61 ± 10.84	500 %	0.89

kernel has grown at a similar pace between each pair of consecutive releases. Thus, we interpret the stabilization of the proportion of globally scattered driver features as an effort to control its preceding growth trend. Hence, $43\,\%$ seems a current upper limit kept by Linux kernel developers.

4.3 Scattering Degrees

RQ 3) How does the extent of feature code scattering evolve over time?

To answer this question, we plot the scattering degrees (*SD*) of all scattered driver features at each kernel release. When measuring *SD* (see Eq. 1), we take the target set of subsystems (S) as the union of all subsystems in the kernel. The boxplot in Fig. 7, which is adjusted for skewness [23], shows that $50\,\%$ of all scattered driver features have a low scattering degree, with $SD \leq 4$ across all stable releases. Above the $50\,\%$ of the distribution, however, the scattering of features considerably increases. In the third quartile (up to $75\,\%$ of the distribution), *SD*-values practically double, lying between seven and eight. In the remaining $25\,\%$, the highest *SD*-values that are not outliers range from 34 to 55, as indicated by the top whiskers. In this range, the average *SD*-value is 44 ± 5.3. Above the top whiskers, outliers (shown as dots) have high *SD*-values, with a minimum of 35 and a maximum of 377 (median of 63). As the kernel evolves, outliers have grown in absolute numbers as well as relatively. Figure 8 displays the corresponding graphs, with summary statistics provided in Table 6. In absolute numbers, outliers show a $500\,\%$ increase, with as little as 7 features in release v2.6.12 and 42 in v3.9. Relatively, however, the *Diff* between the first and last release is only $2.2\,\%$.

The analysis of the *SD*-values of scattered driver features indicates a skewed distribution. In the kernel's evolution, $75\,\%$ of *SD*-values are small (4) to medium (8). A dispersion, however, occurs in the remaining $25\,\%$ (values 34–377), pushing the distribution tail to the right. Consequently, the distribution is skewed to the right, increasing the difference between a typical *SD*-value (4) and the mean (8). In such settings, the mean is a not robust statistic. Instead, practical scattering limits should be relative (e.g., $75\,\%$ of the features should have $SD \leq 8$), rather than a single value to which all features would adhere to.

To ascertain the observed skewness, while summarizing how unevenly *SD*-values are distributed among scattered driver features, we calculate the Gini coefficient [45] for each kernel release. The Gini coefficient is a popular summary statistic in economics, measuring the inequality of wealth (e.g., the value of a software metric, such as *SD*) among the individuals (e.g., features) of a population. Its value is in the range of zero and one; zero means a perfect equality, where all individuals have the same wealth. A high value, in contrast, denotes an uneven distribution.

Figure 9 shows the evolution of the Gini coefficient in the Linux kernel evolution. The coefficient follows a decreasing trend in the first 12 releases, meaning that *SD* is more evenly distributed. From release v2.6.23 onwards, an increasing trend can be observed, indicating that *SD* is more concentrated towards a particular set of features. The absolute difference between the coefficients in v2.6.23 and v3.9, however, is only 0.06, which indicates that *SD* distribution does not vary considerably. At all times, the Gini coefficient is closer to one than to zero, confirming the observed right-skewness.

