A Study of Feature Scattering in the Linux Kernel

Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel,
Krzysztof Czarnecki, and Jesus Alejandro Padilla

Abstract—Feature code is often scattered across a software system. Scattering is not necessarily bad if used with care, as witnessed by systems with highly scattered features that evolved successfully. Feature scattering, often realized with a pre-processor, circumvents limitations of programming languages and software architectures. Unfortunately, little is known about the principles governing scattering in large and long-living software systems. We present a longitudinal study of feature scattering in the Linux kernel, complemented by a survey with 74, and interviews with nine Linux kernel developers. We analyzed almost eight years of the kernel's history, focusing on its largest subsystem: device drivers. We learned that the ratio of scattered features remained nearly constant and that most features were introduced without scattering. Yet, scattering easily crosses subsystem boundaries, and highly scattered outliers exist. Scattering often addresses a performance-maintenance tradeoff (alleviating complicated APIs), hardware design limitations, and avoids code duplication. While developers do not consciously enforce scattering limits, they actually improve the system design and refactor code, thereby mitigating pre-processor idiosyncrasies or reducing its use.

1 Introduction

Scattering of feature code is commonly perceived as an undesirable situation [1–4]. Scattered features are not implemented in a modular way, but are spread over the code base, possibly across subsystems. The tangling of scattered features with different implementation parts can lead to ripple effects and require frequent developer synchronization, which challenges parallel development. Scattered features may significantly increase system maintenance efforts [5, 6]. Yet, feature scattering is common in practice [7–9].

Feature scattering allows developers to overcome design limitations when extending a system in unforeseen ways [5] or when circumventing modularity limitations of programming languages, which impose a dominant decomposition [10–12]. Figure 1 illustrates the scattering of a feature CONFIG_A which has three *ifdef* references, as opposed to the non-scattered feature CONFIG_C, which has only one such reference (cf. Section 2.3). In other cases, the cost of modularizing features might be initially prohibitive or simply too difficult to be handled in practice [13]. In contrast, feature scattering requires little upfront investment [6], although maintenance costs may rise as the system evolves. Many long-lived and large-scale software systems have shown that it is possible to achieve continuous evolution while accepting some extent of feature scattering. Examples span different domains, such as operating systems, databases, and text editors [7, 9, 14].

Surprisingly, there are no empirical studies investigating feature scattering in large and long-lived software systems. Such studies are key in creating a widely accepted set of practices to govern feature scattering and may eventually contribute to a general scattering theory, which could serve as a guide to practitioners—for instance, in identifying implementation decay [15], assessing the maintainability of a system [16], identifying scattering patterns [17] or setting practical scattering thresholds [9].

To contribute to a deeper understanding of feature scattering and its evolution, we present a case study of one of the largest and longest-living software systems in existence today: the Linux kernel. Its features are manifested as compile-time configuration options that users select when deriving customized kernel images. Our analysis covers evolution, practices, and circumstances leading to feature scattering.

The Linux kernel is the operating system kernel upon which free and open-source software operating system distributions, such as Ubuntu, OpenSUSE, Fedora and Android, are built. Its deployment goes beyond traditional computer systems, such as personal computers and servers, to embedded devices, such as routers, wireless access points, and smart TVs, as well as to mobile devices. Introduced in 1991, the Linux kernel boasts over ten million source lines of code (mostly written in C), and 12,000 contributors from more than 200 companies. Contributors work on one or more kernel subsystems, for instance, the device driver sub-system (hereafter referred to as the *driver* subsystem) and the file system (hereafter referred to as the *fs* subsystem). There are two categories of contributors: developers and maintainers. Maintainers are a more restricted group than developers and concentrate more on kernel structure and quality standards of patches contributed to the kernel. They review, modify, and authorize patches, before they are merged into the mainline repository managed by Linus Torvalds.

A feature can be scattered within a subsystem (a.k.a. local scattering), or across subsystems (a.k.a. global scattering). Due to the sheer size of the kernel, we scoped our longitudinal analysis of the source code to features of the *driver* subsystem, which we identified as the largest and fastest growing kernel subsystem (see Sec. 3). We analyzed the scattering of driver features within and across the device-driver subsystem and followed-up with developers and maintainers through a survey and interviews, to understand their practices, circumstances, and perceptions of feature scattering. The survey and interviews, however, were not exclusive to the

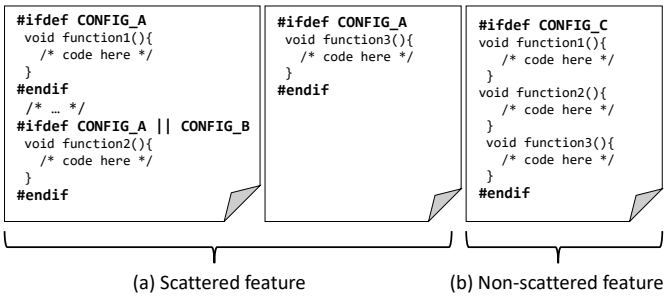


Fig. 1: An example of a scattered and a non-scattered feature

driver subsystem.

Research Questions. We formulated three research questions to empirically investigate the impact of feature scattering on the maintenance of a large and long-lived software system—the Linux kernel: the first question targeted analysis of the kernel's code to investigate trends of feature scattering (how feature scattering evolves with the evolution of the kernel), and the last two questions, as a follow-up to the first, targeted developers of the Linux kernel to investigate their perception of feature scattering and its impact on maintenance effort of the kernel, and how they cope with it.

RQ1: *How does feature scattering evolve?* We conducted a longitudinal[1] analysis of feature scattering in the code base comprising almost eight years of the kernel's 26 years of evolution. In these eight years (from version 2.6.12 to 3.9), the kernel growth has been steady, from 4,752 to 13,165 features, many of which are scattered [14] (note that our analysis began with kernel version 2.6.12 when the Linux development community switched from using a proprietary version control system called Bitkeeper, to Git). To understand this evolution, we formulated three sub research questions:

RQ1.1: *How does the growth of scattered features differ from non-scattered features?* We analyzed the relative and absolute growths of scattered and non-scattered driver features—for instance, to understand whether the proportion of scattered features is increasing, decreasing or stable.

RQ1.2: *How does the growth of locally scattered features differ from globally scattered features?* We analyzed the relative and absolute growths of driver features that are scattered (i) within the *driver* subsystem only (local scattering) and (ii) across, at least, another subsystem (global scattering). We aimed at understanding how scattering is related to the kernel's architecture.

RQ1.3: *How does the extent of feature code scattering evolve over time?* We analyzed the extent (degree) of the scattering of feature code, aiming at understanding the underlying distribution and possible thresholds, as well as how this degree relates to local and global scattering.

The results of RQ1 formed the basis for formulating two further research questions, RQ2 and RQ3, complementing our analysis by a survey with 74 kernel developers (62 of whom contributed to the *driver* subsystem; 17 were subsystem maintainers—13 of whom worked on the driver system), as well as interviews with 9 kernel developers, aiming at understanding phenomena observed in our longitudinal study.

RQ2: *What are the circumstances of feature scattering?* We investigated both possible causes and circumstances leading to feature scattering by analyzing the survey and interview data. We also identified and asked the interviewees about examples of scattered code that they developed and that we identified as such in the kernel's codebase. Furthermore, we studied whether certain kinds of features are more likely to be scattered.

RQ3: *What are practices for coping with feature scattering?* We analyzed the survey respondents' and interviewees' reported practices for coping with feature scattering and whether developers consciously maintain a scattering threshold for the number of scattered features or for the features' scattering degrees.

[1]Longitudinal refers to an observational research method in which data (about feature scattering in our case) is repeatedly gathered for the same subjects (the Linux Kernel in our case) over a long time period (eight years in our case).

Results. With respect to RQ1, we found that the majority of driver features can actually be introduced without causing scattering and that the number of scattered features remains proportionally nearly constant throughout the kernel's evolution. We also found that scattering is not limited to subsystem boundaries and that the implementation of the majority of scattered *driver* features is scattered across a moderate number of four to eight locations in the code. With respect to RQ2, we found that developers introduce scattering in the Linux kernel, among other reasons, to avoid code duplication and to support hardware variability, backwards compatibility, and code optimization. We also learned that the features that are most prone to scattering are those relating to platform devices—devices that cannot be discovered by the CPU as opposed to hotplugging ones. With respect to RQ3, we found that developers try to avoid feature scattering mostly by alleviating the problems of pre-processor use and refactoring existing code to, for instance, improve system architecture, but the majority do not consciously maintain a scattering threshold.

Contributions. In summary, our contributions comprise:

- A *dataset covering almost eight years of the evolution of feature code scattering* extracted from the Linux kernel repository (from version 2.6.12 to 3.9). It serves as a replication package, as a benchmark for tools, and for further analyses.
- *Empirical data from a survey and interviews* aimed at understanding the state of practice of feature scattering in the Linux kernel.
- An *online appendix* [18] with further details on our dataset, scripts to analyze the data, and additional statistics.

An earlier version of this work appeared as a conference paper [19]. Its focus was on analysis of the source code to understand feature scattering trends during eight years of the Linux kernel's evolution. In this article, we broaden the scope of our study to include developer perspectives on feature scattering in the Linux kernel. In particular, we added an analysis and discussion of circumstances that lead to feature scattering from the perspective of developers, including how developers try to limit or avoid it, and what kinds of features are usually scattered.

