# Measuring and Analyzing Code Authorship in 1+118 Open Source Projects

Guilherme Avelino[a,b,*], Leonardo Passos[c], Andre Hora[a], Marco Tulio Valente[a]

[a] *Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil*
[b] *Federal University of Piaui (UFPI), Teresina, Brazil*
[c] *University of Waterloo, Waterloo, Canada*

**Abstract**

Code authorship is a key information about large-scale software projects. Among others, it reveals the division of work, key collaborators, and developers' profiles. Seeking to better understand authorship in large and successful open source communities, we take the Linux kernel as our first case study. In total, we analyze authorship across 66 stable releases. Our analysis is centered around the Degree-of-Authorship (DOA) metric, which accounts for first authorship events (file creation), as well as further code changes. Authorship along the Linux kernel evolution reveals that (a) only a small portion of developers (26%) makes significant contributions to the code base; this ratio is almost constant during the Linux kernel evolution; (b) the number of files per author is highly skewed—a small group of top-authors (2%) is responsible for hundreds of files, while most authors (75%) are responsible for at most 10 files; (c) most authors in Linux (76%) are specialists and the relation between specialists and generalists tends to be constant; (d) authors with a high number of co-authorship connections tend to work with authors with fewer connections. Furthermore, we replicate the study in an extended dataset, composed of 118 well-known GitHub projects. We identify that most of the authorship patterns observed in the Linux kernel are also common to other open source projects.

*Keywords:* *Code Authorship*, *Linux kernel*, *Developer Networks*

## 1. Introduction

Software engineering is essentially a collective effort, requiring the coordination of large developer teams [1, 2, 3]. To tackle complexity and size, software systems are usually partitioned into subsystems, allowing developers to parallelize implementation [4]. Hence, collaborative work and modularization are key players in software development, specially in the context of open source systems.

*Corresponding author
*Email address:* gaa@ufpi.edu.br (Guilherme Avelino)

In a collaborative setup imposed by open source development, code authorship allows maintainers to assess the overall division of work among project members (e.g., to seek better working balance), to identify profiles within the team (e.g., specialists versus generalists), and to find the best fitting of developers for a target task.

Our notion of authorship is broader than the English definition of the word. In the context of code, authorship relates to those who make significant changes to a target file. This may include the original file creator, as well as those who subsequently change it. Hence, code authorship is inherently dynamic as a software evolves. In contrast, authorship in books and scientific papers is static, as given at the time of publication.

In this paper, we initially set to understand authorship in a large and long-lived successful system—the Linux kernel. Our goal is to identify authorship parameters from the Linux kernel evolution history, as well as interpret why they appear as such. At all times, we also check whether those parameters apply to the subsystem level, allowing us to assess their generality across different parts of the kernel. This analysis accounts for 66 stable releases (v2.6.12–v4.17), spanning a period of over 13 years of development (June, 2005–June, 2018). Additionally, in a second study, we contrast the authorship results computed for the Linux kernel with the ones computed for a dataset of 118 popular open source systems, retrieved from GitHub.

**First Study (Linux Kernel).** First, when investigating the Linux kernel authorship history, we provide answers to four research questions:

*RQ1: What is the proportion of developers ranked as authors?*
*Motivation:* In large open source communities, most developers perform occasional and minor contributions [1, 5]. With that insight, not all contributors perform significant changes, but how many do? Hence, this research question allows to reveal the proportion of developers with significant contributions to Linux development, defining the project working force to be studied.

*RQ2: What is the distribution of the number of files per author?*
*Motivation:* Answering such a question provides us with a measure of the work overload within team members, as well as how that evolves over time.

*RQ3: How specialized is the work of Linux authors?*
*Motivation:* Following the Linux kernel architectural decomposition, we seek to understand the proportion of developers who have a narrower understanding of the system (specialists), versus those with a broader knowledge (generalists). Specialist developers author files in a single subsystem; generalists, in turn, author files in different subsystems. This research question seeks to assess how effective the Linux kernel architectural decomposition is in fostering specialized work, a benefit usually expected from a good modularization design [6, 7].

*RQ4: What are the properties of the Linux co-authorship network?*
*Motivation:* The authorship metric we use enables identifying multiple authors per file, evidencing a co-authorship relation among developers [8]. These rela-

tions form a network—vertices denote authors and edges connect authors sharing common authored files. This question seeks to identify co-authorship properties in the Linux kernel evolution. Among others, we compute and discuss several properties, including mean degree, number of solitary vertices, clustering, and assortative coefficients.

**Second Study (118 open source projects).** We extended the initial study (Linux kernel) by applying the same authorship-related metrics to a dataset of 118 popular open source systems, retrieved from GitHub. We contrast the studies results and observe that most of the authorship patterns firstly identified in the Linux kernel are also present in this extended dataset. For example, most projects in the extended dataset present skewed distribution of the number of files per author and a high number of specialists.

**Previous Work.** The work we report here is an expansion of a previous conference paper [9] in which we proposed the use of code authorship measures to study the social organization of software systems. By adopting these concepts, we also investigated authorship in the Linux kernel. In this paper, we improve this initial study in two major directions:

1. By extending our analysis to 118 popular GitHub projects. In this new study, we compare and contrast the authorship results previously computed for the Linux kernel with the ones computed for this larger dataset.
2. By providing new and updated data (release v4.17 of the Linux kernel, June 2018) and better exploring the results of previously investigated research questions. In especial, we improve the investigation of the proportion of authors (RQ1) and the work specialization (RQ3), by contrasting the general results with the ones observed at subsystems level.

**Organization.** The remainder of this paper is organized as follows. Section 2 provides a description of our study design. Section 3 details our results, providing answers for the four research questions. Section 4 discusses our key findings when investigating these questions. Section 5 compares the Linux kernel measures with the ones computed for other 118 open source projects. Sections 6 and 7 discuss threats to validity and related work, respectively. Section 8 concludes the paper, also outlining future work.

## 2. Study Design

### 2.1. Author Identification

At the core of our study lies the ability to identify and quantify authorship at the source code level. To identify file authors, as required by our four research questions, we employ a normalized version of the *degree-of-authorship* (DOA) metric [10, 11]. The metric is originally defined in absolute terms:

$$DOA_A(d, f) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * \ln(1 + AC)$$

From the provided formula, the absolute degree of authorship of a developer $d$ in a file $f$ depends on three factors: first authorship (FA), number of deliveries (DL), and number of acceptances (AC). If $d$ is the creator of $f$, $FA$ is 1; otherwise it is 0; $DL$ is the number of changes in $f$ made by $d$; and $AC$ is the number of changes in $f$ made by other developers. $DOA_A$ assumes that FA is by far the strongest predictor of file authorship. Further changes by $d$ (DL) also contribute positively to his authorship, but with less importance. Finally, changes by other developers (AC) contribute to decrease someone's $DOA_A$, but at a slower rate. The weights we choose stem from an experiment with professional Java developers [11]. We reuse such thresholds without further modification.

The normalized DOA ($DOA_N$) is as given in [12]:

$$DOA_N(d, f) = DOA_A(d, f)/max(\{DOA_A(d', f) \mid d' \in changed(f)\})$$

In the above equation, $changed(f)$ denotes the set of developers who edited a file $f$ up to a snapshot of interest (e.g., release). This includes the developer who creates $f$, as well as all those who later modify the file. $DOA_N \in [0..1]$: 1 is granted to the developer with the highest absolute DOA among those changing $f$; in any other case, $DOA_N$ is less than one.

Lastly, the set of authors of a file $f$ is given by:

$$authors(f) = \{d \mid d \in changed(f) \wedge DOA_N(d, f) > 0.75 \wedge DOA_A(d, f) \geq 3.293\}$$

*Example:* Suppose a file $f$ created by *Bob*; suppose also that after its creation the file was changed 8 times by *Alice*, 3 times by *Bob*, and 2 times by *Carol*. The DOA equation leads to the following absolute results:

$$DOA_A(Bob, f) = 3.293 + (1.098 * 1) + 0.164 * 3 - 0.321 * \ln(1 + 10) = 4.30$$
$$DOA_A(Alice, f) = 3.293 + (1.098 * 0) + 0.164 * 8 - 0.321 * \ln(1 + 5) = 4.02$$
$$DOA_A(Carol, f) = 3.293 + (1.098 * 0) + 0.164 * 2 - 0.321 * \ln(1 + 11) = 2.82$$

After that, we should normalize these values, as follows:

$$DOA_N(Bob, f) = 1$$
$$DOA_N(Alice, f) = 4.02/4.30 = 0.97$$
$$DOA_N(Carol, f) = 2.82/4.30 = 0.68$$

By applying the $DOA_A$ and $DOA_N$ thresholds, we find two authors for $f$:

$$authors(f) = \{Bob, Alice\}$$

Particularly, *Carol* is not an author because her $DOA_A$ and $DOA_N$ do not attend the required thresholds, which are 3.293 and 0.75, respectively.

*DOA Validation and Thresholds:* As presented in the previous paragraphs, the interpretation of DOA results depends on specific thresholds—0.75 and 3.293. Those stem from a calibration setup when applying $DOA$ in previous works.

First, we used DOA as a key metric of an algorithm to estimate truck factors [12]. To support this usage, we manually inspected a random sample of 120 files from six popular GitHub projects, including torvalds/linux. We compared DOA against git-blamed results and concluded that normalized DOA-values below 0.75 frequently lead to false positives.[1] After that, we also validated the truck factors produced using DOA with the principal developers of 67 popular GitHub projects, obtained a positive feedback; 84% of the surveyed developers agree or partially agree with the list of top-developers based on DOA values. In a more recent study, we evaluate the usage of DOA to recommend maintainers to source code files [14]. In this study, we survey 159 developers from 10 projects (including 8 open source and 2 commercial projects) to create an oracle with the maintainers of 654 source code files. Then, we compare maintainers recommendations produced using DOA results with recommendations based on number of commits and git-blame results. We report the three metrics have a similar performance on classifying developers as maintainers and non-maintainers. Particularly, using a threshold of 0.75 to identify software maintainers, DOA has a hit ratio of 62% (in the case of the studied open source projects).