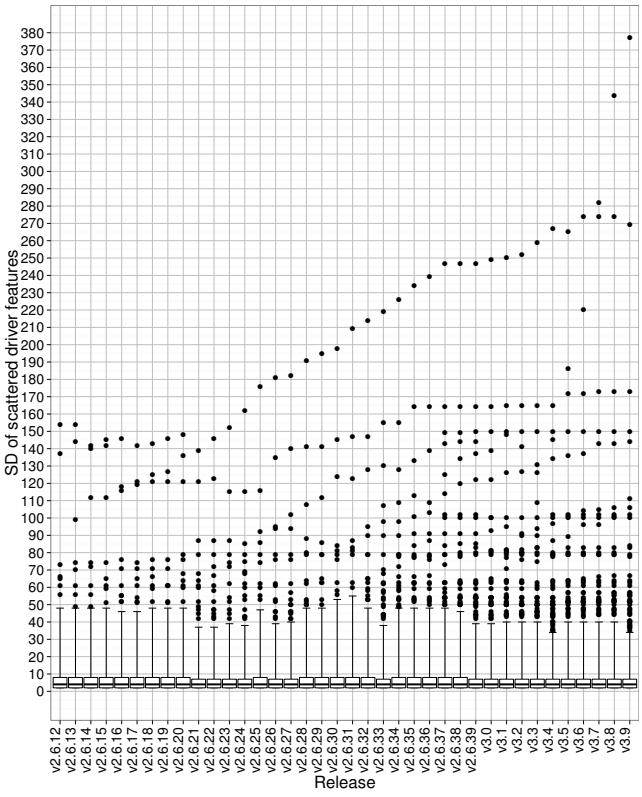


Figure 7: *SD*-values of scattered driver features

Table 7: Summary statistics of the average *SD*-value of scattered driver features

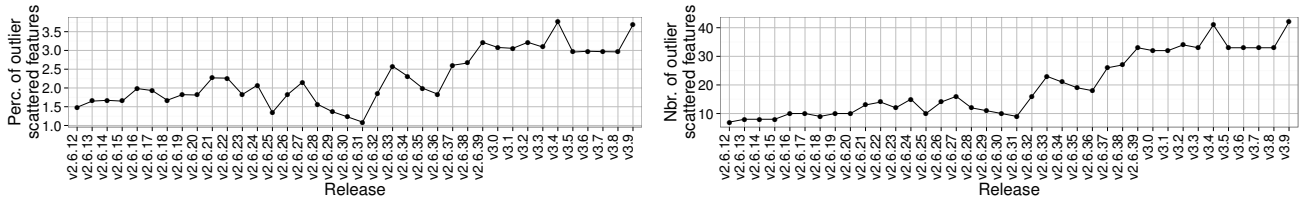
Type	Min	Quartile 1	Median	Quartile 3	Max
Local	2	2	3	6	84.13
Global	2	2.14	4	7.49	199.5

Finally, we partition the *SD* distribution into globally and locally scattered driver features. For each feature, we take the average of all its *SD*-values, as reported at each release where the feature existed. We then compare the distributions of the averages in each partition. As Table 7 shows, starting from the median, globally scattered features have higher average *SD*-values. Thus, globally scattered driver features do not only affect more subsystems, but also tend to have higher prospective maintenance costs, given that more locations in the code base might have to be maintained.

4.4 Causes of Scattering

RQ 4) What are possible factors influencing scattering of feature code?

To answer this research question, we investigate whether specific kinds of features exist that by their nature affect where a feature is scattered across (local versus global scattering) or lead to higher scattering degrees. The characteristics we test are the result from past experience and observations when manually analyzing and classifying features in the Linux kernel [5, 6, 38] and other systems [4].



(a) Relative growth

(b) Absolute growth

Figure 8: Growth of outlier scattered features

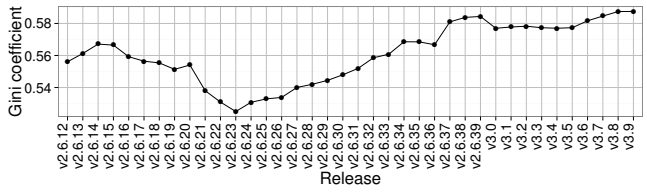


Figure 9: Evolution of the Gini coefficient of the *SD*-value of scattered driver features

The first kind of features we observe relates to so-called *platform* devices. As opposed to hotplugging devices, these cannot be discovered by the CPU. An experienced kernel developer explains:[10]

> "*Happily, we now live in the days of busses like PCI which have discoverability built into them; any device sitting on a PCI bus can tell the system what sort of device it is and where its resources are. [...] Alas, life is not so simple; there are plenty of devices which are still not discoverable by the CPU. [...] So the kernel still needs to provide ways to be told about the hardware that is actually present. 'Platform devices' have long been used in this role in the kernel.*"

In the kernel, a platform driver is any driver that instantiates a `platform_driver` C structure. Since platform devices cannot be discovered by the CPU, the kernel cannot automatically load their corresponding LKMs, as in *hotplugging*. Instead, board-specific code [25] instantiates which devices to support for a target CPU, and with which drivers. However, developers do not instantiate all possible platform devices when porting Linux to a particular CPU, as only some will be present at all times. In the face of such hardware variability, it is intuitive to assume that developers will be more prone to introducing extensions outside the *driver* subsystem (e.g., in the *arch* subsystem, which contains CPU-dependent code), conditioning them on the presence of specific platform devices and their associated drivers and capabilities. For non-platform driver features, the opposite should occur; through hotplugging, devices should be discovered at runtime, triggering the automatic loading of required LKMs.