On a final note, even though our study relies on only one subject system, its size and number of contributors, and subsystems of very different nature (with the driver subsystem being the largest and most diverse one) enhance the study's external validity. Furthermore, we triangulate[2] [20] data obtained from the longitudinal study of the code base with survey and interview data obtained from real kernel developers.

2 Background

We now discuss how the kernel represents its features and how they evolve. We also introduce relevant terminology and definitions.

2.1 Feature Representation

Features in the Linux kernel are explicitly declared in the kernel's variability model written in the Kconfig language [21–24]. Features are referenced in build rules of Linux's Makefiles and in C preprocessor directives, controlling the compilation of entire source files or fragments therein, respectively. Henceforth, we refer to such fragments as *extensions*. The kernel's code base comprises

[2]Triangulation is a technique for validating data through cross verification from two or more sources; it is a common way to enhance research results for a study of a phenomena involving same subjects.

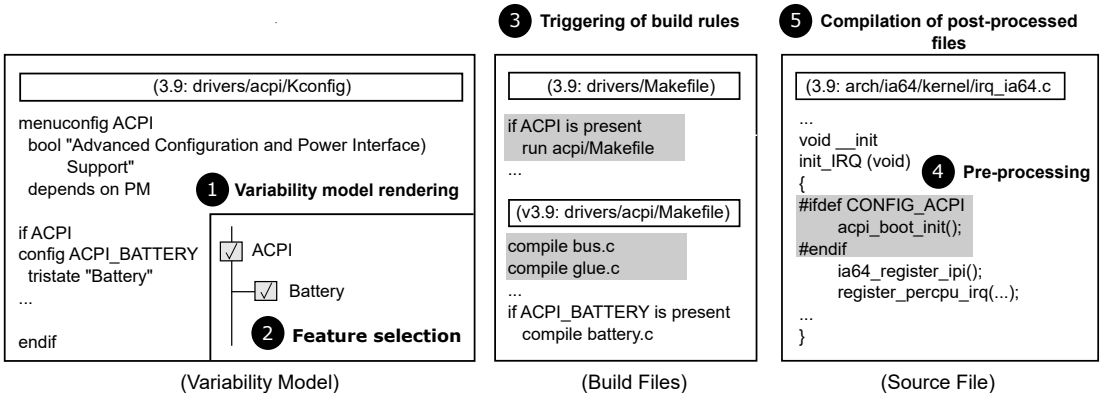


Fig. 2: Binding of different artifacts based on selected features

mostly C implementation and header files (43 % C implementation files, 39 % C header files, 4 % assembly, and 14 % other).

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 [25]. Figure 2 illustrates the steps involved in configuring the ACPI feature, with excerpts of each artifact type (variability model, build file, source file). 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's 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`[3] 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 source files are compiled into 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). Only driver features having the feature type "tristate" in the variability model (e.g., `ACPI_BATTERY`), can become an LKM. 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 [14, 26–28], so 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 pre-processing *ifdefs* and static linking, or at runtime (e.g., when loading/unloading LKMs).[4]

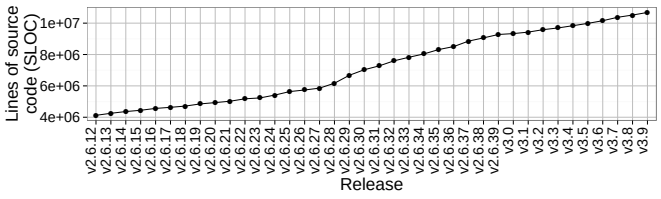
[3]Feature macros are prefixed with `CONFIG_` in the kernel.

2.2 Kernel Evolution

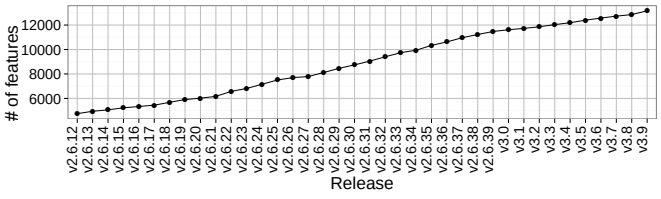
The Linux kernel evolves continuously. Figure 3 summarizes its growth in terms of source lines of code (SLOC) and number of features. As Fig. 3a shows, the code base has increased by $159\,\%$ since the first release recorded in the kernel's git repository (v2.6.12, June 2005), with a growth of $2.6 \pm 1.5\,\%$ between two consecutive main releases.[5] The kernel's feature set, shown in Fig. 3b, displays a similar trend, and strongly correlates with SLOC growth (Pearson product-moment correlation $r = 0.996$). Since release v2.6.12, it has increased by $177\,\%$, growing by $2.8 \pm 1.4\,\%$ between main 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



(a) Kernel growth in SLOC



(b) Kernel growth in number of features

Fig. 3: Kernel growth evolution

[4]Other kinds of runtime variability also exist, such as changing the attributes of a device driver through the sysfs virtual filesystem [29, 30].

[5]The short-hand $2.6 \pm 1.5\,\%$ denotes an arithmetic mean of $2.6\,\%$ with standard deviation of $1.5\,\%$. In the remainder, the mean (or average) should always be understood as the arithmetic mean.

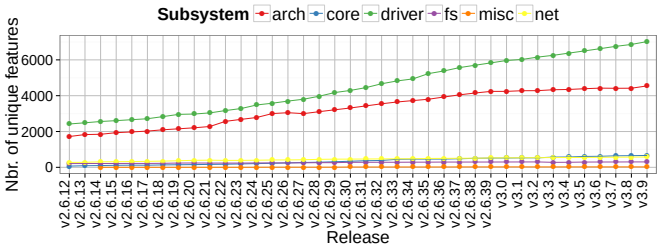


Fig. 4: Feature distribution per subsystem

in the code base (cf. Sec. 3.3.2). We trace these extensions by identifying *ifdefs* that refer to 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 places that a developer may consider upon changing a feature of interest [9].

3 Methodology

Our research method comprises a longitudinal source-code study complemented by a survey and interviews of kernel developers. This section provides details about our methods for collecting and analyzing the evolutionary data of feature code scattering and for conducting the survey and interviews.

3.1 Scoping

Our longitudinal analysis of the source code concentrated on driver features, which are defined in the *driver* subsystem of the Linux kernel. Beside practicality, this decision rests on existing work stating that the Linux kernel evolution is mainly driven by the evolution of its device drivers [14, 27, 31–33], and on our own analyses (which we explain shortly). Setting the scope to features in the *driver* subsystem required us to distinguish them from features of other subsystems.

3.2 Identifying Driver Features

According to Corbet et al. [34] the kernel has seven major subsystems: *arch* (architecture-dependent code), *core* (scheduler, IPC, memory management, etc.), *driver* (device drivers), *firmware* (firmware required by some devices), *fs* (file system), *net* (networking), and *misc* (miscellaneous). To distinguish driver features from features of other subsystems, we sliced the kernel's codebase according to a mapping between files and Corbet et al.'s subsystems[6] created by Greg Kroah-Hartman, a kernel maintainer.[7] After identifying each file's subsystem, we considered the subsystem of a feature's declaring Kconfig file as the feature's subsystem. Some *driver* features, although very few ($0.65 \pm 0.46\,\%$), are also declared in other subsystems (e.g., *core*). As we cannot disambiguate with certainty, we excluded these features from our analysis.

Once we identified the unique features in each subsystem, we were able to confirm that the kernel is actually driven by the evolution of driver features. As Fig. 4 shows, the *driver* subsystem is not only the largest, but also the fastest growing.

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

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

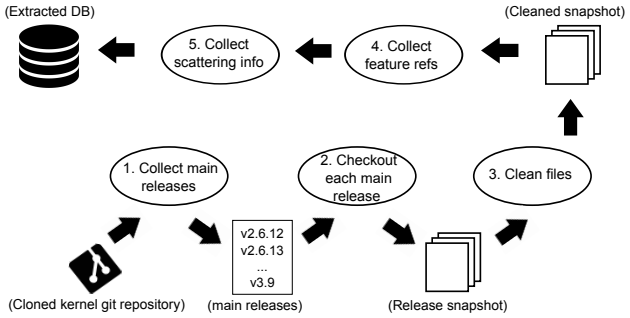


Fig. 5: Data extraction for the longitudinal source-code study

3.3 Longitudinal Source-Code Study

We now describe how we collected and analyzed data from the source code and its history for answering RQ1.

3.3.1 Data Collection

We followed the process shown in Fig. 5. First, we queried the kernel's source management system (git) to list all release tags. From the listing, we filtered the main stable-release identifiers (step 1).[8] We then checked each main release out (step 2), from which we listed all C implementation and header files and cleaned them by removing empty lines and comments, and by transforming multilines[9] into single ones (step 3). We also eliminated white space between in the source code as a means to facilitate pattern matching when mining references to a feature (through its *ifdef* variable) across the code (step 4). Finally, we collected metadata of each identified feature reference (step 5), including the name of the file in which a feature is referenced, the line in which the reference occurs, and the associated *ifdef* pre-processor directive. Note that we did not consider arbitrary preprocessor macros, but only those defined in the kernel's variability model (Kconfig files). All feature references and their associated metadata were 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 main release.

Further details on the data, the database schema, the dataset, and associated scripts, can be found in our online appendix [18].