### *2.2. Linux Kernel Architectural Decomposition*

Investigating authorship at the subsystem level requires a reference architecture of the Linux kernel, as well as a mapping between elements at the source code level to elements in the architectural model.

Structurally, the Linux kernel architectural decomposition comprises seven major subsystems [15]: *Arch* (architecture dependent code), *Core* (scheduler, IPC, memory management, etc), *Driver* (device drivers), *Firmware* (firmware required by device drivers), *Fs* (file systems), *Net* (network stack implementation), and *Misc* (miscellaneous files, including documentation, scripts, etc).

To map files in each subsystem, we rely on expert knowledge. Specifically, we use the mapping rules set by Greg Kroah-Hartman, one of the main Linux kernel developers.[2] Table 1 provides information about the size, as measured by number of files, in each kernel subsystem. As the table shows, *Driver* is the largest subsystem, followed by *Arch*, *Misc*, and *Core*.

### *2.3. Data Collection*

We study 66 stable releases of the Linux kernel, obtained from linus/torvalds GitHub repository.[3] A stable release is any named tag snapshot in which the identifier does not have a *-rc* suffix. To define the *authors* set of a file $f$ in a given release $r$, we calculate $DOA_N$ from the first commit up to $r$. Hence, all

[1] As an important threat to mention here, refactorings can impact on authorship measures based on git-blame, as detailed for example in [13].

[2] `https://github.com/gregkh/kernel-history/blob/master/scripts/stats.pl`

[3] `https://github.com/torvalds/linux`

Table 1: Linux subsystems (release v4.17)

| Subsystem | # Files | % |  |
|---|---|---|---|
| Driver | 28,114 | 46% |  |
| Arch | 15,932 | 26% |  |
| Misc | 9,164 | 15% |  |
| Core | 4,231 | 7% |  |
| Net | 2,039 | 3% |  |
| Fs | 1,849 | 3% |  |
| Firmware | 33 | 0% |  |
| **Total** | **61,362** | **100%** |  |

files from each release have at least one author. It happens, however, that the Linux kernel history is not fully stored under Git. Linus Torvalds explains:[4]

> *I'm not bothering with the full history, even though we have it. We can create a separate "historical" git archive of that later if we want to, and in the meantime it's about 3.2GB when imported into git - space that would just make the early git days unnecessarily complicated, when we don't have a lot of good infrastructure for it.*

We use `git graft`[5] to join the history of all releases prior to v2.6.12 (the first release recorded in Git) with those already controlled by Git ($\geq$ v2.6.12). After joining, we increment the Linux kernel Git history with 64,468 additional commits.

However, by analyzing Linux's pre-git development history, we observed that many of these commits are imports of previous releases. In most cases, they were performed by Linus Torvalds without preserving the full development history. For example, one of the commits has the message "v2.4.3.3 − > v2.4.3.4"[6] and mentions the name of another five developers. In other words, the entire release development was pushed to Git in a single commit, masking individual contributions on behalf of a unique developer, Linus Torvalds. To handle this problem, we ignore commits before the one with the following hashcode: *1ea864f1c53bc771294e61cf9be43b1d22e78f4c*, which is the last commit that explicit mentions a release migration.[7] Furthermore, we also completely discard from our analysis the files added by these commits, since in this case we are not able to identify their creator, which is an essential component of the DOA equation. In the end, we discard the initial 1,206 commits available on Git and 4,366 source code files (9% of the files in v4.17).

[4] https://github.com/torvalds/linux/commit/1da177e4c3f41524e886b7f1b8a0c1fc7321cac2

[5] https://git.wiki.kernel.org/index.php/GraftPoint

[6] d39a11f309a4fdeeed232dd6c0f00604d11a4aea

[7] Their messages follow the pattern `v(\d+\.)*\d+ − > v(\d+\.)*\d+`

Given the Linux kernel evolution history, we then query the Git repository to filter stable release names, checking out each stable release at a time. For a given release snapshot, we list its files, calculating $DOA_N$ for each of them. In the latter case, we rely on `git log --no-merges` to discard merges and retrieve all the changes to a given file prior to the release under investigation. To compute the $DOA_N$, we only consider the author of each commit, not its committer (Git repositories store both) [16].

It is worth noting that prior to calculating $DOA_N$, we map possible aliases among developers, as well as eliminating unrelated source code files. Next, we detail such steps.

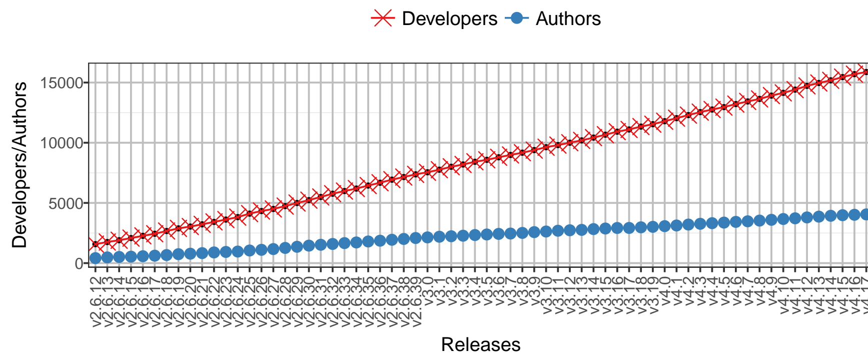
**Alias Detection.** One of core challenges in mining software repositories consists in alias detection. An alias occurs when a single developer uses different identities to commit changes; in such cases, one should track all changes as being of the same developer. In Git, an identity comprises a username and email. To track changes in the face of aliases, we first assign to a single developer all commits with exactly the same e-mail, but with different names. For example, the email `dmonakhov@openvz.org` associates to six different names: "Dmitry Monakhov", "Monakhov Dmitriy", "Dmitri Monakhov", "Dmitri Monakho", "Dmitry", and "Dmitriy Monakhov". Then, we consider developer names to be the same if they have a Levenshtein distance [17] of at most one. Such distance corresponds to the minimal number of single-character insertions, deletions, or substitutions required to make two strings identical, respected the limit of a single-character change. For example, "Haavard Skinnemoen" and "Havard Skinnemoen" are considered to be the same name, since one insertion is needed to make the two strings equal.

**File Cleaning.** Code authorship should consider only the files representing the source code of a target system. Thus, it should ignore documentation, images, examples, third-party source code files, etc. To filter out files unrelated to the Linux kernel, we use the Linguist library.[8] GitHub uses the latter to identify a system's language, as well as files that should not be counted as part of the system (e.g., when collecting repository statistics). We use Linguist to exclude unrelated files at each release we analyze. For example, release v4.17 contains 61,362 files; of those, we remove around 23%. Most of the excluded files consist of documentation (5,902) and device-tree specifications (2,689).[9] We also remove the *Firmware* subsystem from further analysis, as most of its files are blobs.

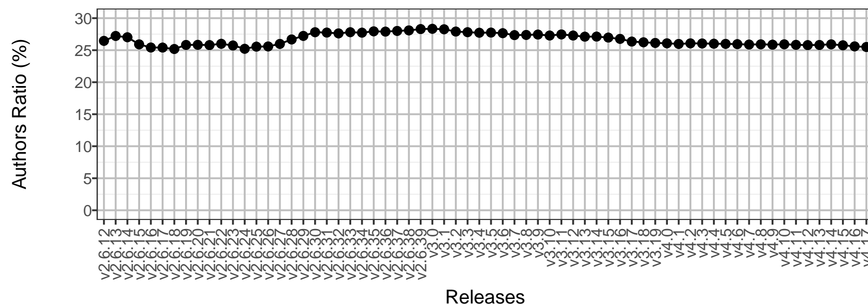
### *2.4. Custom-made Infrastructure*

Using custom-made scripts, we fully automate authorship identification, as well as the collection of supporting data for the claims we make. Our infrastructure is publicly available on GitHub.[10] We encourage others to use it as

[8] `https://github.com/github/linguist`
[9] Files ending in `dts` and `dtsi`.
[10] `https://github.com/gavelino/scp_data`




(a) Number of authors and developers over time




(b) Proportion of authors over time

Figure 1: Authors and developers over time

means to independently replicate/validate our results.

## 3. Results

*RQ1) Proportion of Authors*

*What is the proportion of developers ranked as authors?*

In the latest release in our analysis (v4.17), the Linux kernel has 15,873 developers; of those, 26% author at least one file. Figure 1a contrasts the number of developers with the number of authors across different releases. In both cases, we identify a steady increase, with similar growth rates. Specifically, there is an 9-fold increase in the number of authors, from 443 (first release) to 4,048 (last release). The number of developers, in turn, has grown 10 times.

Historically, the mean proportion of authors is 27%, with alternating periods of small increases and decreases—see Figure 1b. Throughout the kernel development, the proportion of authors is nearly constant (*Std dev*= $\pm$ 0.94%). Thus, the heavy-load maintenance of the kernel has been kept in the hands of a little more than one quarter of all developers.

Table 2: Author Ratio Evolution: Summary Statistics. Avg: Average, Std Dev: Standard Deviation

| Subsystem | Min | Max | Avg ± Std Dev |
|---|---|---|---|
| **Core** | 24.71% | 29.53% | 26.85 ± 1.22% |
| **Driver** | 24.41% | 28.14% | 25.42 ± 0.88% |
| **Arch** | 30.64% | 37.27% | 34.15 ± 1.74% |
| **Net** | 12.53% | 16.18% | 13.69 ± 0.89% |
| **Fs** | 9.38% | 18.58% | 12.71 ± 2.39% |
| **Misc** | 11.20% | 24.14% | 14.30 ± 2.93% |
| **All** | 25.19% | 28.34% | 26.66 ± 0.94% |

Contrasting the authorship proportion at the subsystem level with the global one shows that *Core* (mean 27%) and *Driver* (mean 25%) approximate to the global average.[11] As Table 2 shows, both subsystems display little variance in authorship ratio, which follows directly from their low standard deviation (Std Dev). To a lesser extent, the authorship ratio in *Arch* (34%) also approximates the global parameter; the same does not occur for *Net*, *Fs*, and *Misc*. We interpret such discrepancies as follows.