The second kind of features concerns domain abstractions, which provide a core *infrastructure* from which concrete drivers are built. These abstractions do not bind to a specific vendor, but rather represent a generic set of devices and driver-related capabilities. Examples include generic buses (e.g., `USB`, `PCI`, and `ACPI`), drivers declaring specific device classes (a type of a device, such as an audio or network device),[11] and hardware-description frameworks (e.g., OpenFirmware).[12] Since these features denote abstractions in the operating-system domain, we assume that they should have a higher likelihood of being scattered in comparison to non-infrastructure features. In such cases, extensions in code would check for specific generic functionality and related capabilities, allowing features to react accordingly.

Next, we investigate whether the kind of a feature affects which subsystem the feature's code is scattered across (location) or its scattering degree.

4.4.1 Influence on Scattering Location

We test the effect of being a platform feature on scattering location by first collecting a random sample of $10\,\%$ of all scattered driver features (population size is 1,700). We then manually classify sample features as either platform or not. A platform-driver feature is either a platform driver (i.e, it has at least one compilation unit instantiating a `platform_driver` structure) or it is a capability of a container platform-driver feature. For further details on the classification process, see our online appendix [1].

With the classified sample, we then perform the χ^2 statistical test at a significance level of 0.05. Our hypotheses are:

> *Null hypothesis* (H_0): *being a platform-driver feature has no effect on scattering location*
>
> *Alternative hypothesis* (H_1): *being a platform-driver feature has an effect on scattering location*

Table 8 shows the contingency table used in the test, along with the resulting χ^2 statistic and p-value.

We find strong evidence ($p = 1.933 \times 10^{-5} < 0.05$) of a dependency between being a platform-driver feature and scattering location. Thus, we can reject the null hypothesis in favor of the alternative one. In fact, the analysis of Table 8 indicates that platform-driver features are 2.5 times more likely to be globally scattered than non-platform ones. Conversely, a non-platform driver feature is 1.8 times more likely to be locally scattered. In summary, the test confirms our initial understanding: when facing non-discoverable devices, developers are more likely to introduce *ifdefs* outside the *driver* subsystem. For non-platform devices, the scattering of their driver code is likely local. As Fig. 10 shows, most globally scattered platform-driver features in our sample are scattered across the *arch* subsystem, either only in *arch*, or in both *arch* and *driver* (in the figure, 'either' is captured by the '+' sign, whereas 'and' is denoted by '&'). This evidences a tight relationship between the *arch* subsystem and platform-driver features; since platform devices are not discoverable by the CPU, supporting the drivers of some

[10] Written by Jonathan Corbet, the main author of the *Linux Device Drivers* book [10]. See `http://lwn.net/Articles/448499/`

[11] `https://www.kernel.org/pub/linux/kernel/people/mochel/doc/text/class.txt`

[12] `http://www.openfirmware.info/`

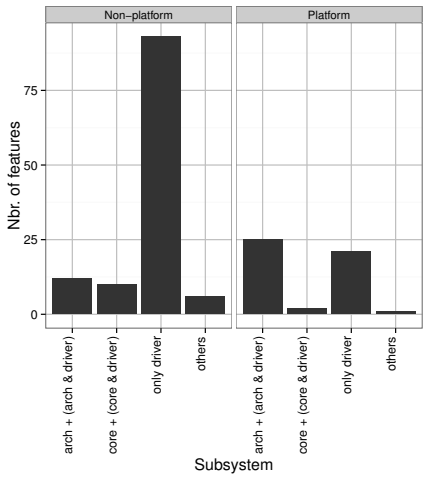


Figure 10: Scattering location of sampled (non-)platform drivers

Table 8: Relationship between being a platform-driver feature and scattering location

$\chi^2 = 18.26, p = 1.933 \times 10^{-5}$

	Is locally scattered?		
Is platform?	No	Yes	Total
No	28	93	121
Yes	28	21	49
Total	56	114	170

Table 9: Relationship between being an infrastructure-driver feature and scattering location

$\chi^2 = 2.43, p = 0.1194$

	Is locally scattered?		
Is infrastructure?	No	Yes	Total
No	44	100	144
Yes	12	14	26
Total	56	114	170

of such devices requires scattering CPU-dependent code, which is mostly found inside the *arch* subsystem.

