UNIVERSITÉ DE MONTRÉAL

TITRE DE MON DOCUMENT

ALEXANDRE COUROUBLE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
NOVEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

TITRE DE MON DOCUMENT

présenté par: COUROUBLE Alexandre
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. NOM Prénom, Doct., président
Mme NOM Prénom, Ph. D., membre et directrice de recherche
M. NOM Prénom, Ph. D., membre

DEDICATION

To all friends at the lab,
I will miss you...

ACKNOWLEDGEMENTS

In the acknowledgements, the author points out the help that various people have provided, including advice or any other type of contribution as the author carried out their research. As appropriate, this is the section where the candidate must thank their thesis or dissertation supervisor, grant-awarding organizations, or companies that provided bursaries or research funds.

If the thesis or dissertation is in French and students wish to thank someone in particular in English, they must insert a second page and title it in English (i.e., “Acknowledgments”). If the thesis or dissertation is in English and students wish to thank someone in particular in French, they must insert a second page and title it in French (i.e., “Remerciements”).

RÉSUMÉ

The résumé (mandatory summary) is a brief explanation in French of the work's topic, its objectives, the research questions or hypotheses put forth, the experimental methods used, and the results analysis. It also includes the key research conclusions and future applications. In general, a summary does not exceed three pages.

The summary must provide an exact idea of the thesis or dissertation's content. It cannot be a simple enumeration of the manuscript's parts. The goal is to precisely and concisely present the nature and scope of the research. A summary must never include references or figures.

ABSTRACT

Written in English, the abstract is a brief summary similar to the previous section (Résumé). However, this section is not a word for word translation of the French.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

LOC Lines of Code
SCM Source Code Managment
OSS Open Source Software
MSR Mining Software Repositories

CHAPTER 1 INTRODUCTION

As an answer to the risks associated with employee turn over in software engineering organizations, researchers have attempted to establish models capable of finding experts in certain code areas. By recommending experienced developers, these models would ease the difficulty of replacing current experts as they leave the organization. Introduced in the early 2000s, these models were using two simple metrics to assess expertise: number of Lines of Code (LOC) and number of commits. However, using these metrics to measure expertise on *modern* large scale software project does not yield accurate results. we believe that, as the teams behind large software projects increase in size, these models tend to lose accuracy.

The Linux Kernel, for instance, is a 26-year-old project that has experienced a drastic growth in terms of code base and in terms of developer community. Wanting to keep a high standard of code quality and a fast release system, the community had to implement a hierarchical contribution system. In this contribution system, some developers are responsible for the maintenance of their subsystems. These developers, or maintainers, are trusted by the rest of the Linux community to be skilled enough to ensure the durability of their subsystem. This trust was acquired by contributing to the subsystem over the years. However, when trying to use traditional expertise determining metrics in the linux kernel, maintainers are rarely recomended.

We believe that previous expertise models fail to take into account some critical aspect of large scale software development. As subsystems become larger and welcome more contributors, maintainers have to dedicate more of their time reviewing other developers' contributions to maintain code quality. This shift towards a managerial position is usually followed by a drop in code contributions by that person. This implies that, in the eye of a model looking at only LOC and commits, the maintainer will lose expertise.

We introduce a new expertise model adapted to large software organization like the Linux community. Our model is better suited for the Linux community because it takes into account all the activities undertaken by both maintainers and developers. This thesis provides an in-depth description of the different steps necessary for the creation of this model.

1.1 Definitions and Concepts

1.1.1 The Anatomy of a Git Commit

A git commit is a fundamental concept in the scope of this research and for the understanding of git in general. The changes brought to source code by developers are contained in a *commit*. If a developer is tasked to fix a bug or to create a new feature in a project, she will have to modify the source code in order to implement these changes. When a developer feels ready to share these changes, she can apply them to the repository in the form of a git commit. The changes are reprented in the *commit diff*, which contains the exact lines to be removed (-) or added (+) in the source code. Git uses the +/- lines to modify the repository of someone who *pulled* the changes form the developer as depicted in Figure 1.1.

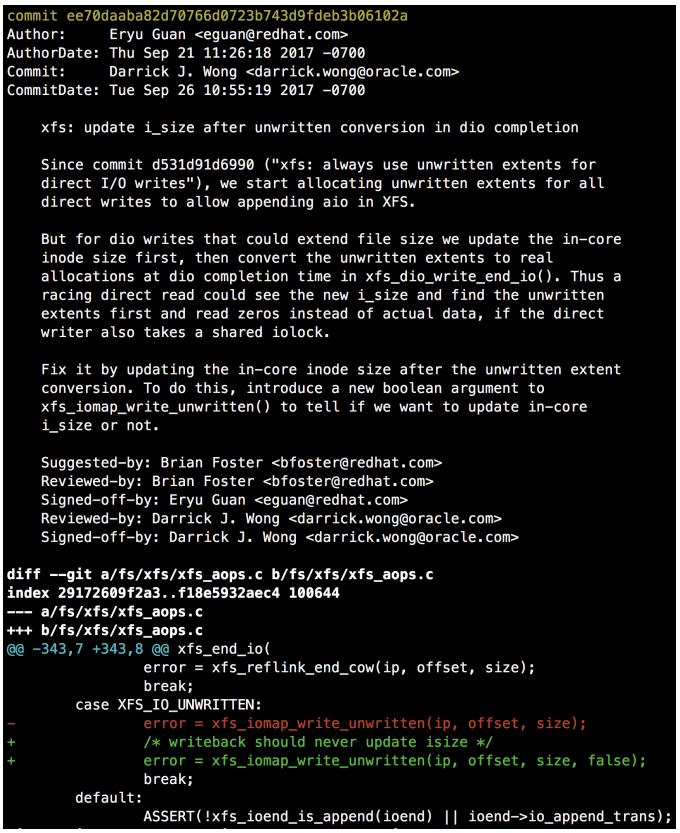
Furthermore, commits contain an array of metadata regarding the changes committed, all of which is accessible to anyone with a copy of the repository through a handful of builtin commands. For example, git log returns information about the past commits in the repository. Figure 1.1 shows one commit in the output printed by the following command: `git log --pretty=fuller --patch`, where the `--pretty=fuller` shows more information and *--patch* shows the commit diff. There are three main parts to the commit as seen in the image: the header, the message, and the diff.

The header contains the following data points:

- **Commit ID:** The "name" of the commit.
- **Commit Author:** Name and email address of the developer who *wrote*, or *authored* the code change.
- **Author Date:** Time, date, and timezone at which the changes were submitted.
- **Commit Committer:** Name and email address of the person who committed the code to the repository.
- **Commit Date:** Time, date, and timezone at which the commit was committed tot he repository.

In the scope of the Linux Kernel, the Commit Author rarely the Commit Committer. As explained in subsection 1.1.2, the author is the person who wrote the code, and then submitted it for review as a patch in an email. The commit committer is the person that recieved, accepted, and commited the changes to their repository.

The commit message contains the following datapoints:



```
commit ee70daaba82d70766d0723b743d9fdeb3b06102a
Author:     Eryu Guan <eguan@redhat.com>
AuthorDate: Thu Sep 21 11:26:18 2017 -0700
Commit:     Darrick J. Wong <darrick.wong@oracle.com>
CommitDate: Tue Sep 26 10:55:19 2017 -0700

    xfs: update i_size after unwritten conversion in dio completion

    Since commit d531d91d6990 ("xfs: always use unwritten extents for
    direct I/O writes"), we start allocating unwritten extents for all
    direct writes to allow appending aio in XFS.

    But for dio writes that could extend file size we update the in-core
    inode size first, then convert the unwritten extents to real
    allocations at dio completion time in xfs_dio_write_end_io(). Thus a
    racing direct read could see the new i_size and find the unwritten
    extents first and read zeros instead of actual data, if the direct
    writer also takes a shared iolock.

    Fix it by updating the in-core inode size after the unwritten extent
    conversion. To do this, introduce a new boolean argument to
    xfs_iomap_write_unwritten() to tell if we want to update in-core
    i_size or not.

    Suggested-by: Brian Foster <bfoster@redhat.com>
    Reviewed-by: Brian Foster <bfoster@redhat.com>
    Signed-off-by: Eryu Guan <eguan@redhat.com>
    Reviewed-by: Darrick J. Wong <darrick.wong@oracle.com>
    Signed-off-by: Darrick J. Wong <darrick.wong@oracle.com>

diff --git a/fs/xfs/xfs_aops.c b/fs/xfs/xfs_aops.c
index 29172609f2a3..f18e5932aec4 100644
--- a/fs/xfs/xfs_aops.c
+++ b/fs/xfs/xfs_aops.c
@@ -343,7 +343,8 @@ xfs_end_io(
                error = xfs_reflink_end_cow(ip, offset, size);
                break;
        case XFS_IO_UNWRITTEN:
-               error = xfs_iomap_write_unwritten(ip, offset, size);
+               /* writeback should never update isize */
+               error = xfs_iomap_write_unwritten(ip, offset, size, false);
                break;
        default:
                ASSERT(!xfs_ioend_is_append(ioend) || ioend->io_append_trans);
```

Figure 1.1 The anatomy of a commit

- **Commit summary:** Often called the commit title, a brief explaination of the purpose of the commit.
- **Commit Message:** In depth explaination of the purpose of the commit.
- **Credit Attribution Tags:** List of people who were involved in the commit and the nature of their involvment.

There are many different types of credit atribution tags, each describing the way the person contributed to the commit. The most common ones, and the ones we use in this study are: *Signed-off-by*, *Reviewed-by*, and *Acked-by* (acknoledged by). According to *[Alex: cite https://www.kernel.org/doc/html/v4.12/process/submitting-patches.html]* the tags have the following meanings. *Signed-off-by* indicates the developer assisted in the creation of the patch or that she committed it upstream. *Acked-by* is used by developers who were not involved in the creation of the patch but wanted to record their approval. *Reviewed-by* is used to credit developers who contributed reviews to the submitted patch.

The commit diff, which sits at the end of the git log output, shows the exact files and lines

that were modified by the author of the commit. Git uses the commit diffs to apply the changes to the files in the repository. The diff can be percieved as the set of instructions to transform the source code into the desired state.

1.1.2 The Linux Contribution Process

Altough Linux uses Git as its version control system, most developer cannot commit directly to the repository. In fact, the developers and maintainers use several different repositories in the development process. The main repository (also called the main tree, of the main line) is maintained by Linus Torvald, the creator of Linux. For a developer to have their code changes integrated into an official release of the Linux Kernel, their changes must be submitted and accepted by Linus Torvalds, as he has the last word on any code being added to the main tree.

Submitting code changes *upstream*, means to submit changes hoping they will be *merged* into the main tree, and thus into an official Linux release. To achieve this, there is a set of guidelines to follow, as the Linux Kernel follows a strict development cadence and has high code quality standards. First, the code submitted must follow the Linux coding style *[Alex: cite coding style https://01.org/linuxgraphics/gfx-docs/drm/process/coding-style.html]*. It is important to impose a strict coding style in projects of the scale of the linux kernel. Code comming from tens of thousands of developers would become very hard to maintain is everyone cubmitted code with their own coding style.

Secondly, developers must follow the submission process guidelines. In the vast majority of cases, developers submit their changes to a maintainer. A maintainer is in charge of maintaining a specifi subsystem. A subsystem can be represented as a series of files that work together to serve a certain purpose. When a developer is unsure of who to send the changes to, they can consult the `MAINTAINERS` file, or use the script `get_maintainer.pl` to discover persone responsible for a certain file. The developer must send her changes in the form of a patch the modified subsystem's mailing list.

Naturally, maintainers must review the patches to ensure that they are worthy of being integrated into their subsytem tree. These reviews usally occur in the email thread following the email patch. The patch author will probably have to improve their code changes according to the maintainer's reviews. If maintainers are satisfied with a patch, they will *commit* it to their repository. As explained in subsection 1.1.1, for each commit, git differenciates between the **commit author** and the **commit committer**.

Before submitting changes to Linus, maintainers usually send the changes acquired in their

repository to *linux-next*. They may do so through an email as a patch, although submitting large changes is easier throught Git. Linux-next is where the integration testing occurs. This is where developers and maintainers make sure new code changes do not interfere with the rest of the code base and do not introduce any bugs. Linus will pull commits that have been in the -next tree for a few weeks *[Alex: cite: https://01.org/linuxgraphics/gfx-docs/drm/process/howto.htmlbecoming-a-kernel-developer]*. Linux-next ensures important bugs introduced by new commits are discovered and dealt with before being committed to the main tree, as these bugs would drastically slow down the release cycle.

