The nature and composition of the Linux kernel developer community: a dynamic analysis[1]

Rishab Aiyer Ghosh
MERIT/Institute of Infonomics, University of Maastricht
e-mail: rishab@dxm.org

Paul A. David
Stanford University & Oxford Internet Institute
e-mail: pad@stanford.edu

draft version, 5 March 2003

Abstract: *This paper presents data gathered from a detailed study of the structure and composition of the Linux kernel developer community, as sampled through three versions of the Linux kernel. Based on previously defined methodology and tools, data is presented on the distribution of authorship across modules, the degree of collaboration between authors, and the distribution, structure and inter-dependence of modules. Using successive versions as a proxy for time, the evolution of author contribution and module structure is also presented. Specific further analysis addresses two questions: the factors behind uncredited (unsigned) source code, and the changes in composition of the "core" and "periphery" author groups across successive versions.*

[1] Support for this research was provided by the Project on the Economic Organization and Viability of Open Source Software, which is funded under National Science Foundation Grant NSF IIS-0112962 to the Stanford Institute for Economic Policy Research.(see http://siepr.stanford.edu/programs/OpenSoftware_David/OS_Project_Funded_Announcmt.htm). Thanks to Kamini Aisola for creating the charts and data analysis.

Table of Contents

Table of Figures

1. Problem definition

1.1. Open Source as a means of production

What most research on free software/open source fails to address is that it is primarily a mode of production. Although the motivation of participants and how they are rewarded is naturally of interest, possibly much more important is to understand what conditions lead to the success and failure of the open source model to produce value, to understand how open source projects allocate resources in order to produce value, and to measure how much value they produce – preferably in a way comparable to the measurement of proprietary software production.

Since conventional measures of resource allocation – time and money – are not directly available for open source communities, alternatives are needed. One alternative, which has been demonstrated as an objective measure in numerous studies[2], is developer time as measured by proxy through source code output. In the LICKS project[3], this measure has been used as a basis for studying the evolving internal resource allocation dynamic within the Linux kernel project by the measurement of authored code distribution in the Linux kernel across successive versions (i.e., over time).

Such empirical study – apart from being interesting in itself – is crucial for the development and validation of simulation models for resource allocation in open source projects. It also validates and develops a suite of techniques for measurement[4] that become increasingly important in order to understand and evaluate the productivity of open source projects, since direct monetary or time estimates are likely to continue to be impossible to obtain.

In the LICKS project, we have examined in considerable detail the structure of resource allocation within the Linux kernel. In this paper, we present the data and an initial analysis of this study.

1.2. Why the Linux Kernel?

The Linux kernel is at the core of the GNU/Linux operating system, and is arguably one of the main factors behind the current success of the Free Software/Open Source phenomenon. It also has the advantage of being a clearly defined (and bounded) piece of software with historical versions available in a continuous progression since at least 1993.

The Linux kernel is an exemplar of a "large integrated project". Quite apart from its importance as the core of most F/OS systems, it is a large project by any measure. The most recent

[2] Ghosh & Prakash 2000, Dempsey et al 2002, FLOSS Part V, Gehring et al 2001

[3] Supported by the NSF, see footnote 1 on cover page.

[4] Ghosh 2002

version studied in the LICKS project had over 3 million lines of code and almost 2,300 identified developers (see section 3, Description of the code base sample). Moreover, due to the nature of the kernel, which provides the core services of the operating system (such as memory management, input/output, filing systems, networking, interfacing with various devices) it must be able to function in a well integrated fashion in order to run at all. The open source development mode does not require the integrated nature of code functioning to be directly represented in a tight structure for developer collaboration, but it does require a strong degree of collaboration on a large scale.

While this may not make the Linux kernel a "typical open source project" we believe it is safe to say that there are no "typical" open source projects per se – at any rate, we are only beginning to collect the sort of empirical data necessary in order to classify projects by any sort of type. So although it is not possible for us to say that the internal resource allocation structure for the Linux kernel as studied in the LICKS project is necessarily representative of all open source projects, we can suggest that it provides useful insights into the functioning of at least one large, integrated and *successful* open source project.

2. Tools and methodology: a brief description

The empirical data gathered for each version of the kernel scanned include:

- Authorship/contribution distribution for the source code and its component parts
- The degree of dependency between component parts of the software
- Identified clusters of authorship (groups of authors who work together, identified by their joint work on components

This data is gathered for three separate versions of the Linux kernel, and is further analysed to identify chronological patterns such as:

- Changes in the distribution of authorship – e.g. does it get less concentrated over time?
- Changes in dependency between components – e.g. do components come together or grow apart?

The purpose of this effort in applying a novel methodology of automatic data extraction, and an appropriate system for analysis of collaborative projects of authorship is two-fold: to demonstrate the applicability of the methodology for a specific product – the Linux kernel, and to produce a substantive analysis of the dynamic development of this technically important and emblematic F/OSS project.

The methodology used in LICKS is described in detail in Ghosh 2002. It depends primarily on the CODD suite of software-analysis tools[5]. These provide the following functions:

- Scan source code for author signatures, to identify authorship distribution (referred to as the Ownergrep method): this looks in source code files for claims of authorship, such as comments with "written by", "author" or copyright notices. The opportunity is provided for manual intervention to associate different "author names" with the same author – developers may use different e-mail addresses, for instance, to sign different packages. Author names are then translated into unique numerical identifiers, used for all further analysis (this is done in order to preserve developer privacy – although developer credit claims are public data).
- Identify duplicates and dependencies between source code modules: in the LICKS project, we have used the CODD "function-definition" method to identify dependencies. This works particularly well for source code in the C and C++ programming languages (accounting for over 95% of the Linux kernel source code). It involves identifying functions declared in *header files* included by code files in each package, and linking these dependent code files to supporting code files (in other packages) that define those functions[6].
- Identify links between groups of authors based on their joint participation in projects: links between packages are found by identifying common authors. The degree of linkage is measured based on the ratio of common authors to total authors (of the linked projects), as well as the proportion of code contributed to the linked projects by the common authors. The resulting weighted graph is traversed to identify clusters of developers linked by their joint participation in projects[7].

Vast quantities of data are extracted as a result of these processes, and are collated for further analysis[8].

[5] CODD, first released in 1998, was designed by Rishab Ghosh and Vipul Ved Prakash, implemented by Vipul Ved Prakash and Gregorio Robles. CODD-Cluster was designed by Rishab Ghosh and implemented with Gregorio Robles.

[6] See Ghosh 2002, section 6.3.1, "Identifying function definitions as an aid to dependency analysis"

[7] See Ghosh 2002, section 6.4, "Clusters of authorship"

[8] Gregorio Robles, co-developer of CODD and CODD-Cluster, was part of the LICKS project team and customised CODD to the needs of the LICKS project. Special credit goes to him for implementing the function-definition identification process for CODD dependency analysis, and for processing and packaging the vast quantities of CODD data into usable form for statistical analysis.

3. Description of the code base sample

The code base used for this analysis is the Linux kernel. Three versions of the software were used: version 1.0, version 2.0.30, and version 2.5.25. The use of three versions with a significant time interval separating them allows meaningful analysis to be performed on the dynamics of changes within the development of the kernel.

It should be noted that the first version used here, version 1.0, is *not* the first version of the Linux kernel to be publicly available. The first self-supporting version of the Linux kernel was version 0.11, released in December 1991 and written more-or-less entirely by Linus Torvalds. By version 0.9x in 1993, (sort-of) stable versions of the Linux kernel were being distributed by semi-commercial vendors such as SLS and Slackware[9].

An overview of the three versions can be found in Figure 1.

Figure 1: Linux kernel code base overview

Linux kernel	Version 1.0	Version 2.0.30	Version 2.5.25
Approximate release date	Mar-94	Apr-97	Jul-02
Number of packages[^]	30	60	169
Number of files	593	2,155	12,451
Number of authors[*]	158	618	2,263
% of code uncredited[*]	18.8%	12.2%	14.9%
Number of defined functions[*]	1,748	7,808	48,006
Physical source lines of code[+]	121,987	537,773	3,157,543
Bytes of source code	4,537,588	21,053,719	133,853,396
Development effort, person-years[+]	31	147	945
Total cost to develop (US$)[+]	4,190,548	19,896,220	127,631,643

Notes: [^] as defined for the LICKS project – see section 4, The internal structure of the Linux kernel. [*]as identified by CODD; uncredited code is code for which CODD was unable to find any author signatures. [+]as identified by David Wheeler's SLOCCount[10], which assumes an average salary of $56,286 per head and overhead factor of 2.4 to calculate the $ cost to develop.