As *Core* and *Arch* provide the basic functionality for the remaining parts of the system, developers must have great confidence on the changes they propose, discouraging volunteers from performing small changes as a means to become kernel contributors. Therefore, in the latter case, we hypothesize that authorship ratio increases. In addition, changing *Arch* requires vendor-specific expertise when maintaining support for different CPUs. Thus, knowledge becomes narrower, lifting author ratio. *Driver* follows a similar rationale, requiring manufacturer-specific knowledge about the hardware to support. Different from the latter three, *Net* and *Fs* require less hardware-specific knowledge; in a sense, maintaining such subsystems is "somehow" easier, making them more attractive for occasional and minor contributions. As a consequence, we assume that there is a decrease in authorship. *Misc*, due to a mix content, fits different goals, not necessarily in-tune with the operating system itself (e.g., infrastructure for building the kernel). Its size, as given in lines of code, tends to be stable across the kernel, indicating that *Misc* does not change frequently [18]. From such insights, we hypothesize that the closer a subsystem is to vendor-specific code (e.g, *Arch* and *Driver*) or to the "brain" of the kernel (*Core*), the harder is to maintain code, inflating authorship. However, this hypothesis and all observations made in this paragraph deserve a separate study, to confirm (or not) their validity.

*Summary: In the last release, 26% of the Linux's developers are authors, con-*

[11]We use the terms *average* and *mean* interchangeably. Both should be interpreted as the arithmetic mean.

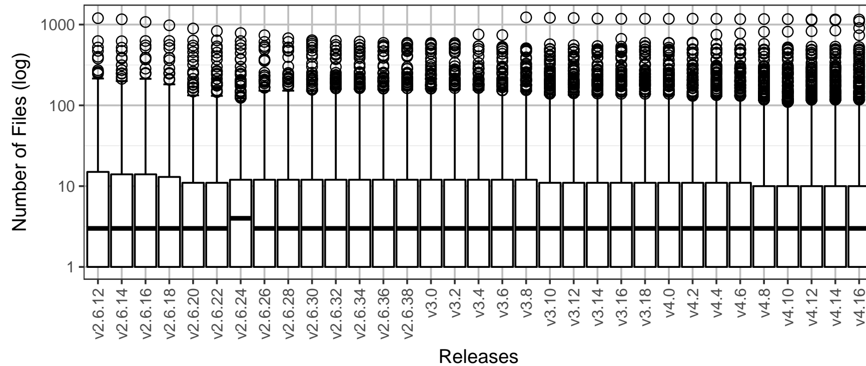



Figure 2: Distribution of the number of files per author in each release

*centrating the heavy-load of the Linux kernel maintenance. Such proportion is almost constant over time. The proportion of authors is higher in subsystem that requires vendor-specific knowledge (e.g., Driver and Arch) or an understanding of the kernel as a whole (e.g., Core).*

*RQ2) Distribution of the Number of Files per Author*

*What is the distribution of the number of files per author?*

The number of files per author is highly skewed. Figure 2 presents the boxplots of files per author across all 66 releases. To simplify the visualization, we present the boxplots at each two releases. Globally, 50% of the authors responds to at most three files (median), a measure that remains constant across all releases except two (v2.6.23 and v2.6.24); along the versions, the third quartile ranges from 10 to 15 files per author. Outliers follow from the skewed distribution. Still, the number of authors with more than 100 files is at most 6% of the authors, ranging from 6% in the first release to 2% in the last one.

As Figure 3 shows, the percentage of the Linux kernel files authored by the top-10 developers has reduced from 43% (first release) to 14%, in v4.17. This suggests that authorship is increasing at lower levels of the pyramid, becoming more decentralized.

To better comprehend the distribution of the number of files per author, we also analyze Gini coefficients (Figure 4). Gini is a widely used metric to express the wealth inequality among a target population [19]. Wealth, in this case, stands for the number of files per author. The coefficient ranges from 0 (perfect equality, when everyone has exactly the same wealth) to 1 (perfect inequality, when a single person concentrates all the wealth). In all releases, the Gini coefficient is high, confirming skewness. However, we notice a slight decreasing trend, ranging from a Gini of 0.82 in the first release to 0.75 (v4.17). Such a trend further strengthens our notion that authorship in the Linux kernel is becoming less centralized (although only slightly). Alternatively, it means that the number of authors per file is becoming more equal.

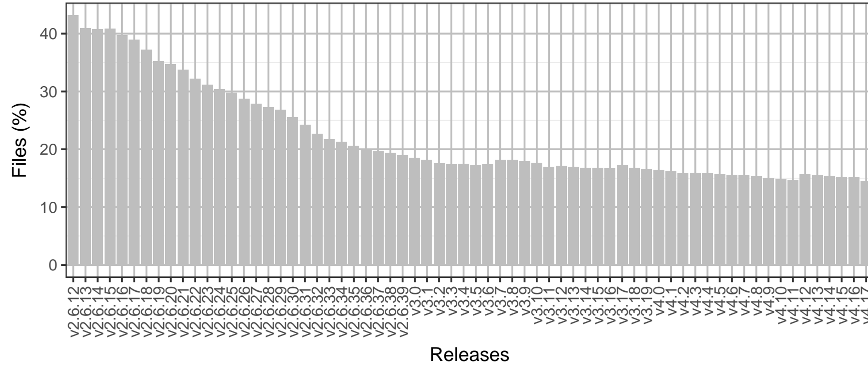



Figure 3: Percentage of files authored by the top-10 authors over time.

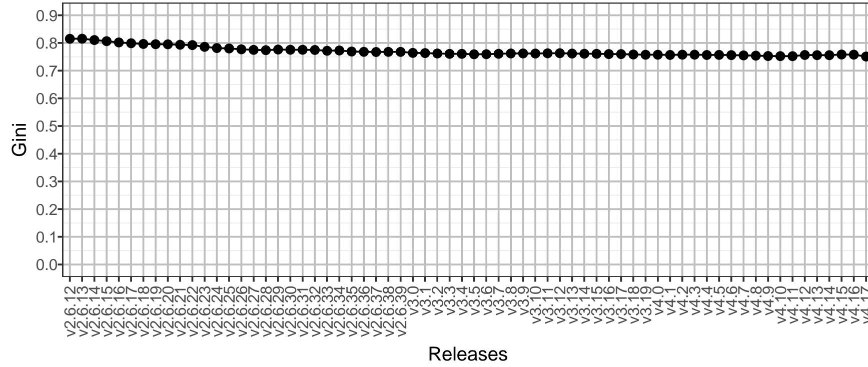



Figure 4: Gini coefficients

As the global distribution of files per author becomes more equal, we note a similar trend at the subsystem level—see Figure 5. In the last release (v4.17), for instance, the number of files per author up to the 75% percentile in *Fs*, *Arch*, and *Driver* closely resemble one-another and the global distribution as a whole—all share the same median (three). *Core* and *Misc*, however, have less variability than the other subsystems, as well as lower median value (one).

*Summary: The number of files per author follows a highly skewed distribution in all analyzed releases. However, it is becoming more equal over time. For example, the top-10 authors owns 43% of the files in the first analyzed release, but only 14% of the files in the last one.*

*RQ3) Work Specialization*

*How specialized is the work of Linux authors?*

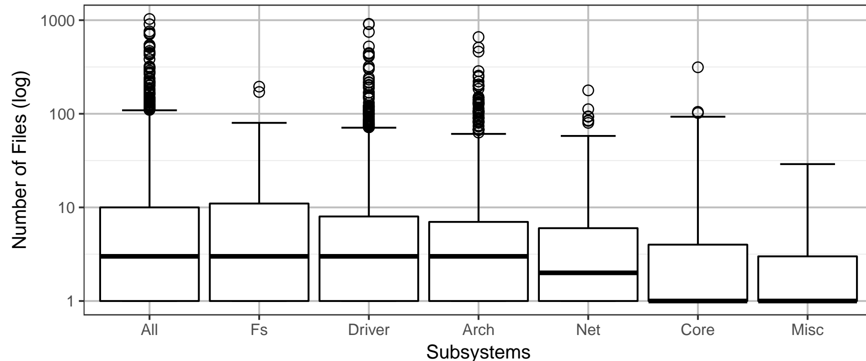


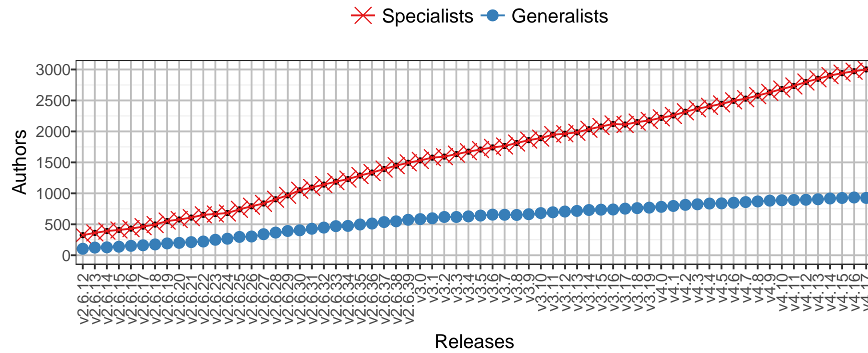

Figure 5: Number of files per author (release v4.17)

To assess work specialization, we introduce two author profiles. We name authors *specialists* if they author files in a single subsystem. *Generalists*, in turn, author files in at least two subsystems. When classifying authors, we ignore files in the **include/** directory. This directory may lead to misclassification because it was designed to store all Linux kernel headers files, independently of what subsystem the file belongs. As Figure 6 shows, the number of specialists dominates the amount of generalists, in both absolute and relative terms.