The main tree uses a specific releace cycle. After a new version is releases, 4.13 for instance, the *merge window* opens for release 4.14. This two-week long merge window is the only opportunity to submit new code changes in hope to have them integrated in release 4.14. Linus pulls most of the changes from linux-next during that time period because these changes are les likely to cause bugs and thus delay the next release. After two weeks, the merge window closes and linux 4.14-rc1 is released. Then, developers work on ensuring that the kernel is as stable as possible. For approximately 6 to 8 weeks, linus will only accept patches that address bugs introduced by commits merged during the merge window. The only exception is new drivers. New drivers can be submitted outside of the merge window because bugs introduced by drivers are self contained, and only affect users using the specific driver. A new -rc comes out about every week, until Linus declares that the kernel is stable enough to be released.

Furthermore, large subsytems impose their own release schedule to developers. For instance, the Net subsystem also has two main trees: `net` and `net-next`. Net-next recieves all changes submitted to the net subsystem. Once the linus' merge window opens, net-next closes, and all the changes accumulated over the last 10 weeks will be submitted to the main tree. At this point, the net tree will recieve all the fixed related to commits that were committed to the main tree. Once Linus' merge window closes, net-next reopens and developers are free to submit new patches again.

As stated above, the contribution process relies heavily on emails. The reason why the linux community does not use tools such as github or gerrit is that those tools would not scale to the size of the linux community (Armstrong et al., 2017a). And even though the email-based system has been very reliable over the lifetime of the project, there is one drawback. Once a patch is sent to a maintainer, there usually is a discussion occuring on the mailing list. This discussion contains many precious information such as design decisions and code reviews. However, once the patch is committed to the subsytem tree, and after the commit makes its way upstream to the main tree, there is no easy way to recover the discussion that took place

in the mailing list. We address this issue with a new tool created during this masters.

1.2 Hypothesis of Thesis

Our main research objective is to create a model capable of detecting experts in a certain code area. section 2.2 describes previous expertise models created over the years. We believe these previous models are not well suited for the linux community. We based our model on a series of new metrics, which were acquired through different techniques. We were able to make some of these metrcis available to the community through two open source projects, which are described in Chapter 3.

Hypothesis: Expertise models targeting large software engineering organizations require a look at all development activities, including code reviews and upstream committing.

1.3 Plan of the Thesis

The structure of this thesis is as follows. We conduct a litterature review in chapter 2, where we describe previous expertise models and other topics of interest. Chapter 3 gives a detailed overview of the general workflow of the thesis. Chapter 4 gives an indepth look at Email2git, one of the opensourcve project we created. Finally, chapter 5 includes the article we submitted to the International Conference on Software Analysis, Evolution and Reengineering (SANER).

CHAPTER 2 CRITICAL LITERATURE REVIEW

Our research project touches upon two areas of software engineering research: *mining software repositories* and *software engineering expertise*. One could argue that the field of mining software repositories enabled the creation of expertise models, as these models are based on metrics only available through the use of mining software repositories techniques. Besides these two research fields, we relied on another type of literature to conduct this research project: Git and Linux documentation. This chapter provides a critical literature review of the our two academic research areas and a description of the information available in the Linux and Git documentation, as well as how it helped us finding solutions for the problems encoutered.

2.1 Mining Software Repositories

To answer the difficulty associated with collaboration in large software projects, tools were created to ease collaboration between team members were created. In addition to providing a contribution platform, Source Code Managment (SCM) systems track and save large amounts of information about each changes brought to the source code. During the lifetime of the project, the SCM acquires a large amount of data about the development of the project. Mining software repositories researchers *mine* this data for their research projects.

Software repositories are not limited to SCM. There are other entities present in software repositories that research mine to gather information about software projects. These entities include bug tracking systems, mailing lists, source code, and issue tracking systems. Over the years, researchers have used mining software reporistories techniques that enabled them to research different topics of software engineering. *[Alex: cite daniel's git paper]*

In the scope of our research, we used mining software repositories techniques in each different part of the project. The data used for both open source project created during this project came from mining the Linux Kernel repository. We eventually used this data for the creation of our expertise model.

One of the difficulty often encourntered by researchers in mining software repositories is the inability to link data coming from different entities of the software repository. In the case of the linux kernel, the dificulty was to link data from the mailing lists to the data from the git repository. A dificulty we addressed by creating Email2git.

2.2 Software Engineering Expertise

Many different studies explored the concept of expertise in software engineering. the authors of two early studies, Expertise Browser (Mockus and Herbsleb, 2002) and Expertise Recommender(McDonald and Ackerman, 2000), expressed the importance of understanding developers' expertise level. McDonald et al. approach the topic from a problem solving percpective. Today's developers have many different resources at their disposal for the purpose of problem solving. Stack Overflow[1], a programming question answer exchange, is used by developers from around the world that are lookiing for solutions for complex problems. Before Stack Overflow's creation in 2008, developers did not have easy access to a large database of solutions provided by *experts* from various topics. Stack Overflow based its business on a common practice of probem solving among developers: *asking an expert*. The authors of the Expertise Recommender were seeking to solve this problem by providing an architecture capable of recommending experts for given parts of the software project, for the sake of problem solving. The authors in (Mockus and Herbsleb, 2002) approach the issue differently. They provide and expertise model to solve the issue of replacing or adding new expert to a distributed software engineering project. They argue that the tool would reduce the time lost by engineers attempting to find a new expert for their team.

Each of these previous studies (Bhattacharya et al., 2014), (Mockus and Herbsleb, 2002), (McDonald and Ackerman, 2000), and (Fritz et al., 2007) base their measures of expertise among developers on the tacit assumption that experience is acquired through development activities, such as number of lines of code contributes or the number of commits authored. The author in (Fritz et al., 2007) examined the reliability of this assumption. Through a review of the many studies in psychology on knowledge and expertise, (Fritz et al., 2007) disovered that there was no sufficient evidence that activity does determine one's knowledge. The authors conducted a quantitative study on 19 java programmer to assess the accuracy of these finding. With this survey, they discovered that multiple activity related heuristics influenced developers' knowledge and expertise. These heurstics include authorship, role, work experience, and activity, which confirms the suitablility of the metrics used in previous work (LOC and commits).

Globally, we found that the previous studies on the topic fail to address several important activities present in software development mentioned in (Fritz et al., 2007). The authors emphasized the importance of the notion of history in determining knowledge. This is an issue we address in our model by including a historic dimention. Furthermore, (Fritz et al., 2007) mention the effect of code stability to code knolwedge. The code stability aspect can

[1] `https://stackoverflow.com/`

be addressed by modifying our proposed historical expertise formula.

2.3 Open Source Participation

Previous work studies developers' motivation in Open Source Software (OSS). In (Wu et al., 2007), warn that the losely organized nature of OSS development could be associated with a high turnover rate and in unexpected departures. Other work (Rigby et al., 2016) studies the impact of a high turn over on the organization. The authors argue that departing developers bring the amount of knowledge they acquired during their time as a contributor. We believe that this implies that OSS projects are at risk of knoledge loss and we believe that accurate expertise modeling could assist in addressing this issue.

CHAPTER 3 RESEARCH GENERAL WORKFLOW

As stated in chapter 1, the goal of our research project is to create an expertise model based on a series of metrics mined from the Linux Git repository and various mailing lists. Our expertise model takes into account many different activities present in the linux contribution process. These activities translate into metrics as we attempt to quantify them. To give back to the linux community, we made those metrics available through two open source tools, as the metrics are complicated to generate.

In this chapter, we describe the general workflow of the research. section 3.1 and section 3.2 introduce the tools we created, providing an explaination of how the metrics will assist the creation of our expertise model. section 3.5 describes the impact that the two tools had on our research project and how they led to the creation of our expertise model, whose aproach and evaluation were submitted as a paper to the IEEE International Conference on Software Analysis, Evolution and Reengineering[1].

3.1 Srcmap

In the interest of offering more visibility to the authors of the Linux Kernel, we built a data visualization tool capable of displaying a wide array of information about directories or files found in the linux git repository. We wanted to display the following data points about each file and directory of the source code:

- LOC
- Median age of the LOC within a file/directory
- Number of lines of code modified since 2016
- A list of the 20 developers with the most lines of code
- A bar plot displaying the distribution of line of code age

We needed an interface that would allow the user to navigate the different files and direcotries of Linux while displaying our list of datapoints, which is why we chose to base the tool on a treemap. Treemaps, which were introduced by (Bederson et al., 2002) as a solution to display large hierarchical dataset on a 2 dimentional plane, were a great fit for our tool's requierments.

[1] `http://saner.unimol.it/`

3.1.1 Srcmap 1.0

In the first version of Srcmap[2], we used the Google Chart treemap implementation[3]. This easy to use library allowed us to create a quick proof of concept.

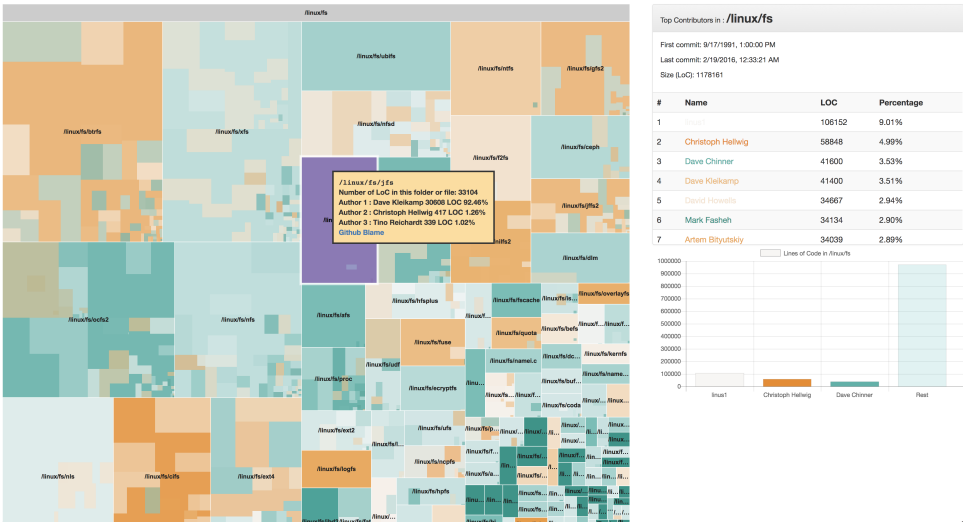


Figure 3.1 First version of srcmap

Figure 3.1 shows the first version of Srcmap. The diferent boxes represent subdirectories of the Linux Kernel. The different colors present within each box give a preview of the content of the box. In this version of the tool, the color represents the developer having contributed the most lines of code in the contained files. Furthermore, the size of the boxes is proportional to the number of lines of code existing within the file or directory represented by the box. The panel on the right of the screen and the tooltip contain most of the data: exact number of lines of code, age of the first and last lines of code to be added, and the top 20 authors and their percentage of lines of code contributed.

3.1.2 Srcmap 2.0

After some research, we discovered a new treemap implementation[4] capable of handling large dataset and deeply nested stuctures, which we used for the second version of the tool[5]. This new version, shown in Figure 3.2, introduced three important features:

- Coloring the files and directories according to three metrics:

[2] `http://mcis.polymtl.ca/~courouble/linux.html`
[3] `https://developers.google.com/chart/interactive/docs/gallery/treemap`
[4] `https://carrotsearch.com/foamtree/`
[5] `http://mcis.polymtl.ca/~courouble/dev/`

  - LOC
  - Median age
  - Number of commits since 2016, or "Hot files"

- File search
- A plot displaying the *age* distribution of LOC present in the file/directory.

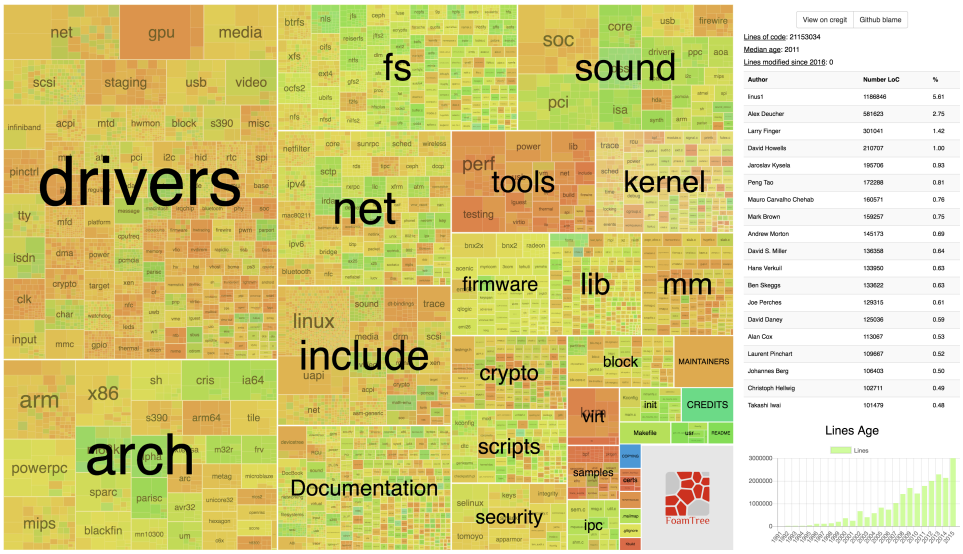


Figure 3.2 Second version of srcmap

Although the foamtree library had a steeper learning curve than the Google Charts Treemap library, the versatility provided by the foamtree library allow us to provide a much more pleasant user experience and easier access to the data.

