Code contribution practice in Linux kernel: How product structure shapes organization

Ben Trovato*
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin†
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld‡
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

Lawrence P. Leipuner
Brookhaven Laboratories
Brookhaven National Lab
P.O. Box 5000
lleipuner@researchlabs.org

ABSTRACT

We investigate how the organization of contribution team and culture are affected by product/module structure in Linux kernel. In FLOSS projects, code commit privilege is often employed to ensure code quality. Code committers are gatekeepers of code repository, and responsible for committing code for contributors who author the code but don't have the privilege to commit. The ratio of number of authors over number of committers represents how the contribution team is organized and suggests a congruency of work load and communication in the team. Different modules of Linux kernel present different structures, which are associated with different business requirements and different contributor activities.

Using code change history and developer interviews we investigate if the modules of Linux kernel with different structures have different ratios and how they evolve in the diverse circumstances. We find that a) Module structure has a dramatic effect on the organization of contribution team (learning reproduces organization through product structure.). In particular, the loose coupled (well modularized) modules like drivers would have a team size fluctuating at 20 (one committer works for 20 authors), and close coupled modules like kernel would have a size of 10. b) The module structure is shaped by business requirements and contribution inflow. c) Compact team tends to work together closely and have a shared control on code. On the contrary, loosely organized contribution team tends to work has stricter code ownership. We expect our findings could be used to improve FLOSS project contribution process by describing ways of organizing contribution teams according to product structure characterized by features and contribution activities. The findings also suggest that software teams maintaining legacy modules are likely to maintain the original (initial) culture and may be able to adjust to changing environment but with a delay.

Keywords

product structure, code commitment organization

1. INTRODUCTION

Linux kernel has been a legend in the Free/Libre and Open Source Software (FLOSS) world. Linux kernel based distributions (operating systems) established a commercial success that many other FLOSS projects would like to pursue. For example, they take 98.8% positions in the top 500 fastest supercomputers in Nov 2015[1]. The statistics from monitoring a substantial number of web sites during the last twelve months (until Dec 2015) show Linux kernel based web clients have a 28.89% share in the market.

The success of Linux kernel can not be achieved without the contributions from participants. Starting from Linus Torvards, volunteers played a crucial role in the development of linux kernel. However, the number of volunteers (unpaid developers) contributing to the Linux kernel has been slowly declining for many years, now sitting at just 12.4% (it was 13.6% in 2014, and 14.6% in 2013). Meanwhile, commercial participation substantially grows in recent years, large companies like RedHat and Intel put substantial resources on the development of Linux kernel, e.g., Intel contributed 10.5% of changes, ReHat 8.4% in 2014[2]. Now more than ever, the de-

*Dr. Trovato insisted his name be first.
†The secretary disavows any knowledge of this author's actions.
‡This author is the one who did all the really hard work.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]https://en.wikipedia.org/wiki/Usage_share_of_operating_systems
[2]https://s3.amazonaws.com/storage.pardot.com/6342/120970/lf_pub_wh

velopment of the Linux kernel is a matter for the professionals, as unpaid volunteer contributions to the project reached their lowest recorded levels in the latest "Who Writes Linux" report[3].

Apparently, Linux kernel experienced dramatic change since the very beginning, e.g., expanding of substantial code and increasing of commercial participation. Different modules of Linux kernel present different nature and attract different contributors. For example, the module of drivers accounts for the largest proportion of linecount (56.6%) in the kernel, probably because various of hardware manufacturers have been trying to get their drivers into the kernel and devoted substantial effort for their cause. As for the module of kernel, the very central one, attracting the most capable and ambitious developers in the world, takes only 1.2%. According to Conway's law [**?**], the structure of software reflects the communication structure of people writing it. It is of interest to understand how different modules present different structure, whether or not that is associated with their contribution practice, In particular, we aim to answer the following questions:

- What are the structures of different modules of Linux kernel?
- Do different modules of Linux kernel have different contribution practice from each other?

On the one hand, it may help us understand the new challenges in the new FLOSS landscape like commercial participation. On the other hand, the understanding can help us utilize the best practices and amplify the effect.

We retrieve the code commits from the mainline repository of linux kernel, and use the data to analyze module structure and quantify the community contribution practice. We observed that different modules of Linux kernel have different structures in terms of modularity. In particular, the module of drivers is better modularized than other modules.