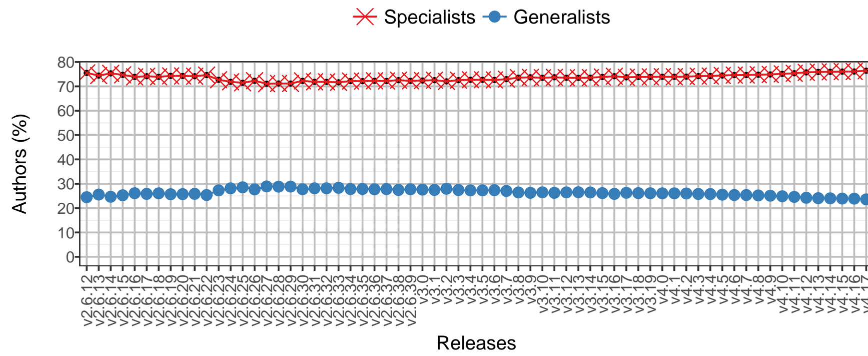
Proportionally, any given release has at least 71% of specialist authors, with a maximum of 76%; at all times, no more than 29% of the authors are generalists. Moreover, the proportion of generalists and specialists appears to be fairly stable across the entire kernel (All) and its constituent subsystems (except for *Misc*)—see Figure 7.

Looking at the division of generalists and specialists working in each subsystem also provides a means to assess how much the Linux kernel architectural decomposition fosters specialized work—a desired by-product of a good modularization design [6, 7]. We notice that the architectural decomposition plays a key role in fostering specialists inside the *Driver* subsystem, but less so elsewhere. Thus, the Linux kernel architecture partially fosters specialization. The reason it occurs so extensively inside *Driver* follows from the plugin interface of the latter and its relative high independence to other subsystems. Device-drivers are supposed to be self-contained modules that are plugged into the kernel and loaded as needed [15, 18]. There are cases, however, when developers must change other parts of the kernel when adding new device drivers or maintaining them. An example includes scattering code in *Arch* due to hardware detection limitations [18]. In cases such as the one described, the knowledge to perform changes increases, leading to the appearance of generalists within the *Driver* subsystem.

Similar to *Driver*, *Net* and *Fs* also follow a plugin-like model of development. Counter-intuitively, however, both subsystems display a dominance of generalists. Our hypothesis is that the maintenance of these systems interplays




(a) Absolute number of specialists and generalists




(b) Percentage of specialists and generalists

Figure 6: Specialists and generalists over time

with other parts of the kernel. For instance, a new distributive file system may require specific network protocols to be added or fine-tuned. Others also report a similar understanding [20].

The results in *Arch* and *Core* match our expectations. When maintaining either subsystem, developers must be aware of possible side effects elsewhere; also, modularization of both systems is harder to achieve, fostering less specialization. Often times, for instance, developers must break programming interfaces to get better performance. Other cases include the maintenance of CPU-specific code that has been historically troublesome (e.g., ARM).[12] Specifically, *Core* is the subsystem with the lowest percentage of specialized workers (17%). This is also expected since *Core* developers tend to have expertise on Linux's central features, which allows them to also work on other subsystems.

[12] `https://lkml.org/lkml/2011/3/17/492`

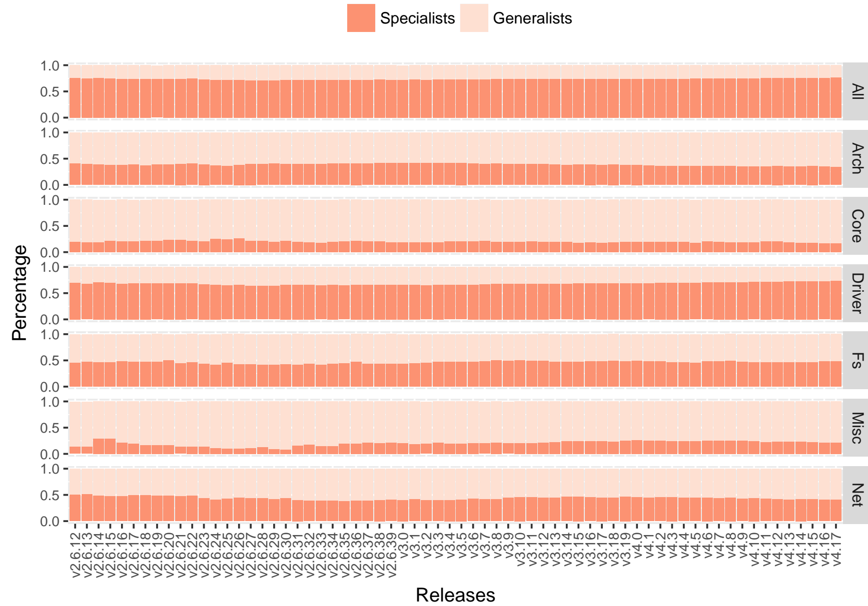



Figure 7: Percentage of specialists and generalists

**Fine-grained specialization.** We also investigate specialization at a fine-grained level. For this analysis, we consider each top-level directory as a Linux kernel module.[13] Therefore, *specialists* are developers who author files in a single top-level directory; the other developers are called *generalists*. By applying these definitions, in the last release (v4.17), we divided the Linux kernel in 16 modules and the percentage of specialists and generalists are, respectively, 74% and 26%. These results are similar to the ones obtained by the subsystems division (respectively, 76% and 24%, in v4.17). In other words, even considering fine-grained modules (16 top-level directories, instead of 6 subsystems) work specialization remains high in the Linux kernel.

*Summary: Specialization is a common practice in the Linux kernel development over the years. In 13 years of development, at least 71% of the Linux authors are specialists, i.e., all files they authored are located in a single subsystem. This is especially valid for Driver authors. The most notable exception are Core developers, who tend to be generalists.*

*RQ4) Co-authorship Properties*

*What are the properties of the Linux co-authorship network?*

[13] We discard files in the root and include directories.

Table 3: Percentage of files with multiple authorship (release v4.17)

| All | Driver | Arch | Core | Net | Fs | Misc |
|---|---|---|---|---|---|---|
| 27% | 27% | 25% | 27% | 31% | 26% | 26% |

Table 4: Co-authorship network properties (release v4.17)

| | All | Driver | Arch | Core | Net | Fs | Misc |
|---|---|---|---|---|---|---|---|
| **Vertices** | 4,076 | 3,180 | 1,219 | 1,251 | 325 | 188 | 89 |
| **Mean Degree** | 3.39 | 2.61 | 2.84 | 1.52 | 2.33 | 2.49 | 0.70 |
| **Clustering** | 0.078 | 0.067 | 0.129 | 0.088 | 0.203 | 0.165 | 0.048 |
| **Assortativity** | -0.080 | -0.111 | -0.040 | -0.035 | -0.015 | -0.151 | -0.243 |

In this section, we investigate the collaborative nature of the Linux implementation work. First, it is important to clarify that the DOA model allows multiple authors per file; they only need to have a DOA value that fits the thresholds defined in Section 2.1. We compute the percentage of files with multiple authorship per subsystem, as presented in Table 3. As we can see, most files have a single author; however, the percentage of files with multiple authors is relevant. It ranges from 25% of the files in *Arch* to 31% in *Net*. When considering the files in all subsystems, it reaches 27%.

As many files in the Linux kernel result from the work of different authors, we set to investigate such collaboration by means of the properties of the Linux kernel *co-authorship network*. We model the latter as follows: vertices stand for Linux kernel authors; an edge connects two authors $v_i$ and $v_j$ if $\exists\ f$ such that $\{v_i, v_j\} \subseteq \mathit{authors}(f)$. To strength the collaboration meaning of the co-authorship network, we filter edges without a temporal overlap between the file authorship. In other word, we only connect the developers $v_i$ and $v_j$ if at least one change in $f$ performed by $v_i$ or $v_j$ occur while the other developer is active (by considering her first and last commit in the repository).

Different from books and scientific papers, the current co-authorship networks do not account for a possible hierarchy among authors (first author, second author, etc). Rather, we take all the authors of a file to be equally important co-authors (and leave an analysis where authors are ranked by their DOA values, for example, to a future work).

To answer our research question, we initially analyze the latest co-authorship network of the entire kernel, as given by the last release in our corpus (v4.17). When doing so, we measure four metrics: *number of vertices*, *mean degree*, *clustering coefficient*, and *assortative coefficient*. At all times, we contrast the system level network with those at the subsystem level. Additionally, we investigate how the values we report came to be, analyzing their historical evolution.

Table 4 presents the values of our target metrics.[14] The number of vertices

[14]We use the R *igraph* (version 1.0.1) to calculate all measures.

(authors) determines the size of a co-authorship network. The mean degree network, in turn, inspects the number of co-authors that a given author connects to. In the system level network for release v4.17 (All), the mean vertex degree is 3.39, i.e., on average, a Linux author collaborates with 3.39 other authors. At the subsystem level, *Driver* forms the largest network (3,180 authors, 78%), whereas *Misc* results in the smallest one (89 authors, 2%). *Arch* has the highest mean degree (2.84 collaborators per author); *Misc* has the lowest (0.70 collaborators per author).

The third metric we look at concerns the *clustering coefficient* of the co-authorship network. Also known as graph transitivity, this coefficient reveals the degree to which adjacent vertices of a given vertex tend to be connected [21]. In a co-authorship network, the coefficient gives the probability that two authors who have a co-author in common are also co-authors themselves. A high coefficient indicates that the vertices tend to form high density clusters. The clustering coefficient of the Linux kernel is small (0.078). Nonetheless, *Net* and *Fs* exhibit a higher tendency to form density clusters (0.203 and 0.165, respectively) in comparison to other subsystems. Excluding *Misc*, the two subsystems are the smallest we analyze, a factor that influences the development of collaboration clusters [22]. Although small, *Misc* has a low clustering degree as result of its reduced number of collaborations (*mean degree* $< 1$).