3.3.2 Analysis

To answer RQ1's three sub-questions, we issued SQL queries through the R statistical environment, which we connected to our database. We plotted the results and calculated different statistics. In particular, we measured the scattering degree (*SD*) of a feature *ft* in terms of its scattering degree at each implementation and header C file f in a set of target subsystems S:

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

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 [9]. Note that sub systems are disjoint, hence no files are shared among them.

The *SD* metric falls under the umbrella of absolute metrics, which count the number of source-code entities relating to a

[8]Unstable releases are suffixed with *-rc* (e.g., v2.6.32-rc1). We do not include updates to stable (main) releases (e.g., v2.6.32.1) in our set of releases.

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

given feature. In contrast, relative metrics assess feature scattering relative to the code size of extensions [35]. Existing research [36] comparing absolute metrics with relative ones shows that the former correlate better with properties of interest (e.g., bugs), which justifies our choice for the *SD* metric.

A feature is *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 (Fig. 2), `ACPI` is a scattered feature; in addition to the reference in the *ifdef* inside `init_IRQ`, it also has 110 corresponding *ifdefs* elsewhere.

3.4 Developer Survey and Interviews

Answering RQ2 (circumstances of scattering) and RQ3 (practices to cope with scattering), we relied on a survey and semi-structured interviews with developers of the Linux kernel. We designed a survey questionnaire to cover several aspects of the developers' experience with scattered features. In particular, our focus was on their perception of the challenges and strategies they used in the face of scattered code. After analyzing the survey results, we performed follow-up interviews with selected survey respondents, investigating RQ2 and RQ3 in more detail, especially by discussing concrete Linux code examples that the interviewees actually contributed to the kernel.

Some Linux developers were not familiar with some concepts and vocabularies commonly used in academia. To ensure that participants knew what scattering is, we provided a brief description on the cover page of the questionnaire and explained it to our interviewees. Additionally, we consistently used the Linux kernel's term *config option* when referring to *feature*.

3.4.1 Developer Survey

We designed the questionnaire to take no more than seven minutes to be completed. All questions were optional and mostly closed-ended. In specific cases, the participant was asked to provide additional options or to elaborate. Table 1 presents an overview of the structure of the survey with a sample of the questions. The online appendix [18] contains the full questionnaire, which was constructed and distributed using the online tool *SurveyGizmo*.

All participants were open-source developers with varied technical and industrial experience. There were no exclusion criteria. We obtained the list of potential survey respondents from the kernel's source code repository, querying git to get the author information of all commits. There was no compensation offered to the participants for their involvement other than the benefit of improved understanding of how feature scattering impacts maintenance tasks and better understanding of how to improve current practices.

3.4.2 Survey Respondents

From a total of 12,248 unique contributors we identified, we selected a random sample of 2000 potential respondents and sent a targeted e-mail inviting them to our survey. A total of 1,966 invites were delivered via SurveyGizmo, while 34 failed (likely due to invalid e-mail addresses). According to the Software Engineering Institute's guidelines for designing an effective survey in Software Engineering [37], when the population is a manageable size and can be enumerated, simple random sampling is the most straightforward approach to avoid selection bias. This is the case for our study with a population of 12,248 Linux kernel contributors.

From the 1,966 e-mails sent, we received 112 responses (5.7% response rate). We disqualified 38 without actual responses (those without responses to any of the survey questions of interest to the study despite responding to basic ethnographic questions, such as role of the respondent and work place), leading to 74 valid responses that we considered. 62 of these were from contributors of the *driver* subsystem. For aspects of feature scattering that were specific to the *driver* subsystem, we considered the 62 responses only, however, for more general aspects, all the responses were considered. Since all questions were optional, certain aspects discussed in this study are limited to a subset of the respondents. In the remainder, we code respondents' quotes using S1, S2, etc. for anonymity reasons.

TABLE 1: Structure of the survey and interviews

	section	#	question examples
survey	respondent background	2.	What is your role in working with the kernel?
	kernel development	4.	Which kernel subsystems do you work on?
		6.	Which of the following methods do you use to create a new device driver?
	perceptions on scattering	9.	Working with highly scattered code is challenging.[1]
		10.	Does highly scattered code impact the following software-development aspects?
	reasons for scattering	12.	Why do you introduce or maintain scattered code? Add more reasons in the three textfields if you want.
	coping with scattering	15.	Do you try to limit scattering? If so, how?
		17.	To what extent do you agree with the following strategies to avoid scattering?[1]
	other development activities	19.	Are you also working as a professional developer in a company?
interview	reasons for scattering	Q1.5.	We want to talk about this specific commit [see Listing 1] you authored. Considering it, what is the reason for adding scattering?
	factors influencing scattering refactoring	Q2.3.	In general, what scenarios make config options prone to scattering? In the previous survey, many contributors (73%) revealed they try to limit scattering by refactoring.
		Q3.1.	When do you refactor a scattered config option? Is there a tipping point?

[1] 5-point Likert-scale question eliciting agreement with the statement

3.4.3 Interviews

We invited 26 survey respondents, who had agreed to be contacted, to participate in structured interviews, of which 9 agreed to be interviewed. We queried the Linux kernel commit history to find concrete code examples of scattering introduced by each participant. The goal was to use real examples authored by the participants to contextualize the questions regarding the reasons for scattering. For instance, Listing 1 states cases of scattering introduced by a commit that interviewee I1 authored; we used this during the interview with I1 to find out why this was the case. The interview guide, including how we personalized it for each interviewee, is available in our online appendix [18]. Table 1 presents an overview of the structure of the interview guide with a sample of questions.

We conducted two interviews via Skype and seven by e-mail. In fact, we observed in a previous study [38] that Linux developers strongly prefer e-mail communication. The Skype interviews were recorded and transcribed. We combined all related answers from the transcripts and the e-mails, and then interpreted and compared them to the individual questions we asked (see Table 1). In the remainder, we code interviewees as I1, I2, etc. for anonymity.

4 Results

In this section, we present our results on the evolution of scattering (RQ1), the possible circumstances that lead to the observed scattering (RQ2), and the practices for coping with it (RQ3).

4.1 Scattering Evolution (RQ1)

To answer RQ1 (*How does feature scattering evolve?*), we analyzed how scattered features evolve compared to non-scattered features, the differences between scattering locality (locally versus globally scattered), and how the scattering degree evolves. In the survey, $80\,\%$ of participants perceived scattering as challenging. We discuss here how this perception is reflected in the kernel's evolution based on our analysis of the code.

4.1.1 Scattered versus Non-Scattered Features

To understand how the growth (increase in the number) of scattered features differs from non-scattered features, we plotted the proportion of scattered driver features in each kernel release, along with their absolute number. In both cases, we compared the growth rate of scattered driver features with the evolution of non-scattered features. Figure 6 displays both plots, with summary statistics provided in the appendix (Tables 1 and 2). When calculating the scattering degree (Eq. 1) to identify scattered features, we took S as the union of all kernel subsystems.

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 v2.6.22 and ends v2.6.26. In absolute terms, the number of scattered driver features grows by $2.5 \pm 2.4\,\%$ between each pair of consecutive main 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.[10] Release v3.9 has over 1,000 scattered driver features. The latter number, however, grows almost six times slower when compared to non-scattered driver features, as given by the ratio of their regression line slope coefficients: 20 for scattered features and 114 for non-scattered features (see appendix Tables 1 and 2). 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.

Our 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 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. Section 4.3, will discuss in more detail what the possible circumstances of such scattering are.

Listing 1: Commit by I1 that introduces scattering

```
I1:

Commit: 47118af076f64844b4f423bc2f545b2da9dab50d
Dec 29, 2011
"mm:␣mmzone:␣MIGRATE_CMA␣migration␣type␣added"

In the files:
include/linux/mmzone.h
Lines 44, and 64

mm/page_alloc.c
Lines 753, and 899

mm/vmstat.c
Line 616

introduced 5 new #ifdefs
with the configuration option CONFIG_CMA

More details on:
https://github.com/torvalds/linux/commit/
47118af076f64844b4f423bc2f545b2da9dab50d
```

[10]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).

4.1.2 Local versus Global Scattering

Next, we investigated to what extent the scattering of driver features was local and to what extent global. Recall, 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 the *driver* subsystem. Ideally, most scattering should be local, contributing to internal cohesion and decreasing coupling (local scattering is better for maintenance since it does not require cross-subsystem coordination when maintaining a feature).

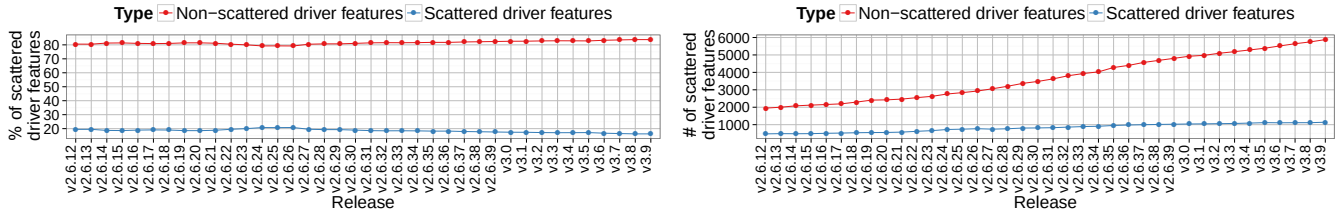
The growth of locally scattered driver features varies along the Linux kernel evolution. Nonetheless, it dominates the growth of globally scattered driver features, both proportionally and in absolute numbers. Figure 7 shows the corresponding plots, with summary statistics provided in the appendix (Tables 3 and 4).