3.1.3 Scalability

The very first Srcmap prototype consisted of a treemap displaying only the `net` subdirectory, which was small enough to provide a smooth user experience. However, Scalability issues started to arise when we attempted displaying the entire Linux Kernel source code. The data would take up to 30 seconds to load, and navigating between each node became very slow.

Furthermore, we discovered the limitations of Google Charts when we tried adding new features to the tool. The API provided did not allow the level of customization we desired.

This is why we decided to create a new version of the tool with a different treemap library. The new library, Foamtree, provided a richer API and allowed for smooth browsing through deeply nested tree.

3.1.4 Community Engagment

After the creation of the first implementation of Srcmap, we attended LinuxCon 2016 to meet with a series of Linux developers and maintainers. The goal of these informal interviews was to recieve early feedback on the tool. In addition to that, I traveled to Santa Fe, New Mexico to present Srcmap and Cregit at the Linux Plumbers Conference. After discussing srcmap and our research to a series of linux developers, we deducted two points.

Firstly, experienced linux developers and maintainers are accustomed to their own development workflow. According to our interviews, experienced developers have acquired *muscle memory* from developing in the same terminal over the years. Moreover, experienced maintainers were not interested in a web-based visualization tool.

Secondly, it became evident that there was real interest for another aspect of linux development: linking Linux git commits to email patches and code reviews.

Access to the original patches and code reviews would save a lot of time

[Alex: more]

3.2 Email2git

The linux contribution process has been a reliable way to pipe code contributions (patches) from developers around the world, to the main Linux repository. With a working copy of the Linux Kernel on their computers, developer can modify the source code and, if desired, submit their changes for review, in hope to integrate the main tree. These changes are submitted as patches, files that contain the lines that are to be removed and the lines that are to be added. To submit a patch, developer send the patch file(s) to a linux maintainer in an *email* with a description of the changes brought by the patch. If accepted, the maintainer will *commit* the changes to his local git repository, and submit the changes *upstream* to another maintainer.

Although this system has been very reliable, it has one major drawback: once committed, it is impossible to easily find the email conversation that eventually led to the creation of the patch. We addressed this drawback by creating an algorithm capable of backtracking the origin of commits in the Linux Git repository. The algorithm and the issues related to scalability are described in chapter 4.

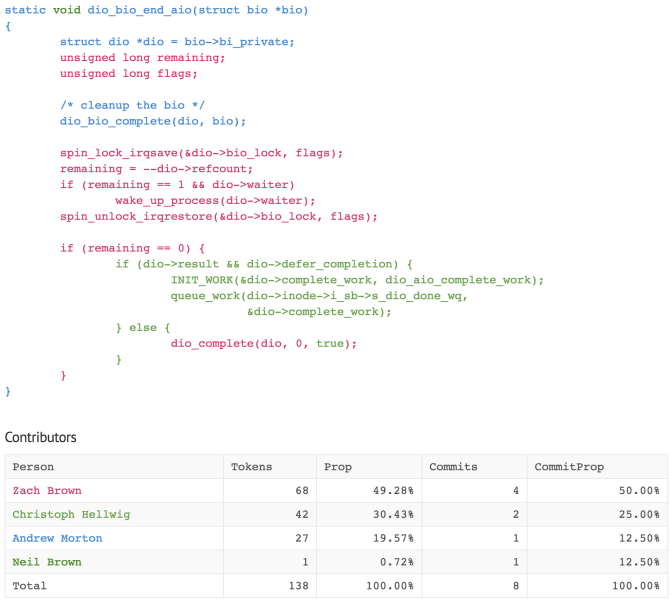
The data generated by the algorithm consists of a list of commit to patch matches. The matches are accessible online through two interfaces: as a commit ID search through the

Email2git interface[6], or though the Cregit interface[7].

3.3 Integrating Email2git with Cregit

Cregit is a project that aim at providing a finer grain approach to *git blame*. The blame option in git returns the name of the developer who last changed a line of code in the source code. It provides a great way to quickly unmask the developers responsible for code in the Source Code. However, it has a serious limitation: git blame assigns a line to a developer even after a small modification to that line. For instance, if developer A writes `print "Hello world"`, this line will then become associated with developer A. However, if developer B modifies the line to read `print "Hello world!"`, git blame will associate the line with devloper B even though developer B only added a character.

Cregit addresses this limitation by tokenizing the source code in a git repository to enables git blame at a token level, instead of a line level. This provides a better understanding of the true authors of the source code. A tokensize version of the Linux kernel source code is available online through the cregit interface[8].



```
static void dio_bio_end_aio(struct bio *bio)
{
        struct dio *dio = bio->bi_private;
        unsigned long remaining;
        unsigned long flags;

        /* cleanup the bio */
        dio_bio_complete(dio, bio);

        spin_lock_irqsave(&dio->bio_lock, flags);
        remaining = --dio->refcount;
        if (remaining == 1 && dio->waiter)
                wake_up_process(dio->waiter);
        spin_unlock_irqrestore(&dio->bio_lock, flags);

        if (remaining == 0) {
                if (dio->result && dio->defer_completion) {
                        INIT_WORK(&dio->complete_work, dio_aio_complete_work);
                        queue_work(dio->inode->i_sb->s_dio_done_wq,
                                   &dio->complete_work);
                } else {
                        dio_complete(dio, 0, true);
                }
        }
}
```

Contributors

Person	Tokens	Prop	Commits	CommitProp
Zach Brown	68	49.28%	4	50.00%
Christoph Hellwig	42	30.43%	2	25.00%
Andrew Morton	27	19.57%	1	12.50%
Neil Brown	1	0.72%	1	12.50%
Total	138	100.00%	8	100.00%

Figure 3.3 Tokenized source code as it appears on Cregit.

Figure 3.3 shows tokenized linux code as it appears on the Cregit interface. In an effort to ease the access to email2git data, we decided to provide access to the matches through cregit.

[6] `http://mcis.polymtl.ca/~courouble/email2git/`
[7] `https://cregit.linuxsources.org/`
[8] `https://cregit.linuxsources.org/`

To this end, I modified the user interface to display a window containing all the available patches after clicking on a token, as shown in Figure 3.4.

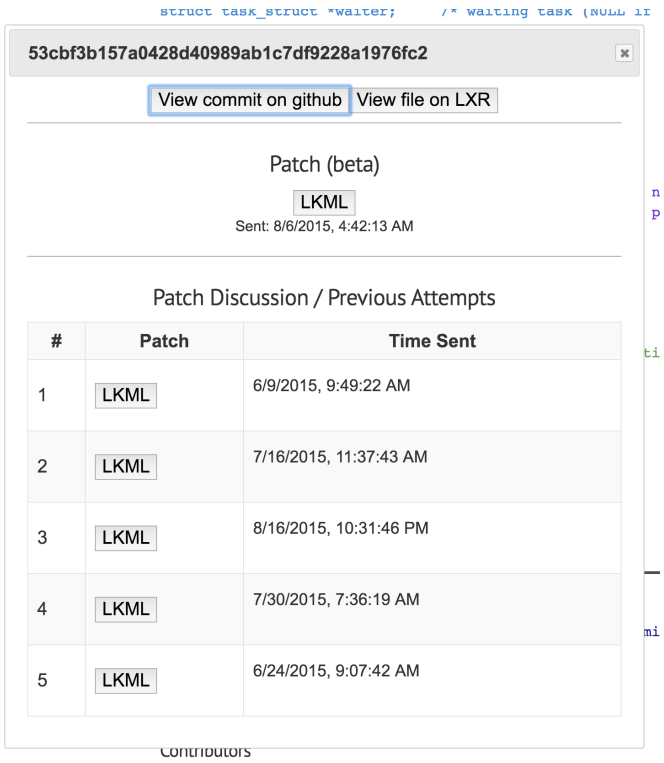


Figure 3.4 Window containing the patches that introduced the commit associated with the clicked token

3.4 Lessons Learned

We learned many important lesson during the creation of Srcmap. From a technical point of view, we learned the importance of selecting a suitable library that will be capable of responding to our demands and requirements. Finding the right tool from the inception of the project would have saved us from weeks of work. From a community perspective, we now understand the importance of understanding the needs and the habits of the targetted community.

We leveraged the lessons learned from Srcmap, while creating Email2git. The purpose of Email2git was to answer a complain coming from multiple developers and maintainers: the difficulty of finding an email discussion about patches that were eventually integratted in the Linux Kernel. Understanding the Linux community allowed us to better introduce our tool when we released it, as explained in Chapter 4.

3.5 Expertise Model

During the creation of srcmap, we discovered that maintainers' contribution frquecies were decreasing overtime. This resulted in their *LOC footprint* to slowly erode because of the contributions coming from other developers. Wondering whether htis decrease in footprint was the result of maintainers reconsidering their involment with the Linux community, we analysed other metrics. Starting with the data collected for the creation of Email2git, we took a look at the other aspects of development. We discovered that as the Linux community becomes larger, maintainers are spending and increasing amount of time review code changes submitted by other developers. With this data available, we were able to improve state of the art expertise model. Chapter 5 presents the paper in which we introduce and evaluate our expertise model.

3.6 Recommendations

[Alex: Recommendations and lessons learned in one section? I feel like I would repeat of lot of talking points.]

CHAPTER 4 EMAIL2GIT: FROM ACADEMIC RESEARCH TO OPEN-SOURCE SOFTWARE

As explained in chapter 1, the linux contribution process is a strong email-based system that has proven to be more reliable and scalable over the years. Like in many other organizations, the linux development community makes extensive use of code reviews to ensure the quality of contributors' code submissions. These code reviews occur in the mailing lists, where the patch was first introduced. After accepting and integrating the patch to the git repository, it is usually very hard to recover the original patch, the code reviews, and the discussion that took place during the code reviews. Other tools have addressed this issue by providing a code review environment and by keeping track of the code review for each commit. *[Alex: examples gerrit, github]* However, the linux community never used these tool for upstream contributions as they would not scale to the size of the linux community. We address this flaw in the development process by creating Email2git, a tool able to find, for a given linux commit, the orignal patches and the code reviews. This chapter introduce Email2git, from its inception, to its deployement to production.

4.1 Previous Publications and Original Algorithm

The original algorithm capable of backtracking patches from commits was introduced in two papers (Jiang et al., 2013, 2014) published by Jiang, a former member of the MCIS Lab. Originally written in Perl, the script was a great proof of concept. The general idea of the script was to compare the +/- lines from both the git commits and the email patches. A match was found if the proportion of identical +/- lines was above a certain threshold. Although this script was a great proof of concept, it had difficulties scalling to 8 years of emails and commits.

4.2 Scalling the Algorithm

Because we wanted Email2git to be a usable and practical tool, we needed a way to display the patches and the code reviews in a browser. Fortunatly, a great existing open-source tool called **Patchwork**[1] perfectly answers our requirements. Patchwork is a tool designed to assist maintainers of open source projects using an email-based contribution process. It tracks the mailing lists used by developers to submit patches and recieve code reviews. The

[1] `https://github.com/getpatchwork/patchwork`

tool extracts each detected patch as well as its associated reviews, then displays them in a web-based user interface.

We were granted read access to the MySQL database behind a patchwork instance hosted on kernel.org[2]. This instance has been tracking 69 of the many linux subsystems mailing lists since 2009, giving us the oportunity to analyse over *1.4 million* patches.

In addition to being a great data source, patchwork.kernel.org is also a great way for us to display the patches and the code reviews associated with commits to the users. The only limitation of this patchwork instance is that it does not track some major mailing lists, particularly some of the `Net` mailing lists.

Since we had access to email patches dating back to 2009, we decided to extract git commits from the Linux git repository from the same date, which represent over *500,000 commits* to analyse. Unfortunatly, this amount of data was too large for the orignal algorithm to parse in a timely fashion, which called for a new, scalable algorithm that leverages heuristics mentioned in (Jiang et al., 2013, 2014).

4.2.1 Patch Email Subject

The most important heuristic that drastically increased the matching speed is the *email subject - commit summary* concept. The built-in git features `git format-patch` and `git send-email` allows developers to easily submit their changes to a maintainer by email according to the Linux Kernel Contribution guidelines[3]. This or these emails contain all the meta-data that will eventually be included in the commit, if the patch is accepted. The meta-data includes heuristics such as: time sent, author, commit message, ... If the patch is accepted, the maintainer can use another git command to integrate the patch into her repository: `git am`[4]. This command will automatically exctract the patch info and keep the relevant information in the commit. The piece of information we are interested in is the email subject. `Git am` automatically saves the email subject and uses it the "commit summary". This commit summary, or commit title, is the first line of the commit message (((TODO: refer to intro for git commit stuff))). Comparing both strings of characters allows for a very quick first phase of matching.