Last, but not least, we compute a measure called *assortativity coefficient*, which correlates the number of co-authors of an author (i.e. its vertex degree) with the number of co-authors of the authors it is connected to [23]. Ranging from -1 to 1, the coefficient shows whether authors with many co-authors tend to collaborate with other highly-connected authors (positive correlation). In v4.17, all subsystems have negative assortativity coefficients, ranging from $-0.243$ in *Misc* to $-0.015$ in *Net* subsystem. This result diverges from the one commonly observed in scientific communities [24]. Essentially, this suggests that Linux kernel developers often divide work among experts who help less expert ones. These experts (i.e., highly-connected vertices), in turn, usually do not collaborate among themselves (i.e., the networks have negative *assortative coefficients*).

In the co-authorship networks we compute, there are a relevant amount of *solitary authors*—authors that do not have co-authorship with any other developer. As Table 5 shows, 25% (1,020) of Linux kernel developers are solitary, authoring few files. Only 14% of solitary authors have more than three files; the solitary author with more files in v4.17, authored 46 files, all in *Driver*. In fact, it is worth noting that 70% of solitary authors work in the *Driver* subsystem. The latter is likely to follow from the high proportion of specialists within that subsystem (see RQ.3).

**Evolution of Co-authorship network metrics.** We set to investigate how the mean degree, clustering coefficient, and assortative coefficients evolved to those in release v4.17. Figure 8 displays the corresponding graphics.

Despite a small increase, the mean degree (see Figure 8a) has little variation from the first release (3.13) to the last one (3.39). Clustering coefficient (see

Table 5: Number of files authored by Solitary Authors (i.e., authors that do not have co-authors)

| Number of Files | 1 File | 2 Files | 3 Files | >3 Files | Total |
|---|---|---|---|---|---|
| **Solitary Authors** | 574 | 217 | 81 | 148 | 1,020 |

Figure 8b), in turn, varies from 0.110 (first release) to 0.078 (v4.17). Since the mean degree does not vary considerably, we interpret such decrease as an effect of the growth of the number of authors (network vertices). The latter creates new opportunities of collaboration, but these new connections do not increase the density of the already existing clusters. A similar behavior is common in other networks, as described by Albert and Barabási [22].

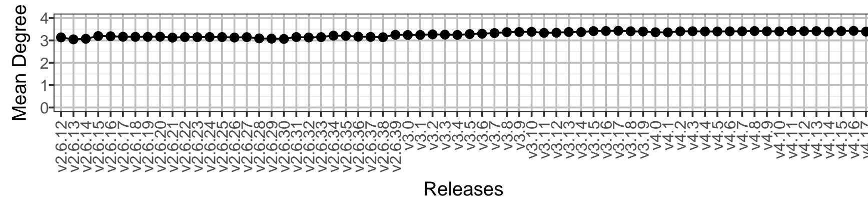
We observe a relevant variation in the evolution of assortativity coefficients—see Figure 8c. Measurements range from -0.202 in the first release to -0.080 in v4.17. Such a trend aligns with the decrease of the percentage of files authored by the top authors (refer to RQ.3). With less files, these authors are missing some of their connections and becoming more similar (in terms of vertex degree) to the their co-authors.

*Summary: On average, a Linux author collaborates with 3.4 other authors and this number is almost constant over time. Moreover, the co-authorship network indicates that authors that collaborate with many other do so in a way to "help" authors who collaborate less, suggesting some sort of mentorship.*

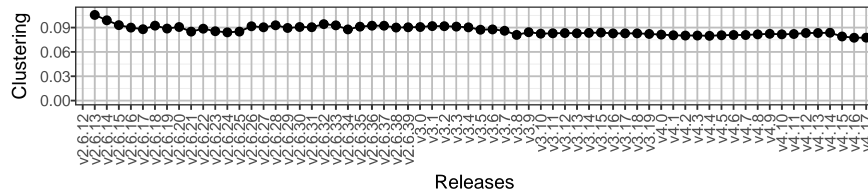
## 4. Discussion

**The Importance of Code Authorship.** Studies on open source communities can lead to misleading conclusions if the relative importance of the developers is not considered. The reason is that open source systems may have thousands of developers, but they usually have much less authors. In this context, authorship measures—like the DOA model used in this paper—can help to automatically identify the *key developers* of each file in a large open source project. By collecting authorship data for each file, we can also reason about many organizational aspects of a development team, as reported in this paper. For example, we used authorship measures to reveal the workload of Linux's authors (75% of them are authors of at most 10 files) and their degree of specialization (more than 71% of the authors are specialists in a single subsystem). Since the DOA model is computed using commit histories, it can also be used to study the evolution of these measures. For example, after collecting authorship data for 66 Linux releases, we showed that the number of files authored by the top-10 authors suffered a major decrease (from 43% of the system's files to 14%). We also reported that ratio of specialists and generalists is almost constant over time (71% vs 29%).

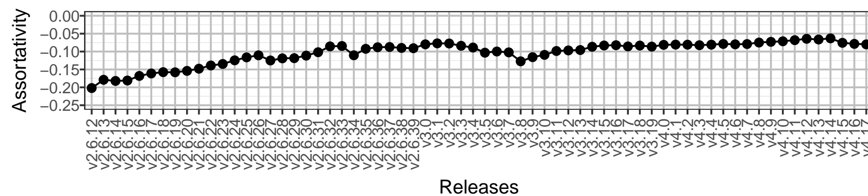
Interestingly, the Linux Foundation releases an annual report on the state of the kernel development [25]. In one of its sections, this document describes




(a) Mean degree




(b) Clustering coefficient




(c) Assortativity coefficient

Figure 8: Co-authorship network properties over time

"who is doing the work" of developing the Linux kernel. To answer this question, the study ranks the developers by number of changes. It reports, for example, that the top-10 individual developers have contributed with 8.4% of the number of changes. However, the report does not provide information about the specialization of such developers and how frequently they collaborate to solve implementation or evolution tasks.

In summary, code authorship measures can be used to check some important properties and practices in software development, like the ability of an open source system to attract not only new developers, but also new authors; to assess the contributions of paid developers in commercial software (since in this case we should expect all developers to be authors of at least some files); to assess the concentration of knowledge in few team members, which can raise concerns in case they leave the project (which can happen both in open source and in commercial projects); to check whether the criteria used to decompose a system in modules is indeed able to foster the specialization of the work force, as usually expected from software modularization; and to use information on multiple authorship to check practices like collective ownership of the code base, as commonly advocated by agile methodologies [26].

**Linux Kernel Evolution and Conway's Law.** Proposed in 1968, Conway's Law asserts that "organizations are constrained to produce application designs which are copies of their communication structures" [27]. Although, proposed in the context of formal organizations, it is also worth to investigate whether it applies to more informal organizations, like open source communities. After reasoning on its use in the context of our study, we are inclined to affirm that Conway's Law does not hold in Linux. Indeed, we collect preliminary evidences that in Linux an *inverse* form of this law better explains the relation between the organization of the Linux development team and the architecture of the system. By inverse, we mean that it is the system's architecture that shaped Linux's development team in the last ten years. Linux follows a monolithic architecture, with a *Core* component responsible for its main features [28]. The other subsystems provide specialized services, such as *Drivers*, *Net*, and *Fs*. This early architectural decision was crucial to define some key characteristics of the Linux's development team along the years. Our study, for example, shows that Linux has a group of top-authors, who are generalists and, therefore, work not only in the *Core* but also in other subsystems. By contrast, the remaining authors tend to focus their work in specific subsystems.

## 5. Measuring Code Authorship in a Large Dataset

In this section, we describe an extension of the Linux kernel study, in which we apply the authorship-related metrics proposed in this work to a large dataset. This dataset is composed of 118 open source systems, which are implemented in six different programming languages; these systems come from our previous truck factor study [12]. The original dataset contains 133 systems, but we removed the torvalds/linux repository because it was previously analyzed (Section 3) and also systems with less than 2 top-level directories (6 repositories) or less than 10 authors (11 repositories).[15] These last two filters are used to enable, respectively, the computation of the specialization and co-authorship measures. Before computing the authorship measures, we updated the repositories regarding our previous study, checking out their repositories on September 27th, 2018.

Figure 9 presents the code authorship measures. In the first violin plot we can observe that the proportion of authors is small in most of the studied systems (the first, second, and third quartiles are 15%, 21%, and 32%, respectively). This behavior was previously observed in the Linux kernel (26%). However, we found systems with a high ratio of authors, which usually are systems with a relevant number of paid developers and some of them are supported by commercial organizations. This is the case of two out of five outliers, including the systems WordPress/WordPress (70%), and JetBrains/intellij-community (60%). We also detected two language interpreters among the top-5 systems: ruby/ruby (67%) and php/php-src (54%).

[15]Three systems match the two filters, thus, in total these filters remove 14 repositories.

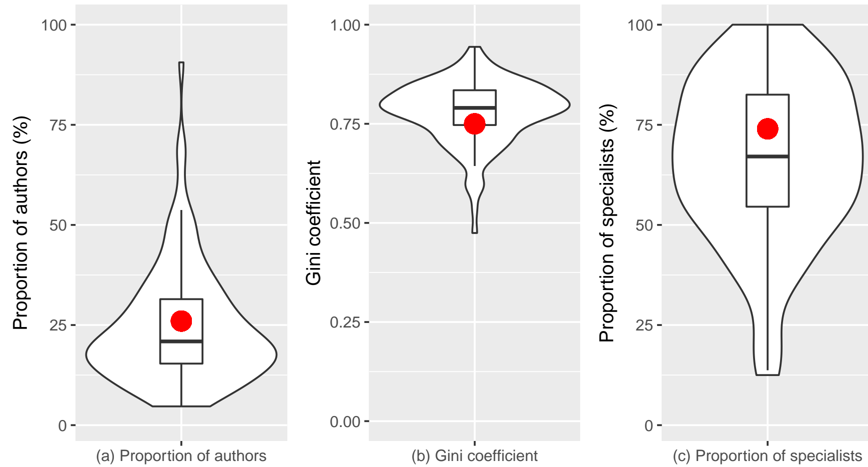



Figure 9: Authorship measures in an extended dataset of 118 open source systems. The Linux kernel results are represented by the red dots.