4. The internal structure of the Linux kernel

4.1. Directory structure and package definitions

The source code for the Linux kernel is distributed, as with other source code packages, in a hierarchical tree form with several directories and sub-directories. CODD treats the entire contents of a

[9] for historical information on Linux kernel versions, see e.g. http://linuxhistory.org

[10] see Wheeler 2001

given directory as part of a single project or package[11]. In order to apply CODD to the Linux kernel, we had to define what directories were to be treated as packages. We based our definition on the scheme devised by kernel developer Paul "Rusty" Russell[12] for the Linux kernel map project, that resulted in a graphical display of the links between modules of the kernel. As such, modules that are coherently defined and relatively self-contained are treated as packages. Figure 2 shows the mapping from package to directory structure used in the LICKS project. Note that not all directories are present in all versions of the kernel.

As can be seen from figure 2, large integrated components such as mm, the memory management module, or the kernel module itself , are treated as single packages. Several other directories are not treated as single packages, but are examined at a more detailed level where each of their sub-directories is treated as a single package. This is the case for, e.g., drivers and filing systems (fs), each of which contain sub-directories with relatively self-contained modules for implementing different filing systems, say, or drivers for accessing different devices.

The number of packages in these sub-directories increases significantly in version 2.0.30 and 2.5.25; as the packages are small relative to, say, the kernel module, this has consequences for analysis, e.g. measures of code concentration.

Figure 2: LICKS package structure

Linux kernel directory name	Treatment by LICKS
arch/	Subdirectories are packages
Boot	Single package
Documentation	Single package
drivers/	Subdirectories are packages
fs/	Subdirectories are packages
Ibcs	Single package
include/	Subdirectories are packages
Infrastructure	Single package
Init	Single package
Ipc	Single package
Kernel	Single package
Lib	Single package
Mm	Single package
net/	Subdirectories are packages
sound/	Subdirectories are packages
Tools	Single package
Usb	Single package
Video	Single package
Zorro	Single package

[11] note that the rest of this paper uses the terms project and package interchangeably to refer modules or contents of sub-directories that have been defined as a single sub-package of the Linux kernel.

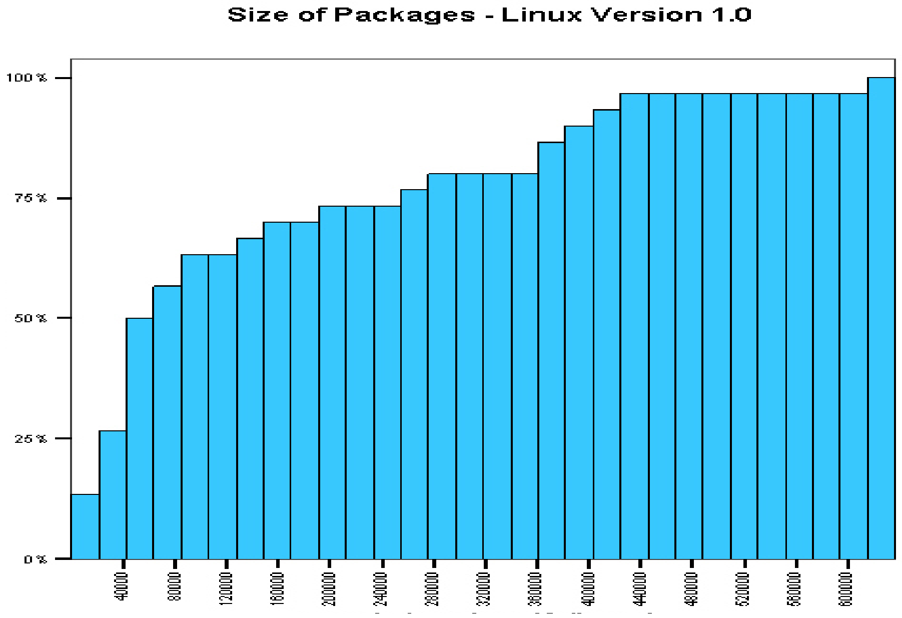
[12] E-mails on file with author. Poster available at http://www.thinkgeek.com/stuff/fun-stuff/3884.shtml

4.2. Package size and distribution

As Figure 1 shows, the growth in number and size of packages from version to version has been huge – a more than four-fold increase in size in each version. What follows is a brief discussion of the distribution of bytes across packages – i.e. the relative size of packages.

It can be seen from figures 3-5 that the majority of packages are small, though given the increasing size of *all* packages, the definition of small varies from below 160 kb (160,000 bytes) for version 1.0 to 1 Mb (1,000,000 bytes) in version 2.5.25.

Figure 3: Cumulative histogram of package size v1.0: bytes against % of packages



A clearer way of visualising the distribution of package sizes is through the Lorenz curves (Fig 6). As the curves and the respective Gini coefficients show, the concentration of packages is increasing in successive versions even as overall total size increases.

This is largely due to the fact that (especially between versions 1.0 and 2.0.30, where the difference in concentration is most significant) the number of modules that are small by definition – drivers for devices, networking etc, and filing systems – increased in number as well as size, while the main modules such as the kernel and memory manager just grew bigger.

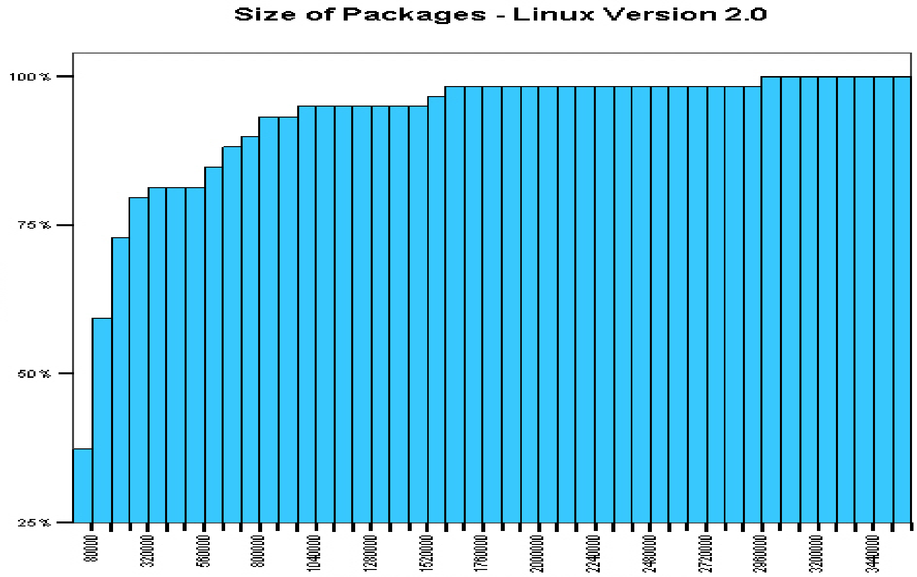
The general tendency for the majority of components to be relatively small is shared across not just versions of the Linux kernel, but appears to be a general rule - with almost "fractal" properties – for any collection of F/OS software. A similar pattern was first observed in the 2000 Orbiten survey[13]

[13] Ghosh & Ved Prakash 2000

and the FLOSS Source Code scan[14] (these were conducted at with a higher-level definition of "package" than LICKS – i.e., the entire Linux kernel was treated as just one "package"). The observation that most packages are small, or developed by small groups of authors, has since been documented in various studies[15].

This general pattern is repeatedly found in Lorenz curves resulting from CODD analysis, regardless of whether it is at the released-package level (where Linux, say, is just one package, as in most previous surveys) or at the sub-package level (where modules of the Linux kernel are treated as distinct packages, as in LICKS).

Figure 4: Cumulative histogram of package size, v2.0.30: bytes against % of packages



[14] FLOSS 2002 Part V

[15] Krishnamurthy 2002, Healy & Schussman 2003

Figure 5: Cumulative histogram of package size, v2.5.25: bytes against % of packages