This first phase will find a match for about 55% of the commits. After this step, we can remove the commits and the patches that were matches from the "search space", reducing the amount of patches and commits to be parsed, reducing the load on the algorithm.

[2] `https://patchwork.kernel.org/`
[3] `https://kernelnewbies.org/FirstKernelPatch`
[4] `https://git-scm.com/docs/git-am`

4.2.2 Author and Affected Files

Even though the number of commits was reduced by half after the first phase, I was unable to make the old script fast enough to parse the rest of the data in a timely fashion. Thus, I had to find a way to use the available meta-data to speed up the matching. The first data point I used was the *author name*. As depicted in Figure 4.1*[Alex: TODO: Improve schema][Alex: Not sure how technical I should get]*, I am able to use the name and email address of the *commit author* to pin point to the patches that were sent by the same person. In other words, to find a match for each commit, the algorithm has to parse a handfull of patches instead of hundreds of thousand. Similarly, the files affected by a commit and a patch can drasticaly help the performance of the +/- line algorithm. Through some regular expression and text parsing, we can retrieve the files that are modified in the patch and the *commit diff*. Since the author-based matching is slighty faster and returns more matches than the file-based matching, we start with the former, removing each matched commit and patch to reduce the workload of the next phase.

In this phase as well as the next one, the matches are found using the same +/- lines as the original algorithm, which is a fairly slow process, although this process could be improved using a hashing-based method to find corresponding +/- lines for further performance improvement.

4.3 The Data

There are two sides to this matching process: the Linux git repository and the archives containing the patches sent in mailing lists over the years. We need to extract the diff (+/- lines), the metadata, and the subject and commit summary from both side. The scripts used for each part of the data extraction are available on the project's github under the GPL-3.0 license [5].

4.3.1 The Commits

First, we need to extract the commit summary from each commit after 2009. This date is our lower bound because our email data from patchwork.kernel.org only has email patches dating back to 2009. The commit summary is the first line of the commit message, which makes it very easy to retrieve. The script `subject_data_gen/commit_subject_generator.py` reads a git log output and stores the commit summary for each commit in a SQLite3 database. The exact git log command used is the following:

[5] `https://github.com/alexcourouble/email2git`

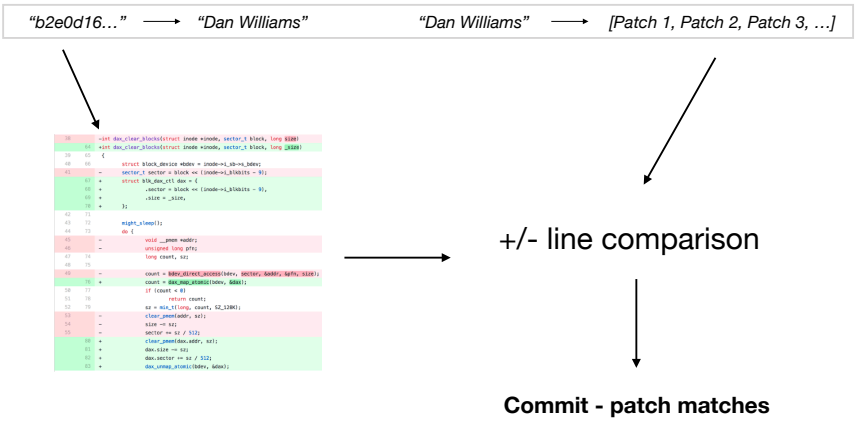


Figure 4.1 Using the patch sender to assist matching

```
git log --no-merges --pretty=format:"%H,%ct,%s" --after={2009-01-01}
```

The `pretty` option formats the output according to the passed parameters.

The next step on this side of the data is to extract the data for the other phases of the matching process. The script `lines_data_prep/git_prep.py` is more complicated, as there is more data to parse and save. This script reads the authors and the files affected by each commit. It will then create two maps: commit ID to author, and commit ID to files affected. These maps, which exist as python dictionaries are then saved to two separate pickle files[6], which make writting, reading, and storing data a fast and easy. This script also extracts the +/- lines from the commit diff and stores them in a pickle file as well.

4.3.2 The Patches

The patches are stored on a remote serve in a MySQL database, the same database that hosts the patchwork.kernel.org data. Through the help of SQL queries, I dumped all the necessary data in csv files to avoid complications arising from handling a production database. Once those csv files created, I could parse them with the help of two python scripts available in

[6] `https://docs.python.org/2/library/pickle.html`

the Email2git githubg repository. `subject_data_gen/patchwork/pwSubjectFull.py` takes care of the subject data and `lines_data_prep/pw_prep.py` takes care of the authors, file names, and +/- lines of the patches. Here again, the subject data is stored in a SQLite3 database, and the line data is stored in pickle files.

4.4 Providing Access to the Matches

Email2git's original inteded contribution was to increase the amount of information existing around a commit by providing access to the conversation that took place during the creation of the patch. And now that matches have been generated and saved, we need a way to make the information available to linux developers. Each match is composed of three elements: the *commit ID*, the *patchwork permalink ID*, the *date*, and the *phase* that found the match (subject, author, or file). The patchwork permalink ID is used to point to the patch and conversation on patchwork.kernel.org.

As discussed in chapter 3, the matches are available through two platforms: cregit and as a standalone commit lookup page. And although both platforms use the same UI and fetching mechanism to display the links to patchwork, the user experience is fundamentally different. On cregit, users navigate the interface by browsing the tokenized files. Once the user clicks on a token, we display a window containing the links to the patches and converstation that introduced that token to the source code. Note that in this case, the user need not to know the commit ID of of the token of interest. The commit ID, which is necessary to retrieve the patches, is hard-coded in the html element containing the token. The HTML element containing a token looks like this:

```
<a onclick="return
windowpopLinux('2ebda74fd6c9d3fc3b9f0234fc519795e23025a5')">
        include
</a>
```

The onclick event calls a function defined in a global javascript file: `cregit.js`. In the original implementation of the cregit interface, this function would open a new browser window and show the commit associated by the token on github. So I modifed `cregit.js` to disable the "popup mechanism" and to instead use the commit ID to fetch the patchwork permalinks IDs from the server. The matches are stored as csv files named after the commit ID they are associated with on the server hosting the interface. The assyncronous requested is done throught Papaparse[7], a powerful opensource javascript library capable of downloading

[7] `http://papaparse.com/`

and parsing csv files from the client. The javascript code that generate the URLs from the permalinks and displays the new window lives in a callback function that executes after the request is complete. We were able to keep the "view commit on github" feature, by showing a button in the new window.

On the standalone commit ID lookup page, the mechanism is almost identical, but the user experience is completely different. Instead of clicking on a token, the user knows the commit ID in advance, as they might have encountered it while trying to fix a bug, or read a `git log` output. The user copies and pastes the commit ID in the search bar, and the match window appears with a list of dated links to patchwork. The lookup page verifies whether the commit ID is a SHA-1 hash with the following regex:

```
// removing white space
cid = cid.replace(/\s/g, '');

// validating input
if (!/\b[0-9a-f]{40,40}\b/.test(cid)){
    window.alert("The input should be a full 40-character SHA1 hash.");
    return;
}
```

We are running analytics on the server hosting the matches to understand the usage of the data and to know the proportion of requested commit IDs are not matched. Removing the whitespace and assuring the validity of the string ensures the accuracy of these statistics, by recuding the number of failed requests due to a poorly formated commit ID.

Figure 4.2 displays the plots created by the analytics script running on the server. We observe two peaks in number of unique IP addresses at two different moments: at the end of July and in mid September. The former date corresponds to the day we introduced Email2git in a blogpost on linux.com and the latter corresponds to the talk I gave at the Open Source Summit North America in Los Angeles.

4.5 Introducing Email2git to the Opensource Community

We undertook various efforts to make our work more visible to the linux and open source community in general. The first effort was a blog post published on linux.com[8]. This blog post discusses email2git and its integration with cregit. This blog post was shared on Facebook and Twiter by the Linux Foundation and by other developers, which helped spreading the

[8] `https://www.linux.com/blog/email2git-matching-linux-code-its-mailing-list-discussions`

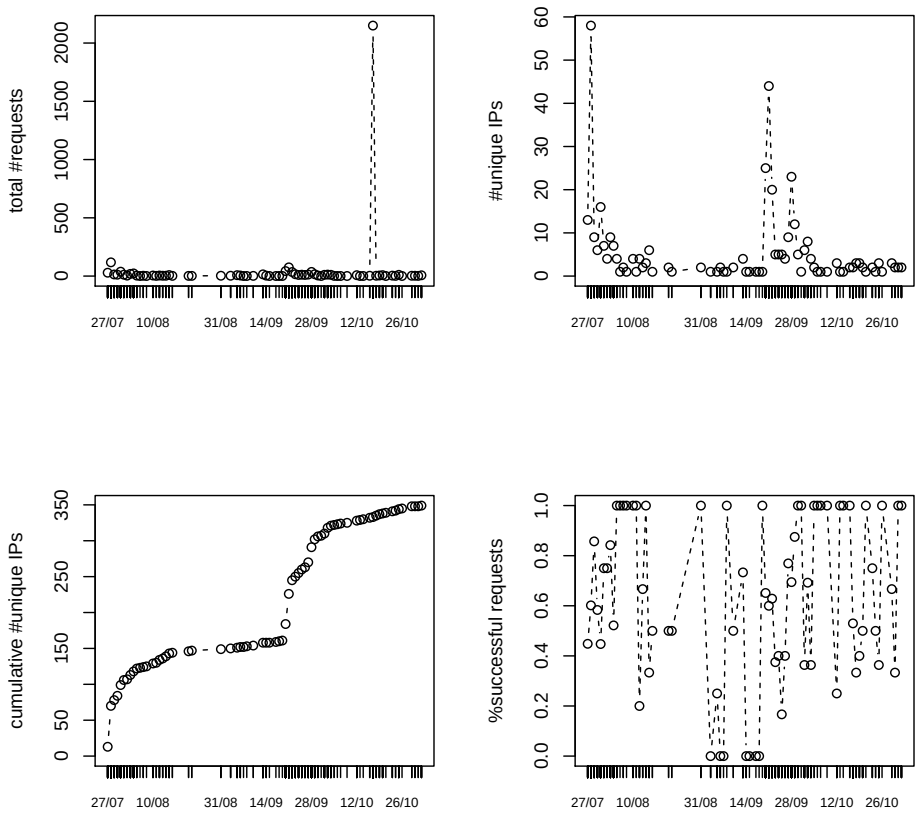


Figure 4.2 Plots created from the analytic

word about our work. In addition to this blogpost, I gave a refereed talk at the Open Source Summit and the Linux Plumbers Conference in Los Angeles. This gave me the oportunity to give a demo, explain the underlying algorithm and finally discuss the project with developer and recieve crucial feedback. An article was published on LWN.net following my talk. It explained the algorithm, the chalenges faced, and mentioned some of the questions the were asked during the talk[9].

[Alex: Should I nclude the entire blogpost? As an annex?]

[9] `https://lwn.net/Articles/734018/`

CHAPTER 5 ARTICLE: ON HISTORY-AWARE MULTI-ACTIVITY EXPERTISE MODELS

5.1 Abstract

As software evolves, a developer's contributions will gradually vanish as they are being replaced by other developers' code, slowly eroding the developer's footprint in the software project. Even though this developer's knowledge of the file did not disappear overnight, to outsiders, the developer and her expertise have become invisible. Through an empirical study on 5 years of Linux development history, this paper analyses this phenomenon of expertise erosion by building a 2-dimensional model of developer expertise involving a range of developer activities and involving activity data on more than one release. Using these models, we found that although many Linux maintainers' own coding footprint has regressed over time, their expertise is perpetuated through involvement in other development activities such as patch reviews and committing upstream on behalf of other developers. Considering such activities over time further improves the expertise models.

5.2 Introduction

As reported by Damien et al. (Joseph et al., 2007), employee turnover is a major challenge of information technology organizations. Estimations of the cost of losing an employee amount to between 0.5 and 1.5 times her salary, with the cost of replacing a software engineer in particular exceeding $100,000 (eco, 2000). These costs are not limited to the software engineer's company, but also spread to open source development. In their 2017 Linux kernel report, Corbet et al. (Corbet and Kroah-Hartman, 2017) noted that "well over 85 percent of all kernel development is demonstrably done by developers who are being paid for their work". In fact, only 8.2% of all kernel commits were made by volunteer developers. Hence, developer turn-over in companies risks to impact open source development as well!

Apart from improving the working conditions and onboarding procedures, software organizations (both closed and open source) need to invest time in finding the "right" expert to replace a parting developer. While it is possible to train newcomers and bring them up to speed (e.g., one third of the kernel contributors in the last 12 months were newcomers, and 60% of those made their contribution as employee), the term "right" refers to having a similar profile, allowing the new expert to seamlessly fit in and continue his or her predecessor's work, without significant loss of knowledge. Thanks to the widespread adoption of

agile development and open source development models, software development has become a collaborative endeavor, in which knowledge is shared across the members of an organization, hence in principle it should be possible to find contributors with similar profiles.