We focus on two particular factors that try to depict contribution practice. The first one is the ratio of number of authors to number of committers(A2C), which tries to measure how many authors each committer commits code for on average, i.e. team size. This factor represents contribution organization. We found that 1), the module of drivers is unique among all the modules in terms of having the biggest team size; 2), the ratio of A2C is decreasing over time for most of modules, even though both the number of authors and committers are increasing (the module of drivers is the single module that has a sharp increase for both that is different from the other modules), apparently, the increase of committers is faster than that of authors. That may suggest a more professional organization of the working teams has been happening in the community; 3), the ratio of A2C on each module correlates with the number of new joiners, and the number of new LTCs. This may imply that a looser control of working team brings more outsiders which requires attention from the community.

The second factor to depict contribution practice is code ownership, defined by number of committers per file. This factor measures how close the cooperation is. We found that the core modules of Linux kernel, like *mm* and *kernel*, have stronger code ownership than other modules.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the research methodology used in our study. Section 5 presents the results of our study. Section 7 discuses limitations of our study. We discuss practical implications of our empirical results in Section 6 and conclude in Section 8.

[3]http://linux.slashdot.org/story/15/02/18/1745246/torvalds-people-who-start-writing-kernel-code-get-hired-really-quickly

2. BACKGROUND AND RELATED WORK

The nature and performance of FLOSS development are subject of numerous investigations, but studies on evolution of contribution practice along the change of project landscape particularly commercial participation are far less common.

An early study of Apache web server and Mozilla web browser [**?**] quantified various aspects of OSS development practices. The results were framed as seven hypotheses that outline key aspects of OSS development. In this paper we focus on a different aspect of contribution organization.

Community strategies and practices are often addressed in the literature, e.g., community architecture [**?**, **?**], license and intelligence property management mechanism [**?**], and code commit privilege or ownership control [**?**, **?**, **?**], etc. Meneely and Williams [**?**] examined the relationship of the number of developers working on parts of the Linux kernel with security vulnerabilities. They found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability.

Unlike in prior work, we observe the contribution practice presented by Linux kernel experiencing various of technical and economic landscape and quantify the important aspects of the development that are likely to be affected by the product structure: the organization of author to committer team.

According to Conway's law [**?**], the structure of software reflects the communication structure of people writing it. Conway's law emphasizes the effect on the artifacts induced by social activities, and provides insights on how to look at software development through the perspectives of organizational science.

3. METHODOLOGY

3.1 Data preparation

We retrieved all the changes from the mainline repository of Linux kernel[4]. We took steps to clean and standardize the data, and obtained a data level for the further analysis. According to the following rules, we obtained the final changes for this study: 1), we only look at changes to *.c* files. We consider *.c* files to be important files as they are implementation of functionality in Linux kernel. These changes account for 65.86% of all changes. 2), we consider the first level directory retrieved from the file path as the module used in Linux kernel. Overall we obtained 22 modules. The top seven modules, in terms of number of changes, include drivers, arch(architecture), net, fs(file system), sound, kernel, mm(memory management).

Table 1 shows an observation used for this study. Each observation corresponds to a change. It records who and when writes the code, who and when commits the code, and

[4]git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

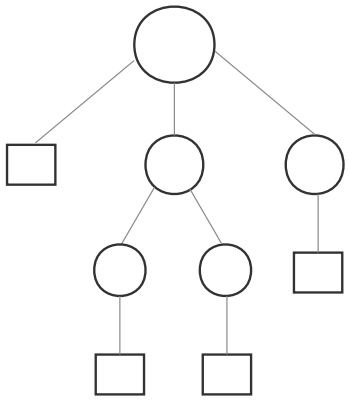

Figure 1: An example of directory structure

which file this change is made on. The author and the committer may be the same person, but most of the time (74%) they are not. The changed file is represented by its directory in the code repository. For example, "drivers/pci/iova.c" illustrates that the changed file is under the directory of pci driver. The last column named module in the table is the first level directory retrieved from the file path.

4. METRICS

Modularity.

Modularity is one useful aspect module structure. We propose a method to compare modularity of different modules of Linux kernel. Source code tree of Linux kernel is organized mainly according to functionality, thus we use directory structure to quantify modularity of one given module. While defining modularity, we have three considerations:
1) one directory may have subdirectories. As an example, in Linux kernel, the *drivers* directory has a sub directory named *staging*, which contains numerous subdirectories with many drivers. All of the contained drivers in *drivers/staging* are drivers that need more development before being added to the mainstream kernel. We will consider the smallest granularity

2) one module's submodules may differ greatly in terms of number of changes. For example, module A has two submodules with 10,000 and 100 changes respectively, and module B has two submodules with 10,000 and 10,000 changes respectively, then we will say module B has a higher degree of modularity than A; 3) modularity of one module may change over time, thus we take three years as a window, sliding month by month.