To check the influence of infrastructure features, we perform similar steps as in the previous test. Using the same sample set of scattered features, we classify them as either infrastructure or not. We test the following hypotheses:

> *Null hypothesis* (H_0): *being an infrastructure-driver feature has no effect on scattering location*
>
> *Alternative hypothesis* (H_1): *being an infrastructure-driver feature has an effect on scattering location*

Here, we do not have a strong evidence suggesting that being an infrastructure-driver feature has an effect on scattering location. As Table 9 shows, running the χ^2 test results in a p-value greater than the chosen significance level. Thus, we fail to reject the null hypothesis.

4.4.2 Influence on Scattering Degree

In order to verify the influence of being a platform or an infrastructure-driver feature on scattering degree, we initially calculate the average *SD*-value of each sample feature across all releases containing it.

To check the influence of being a platform-driver feature, we split the calculated average *SD*-values into those that concern platform-driver features, and those that do not. We then perform a one-tailed Mann-Whitney-Wilcoxon rank sum statistical test to assess whether platform-driver features systematically yield higher average *SD*-values in comparison to non-platform driver features.[13] Our hypotheses are:

> *Null hypothesis* (H_0): *there is no difference in the distribution of average SD-values of platform and non-platform driver features*
>
> *Alternative hypothesis* (H_1): *average SD-values are systematically higher in platform-driver features*

We do not find convincing evidence that average *SD*-values are systematically higher in platform-driver features. With a 0.05 significance level, we are unable to reject the null hypothesis. We also test whether platform-driver features influence the average *SD*-value of non-infrastructure features, as we do not classify platform-driver features as infrastructure. As before, we do not find convincing evidence ($p = 0.8496$).

There seems to be also no significant influence of being an infrastructure-driver feature on scattering degree. Running the one-tailed Mann-Whitney-Wilcoxon rank sum statistical test to compare the average *SD*-values of infrastructure with non-infrastructure features only supports the null hypothesis.

To better understand these results, we place sample features in one of three groups according to the scattering degree limits reported in RQ 3: low (average *SD* ≤ 4), medium ($4 <$ average *SD* ≤ 8), or high (average *SD* > 8). The resulting plot in Fig. 11, which compares infrastructure and non-infrastructure features, shows that both feature kinds have a similar proportional contribution to each scattering degree level bin, yielding similar outcome probabilities. The same occurs between platform and non-platform driver features.

In the case of the outliers observed in RQ 3 (see Fig. 7), however, we do find some influence of being an infrastructure-driver feature on extremely high *SD*-values. By ranking the average *SD*-value of each outlier feature, we plot the histogram of the number of features with an average *SD*-value matching each rank, partitioning the feature set into infrastructure and non-infrastructure outliers. As Fig. 12 shows, infrastructure-driver features are the most scattered features among outliers, with 9 out of the 15 most scattered driver features in the kernel evolution. Considering the total number of outliers, however, infrastructure-driver features are out-performed by non-infrastructure ones. Consequently, most outliers are not infrastructure-related, but are rather narrow in purpose (e.g., target a specific bus-type or particular hardware manufacturer). Among those (Fig. 13), we find that most relate to platform-driver features bound to specific system-on-a-chip devices, such as serial link devices, general input/output (GPIO) capabilities, and video support.

5. Threats to Validity

External Validity. The largest threat to external validity is that our data are based on one case study only. Still, it is one of the largest open-source projects in existence today. Furthermore, our focus on device drivers is justified by the insight that it is the largest and most

[13] Systematically here means that the probability of having an average *SD* greater than a value X among platform-driver features is greater than the probability of having an average *SD* $> X$ among non-platform ones [33].