Unfortunately, there is no consensus on how to measure the profile of a developer, and how to determine whether such a profile indicates the developer to be an expert. The simplest way to measure someone's development activities is to count the number of code changes (e.g., Git commits) authored. This is for example how Corbet et al. determine the most active developers and organizations in Linux kernel development (Corbet and Kroah-Hartman, 2017). Yet, at the same time, they note that "The total number of patches signed off by Linus Torvalds (207, or 0.3 percent of the total) continues its long-term decline. That reflects the increasing amount of delegation to subsystem maintainers who do the bulk of the patch review and merging." Worse, developer rankings based on the number of commits differed substantially based on the period during which this metric was measured (e.g., ranking based on last 10 years vs. last year). At a minimum, one needs to be careful interpreting these and other measures such as a developer's code legacy as shown by "git blame" (Bhattacharya et al., 2014; Mockus and Herbsleb, 2002; McDonald and Ackerman, 2000; Fritz et al., 2007).

To make developer expertise measures more robust and reliable, this paper proposes a 2-dimensional developer expertise footprint, addressing two important issues with current expertise models. First of all, while the amount of code written by a person can be an important indicator of expertise, it does not take into consideration the actions of people who do not directly contribute source code, such as those who review it or discuss it on mailing lists or issue repositories. Our footprint measure combines indicators of multiple kinds of developer activities.

Second, as indicated by the Corbet et al., current measures focus on a given software release or development period, basically ignoring the development activities that happened before. While a person who wrote 50% of the code changes of the previous release could be less of an expert than a person who wrote 50% of the code changes of the current release, the former developer might have been ill or absent, or might have been the one mentoring the latter developer. As such, both developers should be considered as experts, not just the latter developer. Our footprint measure allows to consider a developer's activities across different time periods.

We empirically evaluate the expertise footprint models on 5 years of Linux kernel development history, addressing the following research questions:

RQ1) *How does expertise evolve in an open source project?*

Almost 1 out of 4 subsystems has seen a change in maintainership during the last 22 Linux kernel releases, with the code footprint of maintainers gradually decreasing over time.

RQ2) *How well does dimension 1 explain expertise?*

Models involving a maintainers' own code footprint and coordination activities (committing and/or reviewing) perform the best.

RQ3) *How well does dimension 2 explain expertise?*

Expertise models considering the last R releases perform better than single-release models.

5.3 Background and Related Work

Maintainers ensure the longevity of the Linux kernel by not only contributing new code, but also by reviewing and integrating code submitted to their subsystems by other developers. They are the backbone of Linux kernel development, yet few studies have been done to study their work (Zhou et al., 2017).

In particular, a maintainer's departure of her subsystem calls for her immediate replacement. However, only a developer with extensive experience in the subsystem can take on the task of maintainer. Software expertise and knowledge have been extensively studied in the past (Bhattacharya et al., 2014; Mockus and Herbsleb, 2002; McDonald and Ackerman, 2000; Fritz et al., 2007). Many different models were created to attempt to assess developer expertise.

Earlier expertise models (McDonald and Ackerman, 2000; Mockus and Herbsleb, 2002) made the assumption that expertise is related to coding activities. They both extracted data describing the changes in the source code from the version control system. In other words, they measured a developer's expertise in terms of the number of changes made to the system.

Fritz et al. (Fritz et al., 2007) later expressed their concern about the assumption that activity indicates expertise. Their review of psychology studies indicated a lack of evidence proving that activity can be an indicator of knowledge. However, after a qualitative study consisting of 19 java developers interviews, they were able to confirm a relationship between commit frequency and expertise. In addition to that, they foud evidence proving that authorship (as

obtained from the amount of churn contributed, or through "git blame") is also capable of indicating expertise.

Bhattacharya et al. (Bhattacharya et al., 2014) explored the suitability of different expertise indicators depending on the developer's role. They argue that state-of-the-art metrics (lines of code and commits added), being unaware of the developer's role, can lead to inaccurate results. They add that code activity metrics like the number of lines of code added, only describe expertise at a local level and poorly capture global expertise.

Even though the goal of each study mentioned above varies, they all introduced expertise detection models relying on two simple metrics to measure expertise: the number of lines of code contributed and/or the number of commits.

Although these state-of-the-art techniques are well suited to detect experts among regular developers, we believe that they do not properly evaluate more complex expertise, such as that of subsystem maintainers. Most of the daily tasks, such as reviewing, of maintainers are ignored, creating an inherent bias in the expertise models.

We address this bias by building expertise models based on a variety of metrics capturing the full breadth of software development activities, and also considering a longer the evolution of such activities over time.

5.4 Measuring the Expertise Footprint of Contributors

This section discusses the two-dimensional model of expertise footprint proposed by this paper to enable identification of experienced team members (e.g., developers, testers, etc.). The first dimension of the footprint model considers a wide range of activities performed by a project member, not only focusing on code changes, but also code review or even a developer's code "legacy" (i.e., contributed code that still survives in the code base). The second dimension enhances the first dimension by not only considering the range of activities in the latest release, but *across the last N releases*. As such, accidental lulls or shifts in project activity are accounted for.

Note that the expertise we are interested in is expertise about the *internals* of a particular source code file or component. An alternative form of expertise would consider knowledge on how to *use* a particular component (API). We focus on the former kind of expertise, since it is at the heart of a software organizations needs. For example, it allows to measure the expertise of a particular individual, allowing the organization to better use her skills, evaluate her value to the organization, and assess the risk of her potential departure. Furthermore, it is important for an organization to know—for any section of the system—who are its

experts, and their level of expertise. Finally, in both cases, it is also important to know how this expertise is changing over time (e.g., the areas where a person is gaining and losing expertise, and, for a given area, how it is losing or gaining experts).

5.4.1 Dimension 1: Contributor Activities

The expertise footprint model that we propose explicitly considers a wide range of development activities instead of focusing only on review- or code-related activities. Table 5.1 provides a non-exhaustive list of activities, from very technical to outreach activities. Any activity by a contributor to one of these, can increase (or at least maintain) the contributor's knowledge about the subsystem she is working in. The more measures are considered, the more comprehensive the expertise footprint model becomes, hence the better the expected performance for identification of experts in a project under study, provided the activities are weighted based on their relevance for a given project.

This flexibility comes at the expense of additional effort for mining these activity measures. Fortunately, when developers contribute code to an open or closed source project, data about each code change, code review or other activity is automatically stored in the project's software repositories. The most trivial example are the code changes (commits) recorded in a version control system like git. However, information about the contributor's activity in issue report discussions is also readily available from the project's issue repository (e.g., bugzilla or jira), code review activity from the review repository (e.g., gerrit or mailing list) and mailing list activity from the mailing list archive. Of the metrics in Table 5.1, `represented` and `planned` are the hardest to obtain data for.

Given a set of activity measures $\mathbb{A} = \{a_i | a_i \; is \; activity \; measure\}$, we compute the expertise footprint of a release j as:

$$footprint_j(\mathbb{A}) = \sum_i \frac{w_i \times a_i}{a_i^{tot}}$$

, where w_i is a weight factor given to a_i ($\sum_i w_i = 1$) and a_i^{tot} is the total number of activities of a given type (e.g., number of source code lines, commits or reviews) recorded for a given activity and release. In other words, each activity is normalized, and the weighted sum of the normalized activities yields the $footprint_j(\mathbb{A})$ percentage. Hence, to instantiate the generic $footprint_j(\mathbb{A})$ measure, an organization first has to select the activity measures $\mathbb{A}$ relevant to its context, then determine the relative weight w_i of each selected activity.

It is necessary to normalize each activity's measure to provide a better understanding of the true impact of developers' contributions in the subsystem. This is because the studied subsystems differ in size and the heuristic counts are inherently uneven by nature. For

activity	definition
`legacy`	influential code contributed by C that still survives in R_i
`authored`	code authored by C since R_{i-1}
`committed`	code committed by C since R_{i-1}
`reviewed`	code changes reviewed by C since R_{i-1}
`translated`	involvement in translating/localizing textual strings for R_i
`integrated`	effort spent by C integrating code changes since R_{i-1}
`discussed`	effort spent by C discussing issue reports since R_{i-1}
`represented`	effort spent by C representing S on social media since R_{i-1}
`planned`	effort spent by C planning R_i

Table 5.1 Non-exhaustive list of activity measures that can be measured for a particular contributor C of a specific subsystem S in a given release R_i.

example, the value for `legacy` (in number of lines of code) will likely be much larger than the values of `authored` or `reviewed` (in number of commits).

5.4.2 Dimension 2: Historical Perspective

While the definition of $footprint_j(\mathbb{A})$ takes into account a wide range of activities, it only considers a contributor's activity for one specific release j. As such, this measure might still provide misleading information when used to find the most appropriate expert for a given subsystem (e.g., to help debug a coding issue).

First of all, contributors in both closed and open source development evolve according to a particular career path. Even in open source, many contributors start out translating textual strings, before contributing smaller code fixes and ever larger changes until they are trusted enough to be able to review or even accept other contributors' code. This not only implies that a contributor's volume of contributions is scattered across different activities, but also that this scattering (and volume) might change over time. Hence, depending on the release under study, different $footprint_j(\mathbb{A})$ values are obtained, as if a specific contributor suddenly would have "lost" or "gained" a substantial percentage of expertise (footprint). To counter this noise, one should incorporate past experience to obtain a more robust footprint model.

Second, even when a contributor's responsibilities are stable across a time period, accidental life events such as illness or busy periods at work, or project events such as the scope of the upcoming release (major release vs. bug fix release) could lead to increases or decreases for certain activities. Again, if the contributor was an expert in the previous release, she will not have lost all of this expertise in one release due to illness. Hence, a release-specific

$footprint_j(\mathbb{A})$ measure again would yield the wrong impression.

For this reason, the second dimension of our footprint model explicitly takes into account history by taking the weighted sum of $footprint_j(\mathbb{A})$ over the last R releases. In particular:

$$footprint_j^R(\mathbb{A}) = \sum_{i=j}^{j-R} W_i \times footprint_i(\mathbb{A})$$

, where W_i is a weight factor given to the specific footprint of release i ($\sum_i W_i = 1$). Note that $footprint_j(\mathbb{A}) = footprint_j^0(\mathbb{A})$, i.e., the footprint model obtained based on the first dimension is a special case of the second dimension ($R = 0$).

While the choice of weights w_i for dimension 1 could be chosen arbitrarily based on relevance of individual activities, the weights W_i typically will be decreasing, since recent activity typically is at least as important as older activity. For example, the weights could be linearly decaying (e.g., $[0.33, 0.27, 0.20, 0.13, 0.07]$ for $R = 4$), giving each older release proportionally less influence on the footprint model. Alternatively, an exponential ($[0.64, 0.23, 0.09, 0.03, 0.01]$) or logarithmic ($[0.34, 0.29, 0.23, 0.14, 0.00]$) decay could be used to give older release less or more influence, or (less likely) even a uniform ($[0.20, 0.20, 0.20, 0.20, 0.20]$) decay to give all considered releases the same importance.

5.4.3 Use Cases for Expertise Footprint

Given the footprint models $footprint_j(\mathbb{A})$ and $footprint_j^R(\mathbb{A})$, a number of use cases can be imagined.

The main use case considered in this paper is the identification of experts in a software project. Newcomers to a project typically are not aware who to ask for advice when encountering a specific technical issue. Similarly, when the maintainer of a specific component or library decides to retire, finding a good replacement is not always straightforward for an organization, as important development knowledge (across a range of development activities) risks to be lost.

A less straightforward application was suggested at one of the 2017 OPNFV Summit's panels, where substantial attention went to the issue of non-responsive Linux kernel maintainers. These are experts responsible for a given subsystem who, due to personal events, loss of interest or other reasons, start becoming non-responsive in communication with other developers or management. Having a reliable expertise measure in place would enable monitoring over time of maintainers' activities to spot long-term periods with sub-par performance. Such pro-active detection of issues could also suggest alternative maintainers.

Similarly, an expertise footprint can help an organization guard itself against accidental loss of manpower. For example, the bus factor (Mens et al., 2014) is a known measure of the risk that key personnel might disappear from a project, either out of free will or due to an accident. Organizations with a high bus factor could leverage an expertise footprint to identify backups for key developers or managers. As such, for each subsystem, an organization could have a list of the main people working in it as well as their expertise level.

In order to use the footprint models to find the most appropriate expert of a given subsystem, one needs to calculate $footprint_j(\mathbb{A})$ and/or $footprint_j^R(\mathbb{A})$ for each person who contributed at least once to one of the activities in $\mathbb{A}$. Then, the resulting footprint values should be ranked from high to low. Ideally, the contributor with the highest footprint value is recommended as first candidate expert, followed by the contributor with the next highest footprint value, etc.