Considering three aspects above, we have the definition of modularity of one module in a given 3-year period

$$M = \sum_{i=1}^{N} -p_i \log p_i, \tag{1}$$

where N is the number of submodules of the given module, and p_i is defined by

$$p_i = \frac{n_i}{\sum_{i=1}^{N} n_i}, \tag{2}$$

where n_i is the number of changes of the ith submodule.

Ratio of authors to committers.

We focus on one particular factor that tries to measure the team relationship between code authors and code committers. Code authors in this study are people who write the code that gets into the mainline repository. Code committers are people who have the privilege to write the repository. A committer may be responsible for a specific module (functionality), so responsible for committing whoever's code on that module.

Naturally, the ratio of number of authors over number of committers may represent the load of a committer, or the difficulty (easiness) of the module, or, how hard it is for the author to get her code in. And the change of the ratio on the same module may represent the change of project landscape (e.g., new features may attract more authors but committers may keep at the same level as before), or the maturity of the module (e.g., a mature module may reduce committers).

Code ownership.

Ownership is a key aspect of large-scale software development, a valid proxy for expertise. Strong ownership, i.e., a single key developer responsible for a particular component in a system (whether it be a file, class, module, plugin, or subsystem), might be more effective in carrying out all tasks consistently and to completion [**?**]. Low communication overhead is required. External communications occur through a single channel. However, the disadvantages are obvious. It might be necessary to trade overall development time for development efficiency. The system will have a low truck number – only one developer need be hit by a truck to kill the project. Rogue individuals might "own" their code to the exclusion of organizational goals. It's common for commercial organizations to enforce strong code ownership in order to achieve good quality, see, e.g., [**?**]. Given the informal, distributed way in which FLOSS projects like linux kernel are built, it seems that rather than any single individual writing all the code for a given module, those in the core group have a sufficient level of mutual trust that they contribute code to various modules as needed [**?**]. In FLOSS projects, code "ownership" to be more a matter of recognition of expertise than one of strictly enforced ability to make commits to partitions of the code base.

One measure of ownership is how much of the development activity for a component comes from one developer. If one developer makes 80% of the changes to a component, then we say that the component has high ownership.

The proportion of ownership (or simply ownership) of a contributor for a particular component is the ratio of number of commits that the contributor has made relative to the total number of commits for that component. Thus, if Cindy has made 20 commits to ie9.dll and there are a total of 100 commits to ie9.dll then Cindy has an ownership of 20%.

5. RESULTS

5.1 Module Structure

Table 1: Attributes of an observation

author	author time	committer	commit time	changed file	module
Minfei Huang	Nov 6 16:32:45 2015 -0800	Linus Torvalds	Nov 6 17:50:42 2015 -0800	kernel/kexec.c	kernel

We have observed two aspects of product structure: the architecture, which includes several structures, of which we will primarily focus on the module structure, and the development activity structure.

Based on our interviews and prior experience, developers' tasks are assigned based on these two types of structure. In our study the module structure was organized according to product package/subsystem and functionality (functionality, such as internationalization, may cut across the package/subsystem boundaries).

The activity structure followed common development practices, such as building, installing, configuring, and testing the product. It also included practices used to fix and report problems and to design and develop new features. Furthermore, underlying these generic practices, there were substantial differences in information seeking behavior needed to accomplish these common tasks, for example, knowing when and how to inspect the execution log or where to find information about similar bugs that occurred in the past, and variation in acceptable norms, such as how many defects are acceptable, and what should be tested, how it should be tested, and how extensively it should be tested.

Based on our observations, the way each practice was implemented was carried over from the original practice used to develop the products, often with no individuals serving as conduits. For example, when fixing defects Project A extensively used their rich problem resolution repositories, while project C used almost exclusively the logs of product execution and project B focused on latest code changes.

Our claim is not that the mere fact that the original team and the new team perform testing implies some learning from the product structure (we expect that most software developers know about testing from their undergraduate studies). Rather, the similarity between the ways testing was done in the original and the new team indicates that the product itself has some effect on learning and that product structure incorporates development activity structure. We propose that developers learn through performing regular project tasks under the constraints ("guidance") of the product structure, and, accordingly, change their positions in the project/organization. The centrality of a task, embodies the centrality of the modules or activities the task is related to, and reflects the centrality of the position the developer has in the organizational communication.