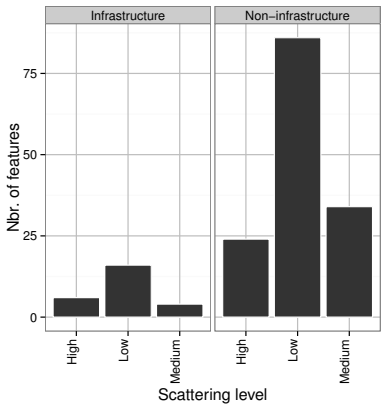


Figure 11: Scattering degree levels in (non-)infrastructure driver features

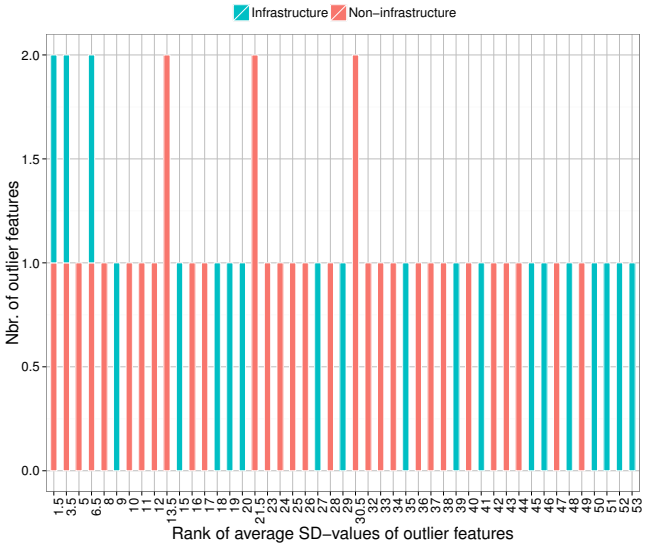


Figure 12: Ranking of outlier features

vibrant subsystem of the Linux kernel. Despite this focus, we study scattering not only within this subsystem, but also investigate how device-driver features affect the other subsystems of the kernel.

To investigate whether two specific kinds of features (platform and infrastructure features) impact scattering degrees and lead to global scattering, we performed hypothesis testing based on a sample of 170 scattered features (population size is 1,700), given that it required manual classification of features. This sampling is justified, and we rely on standard p-value limits to test hypotheses. Recall that the investigation of outliers does not rely on sampling, but on classifying the whole population (54 features).

Finally, our analysis of code scattering relies on pre-processor directives. However, variability in the Linux kernel also affects entire files, as their compilation is controlled by specific features. Thus, we show a partial, yet valid, view of the true story. Of course, our results should be complemented by studying code scattering on the more coarse-grained source file level. Using this information, such as from previous attempts to analyze the Linux kernel's build system, would be valuable future work.

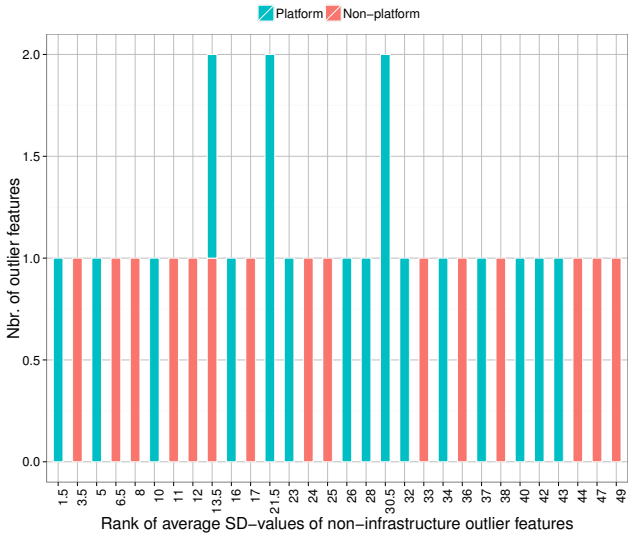


Figure 13: Ranking of features among non-infrastructure outliers

Internal Validity. There is always the risk that bugs in our custom-made tools and scripts impact results. To mitigate this threat, we have performed extensive code reviews, where two authors of the paper performed independent code inspections, summing to almost 16 hours in total. We have also implemented a test suite with over 70 test cases.