The second violin plot (Figure 9b) presents the Gini coefficients. The high coefficients (the first, second, and third quartiles are 0.75, 0.79, and 0.83, respectively) show that the distribution of the number of files per author is highly skewed in most of the systems in the dataset. The systems with a more equal distribution (outliers) are github/linguist (0.47), fzaninotto/Faker (0.54), and alexreisner/geocoder (0.59). Most of these outliers have few authors; however, fzaninotto/Faker, with 180 authors, is an exception. Its low Gini coefficient, when contrasted with the other systems in the dataset, is the result of its plugin-based software architecture. The project has a small core and most of the repository's files are plugins (named *providers*) developed by the community. Again, the Linux kernel's Gini coefficient (0.75) is close to the median value of the distribution in the extended dataset.

Finally, the third violin plot (Figure 9c) presents the proportion of specialists; in this case, we consider as subsystems the top-level directories of the cloned repositories. The high specialization observed in the Linux kernel (74%) is also found in most of the systems in our extended dataset (the first, second, and third quartiles are 55%, 67%, and 83%, respectively).

**Co-authorship measures.** We also build the co-authorship network of the 118 systems in the extended dataset. Figure 10 presents the co-authorship measures. The first violin plot shows the mean degree, which is low in most of the systems (the first, second, and third quartiles are 1.28, 1.80, and 2.41, respectively). Similar to what we observed in Figure 9a, high mean degrees are more common in repositories supported by commercial organizations—e.g., JetBrains/intellij-community (9.01), v8/v8 (7.02), and WordPress/WordPress (4.52)—and in language interpreters—e.g., php/php-src (5.08) and ruby/ruby (4.84). Although not so high

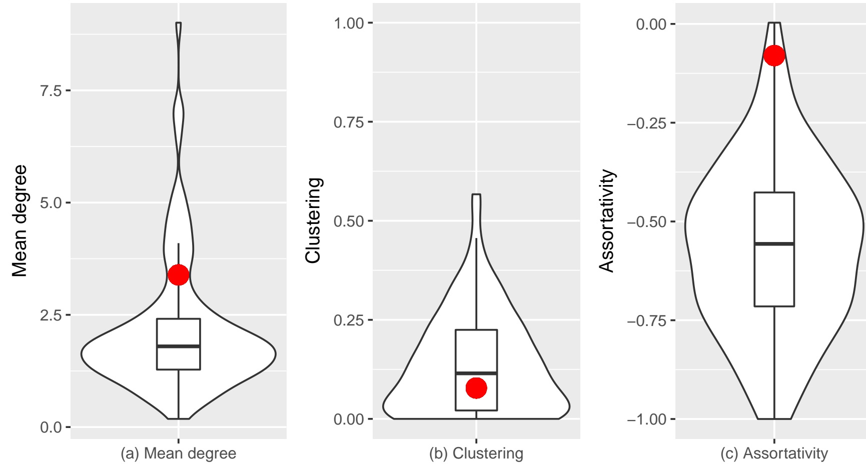



Figure 10: Co-authorship measures in an extended dataset of 118 open source systems. The Linux kernel results are represented by the red dots.

as in these commercial projects, the Linux kernel also presents a high mean degree (3.39), when contrasting with the entire dataset results (mean equal to 2.09). The clustering coefficient (Figure 10b) indicates that most systems have a low probability to form high density co-authorship clusters (the first, second, and third quartiles are 0.02, 0.11, and 0.22, respectively). This trend is also followed by the Linux kernel (0.078). Finally, the assortativity coefficients are presented in the third violin plot. With the exception of webscalesql/webscalesql-5.6 (0.003), all the repositories have negative coefficients. In other words, authors with a high number of co-authorship connections tend to collaborate with authors with less connections, suggesting some sort of mentorship. This behavior was previously observed in the Linux kernel ($-0.08$), but it is more intense in the extended dataset. As example, the assortativity coefficients of Eugeny/ajenti and substack/node-browserify are $-1.00$ (the minimal possible value). These two repositories have one main author and all co-authorship connections are between this main author and authors without any additional connections.

**Analysis by Project Size:** We also divided the projects in the extended dataset by size (measured in lines of code): Small Projects (1st quartile), Medium Projects (2nd and 3rd quartiles), and Large Projects (4th quartile). Then, we compared the distributions of the metrics studied in this paper, considering the projects in each category. First, Table 6 shows the results of Cliff's delta pairwise comparison coefficient [29] to reveal how distinct are the distributions, when considered in pairs. For example, when we compare Small vs Large projects there is a *large* difference in terms of Gini coefficients and also in all three co-authorship measures (mean degree, clustering, and assortativity). For proportion of authors, there is a *medium* difference; by contrast, for proportion

Table 6: Pairwise comparison (Cliff's delta $|d|$)

| Measure | Small x Medium | Medium x Large | Small x Large |
|---|---|---|---|
| **Proportion of Authors** | small (0.260) | small (0.226) | medium (0.456) |
| **Gini coeficient** | medium (0.460) | small (0.213) | **large** (0.680) |
| **Proportion od Specialists** | small (0.185) | small (0.307) | negligible (0.127) |
| **Mean Degree** | medium (0.431) | **large** (0.659) | **large** (0.866) |
| **Clustering** | **large** (0.521) | **large** (0.541) | **large** (0.830) |
| **Assortativity** | medium (0.393) | **large** (0.697) | **large** (0.784) |

of specialists the difference is *negligible*. To clarify these results, Figure 11 shows violin plots with the distribution of proportion of authors, Gini coefficients, and proportion of specialists, per project category. The Linux kernel is represented by a red dot. For proportion of authors, for example, the value computed for the Linux kernel is more similar to the median of the same value computed for Large projects; however, for proportion of specialists, the Linux kernel results is more similar to the median value of the Medium-sized projects. Finally, Figure 12 shows violin plots with the distributions of the three remaining measures: mean degree, clustering, and assortativity. For mean degree, the Linux results is more similar to the median of the Large projects. However, in the case of assortativity, the Linux kernel measure is greater than the measures achieved on each category (Small, Medium, and Large).

*Summary: Most of the authorship patterns previously observed in the Linux kernel are also common in an extended dataset of 118 popular GitHub open source projects. For example, the repositories in this dataset also have few authors, skewed distributions of the number of files per author, and a high proportion of specialists. Additionally, the co-authorship network measures confirm some initial assumptions about co-authorship patterns, such as: most of the repositories have a low mean degree (but this is higher in projects with commercial support), and it is common some kind of mentorship, where authors with many co-authorship connections are connected with authors with fewer connections.*

## 6. Threats to Validity

**Construct Validity.** With respect to construct validity, our results depend on the accuracy of DOA calculations (which for example assume that the file's creator is the one that most contributed to its code). Additionally, we calculate the degree-of-authorship values using weights from the analysis of other systems [10, 11]. Although the authors of the DOA metric claim their weights as general, we cannot eliminate the threat that their choice pose to our results.

To strengthen our confidence on DOA, we also compare the results computed for the Linux kernel (release v4.17) with the number of commits in a file, which can be regarded as an alternative and simple code authorship measure. For this comparison, we consider that authors should have at least 10 commits in

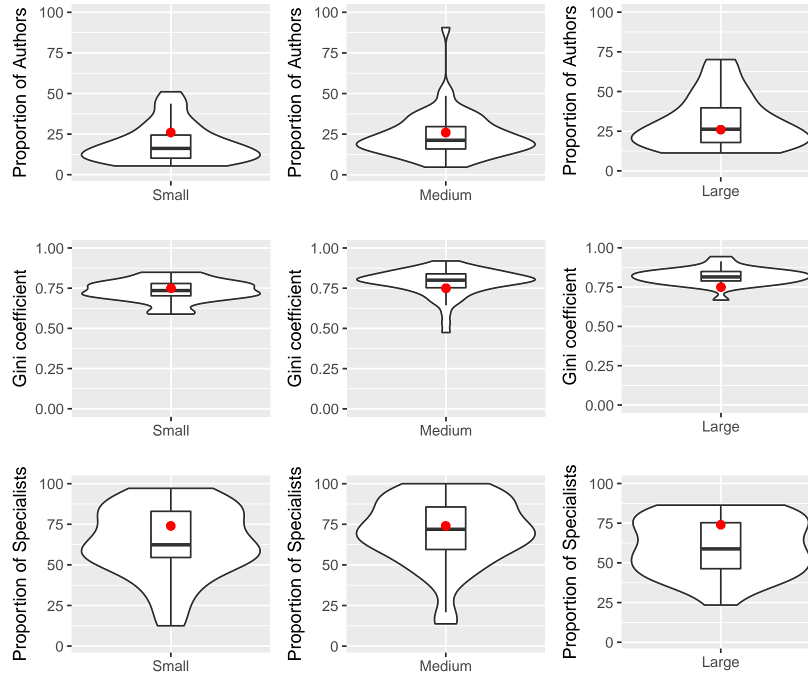



Figure 11: Authorship measures (proportion of authors, Gini coefficient, and proportion of specialists) in an extended dataset of 118 open source systems, by project category (Small, Medium, and Large Size Projects). The Linux kernel results are represented by the red dots.

a file (or be the developers with the highest number of commits, for files with less than 10 commits or files where the developer with the highest number of commits has less than 10 commits). We found that in 58% of the Linux kernel files the authors are exactly the same, despite being computed using DOA (as in our study) or using this threshold of 10 commits. For 17%, DOA-based authors are a subset of the commit-based authors, and for 16% they are a superset.[16] Finally, for 9% of the Linux kernel files, the author sets are disjoint. This analysis reveals a high agreement rate between DOA and a metric based on the number of commits. Indeed, we report a very similar result in a previous study, based on a ground-truth where 159 developers self-assessed their expertise on 654 files from 10 large software systems, including two commercial systems and 8 open-source systems [9].