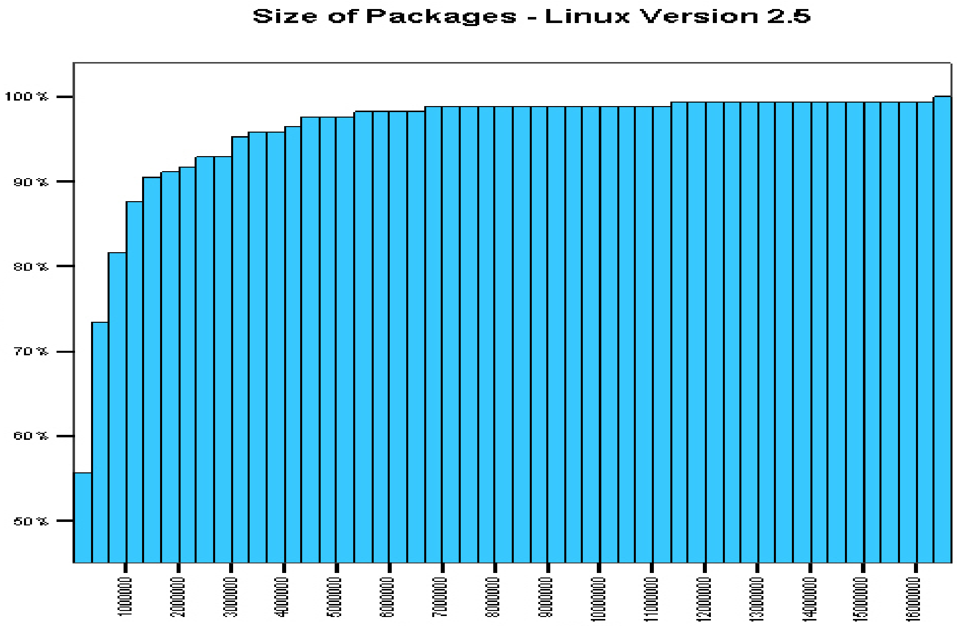
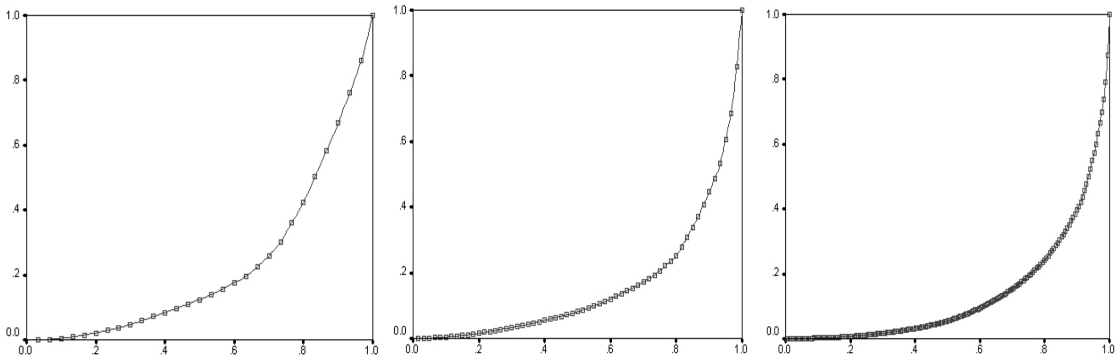


Figure 6: Lorenz curves of package size distribution, v1.0, v2.0.30, v2.5.25



(Gini coefficients: 0.54, 0.68, 0.72).

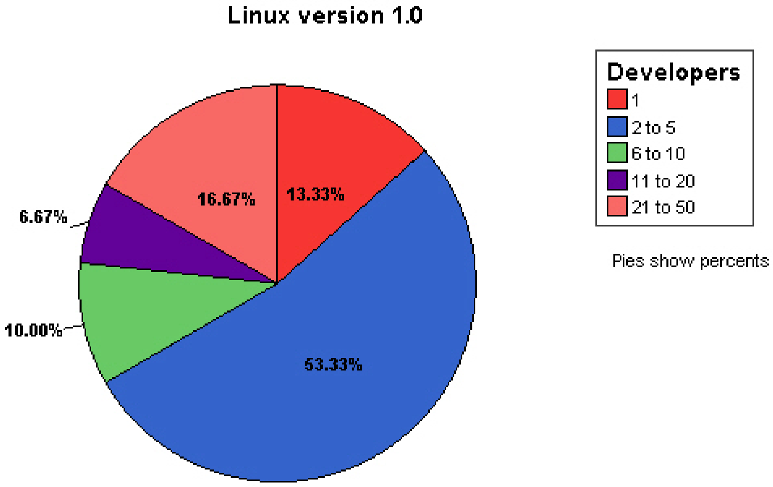
5. Authorship, collaboration and distribution

5.1. Authorship of packages

As Figures 7-9 show, a large fraction of modules have between 2 and 5 developers (53% and 40% for versions 1.0 and 2.0.30, though this reduces to 26% in version 2.5.25) and a significant number of projects have just one developer. Although this reduces from 13% of packages to 5% in version 2.0.30, by version 2.5 it actually increases slightly – probably due to the number of new small packages such as device drivers.

The proportion of packages with a higher number of developers increases with version, but that is largely in line with the increasing number of developers in the newer versions of the kernel as a whole. The distribution of authors across packages is interesting in that it tells us something about the nature and complexity of the packages themselves – in the sense that packages with fewer authors, while possibly of equal *technical* complexity, certainly require less *organisational* complexity. We present more findings on the distribution of authors by package, but it should be noted that these distributions are not weighted by package size.

Figure 7: Number of developers in packages, as % of all packages, v1.0



When we examine author productivity in terms of total code contribution to the kernel, regardless of the number of the distribution of their contribution across packages, we see a surprising consistency in concentration of authorship across all three versions of the kernel. Figure 10 shows the Lorenz curves for authorship in the three versions of the kernel.

Figure 8: Number of developers in packages, as % of all packages, v2.0.30

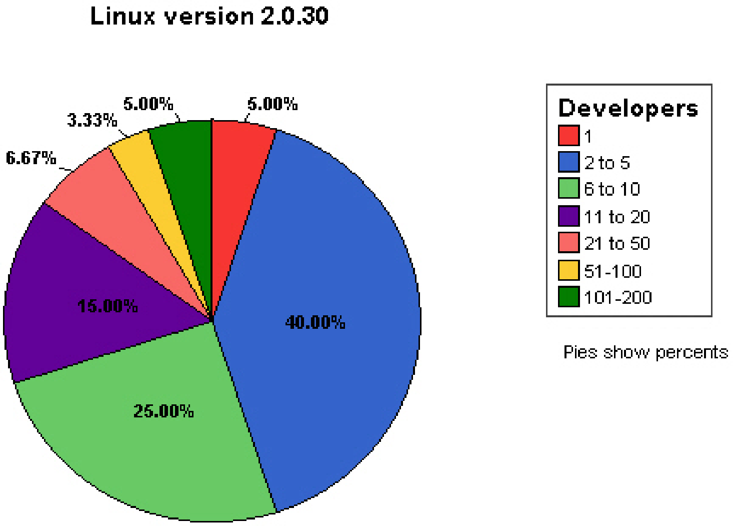
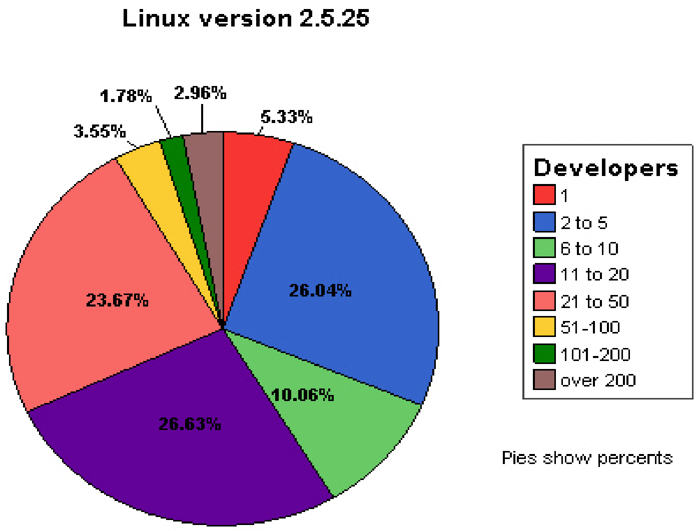
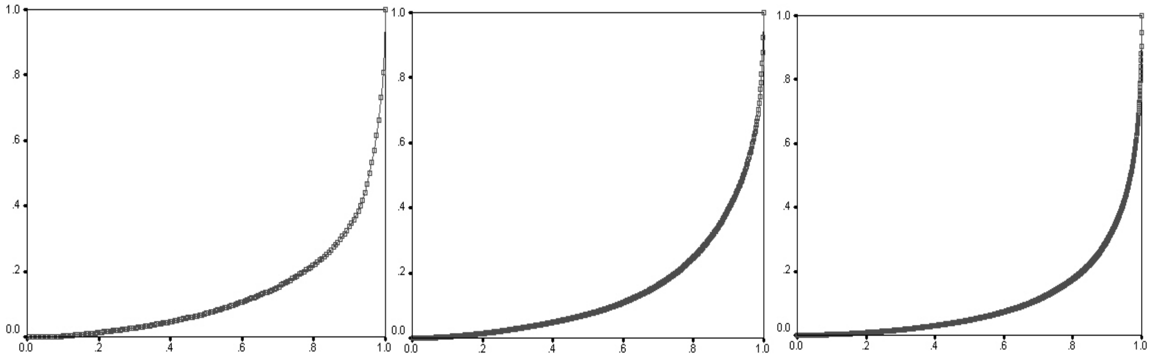


Figure 9: Number of developers in packages, as % of all packages, v2.5.25



The Lorenz curves in Fig 10 show a high degree of concentration. The Gini coefficients are 0.74, 0.71, 0.79 – so, other than a slight dip in version 2.0.30, the concentration of authorship *in terms of total bytes* remains fairly constant, despite the huge increases in the total bytes, number of packages and number of authors. While it is too soon to say without analysis of other projects, this seems to be a typical pattern in many F/OS projects – similar to the curves for package distribution, and similar too to the distribution of authors by size in previous surveys of package authorship[16].