For all analyses, we exclude features that we could not uniquely map to one subsystem. This limitation, however, has no further impact on our results, as only very few driver features ($0.65 \pm 0.46\,\%$ per kernel release) are declared in multiple subsystems. We also exclude references to features that occur in strings in the code, assuming that such references have no impact on maintenance, as opposed to the code parts controlled by pre-processor-directives, which we analyze.

Construct Validity. To measure scattering of feature code, we rely on a very simple metric (*SD*). Given that it is a very low-level metric, it is reliable. Since it measures the parts related to feature code as specified by the original developers (using pre-processor directives), it is also a very valid measurement of scattering. In fact, the ability to rely on this information is a major advantage over previous studies, which had to recover the mapping of features (or concerns, see Sec. 6) to code.

However, it is not completely clear how these syntactic code extensions, which aim at realizing configurability, relate to semantic code extensions, that is, units of functionality from a domain-oriented view. Understanding this relationship constitutes an interesting future research question. In fact, it is subject of our current research.

6. Related Work

Feature scattering is part of a broader research topic: *scattering of concerns*. Scattering of concerns in programs has been studied before. Note that such concerns do not only comprise features (in the sense of how we use the term), but also requirements, design elements, design patterns, or programming idioms [14, 18]. The representation of features in the Linux kernel [4, 6, 34] can be compared to concern models [40], which map concerns to code and support concern location.

A particular research interest in recent years has been on cross-cutting concerns, which are under general suspicion to negatively impact quality and maintainability.

Eaddy et al. [14] investigate the relationship of cross-cutting concerns to defects. They argue that insufficient modularization can lead to increased defects for various reasons. For instance, code maintenance might miss parts of the implementation, leading to inconsistencies. Furthermore, concerns might be tangled with other concerns, thus, changing one concern might accidentally change the other. The authors test their hypothesis on three case studies—open-source projects written in Java. Based on a manual identification of the mapping between concerns, classes, and bugs, their analysis shows a high correlation between scattering and defects, regardless of the system's size. The manual concern identification technique has been proposed in a previous work by Eaddy et al. [13]. There, the authors argue that concern location through execution traces is incomplete, as they miss non-functional concerns, such as logging. In our study, we include these kinds of features and fully trace their implementation through pre-processor directives.

Chaikalis et al. [7] report on a longitudinal case study on feature scattering in four open source Java projects. As opposed to our case study, their projects are relatively small, ranging from 24 KLOC to 177 KLOC. They apply dynamic analysis to trace classes implementing each feature, and use formal concept analysis [19] to model and analyze the feature-to-class/method mapping. Their results show a continuous increase of scattering, with very rare exceptions. Similarly, the Gini coefficient shows an increasing fluctuation in scattering degrees over time. Interestingly, they observe an increasing accumulation of feature implementations in already large classes. In our study, using the Gini coefficient is inspired by their work, and we also observe constant (linear-like) increase. However, Chaikalis et al.'s notion of scattering is very different, as they consider all code involved in the control-flow when executing a feature, which may include methods not related to the feature implementation. We do not consider such dependencies (e.g., function calls to other extensions). Thus, despite small projects, they observe very high scattering degrees (up to 1,467 methods)

Several studies have investigated structural characteristics of cross-cutting concerns. For instance, Figueiredo et al. [18] identify patterns of cross-cutting concerns in the source code of three case studies. Their catalogue of 13 patterns—which may overlap—characterizes patterns in terms of (i) their scattering degree and relative code size of their implementations in the scattered classes or methods, (ii) how concerns scatter along an inheritance hierarchy, (iii) control and data-flows, and (iv) whether they are realized by code-cloning. The authors use pattern-detection techniques and identify patterns in three Java projects. Interestingly, they find a negative correlation between some cross-cutting patterns (inheritance-related) and design stability. In a product-line re-engineering effort, Couto et al. [11] found that six of their total of eight features follow the octopus pattern, a cross-cutting pattern previously reported in Figueiredo's work.