Additionally, our normalized DOA measures may also vary due to possible developer aliases. We mitigate such a threat by handling the most common sources of aliasing—see Section 2.3. Finally, our validation of the DOA results

[16] However, in 69% of these cases—DOA-based authors subset of commit-based authors or vice-versa—only one developer is responsible for the set differences.

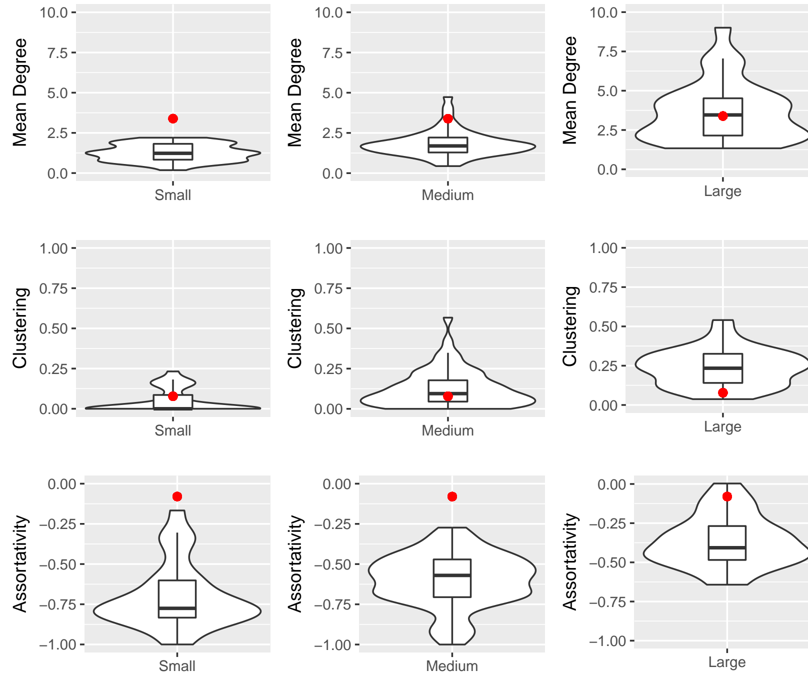



Figure 12: Co-authorship measures (mean degree, clustering, and assortativity) in an extended dataset of 118 open source systems, by project category (Small, Medium, and Large Size Projects). The Linux kernel results are represented by the red dots.

relies on the

**Internal Validity.** We measure authorship considering only the commit history of the official Linux kernel Git repository. Hence, we do not consider forks that are not merged into the mainstream development. Although these changes might be relevant to some (e.g., studies about integration activities, like rebasing and cherry-picking [30]), they are not relevant when measuring authorship of the official Linux kernel codebase. We also consider that all commits have the same importance when computing authorship. As such, we do not account for the granularity of changes (number of lines of code modified, deleted or inserted) nor their semantics (e.g.„ bug fixes, new features, refactoring, etc). In open source systems, it is common that many contributors do not have permission to directly commit their code to the main source code repository. In such scenario, their code, when approved, is integrated by another developer (the committer). Git systems store both information, the author and the commiter of the changes. To give credit to the developer who really performed the change, our approach relies on the author information to compute files' authorship.

**External Validity.** The metrics we use can be applied to any software repos-

itory under a version control system. Additionally to Linux kernel we applied the same metrics to a large dataset of open source systems. Although the results confirm that most of the findings identified in the Linux kernel are also common in the extended dataset, we cannot assume that the findings about workload, specialization, and collaboration among file authors are general. In special to not open source systems. Nonetheless, we pave the road for further studies to validate our findings in such development environment.

## 7. Related Work

**Assessing Code Authorship.** McDonald and Ackerman proposed the "Line 10 Rule", one of the first and most used heuristic for expertise recommendation [31]. The heuristic is based on how developers use change history to identify experts on particular files, generally assuming that the last person who changed a file is most likely to be "the" expert. Expertise Browser [32] and Emergent Expertise Locator [33] are alternative implementations to the "Line 10 Rule". The former uses the concept of experience atoms (EA) to give value for each developer's activity and takes into account the amount of EAs to quantify expertise. The latter refines the Expertise Browser approach by considering the relationship between files that are changed together when determining expertise. A finer-grained algorithm that assigns expertise based on the percentage of lines a developer has last touched is used by Girba et al. [34] and also by Rahman and Devanbu [35]. Version Control Systems (VCS) usually provide their own "blame" tools, such as *git-blame* [16] and *svn-blame* [36]. Essentially, these tools reveal the authors who last modified each line in a file. Girba et al. also propose a graph tool that models how the developers interacted and disputed the code expertise over time. Schuler and Zimmermann [37] consider that developers also accumulate expertise by calling methods.

Our study relies on the *Degree-of-Authorship* (DOA) model [10, 11] to identify the developers who performed the most significant contributions to a file. Different from techniques based on the "Line 10 Rule", the DOA equation considers the whole version history to compute the degree-of-authorship of a developer with respect to a given code element. In fact, DOA is one of the components of the *Degree-of-Knowledge* (DOK) model, which combines authorship and interaction data to identify source code experts. However, DOK-based measures depend on the installation of special monitoring tools on IDEs, which makes difficult its use in large scale, to evaluate systems implemented in different languages, by multiple developers, and without imposing any constraint on their IDEs. For example, the Mylyn plugin [38, 39]—the *de facto* implementation to compute DOK-based measures—is available for a single language (Java) and requires the usage of an specific IDE (Eclipse).

To our knowledge, we are the first to use DOA in a complex project (Linux) and further in a large dataset of open source projects. We are also the first to investigate questions such as proportion of authors per project, distribution of

number of files per author, and authors specialization, using authorship measures as computed by the DOA metric.

**Social Network Analysis (SNA).** Research in this area usually extract information from source code repositories to build a social network, adopting different strategies to create the links between developers. Fernández et al. [40] apply SNA to study the relationship among developers and how they collaborate in different parts of a project. They use a coarse-grained approach, linking developers that perform commits to the same module. Other studies rely on fine-grained relations, building networks connecting developers that change the same file [41, 42, 43, 44]. Joblin et al. [45] propose an even more fine-grained approach. They claim that file-based links result in a dense network, which obscures important network properties, such as community structure. For this reason, they connect developers that change the same function in a source code. Our results, although centered on file-level information, does not produce dense networks, as authorship requires that developers make significant contributions to a file. Consequently, files usually have few authors. Other studies build social networks from mailing lists [46, 47], issue tracks [48, 49], or connect developer that work in the same projects, building inter-projects networks [50, 51] . They also investigate the use of collaboration networks to predict failures [42, 52, 44] and to assess peer review process [41].

**Open Source Development**. One of the first studies on open source software development was conducted by Mockus et al. [5]. The authors investigate the Apache and Mozilla ecosystems, including aspects such as developer's participation, core team size, code ownership, productivity, and defect density. They also compare their results with commercial projects. Robles et al. [53] propose a quantitative methodology to study the evolution of core teams in open source projects. Their methodology can be used to detect unusual events in the evolution of core teams (e.g., break points) or to assess unevenness in the contributions of core team members, when compared to the rest of the contributors. Jiang et al. [54] propose an approach to recover the full reviewing history of patches in mailing-list based reviewing environments, which were common before modern web-based reviewing tools. They apply this approach on Linux and achieve a F-score of 85% when linking commits to mails containing the corresponding patch version. Vasilescu et al. investigate how the workload of the contributors varies across projects in the Gnome community [55]. Recent studies were also conducted using GitHub projects. For example, there is work investigating the impact of GitHub features, such as social coding [56, 57], pull requests [58, 59], stars [60, 61] and distributed version control [62, 63]. Other studies investigate the influence of programming language on software quality [64], the characteristics of community contributions [65], and the reasons for the deprecation of open source projects [66].

In contrast with some of such work, our study does not try to explain the OSS development model or specific characteristics of the GitHub environment. Instead, we propose a set of code authorship based metrics that could be applied to analyze the development and evolution of open source projects.

## 8. Conclusion

Seeking to contribute to a better understanding of how authorship-related measures evolve in successful open source communities, we measure and analyze authorship parameters from 1 + 118 systems. Initially, we propose and investigate authorship measures in the Linux kernel. By mining over 13 years of the Linux kernel commit history, we reveal some organizational aspects of the Linux development team, such as authors workload, degree of specialization and co-authorship patterns. Additionally, we apply the same authorship measures to an extended dataset of 118 popular projects and confirm that the authorship patterns observed in the Linux are also followed by other open source systems.

As future work, we seek to validate our findings directly with the systems' developers. Moreover, we plan to study authorship in commercial systems and contrast the results with the ones presented in this paper.

### Acknowledgments

This study is supported by grants from FAPEMIG, CAPES, CNPq, and UFPI.

### References

[1] K. Crowston, J. Howison, The social structure of free and open source software development, First Monday 10 (2).

[2] J. D. Herbsleb, Global software engineering: the future of socio-technical coordination, in: Future of Software Engineering (FOSE), 2007, pp. 188–198.

[3] I. Mistrík, J. Grundy, A. van der Hoek, J. Whitehead, Collaborative software engineering: challenges and prospects, Springer, Heidelberg, 2010, pp. 389–403.

[4] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

[5] A. Mockus, R. T. Fielding, J. D. Herbsleb, Two case studies of open source software development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology 11 (3) (2002) 309–346.

[6] K. J. Sullivan, W. G. Griswold, Y. Cai, B. Hallen, The structure and value of modularity in software design, in: 9th International Symposium on Foundations of Software Engineering (FSE), 2001, pp. 99–108.

[7] C. Y. Baldwin, K. B. Clark, Design rules: the power of modularity, MIT Press, London, 1999.