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. 7a.

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. In other words, 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 Section 2.2, the 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. $43\,\%$ seems a current upper limit kept by Linux kernel developers. Indeed, as we learned from our interviewees, kernel developers do make efforts to limit global scattering by making drivers as self-contained as possible, except in corner cases which they handle by introducing scattering for reasons that we discuss in Section 4.3.1.

4.1.3 Scattering Degrees

To understand how the extent of feature code scattering evolves over time, we plotted the scattering degrees (*SD*) of all scattered driver features at each kernel release. When measuring *SD* (see Eq. 1), we took the target set of subsystems (S) as the union of all subsystems in the kernel. The boxplot in Figure 8, which is adjusted for skewness [39], shows that $50\,\%$ of all scattered driver features have a low scattering degree, with $SD \leq 4$ across all main



(a) Relative growth

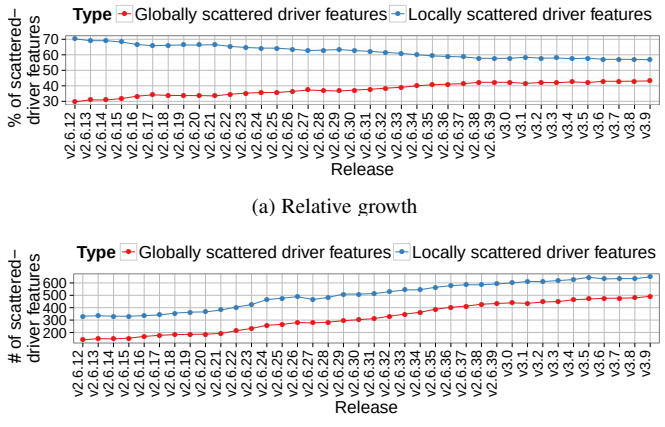
(b) Absolute growth

Fig. 6: Growth of the number of scattered and non-scattered features

releases. However, for the second half of the scattered features, the scattering degree considerably increases. In the third quartile (up to 75 % of the scattered features in the boxplot 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 and relative numbers. Figure 9 shows the respective graphs (summary statistics are in the appendix Table 5). 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 not a 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 should adhere.

To ascertain the observed skewness, while summarizing how unevenly *SD* values are distributed among scattered driver features, we calculated the Gini coefficient [40] for each kernel release. The Gini coefficient is a popular summary statistic in economics, measuring the inequality of wealth (*SD*, in our case) among the



(a) Relative growth

(b) Absolute growth

Fig. 7: Growth of the number of locally and globally scattered driver features

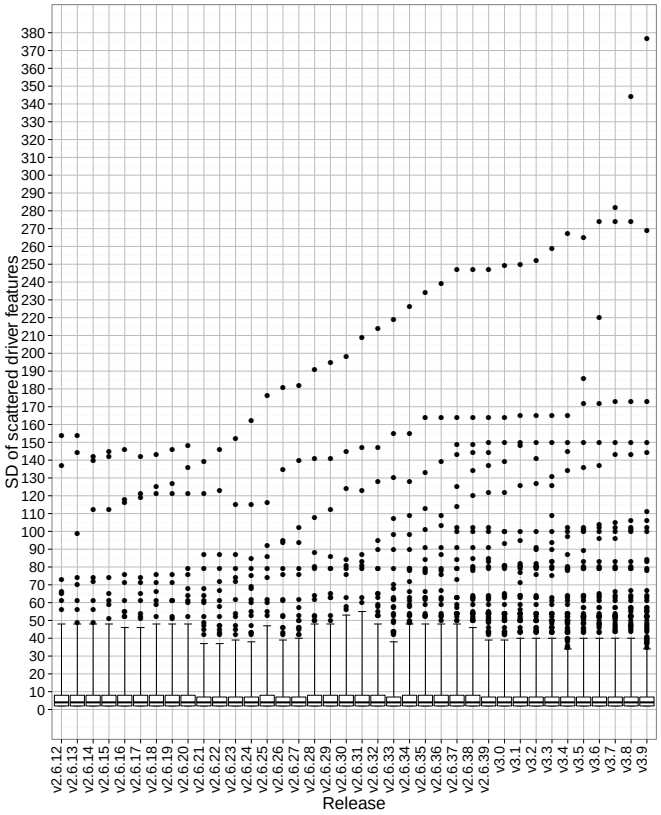
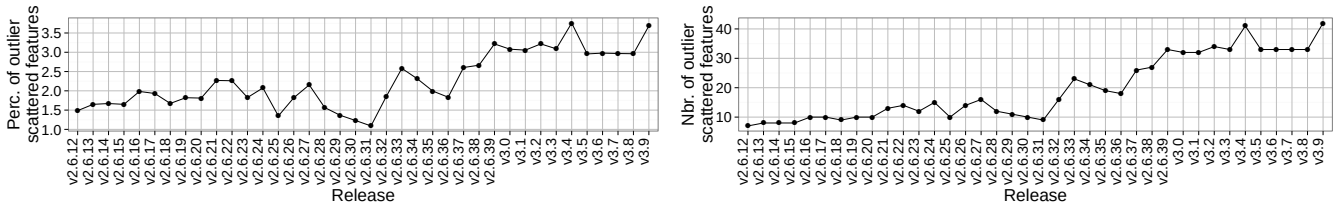


Fig. 8: *SD* values of scattered driver features

individuals (features, in our case) 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 10 shows the evolution of the Gini coefficient in the kernel evolution. The coefficient follows a decreasing trend in the first 12 releases, meaning that *SD* becomes more evenly distributed. From release v2.6.23 on, we observe an increasing trend, indicating that *SD* is more concentrated towards a particular set of features. The absolute difference between the Gini 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.

Finally, we partitioned the *SD* distribution into globally and locally scattered driver features. For each feature, we took the average of all its *SD* values, as reported at each release where the feature existed. We then compared the distributions of the averages in each partition. As Table 6 in the appendix shows, starting from the median, globally scattered features have higher



(a) Relative growth

(b) Absolute growth

Fig. 9: Growth of outlier scattered features

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.

4.2 Survey Respondents and Interviewees

From our 74 survey respondents with valid responses, 49 (66 %) were working as professional developers in companies. Of these, 17 (35 %) were contributing to the kernel as part of their job, 13 (27 %) were contributing as private persons, and 19 (39 %) were contributing both for the company and private. 58 (78 %) of all respondents had worked with the kernel for more than five years.

The respondents had different roles in their work with the Linux kernel; the largest group with 55 (74 %) are developers, followed by 17 (23 %) maintainers. As stated above, maintainers review, modify, and authorize the incorporation of changes into the kernel, therefore we considered their participation relevant for our study.

The majority of survey respondents, 62 (84 %), were contributors to the *driver* subsystem, however, not exclusively. Driver contributors also declared experience with other subsystems: *arch* (52 %), *net* (27 %), *fs* (21 %), *core* (15 %), and others (16 %).

As a subset of survey respondents, the interviewees' characteristics somewhat resembled the population of survey respondents. The only significant difference was with respect to the subsystem to which they contributed, as only 3 (33 %) of the interviewees were contributors to the *driver* subsystem. However, this did not affect our results since our interview questions concerned feature scattering in general, as well as practices for coping with it, regardless of the subsystem.

4.3 Circumstances of Scattering (RQ2)

To answer RQ2, we investigated the circumstances that could possibly explain the scattering phenomena observed in Sec. 4.1.

4.3.1 Reasons for Introducing Scattering

First, we tried to understand what methods developers use to create new device drivers. Four options were listed in the survey: (i) clone an existing driver and adapt it, (ii) develop from scratch, (iii) use a template, and (iv) extend an existing driver by introducing variants using *ifdefs* (without cloning). Additionally, respondents were asked to sort the options they selected according to how frequently they used them. The most frequently used option would be placed first, while the least used would be placed last. The results are shown in Table 2. This question had exactly 62 respondents, which corresponds with the number of respondents contributing to the *driver* subsystem (Sec. 4.2); this was expected considering that the question was about creation of device drivers.

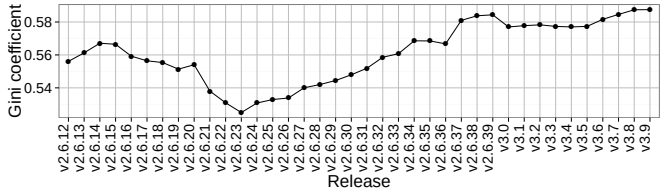


Fig. 10: Evolution of the Gini coefficient of the *SD* value of scattered driver features

As Table 2 indicates, cloning and adapting an existing driver is the most frequently and widely used approach, with 83 % of the respondents stating that they use it and 43 % placing it in first position. This can possibly explain why the number of scattered features remains proportionally nearly constant throughout the kernel's evolution, since code is simply copied and adapted to each new driver, carrying with it all the scattering (see Sec. 4.1.1). However, the fourth ranked option is of particular interest since it can potentially introduce feature scattering: extend an existing driver by introducing variants using *ifdefs* (without cloning). Despite being ranked lowest of all options, 56 % of the respondents stated that they do apply this method when creating new device drivers. Of these respondents, 8 % placed it in first position, 22 % in second position, 18 % in third position, and 8 % in fourth position. This result indicates two things: first, it gives an indication as to how developers perceive working with *ifdefs*; indeed, when asked explicitly in a separate question, 55 % of the survey respondents stated that they perceived working with *ifdefs* as challenging; secondly, it indicates that despite this perception, developers still use *ifdefs* for some aspects of development (as discussed next), which, in turn, may lead to feature scattering. Since the majority of survey respondents stated that they create new drivers by either cloning and adapting an existing driver (83 %), or developing from scratch (69 %), we learned from their additional comments, that generally *ifdefs* are used when extending an existing driver with a new version by adding new features, while cloning is mostly used for new drivers.