Figure 10: Lorenz curves of developer contribution (total bytes), v1.0, v2.0.30, v2.5.25

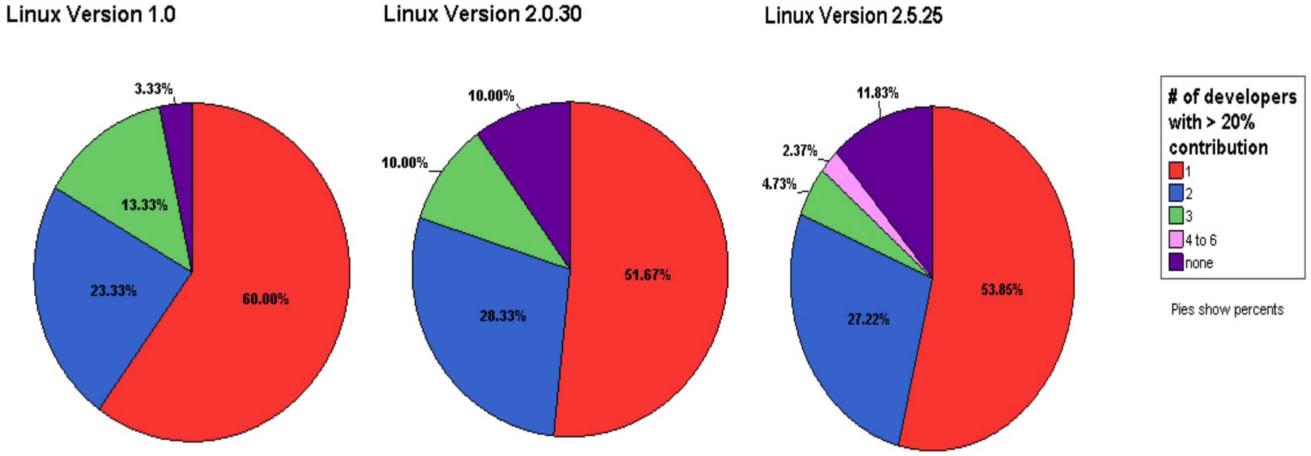


(Gini coefficients: 0.74, 0.71, 0.79).

[16] Ghosh & Ved Prakash 2000, FLOSS 2002 Part V

Turning back to the distribution of authorship by package, figures 11-12 show data on authors who have contributed at least 20% to a given package, followed by authors who have contributed at least 10% to a given package.

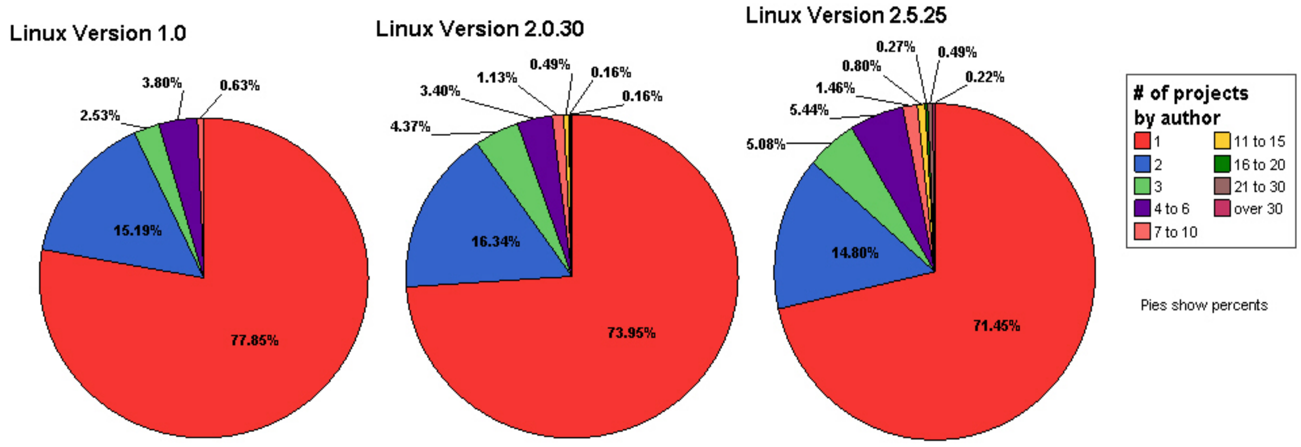
Figure 11: Number of developers contributing >20% to a packages, v1.0, v2.0.30, v2.5.25



5.2. Extent of developer participation

How many packages does a single developer usually contribute to? Figure 13 shows the answer is clearly one. Although the proportion of authors contributing to a single package reduces over versions, from 78% in version 1.0 to 72% in version 2.5.25, there is obviously little correlation between the size and total number of packages in the Linux release and the number of packages to which an individual developer contributes. Very few authors contribute to 7 or more projects – only 0.63% for version 1.0 (i.e. just one author, Linus Torvalds), rising to 1.94% and 3.24% for the next two versions studied. Either way, an overwhelming number of authors contribute to only 1 or 2 packages.

Figure 13: Number of packages authored, v1.0, v2.0.30, v2.5.25



5.3. Collaboration among authors: co-participation and community

It is a cliché that F/OS is a collaborative development model. How collaborative is it, really? The mythology of open source – the “bazaar”, many-eyeballs – suggests that projects are a result of massive collaboration. We do not need to show again how this is simply not true for the majority of projects. Data from the Orbiten and FLOSS surveys, and again from LICKS, show that most projects have one or two developers, or a few more at most; the same data show too that most developers contribute to a handful of projects.

Some researchers have suggested that such data require a reassessment of our understanding of the F/OS model as one of massively collaborating development, especially given that most development occurs in tiny groups[17]. Although there is merit in criticizing the many-eyeballs hype as overly simplistic, the suggestion that most development occurs in small groups without large-scale collaboration implies that F/OS projects develop in a hermetically sealed environment – much as proprietary software development does. Indeed, if one observed a proprietary software project being developed by a team of 3 individuals, one would be correct to assume that only those 3 collaborated in completing that project. This is based on the assumption, though, that developers spend all their (development) time on a given project – as do proprietary software developers – which is simply not true.

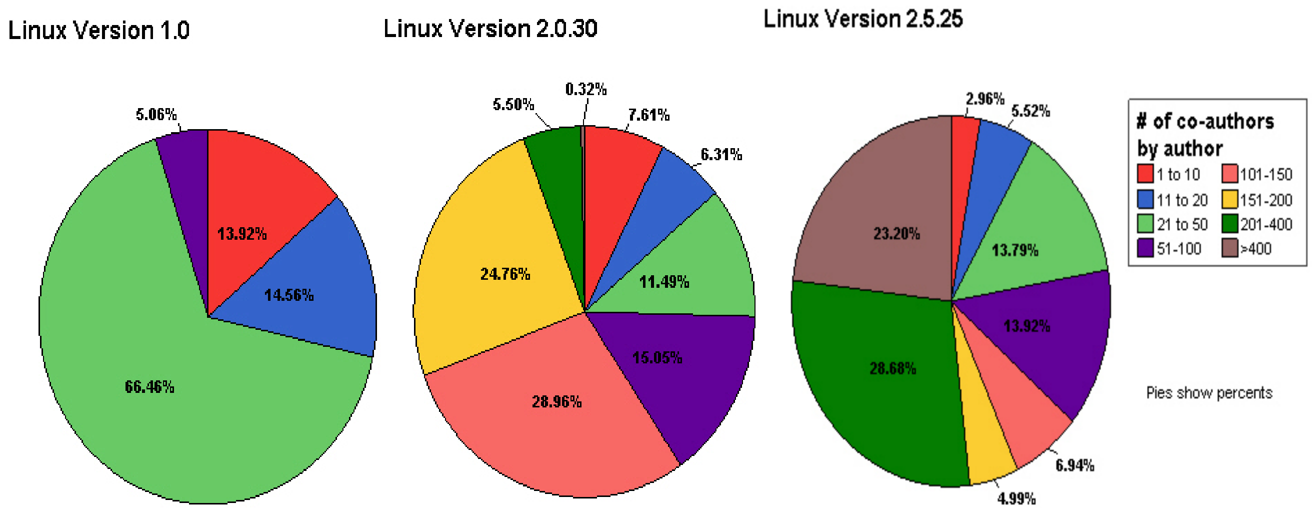
The suggestion that most F/OS development activity occurs in small groups also seems to contradict the motivations presented by developers themselves for participating in F/OS communities – the most important reasons given invariably involve the “learning of new skills” or the “sharing of knowledge”[18].