Finally, researchers have also extracted realistic thresholds for source-code metrics as a prerequisite to effectively assess quality and maintainability of systems. For instance, Oliveira et al. [35] calculate thresholds for common source-code metrics based on two corpora of Java projects, while accounting for the heavy tail of the underlying metric distributions. They argue that certain percentages of classes naturally violate thresholds (e.g., outliers). In the face of heavy-tailed distributions, the authors state that thresholds should not be based on a single limit value (e.g., mean); instead, thresholds should be *relative*. A relative threshold defines a percentage p of code entities that a metric threshold k applies to (e.g., $p = 85\,\%$ of the methods should have McCabe complexity of at most $k = 14$). From p, it follows that $(100 - p)\,\%$ of code entities should not have a metric value greater than k. Our work stresses the importance of relative thresholds, as outliers in the Linux kernel evolution skew the scattering distribution. Consequently, the mean as a threshold value is not representative of the typical scattering degree that most features in the Linux kernel adhere to. Moreover, we show that 75% of scattered driver features have $SD \leq 8$. This relative threshold, however, is based on the analysis of adjusted boxplots, rather than applying Oliveira's calculation technique, which is not directly applicable to our case.

7. Conclusion

We have analyzed almost eight years of evolution history of device-driver features in the Linux kernel and studied the scattering of code used to implement the features. Our goal was not to investigate limitations or maximum degrees of scattering, but to find empirical evidence that scattering can be handled to the extents we can find in one of the largest feature-based software systems in existence today.

Main Results. We learn that the majority of driver features (82 %) can actually be introduced without causing scattering (RQ 1). Classic modularity mechanisms, as employed by the Linux kernel software architecture, seem to suffice. Yet, the absolute number of scattered driver features is still higher than we expected. Proportionally, however, the amount of scattered features remains nearly constant throughout the kernel evolution. Whether such a limit is actively maintained by developers remains an interesting future research question.

We also found that scattering is not limited to subsystem boundaries (RQ 2). While most driver features are in fact only implemented in the *driver* subsystem, a significant proportion (43 %) of features has extensions in other subsystems. This proportion, however, is stable in the last third of releases.

The implementation of the majority (75 %) of scattered driver features is scattered across a moderate number of four to eight locations in code (RQ 3). Moreover, the median is low and constant across the entire evolution ($SD = 4$). Yet, the distribution is skewed to the right, with outliers having scattering degrees up to 377. Thus, the arithmetic mean is not a reliable threshold to monitor the evolution of feature scattering. Outliers, however, are limited in number, accounting for less than $4\,\%$ of the total number of features in the kernel; however, their absolute counting and magnitude grow with the system.

We identify and analyze two kinds of features that are prone to scattering (RQ 4). So-called *infrastructure* features account for 9 out of the 15 most highly scattered outliers in the scattering distribution of driver features, affecting many parts of the code. So-called *platform* features in the Linux kernel are more frequently scattered across subsystem boundaries, but do not necessarily have higher scattering degrees. The cases where platform-driver features affect scattering degree occur within non-infrastructure outlier features, where platform-driver features account for most of the outliers in that group. While the scattering of platform features across subsystem boundaries could be potentially avoided, the necessary generalization of code and abstraction layers might be too expensive or difficult to be achieved in practice, due to hardware detection limitations. Thus, scattering using pre-processor directives is a natural mechanism in this context, yet facing a potential maintenance trade-off.

Open Research Questions. As future work, we aim at studying scattering in a confirmatory manner, running interviews (or surveys) with kernel contributors and device-driver developers. Specifically, we want to uncover whether kernel developers consciously manage feature scattering and whether the limits we found are enforced in practice. We also do not know whether the observed scattering evolution is a model for other systems. Obtaining a more general picture requires further case studies.

Another open research question concerns the effect of scattering on the actual maintenance effort. For instance, are modules with highly scattered features harder to maintain, to what extent, and are they more error-prone, and how? What is the effect of feature-ownership on maintenance effort, especially when features are highly scattered? A single developer maintaining a feature is likely most efficient, but given a high numbers of features and a distributed development model, it is unrealistic. Thus, finding an optimal organizational structure in a project such as the Linux kernel is a difficult problem—solving it requires further empirical measurements.

Finally, we are interested in investigating how scattering in the kernel could be reduced with alternative solutions, either by using better languages or designs. In either case, we need to know from developers whether such alternatives are suitable to their development context—and if not, the reasons for not adopting them.

References

[1] Online Appendix. `http://lpassos.bitbucket.org/modularity15/`.

[2] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[4] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines*, pages 498–499. Springer, 2010.