> S52: "*Sometimes you add features to an existing subsystem, which means extending existing code. Since sometimes*

TABLE 2: Ranking of methods used to create new device drivers

overall rank	method	respondents per rank[1] 1	2	3	4	total[2]
1	clone an existing driver and adapt it	43 %	32 %	7 %		83 %
2	develop from scratch	33 %	13 %	5 %	17 %	69 %
3	use a template	15 %	16 %	17 %	5 %	51 %
4	extend an existing driver by introducing variants using ifdefs	8 %	22 %	18 %	8 %	56 %

[1] the total number of respondents for this question was 62
[2] total per rank (row total)

TABLE 3: Reasons for introducing or maintaining scattered code

reason	stated by
avoid code duplication	44 %
support hardware variability (for limitations in hardware detection)	43 %
keep backwards compatibility	37 %
optimize the code for performance or binary size	33 %
avoid refactoring existing code with scattered features	24 %
introduce support for generic hardware devices	17 %
other	17 %

> *these are optional features that are not necessary or are dependent on hardware/software support, these get the ifdef treatment.*"

> S41: "*Complete new device type –> clone. Extend existing device with new version –> extend, however, avoid ifdef as much as possible. However, sometimes ifdef can prevent unnecessary inclusion of unused data objects.*"

> S24: "*I would use ifdefs in a driver only if I have configurable parts that can only be enabled/disabled before compiling it or if it has dependencies which I cannot control. It depends on how different the new hardware is. If the new hardware is just a variation of another part, extend with ifdefs or runtime checks.*"

In what follows, we analyze why developers introduce or maintain scattering. Table 3 shows results from the respective survey question. We discuss the four most frequent reasons.

(1) Avoid Code Duplication. The most voted reason for scattering is a common situation that leads to scattering using conditional compilation directives. Pre-processor directives are often added on very specific lines of code to implement variability while avoiding code duplication. Listing 2 demonstrates this situation with two of the code fragments from the Linux kernel affected by a commit[11] authored by interviewee I1, which increased the scattering degree of a feature called CMA (Contiguous Memory Allocator—allowing to allocate large, contiguous memory blocks) by 5. The commit introduced five new extensions of the feature in the files *include/linux/mmzone.h* (lines 44, and 64), *mm/page_alloc.c* (lines 753, and 899), and *mm/vmstat.c* (line 616). With these extensions in hand, we asked the commit author why he introduced them.

The author explained that scattering was introduced mainly to avoid code duplication. He further added that one alternative would have been to concentrate the variability in a single conditional compilation directive as much as possible, avoiding or limiting scattering, but the consequence would have been more duplicated code. For instance, moving the *ifdef* from *mmzone.h:44* (see Listing 2) to outside of the *enum* would have required that the enum be duplicated—one enum which would include MIGRATE_CMA to be defined within the #ifdef CONFIG_CMA block, and the second enum excluding MIGRATE_CMA to be defined in the #else block.

(2) Support Hardware Variability. Developers often use pre-processor directives to check platform settings to circumvent limitations in hardware detection. An example of devices with such limitations are platform devices, which we will discuss in Sec. 4.3.2. We asked the interviewees in what circumstances the device driver API was not sufficient for hardware detection. In general, they stated that some devices either cannot be detected entirely or do not provide enough information to accurately detect hardware differences (I1, I4, I10). However, one interviewee (I4) stated that this problem "seems to be rare in modern hardware."

> I1: "*Some buses may provide a way to probe a device and ask the device for its parameters, but that's not always available. For example, if a device is connected to some particular memory address range, there is no way to detect it and it has to be statically configured as being there.*"

> I10: "*When the bus the device is attached to doesn't enumerate devices as robustly as PCI (especially platform devices).*"

While the use of "data structures available at runtime, such as device-tree" (I4) may solve most of these hardware detection problems, it does not always work; firstly, because some buses do not have enumeration capabilities, and secondly, because some vendors, to save production costs, create hacks in the hardware design that make software architecture design particularly difficult (I2, I7). To react to such weird cases, developers rely on code scattering to overcome limitations in the kernel design:

> I7: "*Hardware is [...] not developed by software engineers or [...] reviewed by software engineers. [...] as a programmer you are used to, if you read a value you do not modify it. With some hardware devices it's the other way around. You read a register and you change the internal state. And these are really bad things, but in these cases you have an IFDEF in the Linux kernel configuration.*"

We also asked interviewees about circumstances when developers would prefer *ifdefs* to the device driver API and what the trade-offs would be. While some could not think of such situations (I1, I10), others stated that *ifdefs* should be used only when there is no other alternative, for instance, to address hardware limitations as discussed above (I1), and that drivers I2: *"should be self-contained as far as is possible,"* implying that drivers should be acting as plugins that do not have feature code scattered outside of the subsystem. However, one prominent exception raised to this was when optimization is necessary, that is, optimizing the resulting kernel binary image and how it performs during runtime (I4, I9). Due to limitations inherent in the kernel design, a developer would evaluate trade-offs with questions, such as, does it pay off to avoid scattering; would the cost of implementing a non-scattered solution be worthwhile? Should I invest in a one-size fits all design, which is very expensive to build, or should I build a good-enough solution that handles most of the cases, handling corner cases by scattering? Such trade-offs explain why many features of the *driver* subsystem

Listing 2: Scattering of CONFIG_CMA to avoid code duplication

```
//mmzone.h:44
enum {
        MIGRATE_UNMOVABLE,
        MIGRATE_RECLAIMABLE,
        MIGRATE_MOVABLE,
        MIGRATE_PCPTYPES, /* the number of types on the pcp lists */
        MIGRATE_RESERVE = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
        (...)
        MIGRATE_CMA,
#endif
        MIGRATE_ISOLATE, /* can't allocate from here */
        MIGRATE_TYPES
};

#ifdef CONFIG_CMA
# define is_migrate_cma(migratetype) unlikely((migratetype) == MIGRATE_CMA)
#else
# define is_migrate_cma(migratetype) false
#endif
```

[11] https://github.com/torvalds/linux/commit/47118af

are globally scattered (e.g., inside the *arch* subsystem), which leads to the global scattering observed in Sec. 4.1.2.

In addition to addressing limitations in hardware detection, 17 % of survey respondents stated that they introduce scattering to provide support for generic hardware devices, such as `USB`, `PCI`, and `ACPI`. We also discuss these in Sec. 4.3.2.

(3) Maintaining Backwards Compatibility. The third most voted reason for introducing scattering was maintaining backwards compatibility. We asked interviewees what compatibility meant in the context of Linux kernel development. Foremost among the explanations was that scattering is a mechanism for keeping backwards compatibility affecting user space (I1, I2, I4, I7, I10), but less so elsewhere, since interfaces within the kernel have no obligation of backwards compatibility.

> I2: "*It's particularly applicable to user-space interfaces: Linus has a policy of never breaking the user-space API, except in highly restricted circumstances.*"

> I4: "*This is almost never an issue in the context of the Linux kernel, because interfaces within the kernel have no obligation to maintain backwards compatibility. However, this can arise in user-space interfaces, when attempting to consolidate code.*"

In addition, maintaining backwards compatibility also involves scenarios where refactoring code that is well up and running is perceived as a regression risk not worth taking, or:

> I2: "*where a driver is restructured and provision of compatibility with the old version within the structure of the new might mandate the use of code scattering.*"

Avoidance of perceived regression risks is what accounts for the 24 % of survey respondents in Table 3 who stated that they avoid refactoring existing code with scattered features.

(4) Optimizing for Performance or Binary Size. Lastly, since this reason is one of the top four reasons for scattering feature code, we asked interviewees why this was so. They explained that (i) this is usually necessary when dealing with resource-constrained architectures, such as ARM (I3), (ii) when the exclusion of *ifdefs* in writing code that accounts for many variants would lead to bugs, such as someone using a feature that is not actually supported in a particular configuration of the kernel (I1), and (iii) to avoid code duplication, dead code, and unused data related to a feature (I1, I2, I4). One interviewee explains:

> I4: "*Runtime detection still leaves code in the final binary for the path not taken. Compile-time ifdefs eliminate that code. Most of the time, static inline functions and similar techniques allow avoiding scattering of these ifdefs. However, in occasional cases, code becomes more readable by repeating the ifdef more than once within the same file, to avoid repeating other code instead.*"

Other. We had nine other reasons for scattering, indicated as 'other, 17 %' in Table 3. Three of these are covered by our discussion of 'maintaining backwards compatibility' as a reason for scattering since respondents indicated that they opt to maintain scattered code because refactoring such code is perceived as a regression risk. Two have been ommitted for lack of clarity from respondents. The other four were: (i) sometimes it's the one-eyed among blind option, (ii) support OSes other than Linux, (iii) to group pieces of code on the functional basis where it is more natural to place them, and (iv) to introduce debugging version of a function. In the case of (ii), an example use of preprocessor directives would be the construction of a path either using the backslash (\) or forward slash (/)–which depending on the OS, or supporting OS specific data types. These are cases where makefiles may not suffice.