The fact is that while much production of code integrated into single packages occurs through the collaboration of small groups of developers, this tightness of collaboration is probably limited to the execution of specific projects, and relies on inputs received from other developers external to the project through wider collaborative links.

[17] Krishnamurthy 2002, Healy & Schussman 2003

[18] FLOSS 2002 Part IV, Survey of Developers.

Figure 14: Collaboration with other authors, v1.0, v2.0.30, v2.5.25



Since developers are only human, and usually devote only part of their F/OS development time to any single project, they are bound to feed in results from their collaboration with other developers in external F/OS projects. Evidence of such cross-project collaboration (or extra-project collaboration) has largely been hard to find, and has often been assumed to result from the wide-spread informal communication systems that F/OS relies upon (specifically e-mail discussion lists). However, we are able to show at least for the Linux kernel how authors collaborate[19] in terms of actual code production, and do so extensively.

As has been noted above, most authors contribute to only a single package. However, figure 14 shows that most authors in the Linux kernel collaborated – i.e. contributed code to modules with others – with several developers. Indeed, there are few authors in any of the three versions of the kernel studied who wrote *only* one single package all on their own, as almost all have at least one collaborator[20].

[19] What is found in the source code is not, strictly speaking, evidence of collaboration among authors, but their "co-participation" in the authorship of a given project or module – i.e. appearance of authorship credits for multiple authors of a single source code module. There is a strong argument that "co-participation" in itself implies a high degree of collaboration in the F/OS arena (unlike in publishing, where joint authors of a paper could have contributed different sections with no collaboration at all). For a computer program at the level of a single file or source code module, collaboration in the form of awareness of other developers' contributions is a pre-requisite in order for the program to function at all. This argument is detailed in Ghosh 2002, see footnote 39.

[20] From our findings, the number of developers with no collaborators was 0, 1 and 2 for versions 1.0, 2.0.30 and 2.5.25 respectively. It should be noted that the CODD Ownergrep method rarely provide false positives, in that any two authors identified as co-developers would have credit claims on a common package, which means they would have contributed to the package – though not necessarily at the same time. The exception to this is when two developers are

True, developers contribute to only one package, and many packages have only one developer. However, most developers who contribute to just one package choose to contribute to a large one – i.e. one with several developers. This explains the fact that most (66%) developers in version 1.0 have collaborated with between 21 and 50 others, and 69% of version 2.0.30 developers have collaborated with between 51 and 200 others. The fact that single-package contributors tend to contribute to packages with many developers suggests that reputation – or simply social group-formation – does indeed attract new developers to major packages.

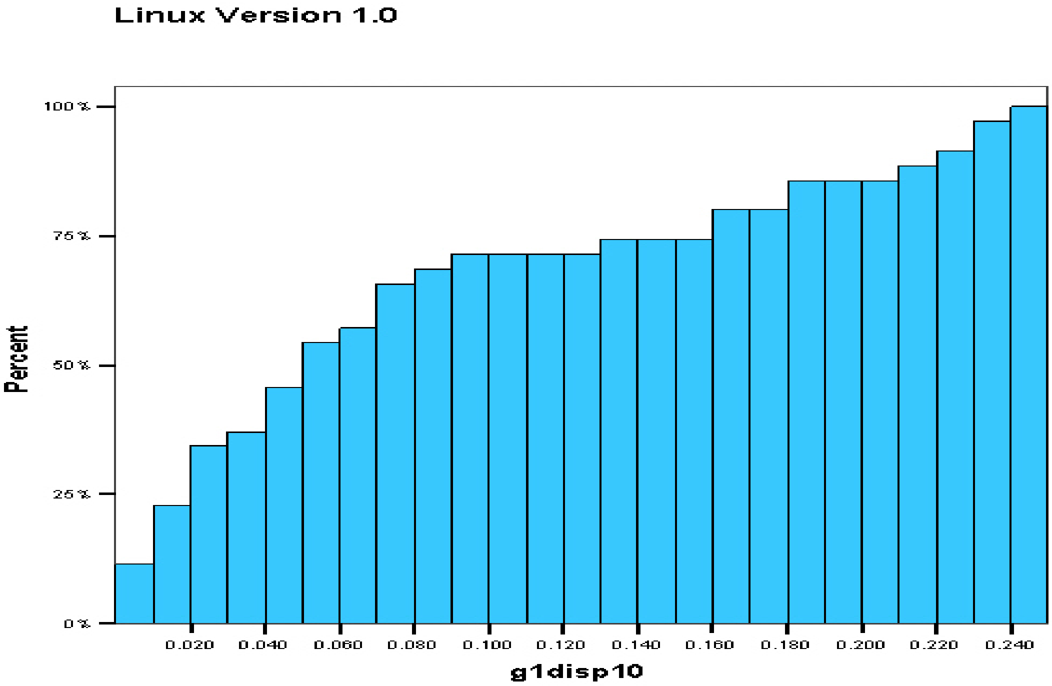
It should be noted that these data are only for the Linux kernel, where, given the nature of the project as a whole, it is unlikely that a new developer will create an entirely new module without previous (or simultaneous) participation in some other Linux kernel package as well. Of course, in the F/OS universe there may well be new developers whose first contribution is an entirely new and self-contained module. That may be unlikely, given that developers would tend to first gain experience in other projects, but it cannot be ruled out (there are programming geniuses, and those with experience in commercial software development). However, such developers also have a much larger opportunity to collaborate to some project or other than do contributors to the Linux kernel which is, after all, an integrated project with a governance structure. So it is unlikely that there are many small groups of developers – let alone individuals – who are totally isolated from the wider developer community, even if measured strictly in terms of collaborative code production as determined by co-participation in code authorship.

5.4. Distribution of author contribution across projects

Among developers who contribute to more than one package, as may be expected, the distribution of their contribution across packages is not uniform. The dispersion measure used here is, for each author, the variance of contribution to individual packages measured as a fraction of the authors contribution to all packages – so the more evenly the authors' contribution is distributed, the closer the dispersion measure is to zero. As shown in Figure 15, a significant percentage (over 25%) of developers distribute their contribution evenly across the packages to which they contribute, and a small number of authors (about 10%) are at the other end of the spectrum, concentrating most of their contribution in a few of their authored packages. Figures 16 and 17 show the corresponding cumulative percentage histograms for dispersion of author contribution for versions 2.0 and 2.5.25.

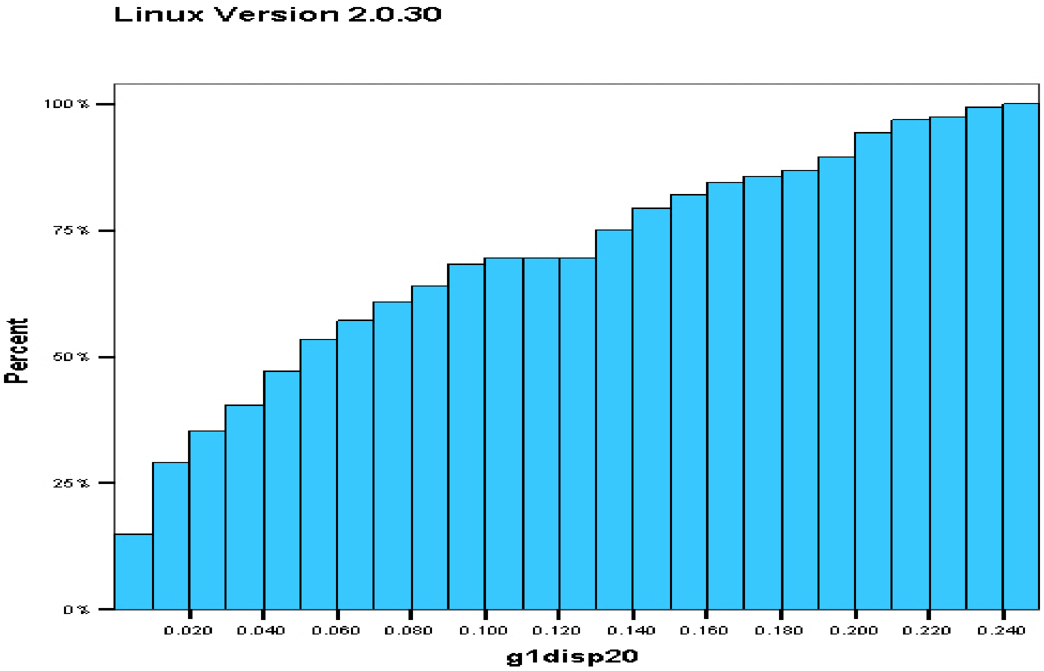
actually the same individual with different identities in the source code files – something for which CODD allows correction, but cannot do fully automatically. Much more likely – as can be seen in the section on unsigned code – is false *negatives*, where CODD identifies fewer developers (and hence co-developers) than actually exist. Given that the true number of co-developers are thus likely to be higher than represented by our data, it would be reasonable to argue that *no* authors contributing to the Linux kernel versions studied have done so with absolutely no co-participation with other authors.