[8] A. Meneely, L. Williams, Socio-technical developer networks: Should we trust our measurements?, in: 33rd International Conference on Software Engineering (ICSE), 2011, pp. 281–290.

[9] G. Avelino, L. Passos, A. Hora, M. T. Valente, Assessing Code Authorship: The Case of the Linux Kernel, in: 13th International Conference on Open Source Systems (OSS), 2017, pp. 151–163.

[10] T. Fritz, J. Ou, G. C. Murphy, E. Murphy-Hill, A degree-of-knowledge model to capture source code familiarity, in: 32nd International Conference on Software Engineering (ICSE), 2010, pp. 385–394.

[11] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, E. Hill, Degree-of-knowledge: modeling a developer's knowledge of code, ACM Transactions on Software Engineering and Methodology 23 (2) (2014) 14:1–14:42.

[12] G. Avelino, L. Passos, A. Hora, M. T. Valente, A novel approach for estimating truck factors, in: 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10.

[13] A. Hora, D. Silva, R. Robbes, M. T. Valente, Assessing the threat of untracked changes in software evolution, in: 40th International Conference on Software Engineering (ICSE), 2018, pp. 1102–1113.

[14] G. Avelino, L. Passos, F. Petrillo, M. T. Valente, Who can maintain this code? assessing the effectiveness of repository-mining techniques for identifying software maintainers, IEEE Software 1 (1) (2018) 1–15.

[15] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux device drivers, O'Reilly, Sebastopol, 2005.

[16] S. Chacon, B. Straub, Pro Git, Apress, New York, 2014.

[17] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.

[18] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, M. T. Valente, Feature scattering in the large: a longitudinal study of Linux kernel device drivers, in: 14th International Conference on Modularity, 2015, pp. 81–92.

[19] C. Gini, Measurement of inequality of incomes, The Economic Journal 31 (121) (1921) 124–126.

[20] I. T. Bowman, R. C. Holt, N. V. Brewster, Linux as a case study: Its extracted software architecture, in: 21st International Conference on Software Engineering (ICSE), New York, 1999, pp. 555–563.

[21] D. J. Watts, S. H. Strogatz, Collective dynamics of small-world networks, Nature 393 (1998) 440–2.

[22] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Rev. Mod. Phys. 74 (2002) 47–97.

[23] M. E. J. Newman, Mixing patterns in networks, Physical Review E 67 (2003) 026126.

[24] M. E. J. Newman, Coauthorship networks and patterns of scientific collaboration, Proceedings of the National Academy of Sciences 101 (2004) 5200–5205.

[25] J. Corbet, G. Kroah-Hartman, A. McPherson, Who writes Linux: Linux kernel development: how fast it is going, who is doing it, what they are doing, and who is sponsoring it, Tech. rep., Linux Foundation (2015).

[26] K. Beck, C. Andres, Extreme programming explained: embrace change, Addison-Wesley, Boston, 2004.

[27] M. E. Conway, How do committees invent, Datamation 14 (4) (1968) 28–31.

[28] R. Love, Linux kernel development, Addison-Wesley, Boston, 2010.

[29] R. J. Grissom, J. J. Kim, Effect Sizes for Research: A Broad Practical Approach, 2005.

[30] D. M. German, B. Adams, A. E. Hassan, Continuously mining distributed version control systems: an empirical study of how Linux uses Git, Empirical Software Engineering 21 (1) (2015) 260–299.

[31] D. W. McDonald, M. S. Ackerman, Expertise recommender: a flexible recommendation system and architecture, in: 7th Conference on Computer Supported Cooperative Work (CSCW), 2000, pp. 231–240.

[32] A. Mockus, J. D. Herbsleb, Expertise browser: a quantitative approach to identifying expertise, in: 24th International Conference on Software Engineering (ICSE), 2002, pp. 503–512.

[33] S. Minto, G. C. Murphy, Recommending emergent teams, in: 4th Workshop on Mining Software Repositories (MSR), 2007, pp. 5–5.

[34] T. Girba, A. Kuhn, M. Seeberger, S. Ducasse, How developers drive software evolution, in: 8th International Workshop on Principles of Software Evolution (IWPSE), 2005, pp. 113–122.

[35] F. Rahman, P. Devanbu, Ownership, experience and defects, in: 33rd International Conference on Software Engineering (ICSE), 2011, pp. 491–500.

[36] C. M. Pilato, B. Collins-Sussman, W. F. Brian, Version Control with Subversion, O'Reilly, Sebastopol, 2008.

[37] D. Schuler, T. Zimmermann, Mining usage expertise from version archives, in: 5th International Working Conference on Mining Software Repositories (MSR), 2008, pp. 121–124.

[38] M. Kersten, G. C. Murphy, Using task context to improve programmer productivity, in: 14th International Symposium on Foundations of Software Engineering (FSE), 2006, pp. 1–11.

[39] G. C. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Software 23 (4) (2006) 76–83.

[40] L. López-Fernández, G. Robles, J. M. Gonzalez-Barahona, I. Herraiz, Applying social network analysis techniques to community-driven libre software projects, International Journal of Information Technology and Web Engineering 1 (2006) 27–48.

[41] X. Yang, Social network analysis in open source software peer review, in: 22nd International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 820–822.

[42] A. Meneely, L. Williams, W. Snipes, J. Osborne, Predicting failures with developer networks and social network analysis, in: 16th International Symposium on Foundations of Software Engineering (FSE), 2008, pp. 13–23.

[43] A. Jermakovics, A. Sillitti, G. Succi, Mining and visualizing developer networks from version control systems, in: 4th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2011, pp. 24–31.

[44] C. Bird, N. Nagappan, B. Murphy, H. Gall, P. Devanbu, Don't touch my code!, in: 19th International Symposium on Foundations of Software Engineering (FSE), 2011, pp. 4–14.

[45] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, D. Riehle, From developer networks to verified communities: a fine-grained approach, in: 37th International Conference on Software Engineering (ICSE), 2015, pp. 563–573.

[46] C. Bird, D. Pattison, R. D'Souza, V. Filkov, P. Devanbu, Latent social structure in open source projects, in: 16th International Symposium on Foundations of Software Engineering (FSE), 2008, pp. 24–35.

[47] W. Zhang, Y. Yang, Q. Wang, Network analysis of OSS evolution: an empirical study on ArgoUML project, in: 12th International Workshop on Principles of Software Evolution (IWPSE), 2011, pp. 71–80.

[48] S. Panichella, G. Canfora, D. M. Penta, R. Oliveto, How the evolution of emerging collaborations relates to code changes: an empirical study, in: 22nd International Conference on Program Comprehension (ICPC), 2014, pp. 177–188.

[49] Q. Hong, S. Kim, S. C. Cheung, C. Bird, Understanding a developer social network and its evolution, in: 27th International Conference on Software Maintenance (ICSM), 2011, pp. 323–332.

[50] G. Madey, V. Freeh, R. Tynan, The open source software development phenomenon: An analysis based on social network theory, in: Americas Conference on Information Systems (AMCIS), 2002, pp. 1806–1813.

[51] J. Xu, Y. Gao, S. Christley, G. Madey, A topological analysis of the open souce software development community, in: 38th Annual Hawaii International Conference on System Sciences, 2005, pp. 198a–198a.

[52] M. Pinzger, N. Nagappan, B. Murphy, Can developer-module networks predict failures?, in: 16th International Symposium on Foundations of Software Engineering (FSE), 2008, pp. 2–12.

[53] G. Robles, J. M. Gonzalez-Barahona, I. Herraiz, Evolution of the core team of developers in libre software projects, in: 6th International Working Conference on Mining Software Repositories (MSR), 2009, pp. 167–170.

[54] Y. Jiang, B. Adams, F. Khomh, D. M. German, Tracing back the history of commits in low-tech reviewing environments: a case study of the Linux kernel, in: 8th International Symposium on Empirical Software Engineering and Measurement, (ESEM), 2014, pp. 1–10.

[55] B. Vasilescu, A. Serebrenik, M. Goeminne, T. Mens, On the variation and specialisation of workload – a case study of the Gnome ecosystem community, Empirical Software Engineering 19 (4) (2014) 955–1008.

[56] B. Vasilescu, Human aspects, gamification, and social media in collaborative software engineering, in: 36th International Conference on Software Engineering (ICSE), 2014, pp. 646–649.

[57] B. Vasilescu, S. V. Schuylenburg, J. Wulms, A. Serebrenik, M. G. Brand, Continuous integration in a social-coding world: empirical evidence from GitHub, in: 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 401–405.

[58] G. Gousios, M. Pinzger, A. V. Deursen, An exploratory study of the pull-based software development model, in: 36th International Conference on Software engineering (ICSE), 2014, pp. 345–355.

[59] J. Tsay, L. Dabbish, J. Herbsleb, Let's talk about it: evaluating contributions through discussion in GitHub, in: 22nd International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 144–154.

[60] H. Borges, M. T. Valente, What's in a GitHub star? understanding repository starring practices in a social coding platform, Journal of Systems and Software 146 (2018) 112–129.

[61] H. Borges, A. Hora, M. T. Valente, Understanding the factors that impact the popularity of GitHub repositories, in: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 334–344.

[62] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, P. Devanbu, Cohesive and isolated development with branches, in: 15th International Conference on Fundamental Approaches to Software Engineering (FASE), 2012, pp. 316–331.

[63] C. Rodriguez-Bustos, J. Aponte, How distributed version control systems impact open source software projects, in: 9th Working Conference on Mining Software Repositories (MSR), 2012, pp. 36–39.

[64] B. Ray, D. Posnett, V. Filkov, P. Devanbu, A large scale study of programming languages and code quality in GitHub, in: 22nd International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 155–165.

[65] R. Padhye, S. Mani, V. S. Sinha, A study of external community contribution to open-source projects on GitHub, in: 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 332–335.

[66] J. Coelho, M. T. Valente, Why modern open source projects fail, in: 25th International Symposium on the Foundations of Software Engineering (FSE), 2017, pp. 186–196.