[5] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 73–82. ACM, 2010.

[6] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[7] T. Chaikalis, A. Chatzigeorgiou, and G. Examiliotou. Investigating the Effect of Evolution and Refactorings on Feature Scattering. *Software Quality Journal*, pages 1–27, 2013.

[8] J. Conway, D. Eddelbuettel, T. Nishiyama, S. Kumar, and T. Neil. *Package 'RPostgreSQL'*, 2013. R package version 0.4.

[9] J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux Kernel Development: How Fast It is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. `http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013`. Last seen: February 14, 2015.

[10] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.

[11] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 191–200. IEEE, 2011.

[12] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the 2nd Workshop on Modularity in Systems Software*, pages 15–20. ACM, 2012.

[13] M. Eaddy, A. Aho, and G. C. Murphy. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques*, pages 2–7. IEEE, 2007.

[14] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[15] J.-M. Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the 12th International Conference on Software Maintenance*, pages 329–339. IEEE, 1996.

[16] D. G. Feitelson. Perpetual Development: A Model of the Linux Kernel Life Cycle. *Journal of Systems and Software*, 85(4):859–875, 2012.

[17] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE, 2008.

[18] E. Figueiredo, B. C. da Silva, C. Sant'Anna, A. F. Garcia, J. Whittle, and D. J. Nunes. Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 138–147. IEEE, 2009.

[19] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1st edition, 1997.

[20] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 131–142. IEEE, 2000.

[21] G. Grothendieck. *Package 'sqldf'*, 2014. R package version 0.4-7.1.

[22] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd. and Toshiba Corp. Advanced Configuration and Power Interface Specification, Revision 5.0. `http://www.acpi.info/spec50a.htm`. Last seen: February 14th, 2015.

[23] M. Hubert and E. Vandervieren. An Adjusted Boxplot for Skewed Distributions. *Computational Statistics & Data Analysis*, 52(12):5186–5201, 2008.

[24] C. Izurieta and J. Bieman. The Evolution of FreeBSD and Linux. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 204–211. ACM, 2006.

[25] M. T. Jones. Anatomy of the Linux Kernel. *IBM Developer Works*, 2009.

[26] C. Kästner, S. Apel, and K. Ostermann. The Road to Feature Modularity? In *Proceedings of the 15th International Software Product Line Conference*, pages 5:1–5:8. ACM, 2011.

[27] Kbuild. The Kernel Build Infrastructure. `www.kernel.org/doc/Documentation/kbuild`. Last seen: February 14th, 2015.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[29] G. Krone, M.; Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of the 6th International Conference on Software Engineering*, pages 49–57. IEEE, 1994.

[30] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 105–114. ACM, 2010.

[31] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development*, pages 191–202. ACM, 2011.

[32] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines*, pages 136–150. Springer, 2010.

[33] D. S. Moore, G. P. McCabe, and B. Craig. *Introduction to the Practice of Statistics*. W. H. Freeman, 6th edition, 2009.

[34] S. Nadi and R. Holt. The Linux Kernel: A Case Study of Build System Variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

[35] P. Oliveira, M. T. Valente, and F. P. Lima. Extracting Relative Thresholds for Source Code Metrics. In *Proceedings of the Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 254–263. IEEE, 2014.

[36] L. Passos and K. Czarnecki. A Dataset of Feature Additions and Feature Removals from the Linux Kernel. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 376–379. ACM, 2014.

[37] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-Oriented Software Evolution. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 17:1–17:8. ACM, 2013.

[38] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of Variability Models and Related Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering*, 2015. To appear.

[39] R. Queiroz B., L. Passos, M. T. Valente, S. Apel, and K. Czarnecki. Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Software Families. In *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*, pages 23–29. ACM, 2014.

[40] M. P. Robillard and G. C. Murphy. Representing Concerns in Source Code. *Transactions on Software Engineering Methodologies*, 16(1), 2007.

[41] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines*, pages 30–33, 2007.

[42] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX*, pages 185–197, 1992.

[43] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the 10th European Software Engineering Conference*, pages 166–175. ACM, 2005.

[44] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[45] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 179–188. IEEE, 2013.

[46] S. Venkateswaran. *Essential Linux Device Drivers*. Prentice Hall Press, 1st edition, 2008.