5.2 Organization of Code Contribution

we focus on one factor that tries to measure the team relationship between code authors and code committers. Code authors in this study are people who write the code that gets into the mainline repository. Code committers are people who have the privilege to write the repository. A committer may be responsible for a specific module (functionality), so responsible for committing whoever's code on that module. Naturally, the ratio of number of authors over number of committers may represent the load of a committer, or the difficulty (easiness) of the module, or, how hard it is for the author to get her code in. And the change of the ratio on the same module may represent the change of project landscape (e.g., new features may attract more authors but committers may keep at the same level as before), or the maturity of the module (e.g., a mature module may reduce committers).

We use this measure to investigate how the code contribution is organized and how the balance between authors and committers is achieved in the community. We take three years as a window, start from January 2005 (the window is from January 2005 to December 2007) and roll the window by month until the window getting to December 2015, i.e., the date we retrieved the data. We calculate the metrics on these windows. Considering the variations of the modules, we look at the team organization on each module separately. Figure 2 presents the ratio changes in a three-year window rolling from January 2005 month by month.

Overall, the ratio is decreasing over the years. Even both the number of authors and committers are increasing (drivers is the single module that has the sharpest increase for both), apparently, the increase of committers is faster than that of authors. That may suggest a more professional organization of the working teams has been happening in the community.

As we can see, the module of drivers (the top line) has a much higher ratio than the other modules. It fluctuates at around 20 compared to 10 that the other modules move around. If we look at the number of authors and committers separately on the modules, we can see that they both grow over time on all the modules, but the drivers module has the biggest growth. At the same time, even both authors and committers grow, apparently committers have a slower growth, therefore the ratio drops.

We discovered that the ratio is around 10 for most modules but high above 10 for the module of drivers. The ratio for modules like kernel, mm stabilizes at 6 or 7, a reasonable size that is a controllable number for a team. A further investigation reveals that the drivers module have a much looser control over the team, the fact that a committer works for 20 authors suggests that the tasks may be easy on this feature. Net drivers and staging drivers appear to be on the easiest tasks, suggesting a place where newbies like to start. The fact that the drivers module has more than 50% share of newcomers supports this hypothesis (what's the proportion of changes of drivers?). In fact more than 40% of all the committers in the kernel community started from the drivers module (no comitters of kernel module or mm module started from kernel or mm).

We take a further investigation to understand the story in the module of drivers. Figure 3 shows the ratios of subdirectories under the drivers. We can see net, staging and media are the three sub-modules that are different from the others that have the similar trend compared to the modules at the first level, i.e., they fluctuate at 10.

The investigation on net module shows 10 is a common number for this metric, as presented in Figure 4. Actually except mac80211, all the other sub-modules fluctuates at 10 (sunrpc seats at five). A closer look at mac80211 clarifies its

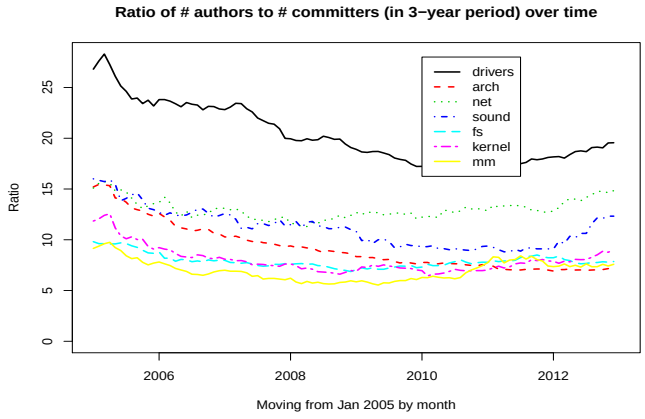

Figure 2: Ratio of #authors to #committers

./pics/testSmall.pdf

Figure 3: Ratio of #authors to #committers on Drivers

change: the number of committers is from 3 to 8 and back to 3, but the number of authors stays the same, making the curve change sharply.

In summary, 10 might be a reasonable number for organizing the contribution team. If the code is well modularized, the ratio could be bigger, otherwise it could be smaller.