Figure 15: Dispersion of author contribution, cumulative histogram, Linux kernel v1.0



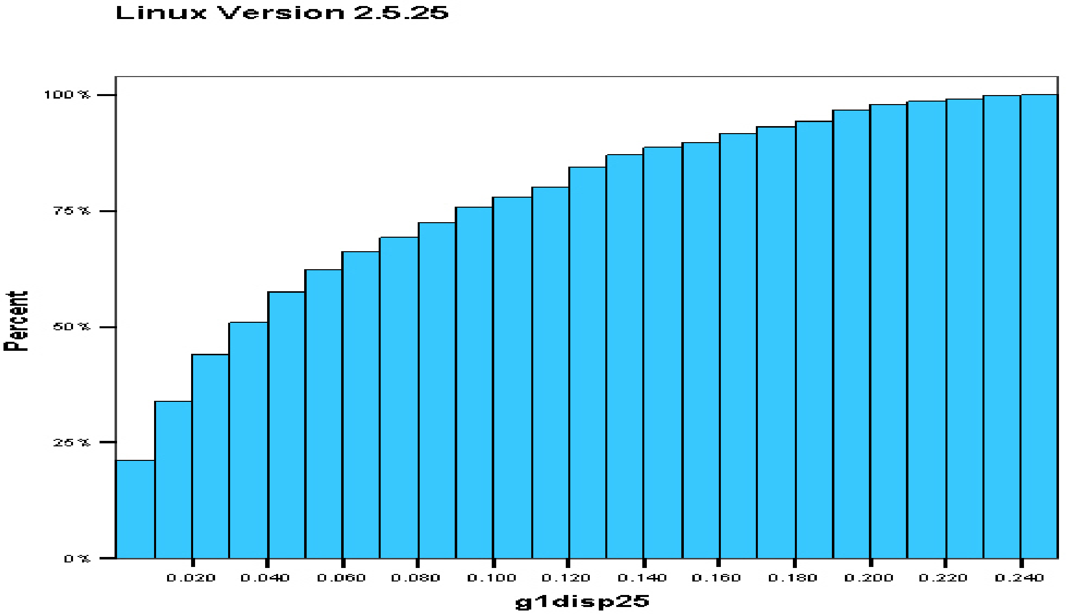
(Variance of author’s contribution across individual packages contributed to, as proportion of author’s total contribution)

Figure 16: Dispersion of author contribution, cumulative histogram, Linux kernel v2.0.30



(Variance of author’s contribution across individual packages contributed to, as proportion of author’s total contribution)

Figure 17: Dispersion of author contribution, cumulative histogram, Linux kernel v2.5.25



(Variance of author's contribution across individual packages contributed to, as proportion of author's total contribution)

There are many things that may influence dispersion of developer contribution across packages, but in general there appears to be a negative correlation[21] between dispersion and total number of packages contributed to – i.e., as author's contribute to more packages, they tend to spread their contribution more evenly across them. On the other hand there is a positive correlation between dispersion and mean contribution (in bytes) to packages – i.e. as an author's mean contribution per package increases, the contribution tends to be more concentrated in a few packages.

There is, however, *no correlation between dispersion and total author contribution* in bytes, which suggests that there may be two categories (at the extrema) of equally productive authors, the first including authors who tend to spread their contribution across several projects evenly, the second of authors who tend to concentrate their contribution in a few of the projects in which they are involved.

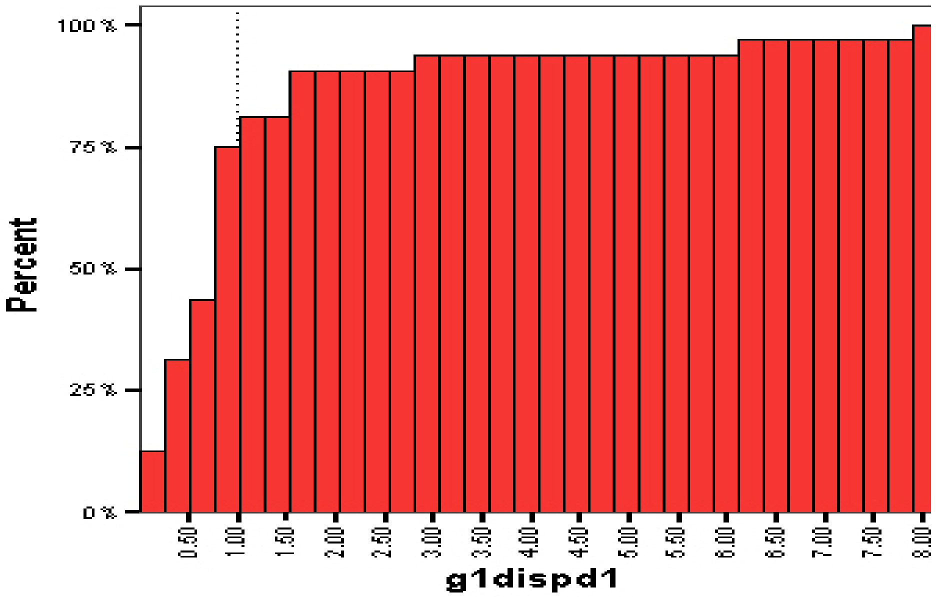
Whether this is a result of a conscious decision of developers to focus most of their contribution on a few projects or spread their code across projects (possibly focusing on specific tasks that are useful for a number of projects) is a subject for further research. Further analysis of the existing data could provide an insight into whether these categories of developers code differently or

[21] For v2.5.25, Pearson two-tailed correlation coefficient of 0.203 between mean contribution and dispersion; -0.259 (negative) correlation coefficient between number of projects contributed to and dispersion

contribute to different types of projects, such as projects defining libraries that support several other projects.

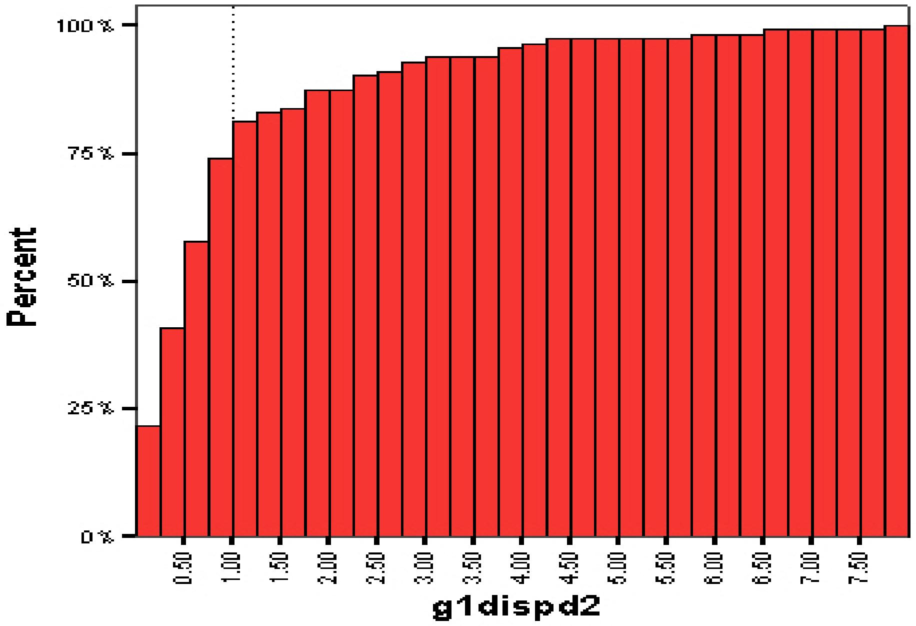
It is arguable that developers who focus on specific projects and those who distribute their contribution more evenly may have different incentives – though it is not necessary that concentrating on a few projects (leading to a high-profile in those projects) must lead to a better reputation than spreading oneself across several projects (leading to a wider reputation possibly for specific, in-demand types of coding that are needed in several projects).

Figure 18: Growth ratio for dispersion of author contribution, change from v1.0 to 2.0.30



(g1dispd1: dispersion in version 2.0.30 / dispersion in v1.0; i.e. 1.0 = no change in dispersion)

Figure 19: Growth ratio for dispersion of author contribution, change from v2.0.30 to v2.5.25



(g1dispd2: dispersion in version 2.5.25 / dispersion in v2.0.30; i.e. 1.0 = no change in dispersion)

Figures 18 and 19 show cumulative histograms of growth ratio in the dispersion value (dispersion divided by dispersion in previous version). As the reference lines at value 1.0 show, for about 75% of developers the dispersion measure reduced or stayed constant (i.e. contribution was more evenly spread across packages in the new version as compared to the previous version of the kernel).