5.5 Case Study Setup

This section presents the design of an empirical study on the Linux kernel to evaluate the 2-dimensional footprint model introduced in the previous section. The study addresses the following research questions:

RQ1: How does expertise evolve in an open source project?

RQ2: How well does dimension 1 explain expertise?

RQ3: How well does dimension 2 explain expertise?

5.5.1 Subject Data

Our study evaluates the expertise footprint models in the context of the Linux kernel. First of all, the Linux kernel is one of the hallmark open source projects, with a long history, large code base and vast supply of contributors. Second, the kernel is one of the few open source projects in which expertise is documented explicitly. The code base contains a file named `MAINTAINERS` that lists, for each subsystem, the experts in charge. Just as for source code, changes in maintainership are recorded through regular commits. This provides us with a unique oracle for our expertise measures.

Furthermore, the Linux Foundation (who governs and mentors the development of the Linux kernel and related open source initiatives) recently has started up the CHAOSS committee

on Community Health Analytics for Open Source Software[1]. Amongst others, the aim of this committee is to identify explicit measures of expertise that can help prospective adopters of open source projects in choosing the right developer or maintainer to contact. As such, our study can help this concrete initiative, and we are in contact with the CHAOSS consortium.

Determining expertise footprints in the Linux kernel, especially taking into account the second dimension of our measure, requires a large set of historical data. We conducted our analysis on a set of 27 releases of the Linux kernel, spanning releases *v3.5* to *v4.11*, which corresponds to approximately 5 years of development and release history.

5.5.2 Filtering of the Data

Because of the constantly changing nature of the kernel, new subsystems are being added to the Linux kernel in every release to meet the demands associated with new hardware and changes in user expectations. Furthermore, it is not uncommon to see a subsystem disappearing, or, more precisely, becoming obsolete or orphaned (lin, v4.11, `MAINTAINERS`, Line 84).

On the other hand, the importance of the historical aspect of our analysis forces us to choose long-standing subsystems that would best reflect the evolution of expertise of the subsystems' maintainers. Hence, we filter our Linux kernel data set to keep only subsystems that existed throughout the studied timespan. This subset reduces the number of subsystems from 1,662 to 734 subsystems.

For RQ2 and RQ3, we need further filtering to ensure a data set of subsystems for which there is ongoing activity in each studied release. To achieve this, we parsed, for each release, the `MAINTAINERS` file to extract each *active* subsystem along with its name, list of maintainer names, and the list of files and directories belonging to that subsystem.

We then retrieved the list of commits made to each subsystem, for each release that we considered. This allows us to compute, for each subsystem, the average number of commits across its releases. After matching each commit to its code reviews (see below), we also compute the average proportion of matched commits per release.

We then set minimum thresholds of 50 commits per release and 60% matched commits per release. This filtering reduces the 734 subsystems to a set of 78 subsystems for RQ2 and RQ3. This subset contains well know subsystems like `ARM PORT`, `XEN HYPERVISOR INTERFACE`, `SOUND`, `SCHEDULER`, and `CRYPTO API`.

[1] `https://chaoss.community/`

5.5.3 Instantiation of the Footprint Models

Table 5.2 shows the five concrete activity measures considered in our empirical study on the Linux kernel. These measures cover influential source code contributed (`legacy`), the volume of code changes since the last release (`authored` and `committed`), and code review activities since the last release (`attributed` and `reviewed`).

Our $footprint_j(\mathbb{A})$ and $footprint_j^R(\mathbb{A})$ models are calculated based on the above metrics, using $w_i = 0.20$ as weights for dimension 1 and a linear decay with $R = 4$ (i.e., $W_i \in [0.33, 0.27, 0.20, 0.13, 0.07]$) for dimension 2. A more judicious choice of w_i and/or W_i could improve the results in RQ2 and RQ3, however our empirical study aims to provide a lower bound on the expected performance.

5.5.4 Git-related Activity Measures

To calculate the Git-related activity measures of Table 5.2, i.e., all measures excluding `reviewed`, we cloned the official Linux kernel git repository, then checked out the Git tag corresponding to each analyzed release.

Bread-and-butter analysis of the Git log commits in the time span since the previous official release yields `authored` and `committed`, while simple regular expressions of the commit messages in the same logs obtains `attributed`. Finally, a standard git blame command yields, for each code line in the release under analysis, the last person touching it. This information allows to calculate `legacy`.

In kernel development, tags like "Signed-off-by", "Reviewed-by" and "Acked-by" are used as "a loose indication of review, so [...] need to be regarded as approximations only" (Corbet and Kroah-Hartman, 2017). Despite the warning of Corbet et al., `attributed` information is straightforward to obtain from commit messages, which is why we included this measure to complement the more strictly defined `reviewed` measure (calculated from reviewing data).

The next step is to lift up each contributor's Git-related activity measures to the subsystem-level, leveraging the file path information for each subsystem in the `MAINTAINERS` file. To do this, we identify for each commit the changed files, then map the commit to the subsystem(s) to which these files belong and aggregate the file-level measures to the subsystem level, for each contributor.

activity	source	definition
`legacy`	git blame	#lines of code contributed by C that still survive in R_i
`authored`	git log	#commits authored by C since R_{i-1}
`committed`	git log	#commits committed by C since R_{i-1}
`attributed`	git log	#commits since R_{i-1} for which C is credited in the commit message under “
`reviewed`	mailing list	#commits since R_{i-1} for which C has written at least one code review email

Table 5.2 Concrete activity measures used for our empirical study on the Linux kernel. Each activity is measured for a particular contributor C of a specific subsystem S in a given release R_i.

5.5.5 Linking Commits to Review Emails

In contrast to the developer `attributed` data obtained from the Git repository, the `reviewed` metric considers a second repository, i.e., the review environment. For the Linux kernel, code reviews are performed through mailing list discussions (Armstrong et al., 2017b; Jiang et al., 2013, 2014). Patches are sent to one or more of the various linux mailing lists (typically one per kernel subsystem), where anyone can step up and provide review comments simply by replying to the patch email. As such, the review comments of a specific patch are spread across one or more email threads.

Jiang et al. (Jiang et al., 2013, 2014) have introduced a number of heuristics to link an accepted patch stored as a commit in the official Git repository to the (different versions of the) reviewed patch in the mailing list. We adopted the best performing heuristic of Jiang et al., which uses simple set intersection of the changed code lines between each email patch and Git commit. The heuristic matches a given Git commit C to the email patch P with which the change code line intersection is the largest and exceeds a given threshold of 4%. All emails in P's email thread are said to correspond to the review comments on P (and hence C).

To improve the line-based heuristic of Jiang et al., we have combined it with other heuristics. First of all, we observed that more and more kernel developers are using the commit message summary as the subject of their email threads. This summary is recommended to be between 50 and 72 characters[2] and appears before the body of a commit message. Hence, before applying the line-based matching of Jiang et al., we first check if there is a unique email patch P with subject identical to a commit C's commit summary. If so, we consider P and C to be a match, and do not need to run the more complex line-based matching algorithm.

[2] `https://medium.com/@preslavrachev/what-s-with-the-50-72-rule-8a906f61f09c`

If there is no such P, or multiple patches P have been found, we extract for each remaining commit the *author* and the *changed files*. We do the same for each remaining email patch. This information is then used to narrow down the search space of the line-based matching, by trying to match a commit only to email patches authored by the same developer and/or touching the same files. This substantially speeds up the matching process.

The remaining commits, i.e., the commits still not matched to a review, introduce noise to our measures. The reason for not finding a code review could be due to the reviews being sent to a mailing list that we did not analyze, or not being reviewed at all. We were granted read access to the database behind the Patchwork mailing list archive hosted by linux.org[3]. This Patchwork instance has been tracking 69 different Linux mailing lists since 2009, providing us with about 1.4 million patches. However, patches that were submitted through untracked mailing lists are not in our dataset, which explains the variability of matched commits across subsystems. Alternatively, the code change in the accepted commit could also have undergone substantial changes compared to the reviewed commit, for example due to rebasing, cherry-picking or squashing (Bird et al., 2009).

Figure 5.1 shows the total percentage of matched commits from 2009 to the time of writing this paper, across the largest subdirectories of the kernel. It shows that this percentage varies greatly among the different subdirectories, with a minimum of 25% for the "net" subdirectory. This is why, in subsection 5.5.2, we filter out those subsystems whose average percentage of matched commits across the studied releases is lower than 60%. Note that we do not show the percentage of unmatched *patches* (only unmatched *commits*), since the unmatched email patches include those patches that were rejected during code review, and hence never showed up in the Git repository.

5.6 Case Study Results

This section discusses for each research question its motivation, specific approach and results.

RQ1. How does expertise evolve in an open source project?

Motivation:

Open source software maintainers are responsible for the health of their subsystem. For example, each Linux kernel maintainer manages the changes proposed by developers to the subsystem they are responsible for, and shepherd those changes upstream towards the Git

[3] `https://patchwork.kernel.org/`

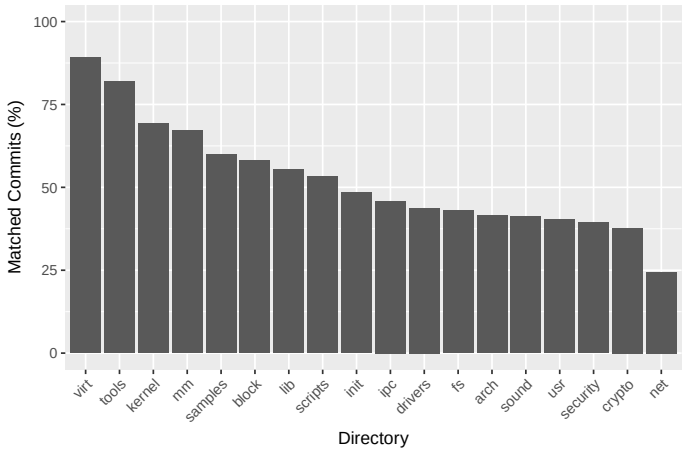


Figure 5.1 Percentage of matched commits in Linux subdirectories, from 2009 to the time of writing this paper.

repository of Linus Torvalds (i.e., the official Linux kernel repository). Hence, their presence is vital to the kernel community.

Unfortunately, due to the unpredictable nature of life in general and open source software development in particular (Wu et al., 2007; Zhou and Mockus, 2015), maintainers, for various reasons, one day will be forced to give up their responsibilities. In most cases, this means that another developer will have to take over the responsibility of maintainer.

Hence, this research question aims to analyze how often maintainership changes in kernel development. Furthermore, we are interested in understanding how much of the code base of official releases is "owned" by the subsystem maintainers, i.e., was originally developed by a maintainer. Since "git blame" is a popular means for finding experts (Rahman and Devanbu, 2011), our results will help us understand to what extent such a measure is reliable to measure expertise.

Approach:

To confirm the presence of changes in maintainership during the evolution of the Linux kernel, we analyzed the maintainers recorded in the `MAINTAINERS` file of releases *v3.5* to *v4.11* to identify how often maintainers (dis)appeared. Furthermore, for each studied release, we measure and plot these maintainers' `legacy`, which corresponds to the number of surviving code lines of a maintainer, as given by "git blame". We then validated the statistical significance of the change in `legacy` distribution between the first and last analyzed release using a Wilcoxon paired test. In case of a significant test result, we also provide the Cliff Delta effect size ((Cite Romano stats paper)). An effect size smaller than 0.147 is deemed a "negligible"

difference, smaller than 0.33 a "small" difference, smaller than 0.474 a "medium" difference and otherwise a "large" difference.

Results:

23% of the studied subsystems saw changes in maintainership over the last 5 years. Out of the 734 subsystems studied for RQ1, we counted 168 subsystems that experienced some sort of maintainership change. We counted 100 maintainer arrivals, 63 departures, and 88 replacements. These numbers confirm that maintainership change is common, even in mature open source systems like the Linux kernel. Furthermore, the median percentage of developers who are maintainers in the analyzed subsystems is 0.50% (mean of 0.90%), indicating that it is not straightforward to guess the next maintainer. These observations strengthen our case for more advanced expertise measures.

The median maintainer `legacy` significantly decreases over time. Figure 5.2 shows the evolution of the median percentage of maintainer `legacy` across all subsystems in each studied release. The plot shows a clear, steady decrease of this measure across releases in terms of median and variance. We confirm the significance of this decrease with a Wilcoxon paired test ($\alpha = 0.01$) between the first and last studied version, which yields a p-value of 2.2e-16.

Although the Cliff Delta value of 0.07 indicates only a negligible difference, this decreasing trend suggests that, if one limits the measure of expertise to the amount of surviving code originally authored by a maintainer, as was done by earlier work (Rahman and Devanbu, 2011), the expertise of maintainers globally seems to be decreasing over time. The next two research questions evaluate the use of a wider range of activity measures, across a range of releases, to obtain a more accurate measure of expertise.

RQ2. How well does dimension 1 explain expertise?