4.3.2 Types of Features

Motivated by previous experience [9], we investigated whether specific kinds of features exist that by their nature affect where a feature is scattered across (i.e., locally or globally) or that lead to higher scattering degrees. First, we used the code analysis (longitudinal study) to hypothesize on the types of features prone to scattering. The characteristics we tested are the result of past experience and observations when manually analyzing and classifying features in the Linux kernel [14, 21, 22] and other systems [41]. Later, we verified our assertions with data obtained from the interviews with developers.

The first kind of features we observed relates to *platform* devices. As opposed to *hotplugging* devices, platform devices cannot be discovered by the CPU. Corbet [29] explains[12] that, while any device sitting on a bus like PCI (with built-in discoverability) can tell about its type of device and its resources; platform devices, however, are not discoverable and still need the kernel to provide ways to be told about the hardware that is actually present.

In Linux, a platform driver is any driver that uses the C structure `platform_driver`. Since platform devices are not discoverable by the CPU, the kernel cannot automatically load their LKMs, as in hotplugging. Instead, board-specific code [42] 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 *arch*, which contains CPU-specific 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 scattering-prone features concerns abstractions that provide a core *infrastructure* for developing drivers. 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 (such as audio or network devices),[13] and hardware-description frameworks (e.g., OpenFirmware). Since these features denote abstractions in the operating-system domain, we assume that they should be more likely to be scattered compared to non-infrastructure features. In such cases, extensions in code would check for specific generic functionality and related capabilities,allowing features to react accordingly.

We investigated whether the kind of a feature affects the scattering locality (i.e., local or global) or scattering degree.

Influence on Scattering Locality. We tested the effect of being a platform feature on scattering locality by collecting a random sample of 10 % of all scattered driver features (population size is

[12] http://lwn.net/Articles/448499/

[13] http://www.kernel.org/pub/linux/kernel/people/mochel/doc/text/class.txt

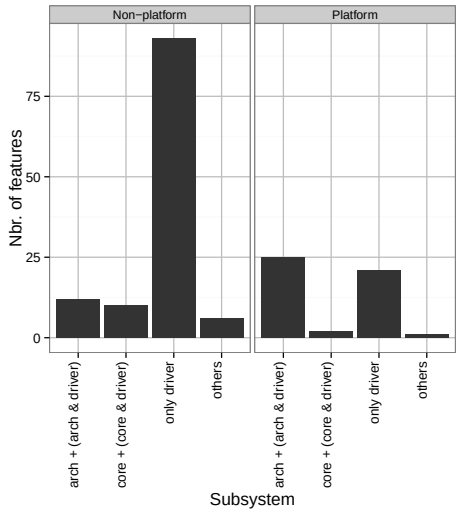


Fig. 11: Scattering location of sampled (non-)platform drivers

1,700) and by manually classifying them as either platform or not. A platform-driver feature is either part of 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 criteria we used, see our online appendix [18]. We performed the χ^2 statistical test on the sample with at a significance level of 0.05. See appendix (Sec. 4 and Table 7) for the hypotheses and results.

In summary, we found strong evidence ($p = 1.933 \times 10^{-5} < 0.05$) of a dependency between being a platform-driver feature and scattering locality. In fact, our analysis 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. 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. 11 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* ('either' is captured by the '+' sign, whereas 'and' is denoted by '&'). This is evidence for 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 of such devices requires scattering CPU-dependent code, which is mostly found inside the *arch* subsystem.

To analyze the influence of infrastructure features on scattering locality, we classified the same sampled features as either infrastructure or not, and performed the same statistical test as before (see appendix Sec. 4 and Table 8). However, there was no strong evidence suggesting that being an infrastructure-driver feature affects scattering location; the χ^2 test resulted in a *p*-value greater than the chosen significance level.

Influence on Scattering Degree. To analyze the influence of being a platform or an infrastructure-driver feature on scattering degree, we calculated the average *SD* value of each sampled feature across all releases containing it. We performed two one-tailed Mann-Whitney-Wilcoxon rank sum statistical tests to assess whether: (i) the platform-driver features and (ii) the infrastructure-driver features in this set systematically yield higher average *SD* values as opposed to all other features.[14] See Sec. 4 in the appendix.

Overall, we did not find convincing evidence that average *SD* values are systematically higher in platform-driver features.

Likewise, there also seems to be no significant influence of being an infrastructure-driver feature on scattering degree. The test only supports the null hypothesis (i.e., that there is no difference in the distribution of average *SD* values of platform and non-platform driver features). However, in the case of the outliers observed in RQ1.3 (see Fig. 8), we did find some influence of being an infrastructure-driver feature on extremely high *SD* values, as 9 out of the 15 most scattered driver features in the kernel evolution were infrastructure features. For more details see the appendix.

Developers' perspective. To better understand our results, we asked our interviewees about our observations with regard to platform and infrastructure driver features. Interestingly, all confirmed our understanding that these types of features are prone to scattering due to hardware limitations in discoverability, as we discussed in Sec. 4.3.1. For instance, two interviewees expressed:

> I1: "*Platform drivers don't surprise me. [...] the more low level, the harder it may be to probe/detect a device. Unless we know where the device is, we may be unable to discover it and even if there is a way to discover its existence, it may not provide a way to ask for its parameters.*"

> I10: "*I believe OF [an infrastructure driver feature] is scattered so widely because it is the primary, evolving, interface for device discovery for SOC systems [...]. I think we're stuck with this because of the way ARM support was driven into the kernel. Several competing vendors trying to find the most expeditious (not best) common framework to integrate their disparate implementations.*"

In response to our question "Are there other kinds of devices that cannot be detected? Why can they not be under the hood of the device driver API?" two interviewees stated:

> I9: "*Some HW needs to be probed by direct register read, if not present you will read an invalid pointer.*"

> I10: "*Any device that isn't attached to an enumerable bus. That's devices on most any system that isn't based on PCI. The device driver API would need to support an architecture independent device enumeration interface. We currently have a hardware based enumeration (PCI) and a software based enumeration (devtree). There may be more. But a layer of abstraction above the existing enumeration mechanisms would be needed to fully fold discovery under the device driver API.*"

Since most of device drivers are PCI-based, the Linux kernel provides a good enough abstraction to handle most common drivers. In cases where the software architecture fails to integrate new drivers (e.g., those very close and dependent on specific CPU architectures), scattering is used as an alternative solution.

Lastly, we asked interviewees what scenarios, in general, make some features prone to scattering. As expected, the most prominent scenario is the non-discoverability of devices. Secondly, we observed that scattering is indeed a choice in some particular

[14] "Systematically" means that the probability of having an average *SD* greater than a value X among platform/infrastructure-driver features is greater than the probability of having an average $SD > X$ among other features [43].

cases. Developers do it consciously based on trade-off decisions they make, such as for optimization. In addition, time-to-market concerns by the vendors contributing device support also make some options prone to scattering.

> I10: " *[...] especially [from] vendors that don't yet have a long term relationship with the kernel development community. When a TI or an Altera (before Intel) start building Linux support for their products, they often start in a vacuum. Their development cycle is based on time-to-market, not kernel design and support. Then, they partner with a large distribution for help getting their code submitted to the kernel. There's some pushback from the community. But in the end, it's more expeditious to take their submission and work on cleaning it up than it would be to lose the device support.*"

4.4 Coping with Scattering (RQ3)

As stated above, 80 % of survey respondents perceived scattering as challenging. Some of the reasons for this perception are that: scattering complicates program comprehension (84 %), introduces more bugs (53 %), decreases code quality (51 %), hinders evolution (46 %), and lowers patch acceptance (41 %). With RQ3, we tried to understand what strategies developers employ to cope with challenges posed by scattering. Recall that, as observed in Sec. 4.1.1, the proportion of scattered driver features is nearly constant throughout the eight-year history of the kernel. So, it is natural to ask whether developers might consciously manage scattering degrees or enforce limits in practice, or whether they limit their use of *ifdefs*, thereby limiting scattering?

(1) Adherence to Coding Guidelines. The Linux kernel development documentation provides coding guidelines that advise developers to avoid the use of *ifdefs* in code. Specifically, the *patch submission guide*[15] states: "Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define 'static inline' functions, or macros, which are used in the code. Let the compiler optimize away 'the no-o' case." Furthermore, the kernel development process[16] reinforces this as follows: "Conditional compilation with #ifdef is, indeed, a powerful feature, and it is used within the kernel. But there is little desire to see code which is sprinkled liberally with #ifdef blocks. As a general rule, #ifdef use should be confined to header files whenever possible."

We asked our survey respondents if they adhere to the above coding guidelines to avoid the use of *ifdefs*. Table 4 indicates that almost all participants try to avoid *ifdefs* following different strategies, chief among them being those given by the Linux kernel coding guidelines, that is, the use of static in-line functions (77 %) and putting *ifdefs* into header files (54 %). These strategies have the potential to limit or reduce scattering as a consequence.

Putting *ifdefs* only in header files prevents a feature from being scattered in different parts of the same code file or in undisciplined ways (not aligned with the structure of the language) [8]. Specifically, scattered feature code is refactored into a (potentially parameterized) static inline function, whose body (the feature code) is protected by an *ifdef* with an else branch that will return null. So, if the feature is disabled, the compiler will optimize away the inlined function. So, using static inline functions, any other occurrence of the feature code is replaced by a function call without a surrounding *ifdef*, effectively reducing scattering.