Through chcking with the core members of kernel, we found the ratio changes because of the following reasons:
We use directory structure to measure 1), when there is new feature, a bulk of authors would cause fluctuation of this measure, because the committers wouldn't increase at the moment, e.g., drivers/staging/iio.
2), when there is new feature, the increase of committers is likely to be behind the increase of authors, e.g., net/wireless, drivers/staging/comedi, drivers/staging/iio.
3), for the matured feature, it's likely the committer would leave, and a small amount of leaving committers would lead to a drama increase of this ratio because the proportion is relatively high, e.g. net/mac80211.
4), in general the development is stable.

In summary, we have the following observation:

Observation 1. *The suitable way of organizing code contribution team in Linux kernel is to have a committer work for ten authors. If the code is well modularized, the ratio could be bigger, otherwise it could be smaller.*

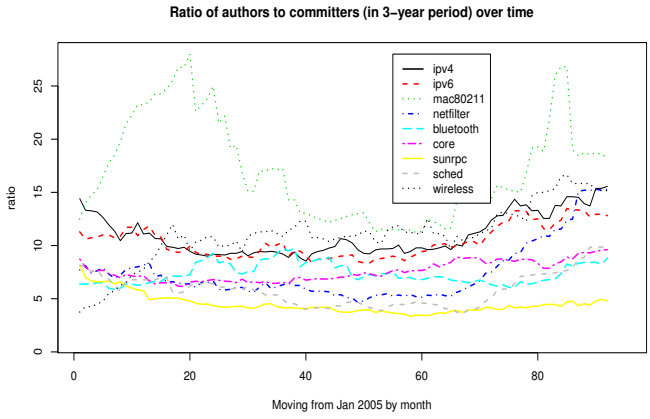

Figure 4: Ratio of #authors to #committers on Net

The ratio for modules like kernel, mm stabilizes at 6 or 7, a reasonable size that is a controllable number for a team. A further investigation reveals that the drivers module have a much looser control over the team, the fact that a committer works for 20 authors suggests that the tasks may be easy on this feature. Net drivers and staging drivers appear to be on the easiest tasks, suggesting a place where newbies like to start. In fact more than 40% of all the committers in the kernel community started from the drivers module (no comitters of kernel module or mm module started from kernel or mm).

5.3 Code Ownership

The kernel which forms the core of the Linux system is the result of one of the largest cooperative software projects ever attempted [**?**]. With the 2.6.x series (Linux 2.6.0 Released 18 December, 2003), the Linux kernel has moved to a relatively strict, time-based release model. The stricter code ownership may be developed to simplify the cooperation in the building of more and more complicated kernel. However, we can still see the differences among different modules.

The close coupled module is associated with close team, and we hypothesize that the close team are more likely to have shared code ownership in contrast to the loose team.

Stricter code ownership would manifest itself as a smaller number of developers changing a single file. The average number of changers per file ($Changer_{avg}$) may, therefore, be used to measure the strictness of code ownership. We calculated $Changer_{avg}$ for each month of each module and the resulting time series of this measure of code ownership are shown in Figure 5. We can see the modules of drivers and arch have a much stricter ownership as compared to the other modules. The module of kernel has the loosest control, next to it is the module of mm.

The boxplot in Figure 6 and Figure 7 show a clear comparison between the modules of kernel and drivers. (every boxplot has 12 numbers for every year) shows a clearer trend[5].

[5]The lower and upper boundaries of the rectangle of the boxplot represent the first and the third quartile and the thick line is the median. The lowest and upper horizontal lines

Figure 5: Code ownership of different modules

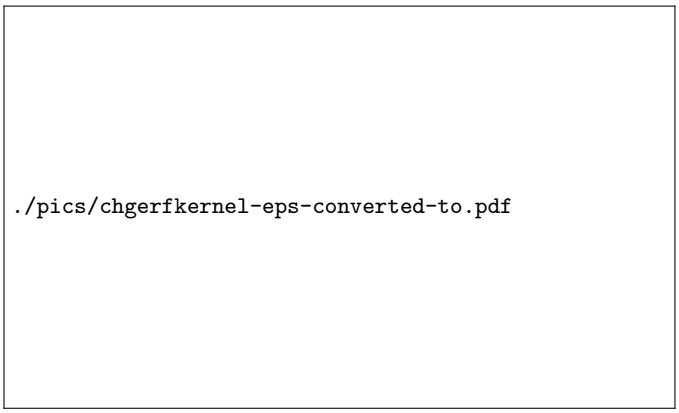

Figure 6: Boxplot of code ownership of kernel module

For example, in 2005 when drivers has a code ownership of 1.36 being the median, kernel has 2.62, almost twice of that of drivers. In 2015 drivers has 1.50 and kernel has 2.89. This is an indication that close team has a closer relationship on sharing code.