Motivation:

Prior work on expertise measures (Anvik et al., 2006; Bhattacharya et al., 2014; McDonald and Ackerman, 2000; Minto and Murphy, 2007; Mockus and Herbsleb, 2002) primarily are based on code activity, which can be defined in terms of `legacy` and `committed`. As motivated in subsection 5.4.3, we believe that these two metrics do not capture the full breadth of contributor expertise activities. Indeed, the results in RQ1 indicate that the `legacy` of long-standing maintainers crumbles over time. Unless one assumes that this reflects a real drop in expertise over time, the only explanation is that existing experts reorient their focus to other activities, such as code review and email communication. Hence, this research question

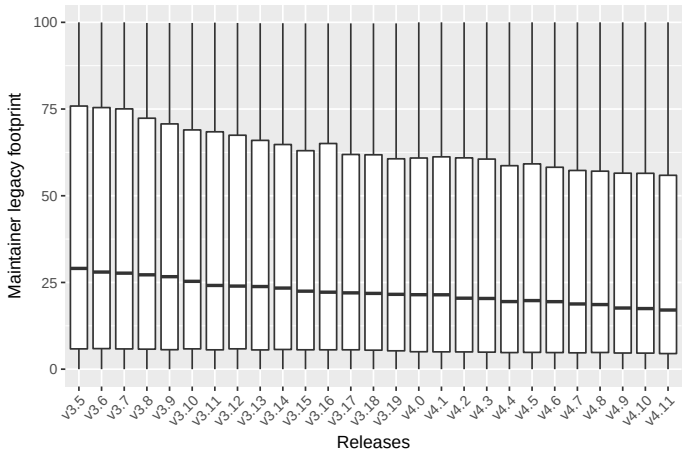


Figure 5.2 Median maintainer `legacy` across releases.

evaluates whether considering such additional activities is able to improve the identification of experts.

Approach:

To validate the ability of the measures in Table 5.2 to explain expertise, we evaluate how well the $footprint_j(\mathbb{A})$ measure involving those measures is able to identify the maintainers of Linux kernel subsystems. Those maintainers are the experts listed in the `MAINTAINERS` file of a kernel release.

For a given release and subsystem, we should find the maintainers in the top positions when ranked based on footprint values. The combination of activity measures $\mathbb{A}$ that is able to systematically yield the correct maintainers across subsystems and releases could be assumed to be a better indication of expertise.

In particular, we calculate two performance metrics:

POS_N Percentage Of Subsystems for which at least one maintainer was ranked in the top N expert candidates

POM_N Percentage Of Maintainers in the *whole* project who were ranked in the top N expert candidates for their subsystem

For these performance metrics, N is a threshold that can be varied. Our case study uses thresholds ranging from 1 to 5. It is important to note that, if the number of maintainers of a subsystem is larger than N, POM_N could be penalized. To avoid this, we slightly changed the definition of POM_N to be calculated only for the maintainers of all subsystems

with at most N maintainers, instead of for all maintainers of the whole project. For example, POM_1 measures the percentage of top-recommended maintainers of subsystems with at most 1 maintainer.

To structure our analysis, we analyzed the performance of expertise models involving only one metric of Table 5.2, then analyzed models involving all combinations of `legacy` with one of the other 4 measures, and finally one model with all measures combined. We focus explicitly on models involving `legacy` because it is a commonly used measure (Rahman and Devanbu, 2011), and hence we use its performance as baseline.

Since, for a given release, there is one POS_N value and one POM_N value, we calculate these metrics for each release, then study their distribution across the analyzed releases using boxplots. Figure 5.3 and Figure 5.4 show for each analyzed $\mathbb{A}$ the distributions of POS_N for N=1 and N=3, respectively, across the 22 studied Linux releases, while Figure 5.5 and Figure 5.6 the distributions of POM_N for N=1 and N=3, respectively. We only show the plots for N=1 and N=3, as for higher values of N the plots remain more or less stable.

Results:

`attributed` is the only single-measure model able to keep up with the multi-measure models.

The results in Figure 5.3 and Figure 5.5 indicate that the first two individual measures, i.e., `legacy` and `authored`, are bad indicators of expertise compared to the other studied metrics. For example, in Figure 5.3, `legacy` only reaches a median POS_N value of 47.22%, while `attributed` reaches a median POS_N percentage of 58.4%.

The models combining `legacy` with `committed`, `attributed` and/or `reviewed` perform the best. Figure 5.3 shows indeed how only these four models read median percentages of 69.9%, while the other multi-measure models, especially the one involving only `legacy` and `authored`, are not able to outperform the best individual measure models.

These findings confirm the intuition that maintainers shifted focus from doing development (`authored`) themselves to mentoring others by controlling access to their subsystem's Git repository through committing and/or reviewing. As such, an expertise model only involving their own development (i.e., `legacy` and `authored`) is unable to explain the current kernel maintainers' expertise. In other words, modern expertise models should take into account the time spent reviewing code and pushing changes upstream.

POS_N increases to a median of 87.5% for larger N, with multi-measure models outperforming single-measure models by at least 17%. Comparing Figure 5.4 to Figure 5.3 shows how the top multi-measure models for N=1 are able to increase their

Type	Measure	N = 1	N = 3
POS_N	P-value	4.27e-05	4.28e-05
POS_N	Cliff's delta	0.99	1.0
POM_N	P-value	4.27e-05	4.77e-07
POM_N	Cliff's delta	0.84	1.0

Table 5.3 P-values and Cliff's delta values for the Wilcoxon paired tests ($\alpha = 0.01$) between `attributed` and `legacy + committed` for ranking thresholds N=1 and N=3 and for POS_N and POM_N.

distance compared to even the best single-measure models (`attributed`). This, compared with a change in best performing single-measure models, indicates that a larger diversity in activity measures enables better identification of the two additional candidate maintainers. Indeed, by considering top performing contributors across a wider range of activities, there is a larger chance at least one real maintainer is found. Although the percentages of Figure 5.5 and Figure 5.6 cannot be compared directly to each other (cf. modified definition of POM_N), Figure 5.6 (for POM_N) shows a similar ranking of models as Figure 5.3 (for POS_N), confirming the findings for POS_N.

Table 5.3 shows the p-value and effect size of the Wilcoxon test between `attributed` and `legacy + committed` for figures 5.3, 5.4, 5.5, and 5.6. Each effect size being close to 1, we notice a **large** performance increase between `attributed` and `legacy + committed`.

Interesting to note is that, across all analyzed releases, the boxplots show a remarkable small variance, especially for N=3. Although this is partially due to the fact that less than 25% of the subsystems saw at least one maintainer change, it also indicates that our measures are stable across changes in the 5 activity measures used.

RQ3. How well does dimension 2 explain expertise?

Motivation:

The metrics analyzed in RQ2 reveal that traditional expertise metrics based solely on a contributor's own development productivity are not well suited to identify maintainers. Expertise models exploiting only the information available for the release under study, are able to obtain median POS_N performance of up to 75% (N=1) and 90% (N=3).

However, we believe that adding a historical dimension considering also the activity in the last R releases would assist the model in two ways. On the one hand, long standing kernel developers' contributions should carry more weight than newcomers' contributions. On the other hand, analyzing data on multiple releases would control for cases where contributors'

productivity was lower due to a variety of reasons, such as illness, vacation or work on other projects.

Approach:

For each studied kernel release, we calculated $footprint_j^R(\mathbb{A})$ for R=4, since this covers a time span of 60 to 70 days. For example, when looking at experts in release *v4.11*, we need to take into account data found for releases *v4.7, v4.8, v4.9, v4.10, and v4.11*. We repeat such analysis for each of the 22 releases. In this paper, we use linearly decaying weights W_i to combine the individual $footprint_j(\mathbb{A})$ values across the five considered releases, since this scheme is less extreme than the exponential and logarithmic ones.

Similar to RQ2, we then use the footprint values to create, for each subsystem and release, a ranking of all contributors active in the five considered releases. We also use the same performance metrics as for RQ2, which allows us to compare the results of RQ3 to those of RQ2 to validate whether the historical dimension improves the model.

To save space, and since we found that, similar to RQ2, the combination of `legacy` and `committed` performs the best, we only show the results for this model (the rest of the data will be made available after the double-blind review). In particular, Figure 5.7 and Figure 5.8 show the POS_N performance of the combined `legacy`+`committed` model without and with the history dimension, for ranking thresholds N ranging from 1 to 5. Figure 5.9 and Figure 5.10 show the corresponding POM_N results.

Table 5.4 contains the results of Wilcoxon paired tests between the POS_N values without and with history, for each N, and (similarly) between the POM_N values without and with history, for each N. For each test, we also provide the Cliff Delta effect size(((cite romano))).

Results:

The history-aware `legacy`+`committed` footprint models perform significantly better than the history-unaware models. Figure 5.7 and Figure 5.8 show how, except for N=3, the median performance of the history-aware expertise measure improves upon the history-unaware measure. If one considers only the first recommendation of the measure, there is a median 73.6% chance that at least one maintainer is identified for a history-aware expertise model compared to 69.9% with the single-release model. This difference progressively decreases for higher N, which means that, for higher N, an expertise model considering `legacy`+`committed` on one release only is robust enough to assess expertise.

We find similar improvements for Figure 5.9 and Figure 5.10, except that the improvements due to history increase for larger N (and for N=1 there is no significant improvement). This is clearly shown by the p-values and effect sizes of the Wilcoxon paired tests in Table 5.4

Fig.	Measure	1	2	3	4	5
5.7/5.8	P-value	1.12e-4	8.76e-3	1.15e-2	1.57e-3	7.70e-4
5.7/5.8	Cliff's delta	0.62	0.50	-	0.55	0.51
5.9/5.10	P-value	1.27e-2	4.63e-3	5.13e-4	1.57e-5	1.88e-4
5.9/5.10	Cliff's delta	-	0.46	0.56	0.66	0.69

Table 5.4 P-values and Cliff's delta values for the Wilcoxon paired tests ($\alpha = 0.01$) between the POS_N results of Figure 5.7 and Figure 5.8, and of Figure 5.9 and Figure 5.10, for ranking thresholds N=1 to N=5. A Cliff delta "-" indicates a non-significant test result, with p-value> α

($alpha = 0.01$). As an effect size greater than 0.474 indicates a **large** increase in performance, we notice that 7 of the 8 significant differences have an effect size of at least 0.50.

5.7 Discussion

Threats to validity:

Threats to external validity prevent generalization of empirical results to other contexts. In particular, due to the abundant volume of data and presence of an oracle for expertise, our empirical evaluation only focused on 22 releases of the Linux kernel project. Hence, the study should be expanded to cover not only more kernel releases, but also other open (and closed) source projects. Furthermore, we considered only 5 expertise measures for our footprint models. Other measures, such as those mentioned in Table 5.1, should be studied to understand their impact on expertise.

Threats to construct validity involve risks regarding the measures used in the empirical study. Of the five considered expertise measures, `reviewed` was the only one requiring noisy approximations. Except for cases where the email patch subject was identical to the Git commit message summary, there is a definite risk of false positive and false negative matches, as identified earlier by Jiang et al. (Jiang et al., 2013, 2014). This might explain the relatively weak performance of expertise models involving `reviewed`. However, no better alternatives exist for projects that use mailing lists for code review. Projects using web-based review environments like Gerrit do not have this issue, and will have perfect matching between commits and their reviews.

Finally, regarding threats to internal validity (i.e., confounding factors potentially explaining our findings), we mention the limited number of subsystems considered for RQ2 and RQ3. This number was the result of the data filtering in subsection 5.5.2 used to eliminate temporarily inactive subsystems. Furthermore, we used the `MAINTAINERS` file as oracle for

expertise. Although this is the known reference in the Linux kernel community for finding the right maintainer to contact, this is a manually maintained text file that hence could contain inconsistencies (even though changes to it are peer-reviewed).

Finally, although maintainership is a form of expertise, there are other forms of expertise that our footprint models be indicators of that were not considered in our empirical study. As such, some of the false positive recommendations of our footprint rankings might actually be correct suggestions based on a different interpretation of expert, in which case our POS_N and POM_N results are lower bounds for the actual performance.

Future work:

Apart from addressing the threats to validity, other future work should consider different weights w_i and W_i. The former weights consider different activities to be more relevant than others, while the latter weights would give more or less weights to older vs. newer releases. For example, comparison of exponential and logarithmic decaying weights to the linear decay used in our study could be interesting. Similarly, different values of R for $footprint_j^R(\mathbb{A})$ should be evaluated.

Finally, whereas we used a top-down approach from expertise model to evaluation on an actual open source project, a bottom-up approach starting from the analysis of a project's or subsystem's maintainers before formulating expertise measures and models could provide complementary insights into different kinds expertise.