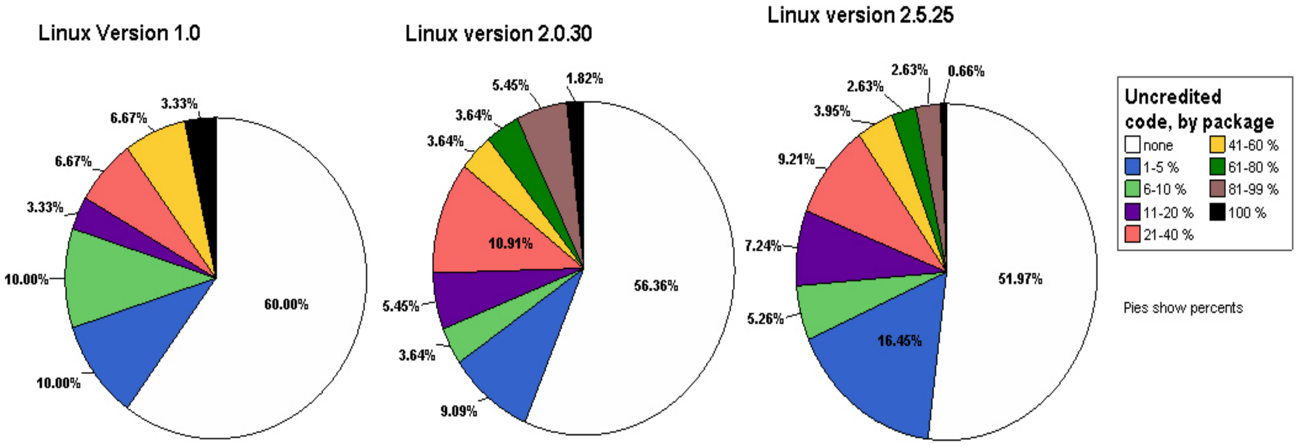
At the other extreme, about 10% of developers increased their dispersion value by a factor of over 6 (i.e. developers' concentration of code in a few of their projects increased very sharply). Of course, much of this change in dispersion is a result of the manifold increase from version-to-version in the number of packages contributed to, and size of packages.

5.5. Unsigned code

As was shown in Figure 1, there is a significant fraction of the total Linux kernel source code that is "uncredited". Generally, this means that it was unsigned code, as CODD was unable to find author signatures or other indications of author credit in the source code. Part of this may be due to signatures that CODD couldn't recognize, but given that CODD looks for several different phrases that could indicate an author signature, the main reason for this is probably that developers simply do not bother to sign all their code.

In some F/OS projects, this is intentional – e.g. the developers of the web server Apache do not put their names in the source code. In the Linux kernel, though, there are no "official" policies towards signing source code, so the presence of unsigned code is probably due to authors simply not bothering to sign. Figure 20 shows the proportion of unsigned code in packages of the Linux kernel. This does not show data weighted by package size, but simply the percentage of packages with specific degrees of unsigned code.

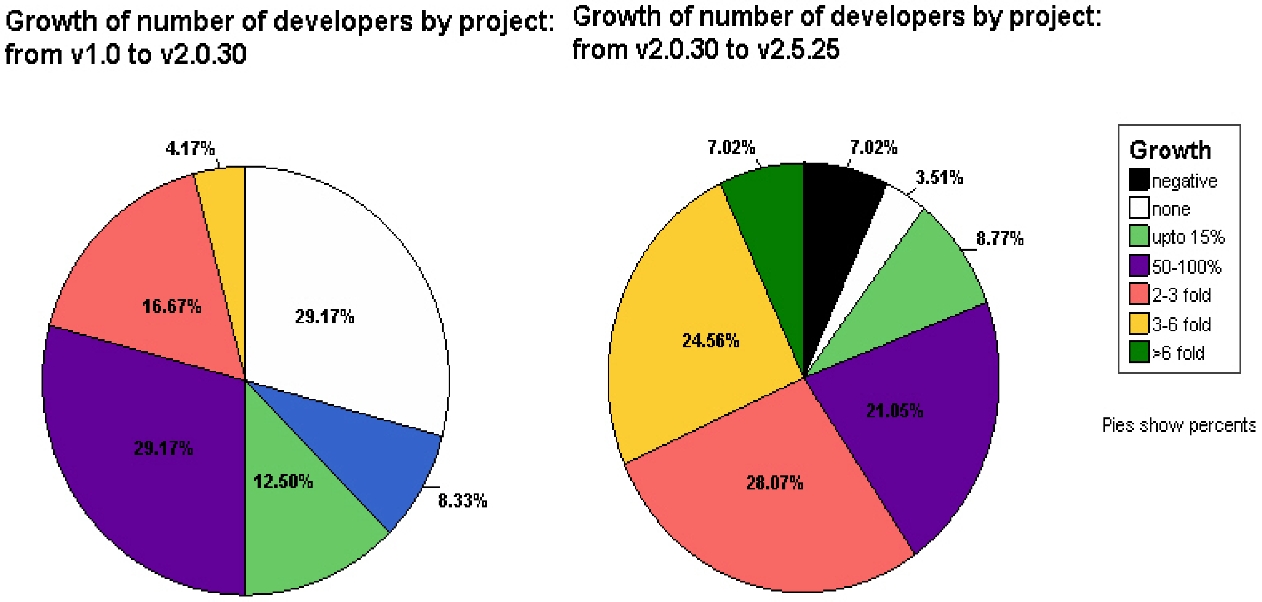
Figure 20: Unsigned code by package, Linux kernels v1.0, v2.0.30, v2.5.25



5.6. Growth in authorship of projects

As size and number of developers have been increasing from version to version by orders of magnitude – see figure 1 in section 3 – the number of developers for each package has also, naturally, increased rapidly. This increase is not uniform, though. Figure 21 shows the growth in number of developers for each project, between versions 1.0 and 2.0.30, and between 2.0.30 and 2.5.25.

Figure 21: Growth (in %) of number of developers per project, by package



What is interesting to note is that between version 1.0 and 2.0.30 a significant proportion of all packages (29%) saw no growth at all in the number of contributing developers. This is explained by the fact that the huge growth in size was more due to new modules and packages being added than to growth in existing modules. On the other hand, in the next change (to version 2.5.25) added a large number of developers across all packages, and the increase in size was also more evenly distributed across all packages (though there was, at the same time, the development of several new projects). The next change also shows a *negative* growth for 7% of packages, where the number of developers actually *reduced.* This is a small measure of the fact that code (and developers) are removed and replaced, not just added to, and this net/gross difference is not possible to resolve without looking at differences between versions with a much smaller time-interval gap (e.g. between 2.0.29 and 2.0.30).

6. Dependency information

As described previously[22], software is by nature collaborative in functioning and software packages usually depend on features and components from several other packages. Such dependencies must be explicitly detailed in a way that they can be determined automatically, in order for an application to run. In the LICKS project, these dependencies were identified in some detail, using the

[22] Ghosh 2002, section 6.3

function-definition identification method described in section 2. The scope of this analysis was very large: to illustrate, dependency analysis for the Linux kernel version 2.5.25 alone generated 600Mb of data, identifying over 5 million function dependencies for some 50,000 functions defined across more than 12,000 source code files (about 175 Mb of source code). This was then summarised to 8,328 dependencies between 178 projects.

There is an line of analysis possible in studying the correlation between dependency links and links based on, e.g. common authorship (found through the clustering analysis, not described in this paper). This would help understand whether the increasing communication between developers also leads to increasing interdependence between the code they produce.

One highlight of the LICKS dependency study, though, is provided in Figures 22-24. These plot the Lorenz curves for the number of packages supported by and depended upon, for each package.