TABLE 4: Strategies for avoiding the use of *ifdefs*

strategy	stated by
use static inline functions	77 %
put ifdefs into header files	54 %
use selection statements (e.g., if and switch) from the C language instead of ifdefs	48 %
use optional code confined in one single ifdef	14 %
other	14 %
reject patches containing ifdefs	11 %
none (no attempt to avoid using ifdefs)	5 %

TABLE 5: Strategies for limiting scattering

strategy	stated by
improving system design	73 %
refactoring existing code	73 %
copying code (allowing duplicates)	13 %
none (no attempt to limit scattering)	9 %
other	5 %
enforcing max. number of ifdefs an option can appear in	2 %

One of the most voted strategies is the use of selection statements from the C language (*if* or *switch*) instead of *ifdefs* (59 %). From a practical perspective, this allows the compiler to syntax- or type-check even disabled code, which would not be seen by the compiler when cut out by the pre-processor (since the build always relies on one specific feature selection). Yet, it does not alleviate the feature scattering problem as such.

(2) Conscious Management of Scattering (Better System Design). To learn if the survey respondents consciously manage feature scattering (not only *ifdefs*), we asked them if they try to *limit* it, and if so, how. Table 5 indicates that only 10 % do not try to limit scattering, but the majority (90 %) uses at least one strategy to limit it. The most used strategies are (i) improving system design (81 %), and (ii) refactoring existing code (74 %). These results indicate that developers indeed do consciously manage feature scattering in the Linux kernel by actively and proactively limiting it. However, generally they do not enforce any limits by use of a threshold as only 2 % (one respondent) reported doing so.

In addition, using a Likert-scale of 1 (strongly disagree) to 5 (strongly agree) and 3 as neutral, we asked survey respondents which of the suggested strategies they perceived suitable to completely *avoid* scattering. Fig. 12 indicates that the trend of responses leans towards using *a better system design* (86 %) and *better modularity mechanisms in the C language* (58 %). And, as expected, the majority (72 %) disagreed that code duplication could be one such strategy. Recall that in Sec. 4.3.1, we found that avoiding code duplication was the top most reason for why developers of the Linux kernel introduced scattering, hence the disagreement observed here. One other proposed option was, 'I don't see any alternative', to which many respondents expressed neutrality (46 %) or agreement (18 %). This apparent hesitation to adopt the alternatives proposed might indicate an inclination towards the code-size and performance optimization trade-offs discussed earlier in Sec. 4.3.1. Investigating this in a follow-up study would be valuable future work.

> S12: "*Mostly I try to follow the Linux guidelines, since they are good practice hints. However, there may be exceptions when scattered ifdefs are needed for goals like performance, code size and so on.*"

(3) Conscious Management of Scattering (Refactoring). Since

[15] https://www.kernel.org/doc/Documentation/SubmittingPatches

[16] https://www.kernel.org/doc/Documentation/development-process/4.Coding

refactoring was stated by 74 % of the survey respondents as one of the top strategies for limiting scattering, we asked interviewees: 'When do you refactor a scattered configuration option? Is there a tipping point that triggers such a decision?' Even though there was no general agreement, some interviewees stated that refactoring is only performed in instances where either the targeted *ifdefs* are no longer needed (I1, I7, I8) or the refactoring does not break the logical structure of the driver code and offers code optimization and performance trade-offs (I2). The decision to refactor is often made when the developer realizes that making a new change to the code is made difficult by the scattered feature, at which point trade-offs are evaluated, and if the refactoring is worth the effort then it is performed (I1, I2, I4). Even though interviewee I2 stated that no common approach to refactoring exists, as each driver and piece of hardware is different, some refactoring strategies include the use of static inline functions and header files as provided by the Linux kernel coding guidelines, and the use of *"macros that serve a similar function, such as for access to a structure field that may not always exist,"* (I4), provided that specific values signal for the absence of a given field information. Lastly, interviewee I4 suggested an interesting strategy:

> I4: "*In the future: make compilers smarter, to avoid needing certain config options at all. For instance, compilers with link-time optimization (LTO) can include smarter ways of detecting unused code or data throughout the program, and omit that code or data without requiring an explicit config option for it.* "

In summary, we observed that developers focus their efforts on removing *ifdefs* individually, but did not find any indication that they take the overall scattering degree of a feature into account.

5 Threats to Validity

External Validity. A significant 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 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) relate to scattering degrees and lead to global scattering, we performed hypothesis testing based on a sample of 170 scattered features (population size is 1,700), since it required a manual classification of features. This sampling is justified, and we rely on standard p-value limits to test hypotheses.

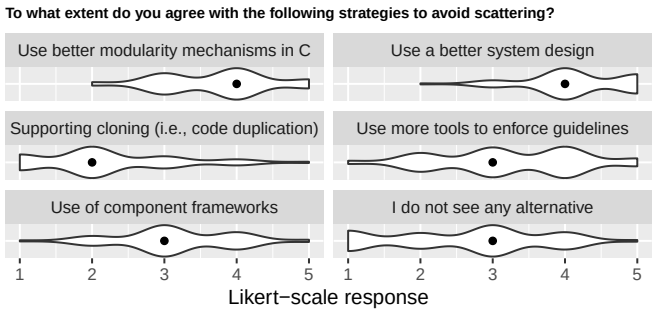


Fig. 12: Strategies for avoiding scattering (*1–completely disagree, 2–disagree, 3–neutral, 4–agree, 5–completely agree*)

Recall that the investigation of outliers does not rely on sampling, but on classifying the whole population (54 features).

Our analysis of code scattering relied 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 only a partial, yet valid, view of the true story. Our results can be complemented by studying code scattering on the more coarse-grained source file level. Using this information would be valuable future work.

Finally, the majority (66 %) of our survey respondents and interviewees work as professional developers in different companies besides contributing to the Linux kernel. Hence the insights they provided on feature scattering may not be specific only to the Linux kernel but may be applicable to other systems.

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. Two authors inspected the code for almost 16 hours in total, and created over 70 test cases.

For all code analyses, we excluded features that we could not uniquely map to one subsystem. This limitation, however, has no further influence on our results, as only very few driver features (0.65 ± 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 influence on maintenance, as opposed to the code parts controlled by pre-processor directives, which we analyze.

To avoid limiting conclusions to individual perspectives, the survey covers a broad range of roles of respondents that contribute to more than one subsystem of the Linux kernel. In addition, owing to the substantial technical and industrial experience of our interviewees, our work provides both a general perspective on feature scattering as well as in-depth insights on technical issues.

Construct Validity. To measure scattering of feature code, we relied 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.

It is not completely clear, however, 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 [44].

With respect to the survey and interviews, we limited the effect of potential subjectivity by considering statements affirmed by more than one interviewee and several survey responses. Introductory explanations to non-trivial questions were given to respondents to mitigate misinterpretation. Furthermore, to ensure completeness, both survey respondents and interviewees were given opportunity to provide additional answers not covered by devised questions, but relevant.

6 Related Work

Feature scattering is part of a broader research topic: *scattering of concerns*. Concerns do not only comprise features [45], but also requirements, design elements, design patterns or programming idioms [17, 36]. The representation of features in the Linux kernel [21, 41, 46] can be compared to concern models [5], which map

concerns to code and support concern location. A particular research interest has been on cross-cutting concerns, which are under general suspicion to negatively influence quality and maintainability.

Eaddy et al. [36] investigate the relationship of cross-cutting concerns to defects, arguing that insufficient modularization can lead to increased defects. For instance, code maintenance might miss parts of the implementation, leading to inconsistencies. Furthermore, concerns might be tangled with other concerns, so changing one concern might accidentally affect the other. The authors analyze three open-source projects written in Java, where they manually mapped concerns to classes and bugs. They identified a high correlation between scattering and defects, regardless of the system's size. Their manual concern identification technique was proposed in previous work [35], where they argued that automated concern location through execution traces would have been incomplete, missing 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. [47] present a longitudinal case study on feature scattering in four, relatively small (24 KLOC to 177 KLOC) open source Java projects. They use dynamic analysis to trace classes implementing features, together with formal concept analysis [48] to analyze the feature-to-class/method mapping. Their results show an almost continuous increase of scattering, and the Gini coefficient of the scattering degrees increasingly fluctuates over time. Interestingly, feature implementations increasingly accumulate in already large classes. Their study inspired our use of the Gini coefficient, and we also observed an almost linear scattering increase. However, their notion of scattering is very different from ours, as they consider all code involved in the control-flow when executing a feature, which may include methods not related to the feature implementation. Consequently, despite small projects, they observe very high scattering degrees (up to 1,467 methods). We did not consider such dependencies.

Several studies investigate structural characteristics of cross-cutting concerns. Figueiredo et al. [17] identify patterns of cross-cutting concerns in the source code of three case studies. Their catalog of 13 patterns 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 patterns and design stability. In a product-line re-engineering effort, Couto et al. [49] find that six of their total of eight features follow the octopus pattern—one of Figueiredo's et al's patterns.