At the same time, we found the average number of unique files touched by a developer (per month) on different modules differ from each other with a similar pattern. Figure **??** and Figure **??** show the numbers of kernel and drivers respectively, as we can see, developers of kernel module tend to touch more different file as compared to developers of drivers module. The number fluctuates around 10 for kernel module, but 5 for drivers module. This may indicate that the sharing ownership brings more chances for the expertise diversity. On the contrary, for modules like drivers, it may require more effort to work on multiple files, and the interconnections between files may be reduced in order to reduce the needs of cooperations. Developers need to put more effort on single files instead of working with others on the same files. This indicates a possible phenomenon that requires attention from the kernel community: if people start to work on their own, there is a risk that the cooperation critical for the health of the community is reduced.

In summary, we have the following observation:

represent the 1.5 quartiles away from the mean. The points below or above the horizontal lines are suspected outliers.

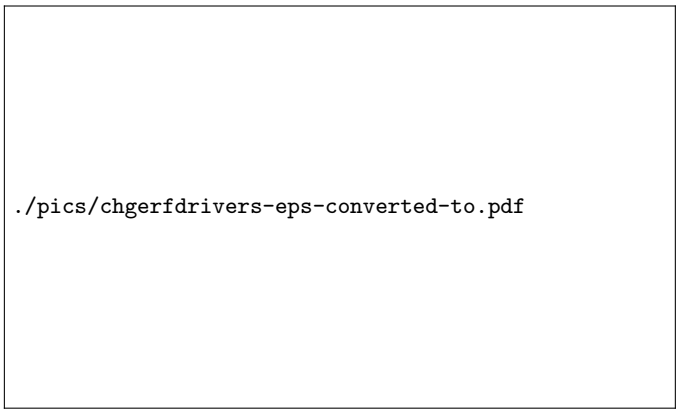

Figure 7: Boxplot of code ownership of drivers module

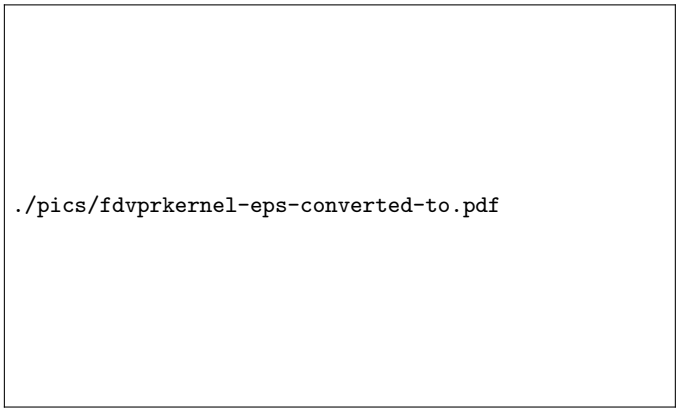

Figure 8: Number of files touched by a developer each month on kernel

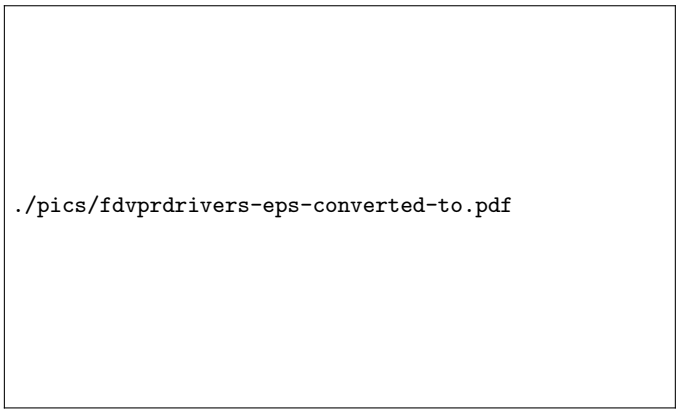

Figure 9: Number of files touched by a developer each month on drivers

Observation 2. The developers working on different modules present different extent of control on the code. In particular, more structured modules have stricter code ownership and compact modules are more likely to share the ownership.

Moreover, the ratio of each module correlates with the number of new joiners, and the number of new LTCs. This may imply that a looser control of working team brings more outsiders.

6. DISCUSSION

7. LIMITATION

8. CONCLUSION