Figure 22: Number of other packages supported by and depended on, Lorenz curve, v1,0

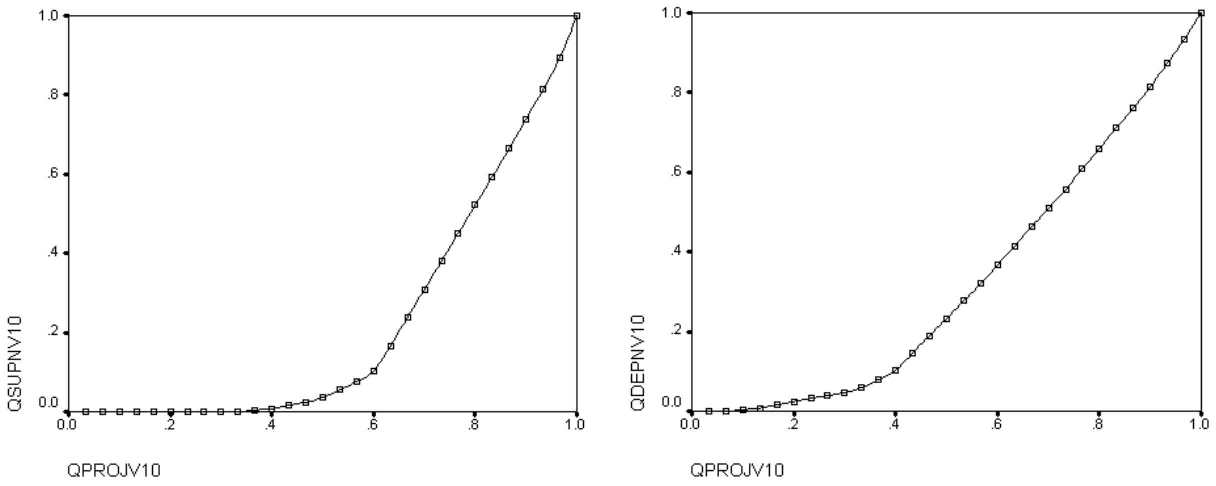
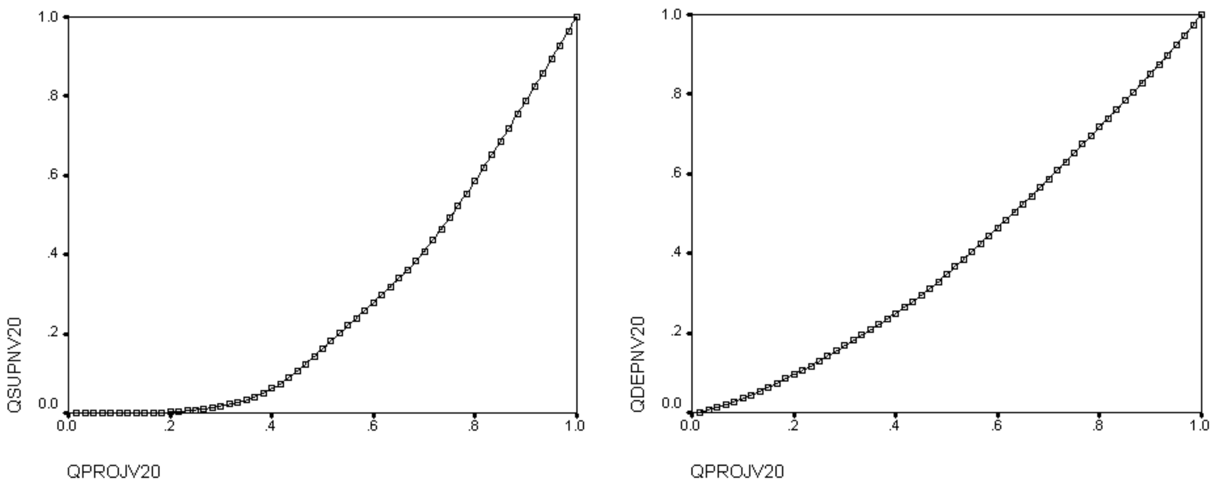


Figure 23: Number of other packages supported by and depended on, Lorenz curve, v2.0.30



The difference in concentration between supporting and depending packages is immediately apparent: support is much more concentrated, in that a few packages tend to account for most of the support functions. Put another way, most packages depend on a small subset of "highly supporting" packages. The distribution of dependence, in contrast, is almost uniform: all packages tend to depend on a more-or-less equal number of other packages.

Figure 24: Number of other packages supported by and depended on, Lorenz curve, v2.5.25

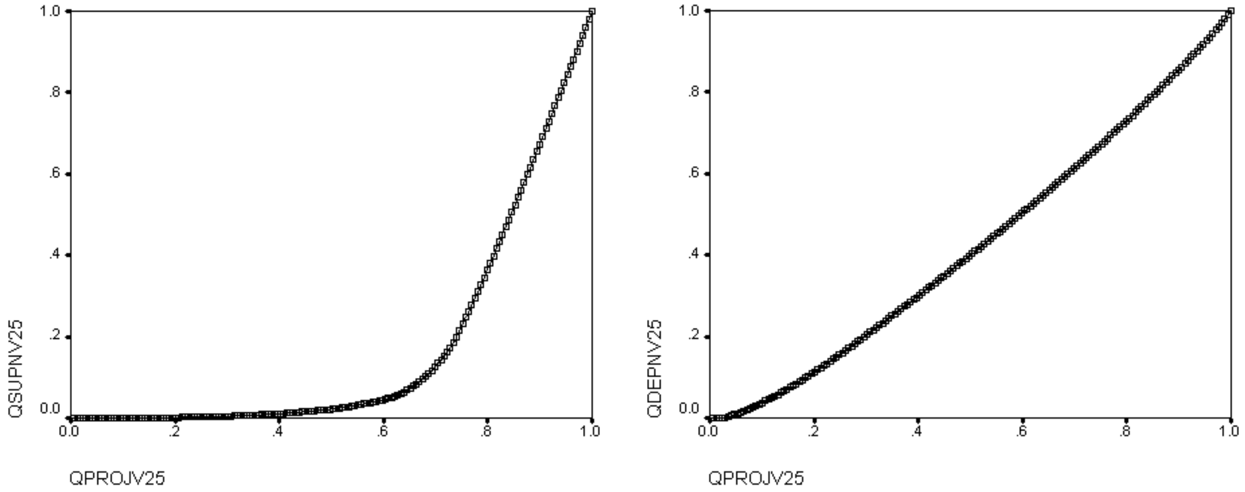


Figure 25 shows the Gini coefficients for support and dependence distribution across packages in the three versions of the Linux kernel. They show that the contrast between the high degree of concentration of support in packages and the low degree of concentration in dependence among packages is increasing over successive versions of the Linux kernel. In later versions, dependence becomes more evenly distributed across packages (more packages depend on a near-equal number of other packages) till by version 2.5.25 the Lorenz curve is almost flat.

Figure 25: Gini coefficients of support and dependence

Linux version	Support	Depend
1.0	0.56	0.35
2.0.30	0.44	0.20
2.5.25	0.66	0.15

7. Conclusions

This paper has provided an overview of the data extracted from the Linux kernel and initiated an analysis into attributes of the Linux kernel development process based on these data. This provides a validation of a set of methodologies developed in recent years[23] and demonstrates, hopefully, that there is a range of possibilities for understanding the F/OS development process at a previously

[23] Ghosh 2002

unavailable level of detail by using these tools to study the source code itself. The LICKS project has generated much more data and analysis, which is the subject of further papers.

8. References

Boehm, Barry W., *Software Engineering Economics*, Prentice Hall, 1981. More details and updates at: http://sunset.usc.edu/research/COCOMOII/

Dempsey, Bert J, Debra Weiss, Paul Jones, and Jane Greenberg, “A Quantitative Profile of a Community of Open Source Linux Developers,” Communications of the ACM. April, 2002. [http://www.ibiblio.org/osrt/develpro.html]

FLOSS: Free/Libre/Open Source Software Study, Rishab Ghosh, Ruediger Glott, Bernhard Krieger & Gregorio Robles, International Institute of Infonomics/MERIT, http://floss.infonomics.nl/report/

Robles, Gregorio et al, 2001, “WIDI: Who Is Doing It?”, http://widi.berlios.de/paper/study.html

Ghosh, Rishab Aiyer, and Vipul Ved Prakash, “Orbiten Free Software Survey”, First Monday, Vol 5, No. 7 – (July 2000) [available at http://www.firstmonday.org/issues/issue5_7/ghosh/]

Ghosh, Rishab Aiyer, “Clustering and Dependencies in Free/Open Source Software Development: a Methodology”, IDEI/CEPR Open Source Economics Workshop, Toulouse, June 2002 [available at: http://dxm.org/papers/toulouse2/]

Healy, Kieran and Alan Schussman, “The Ecology of Open Source Software Development”, January 2003 draft, http://opensource.mit.edu/papers/healyschussman.pdf

Krishnamurthy, Sandeep, “Cave or Community? An empirical examination of 100 mature open source projects”, First Monday vol 7, no. 6 (June 2002), http://www.firstmonday.org/issues/issue7_6/krishnamurthy/

Tuomi, Ilkka, "Evolution of the Linux Credits File: Methodological Challenges and Reference Data for Open Source Research” – working paper, 2002, available at http://www.jrc.es/~tuomiil/moreinfo.html

Wheeler, David, “More Than a Gigabuck: Estimating GNU/Linux's Size”, July 2001, http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html