In our study, the features we considered were all optional features—that is, units of variability—which can be switched on or off using the pre-processor. Since optional features are the predominant type of features in the Linux kernel [21], this focus is justified. However, features are typically used as units of functionality [45] and rather mandatory. As such, they are used for communication, planning or keeping an overview understanding on the development, among others. Since they are not optional, their locations are usually not explicitly recorded in the source code and instead need to be recovered—a discipline called feature location [50–55]. To avoid feature-location efforts, researchers have proposed to use embedded annotations to record feature locations directly in the code (similar to ifdefs) together with respective tooling and case studies [56–59]. Notably, Krüger et al. [44] manually annotated feature locations of mandatory features in the open-source 3d printer firmware Marlin and found that their code-level characteristics differ from optional features. Specifically, mandatory features are less scattered, probably since they align better with the software architecture. Since no substantial history of such a project with annotated mandatory features exists, we cannot conduct a longitudinal study and contrast the results, which however would be an interesting future work.

Finally, researchers have also extracted realistic thresholds for source-code metrics as a prerequisite to assess quality and maintainability of systems. For instance, Oliveira et al. [60] calculate thresholds for common source-code metrics upon 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. Similarly, Queiroz et al. [9] propose thresholds by analyzing the statistical distribution of three feature-related metrics (scattering degree, tangling degree, and nesting depth of pre-processor annotations) collected from 20 pre-processor-based systems. 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 studied scattering of feature code in the Linux kernel by analyzing almost eight years of its evolution history, surveying 74 kernel developers, and interviewing nine of them. Our goal was not to investigate limitations or maximum degrees of scattering, but to obtain empirical data on whether and how scattering can be handled to the extents found in one of the largest feature-based systems in existence today.

We learned that the majority of driver features (82 %) is introduced without causing scattering (RQ1.1). 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 expected. Proportionally, however, the amount of scattered features remains nearly constant throughout the kernel's evolution. We also found that scattering is not limited to subsystem boundaries (RQ1.2). 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 majority (75 %) of scattered driver features is scattered across moderate four to eight locations in code (RQ1.3). Moreover, the median is low and constant across the entire evolution ($SD = 4$). Yet, the distribution is skewed, 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; even though their absolute counting and magnitude grow with the system size.

We identified and analyzed two kinds of features that are prone to scattering. *Infrastructure* features account for 9 out of the 15 most highly scattered features (outliers) in the scattering distribution of driver features, affecting many parts of the code. *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 the scattering degree occur within non-infrastructure outlier features, where platform-driver features account for most of the outliers in that group. While 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. This result was confirmed by our survey and interviews.

From the developers' perspective, we found that feature scattering is introduced in the Linux kernel for four main reasons (RQ2): avoid code duplication, support hardware variability due to limitations in hardware detection, support backwards compatibility for the user-space and preservation of the logical structure of drivers, and optimize code for binary size and performance. As established during code analysis, we also confirmed that features most prone to scattering are those relating to platform devices, which cannot be discovered automatically by the CPU. Upon evaluating maintenance and performance trade-offs, developers do try to avoid feature scattering mostly by improving system design, such as by adhering to coding guidelines to use static functions and placing *ifdefs* in header files, and refactoring existing code, but they do not consciously maintain a scattering threshold (RQ3).

Our results suggest the following research directions. First, considering that full feature modularity is almost impossible to achieve, and also does not appear to be necessary according to our study, we suggest considering lightweight modularity techniques. An interesting technique could be module projections [61–63], where feature code is physically scattered, but developers can easily obtain projections of modules in their IDE. Second, feature scattering should already be considered when designing the system architecture, accounting for some extent of scattering and possibly a limited number of highly scattered outliers. Identifying early indicators for how well architectures cope with feature scattering, or conceiving decision support for making the right design decisions with respect to feature scattering, appears to be an interesting research direction. Finally, recall our observation that scattering partly originates from badly designed hardware. Improving hardware design methodologies, potentially incorporating principles known from software design could lead to better hardware design practices that alleviate this problem.

Acknowledgments

Supported by Vinnova Sweden (project 2016-02804), the Swedish Research Council (project 257822902), and the German Research Foundation (AP206/5, AP 206/6, and AP 206/11).

References

[1] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C news," in *USENIX*, 1992.

[2] G. Krone, M.; Snelting, "On the inference of configuration structures from source code," in *ICSE*, 1994.

[3] J.-M. Favre, "Preprocessors from an abstract point of view," in *ICSM*, 1996.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997.

[5] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *Transactions on Software Engineering Methodologies*, vol. 16, no. 1, 2007.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines: Concepts and Implementation*. Springer, 2013.

[7] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE*, 2010.

[8] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *AOSD*, 2011.

[9] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki, "The shape of feature code: an analysis of twenty c-preprocessor-based systems," *Software & Systems Modeling*, vol. 16, no. 1, pp. 77–96, 2017.

[10] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: multi-dimensional Separation of Concerns," in *ICSE*, 1999.

[11] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information hiding interfaces for aspect-oriented design," in *ESEC/FSE*, 2005.

[12] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 162–180, 2008.

[13] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *SPLC*, 2011.

[14] 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.

[15] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo, "Feature-oriented software evolution," in *VaMoS*, 2013.

[16] 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 *CSMR*, 2008.

[17] 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 *ICPC*, 2009.

[18] "Online appendix," https://github.com/Mukelabai/featurescattering18/.

[19] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: a longitudinal study of Linux kernel device drivers," in *MODULARITY*, 2015.

[20] P. Rothbauer, "Triangulation," *The SAGE encyclopedia of qualitative research methods*, vol. 1, pp. 892–894, 2008.

[21] 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*, vol. 39, no. 12, pp. 1611–1640, 2013.

[22] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE*, 2010.

[23] Kbuild, "The kernel build infrastructure," www.kernel.org/doc/Documentation/kbuild, last seen: Feb. 14th, 2015.

[24] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the Linux kernel," in *VaMoS*, 2010.

[25] 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: Feb. 14th, 2015.

[26] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the Linux Kernel a software product line?" in *OSSPL*, 2007.

[27] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *SPLC*, 2010.

[28] 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*. ACM, 2012, pp. 15–20.

[29] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*, 3rd ed. O'Reilly, 2005.

[30] S. Venkateswaran, *Essential Linux device drivers*, 1st ed. Prentice Hall Press, 2008.

[31] M. W. Godfrey and Q. Tu, "Evolution in open source software: a case study," in *ICSM*, 2000.

[32] C. Izurieta and J. Bieman, "The evolution of FreeBSD and Linux," in *ESEM*, 2006.

[33] D. G. Feitelson, "Perpetual development: a model of the Linux kernel life cycle," *Journal of Systems and Software*, vol. 85, no. 4, pp. 859–875, 2012.

[34] 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: Feb. 14, 2015.

[35] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *ACoM*, 2007.

[36] 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*, vol. 34, no. 4, pp. 497–515, 2008.

[37] M. Kasunic, *Designing an effective survey. Technical report, handbook CMU/SEI-2005-HB-004*. Software Engineering Institute, Carnegie Mellon University, 2005.

[38] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, 2015.

[39] M. Hubert and E. Vandervieren, "An adjusted boxplot for skewed distributions," *Computational Statistics & Data Analysis*, vol. 52, no. 12, pp. 5186–5201, 2008.

[40] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *ICSM*, 2013.

[41] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, "Feature-to-code mapping in two large product lines," in *SPLC*, 2010.

[42] M. T. Jones, "Anatomy of the Linux kernel," *IBM Developer Works*, 2009.

[43] D. S. Moore, G. P. McCabe, and B. Craig, *Introduction to the practice of statistics*, 6th ed. W. H. Freeman, 2009.

[44] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: a case study of marlin," in *VaMoS*, 2018.

[45] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.

[46] S. Nadi and R. Holt, "The Linux kernel: a case study of build system variability," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 730–746, 2014.

[47] T. Chaikalis, A. Chatzigeorgiou, and G. Examiliotou, "Investigating the effect of evolution and refactorings on feature scattering," *Software Quality Journal*, pp. 1–27, 2013.

[48] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*, 1st ed. Springer, 1997.

[49] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: a case study using conditional compilation," in *CSMR*. IEEE, 2011, pp. 191–200.

[50] S. Paul, A. Prakash, E. Buss, and J. Henshaw, "Theories and techniques of program understanding," in *CASCON*. IBM Press, pp. 37–53.

[51] M. P. Robillard and G. C. Murphy, "Feat: a tool for locating, describing, and analyzing concerns in source code," in *ICSE, Demonstrations Track*, 2003.

[52] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[53] J. Rubin and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*, 2013.

[54] A. Burger and S. Gruner, "Finalist 2: Feature identification, localization, and tracing tool," in *SANER*, 2018.

[55] J. Krüger, T. Berger, and T. Leich, *Features and how to find them: a survey of manual feature location*. LLC/CRC Press, 2018.

[56] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining Feature Traceability with Embedded Annotations," in *SPLC*, 2015.

[57] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *VaMoS*, 2017.

[58] M. Seiler and B. Paech, "Using tags to support feature management across issue tracking systems and version control systems," in *REFSQ*, 2017.

[59] H. Abukwaik, A. Burger, B. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *ICSME, NIER Track*, 2018.

[60] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *CSMR-WCRE*, 2014.

[61] M. Voelter, J. Warmer, and B. Kolb, "Projecting a modular future," *IEEE Software*, vol. 32, no. 5, pp. 46–52, 2015.

[62] B. Behringer, J. Palz, and T. Berger, "Peopl: projectional editing of product lines," in *ICSE*, 2017.

[63] M. Mukelabai, B. Behringer, M. Fey, J. Palz, J. Krüger, and T. Berger, "Multi-view editing of software product lines with peopl," in *ICSE, Demonstrations Track*, 2018.