5.8 Conclusion

This paper argued about the need for expertise models considering a wide range of developer and other activities, and doing so across different snapshots of a project instead of just for one snapshot. Through an empirical study on 22 releases of the Linux kernel, we empirically showed how measures about an expert's own coding footprint (`legacy`) and her involvement in coordinating other project members (e.g., committing their commits and/or reviewing their code changes) significantly improves on coding-only expertise models. Furthermore, considering those measures across different releases significantly improved performance, with large effect size.

The simplest incarnation of our expertise model that software organizations should consider adopting involves (1) a developer's code `legacy` and number of changes `committed`, which are both readily obtainable from a Git log, calculated across (2) the last 5 releases. In future work, we will consider additional activity measures and empirically analyze other open source projects.

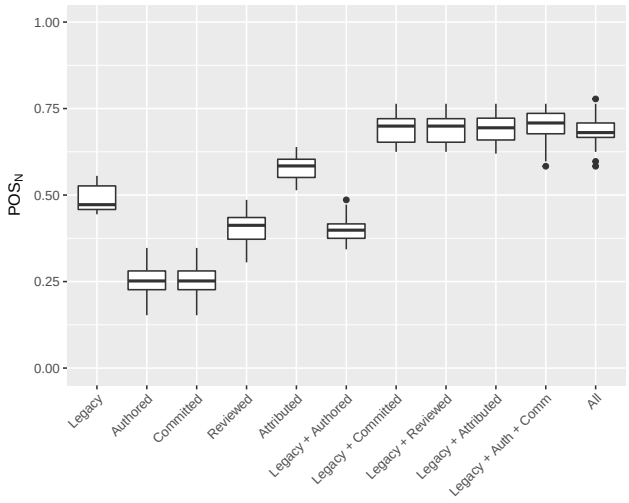


Figure 5.3 Distribution of POS_N for each combination of activity measures, for ranking threshold N = 1.

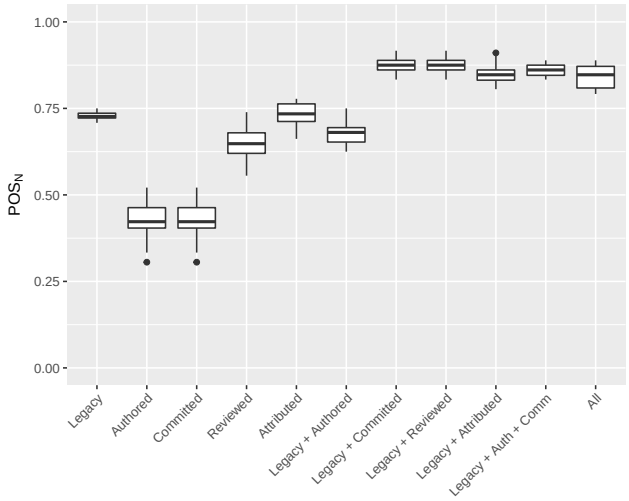


Figure 5.4 Distribution of POS_N for each combination of activity measures, for ranking threshold N = 3.

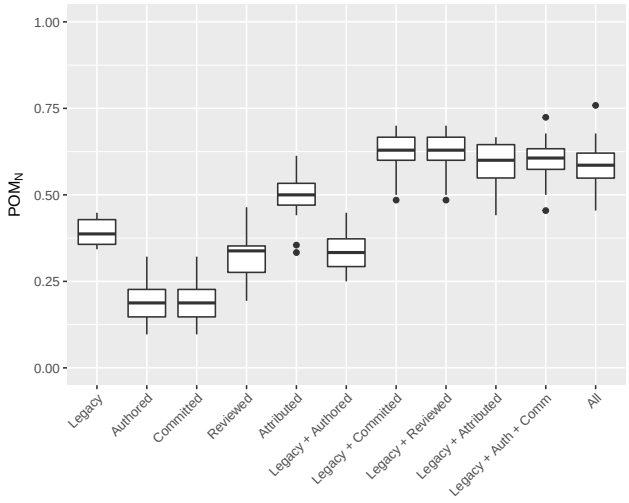


Figure 5.5 Distribution of POM_N for each combination of activity measures, for ranking threshold N = 1. These boxplots only consider subsystems with at most 1 maintainer.

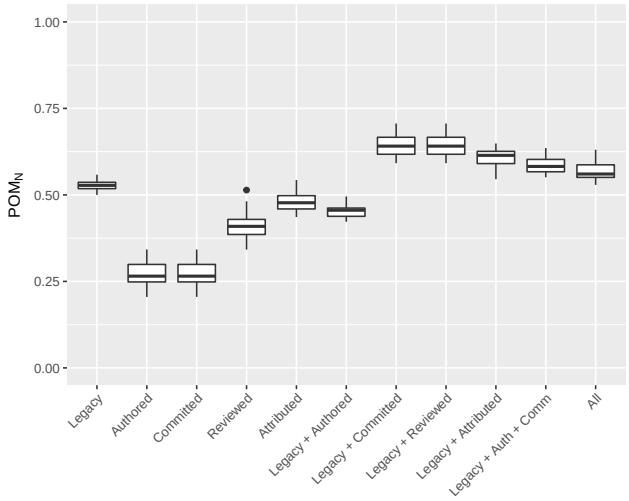


Figure 5.6 Distribution of POM_N for each combination of activity measures, for ranking threshold N = 3. These boxplots only consider subsystems with at most 3 maintainers.

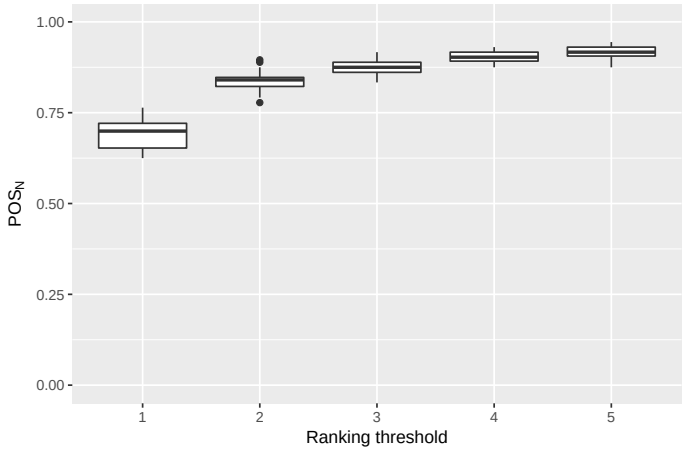


Figure 5.7 Distribution of POS_N for the combined `legacy+committed` model (**without** history dimension), for different N.

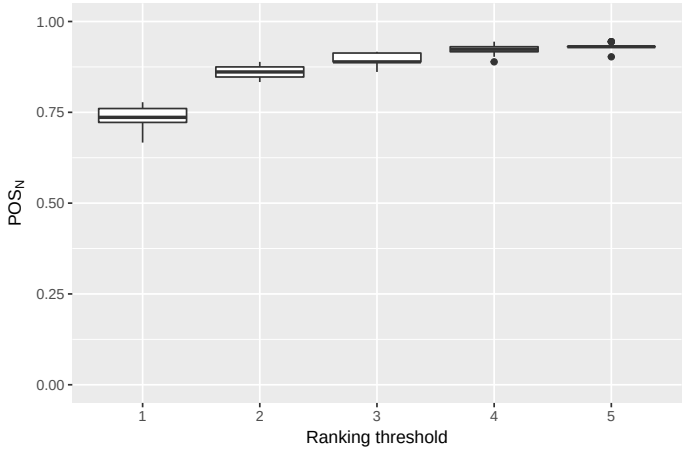


Figure 5.8 Distribution of POS_N for the combined `legacy+committed` model (**with** history dimension), for different N.

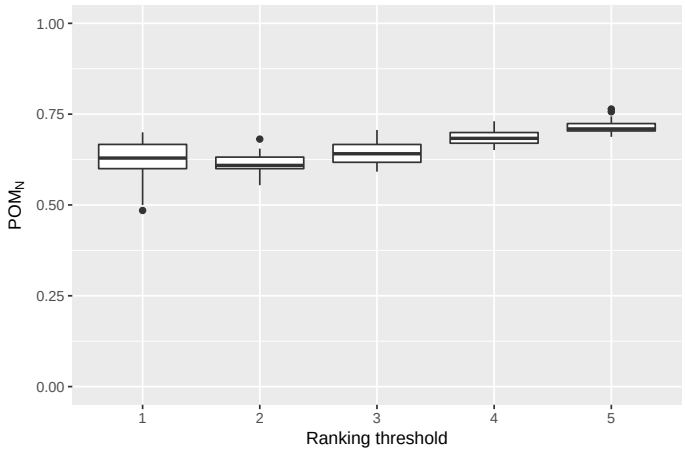


Figure 5.9 Distribution of POM_N for the combined `legacy+committed` model (**without** history dimension), for different N. For each N, the boxplot only considers subsystems with at most N maintainers.

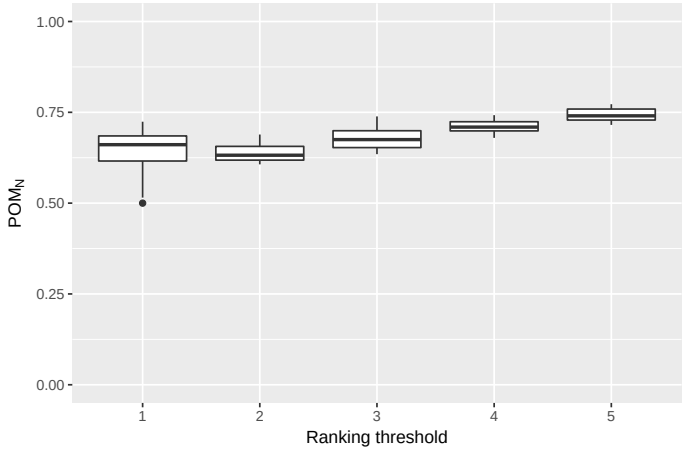


Figure 5.10 Distribution of POM_N for the combined `legacy+committed` model (**with** history dimension), for different N. For each N, the boxplot only considers subsystems with at most N maintainers.

CHAPTER 6 CONCLUSION

The research project described in this thesis establishes a history-aware model capable of predicting experts in a large software project. We describe the different metrics acquired for the creation of the model as well as the techniques used to mine the necessary data. We were able to make some of the data available to the Linux community through two open source tool available online. We describe the different steps undertaken during the deployment of those tools. We also provide an evaluation of our expertise model.

6.1 Advancement of Knowledge

In addition to communicating our findings regarding expertise models in a submitted scholarly article, we were able to build two interfaces to further share our data with the rest of the Linux community. The main advancement of knowledge carried by our research project is the new dimentions used in our expertise model. State of the art techniques fail to include other aspects of software engineering, like code reviews and upstream committing. Our model is more appropriate to large organization where mainainers or managers are usually too busy to continue contributing code to the project.

6.2 Limits, Constraints, and Recommendations

6.2.1 Srcmap

Srcmap, our visualization of the kernel and its authors, has a few constrains and a lot of possible future work. The main constrain is the lack of a fluid user experience. The amount of data to process in the browser is too high to allow smooth browsing of the main tree. A way to address this issue would be to configure the interface to only download the required data as users browse the visualization. This way, the internet browser uses to display the tool would not have to save the entire dataset in memory and would only process the desired area.

6.2.2 Email2git

Email2git, our code reivew tracking system, has a few important limitations. The first limitation to consider is the missing mailing list. Although our patch data source, patchwork.kernel.org, already track many precious mailing lists, some major mailings list like

`net-dev` are not tracked. This is reflected in the low number of commit matched in the `net` subdirectory.

We recieved a lot of valuable feedback from linux developers after our refereed talk the Open Source Summit North America. A developer mentioned the absence of the *Patch 0* from our current implementation of Email2git. The Patch 0 is a summary of the changes submitted, often in multi-patch submissions. Another suggestion was to track *linux-next*. This would allow developers to access discussion behind commits that have not been integrated in the main tree.

For the future of this project, we recommend running our own instance of Patchwork 2.0, which automatically track the Patch 0 of each patch. In addition to ansering the lack of Patch 0, it would allow to have control on the tracked mailing lists. If we have access to old archives of the desired mailing lists, we could be able to create matching data dating to before 2009. We also recommend tracking the linux-next tree, as Email2git could ease the integration debugging process.

6.2.3 Paper

There are two main limitation to the model proposed in our submitted paper. The first limitation is a direct concequence of Email2git's limitation. The missing mailing lists cause an uneven distribution of the matched commits accross the different subdirectories of the kernel. To address this limitation, we only studied subsystems with a certain percentage of matched commits. This ensured homogeneity of the data among the different subsystems.

The other limitation is the validation technique we implement to assess the performance of our model. We use the maintainers currently active in the subsystem for the studied release. The issue with the technique is that since our model is partially based on activities usually related to maintainership, such as code reviews and upstream committing. To evaluate our model as **maintainer recommender** instead of an **expert recommender**, we would have to look at the developers that were selected as a replacement for a departing maintainer. A strong model would be capable of detecting the chosen developer for the release before the maintainer change.

At last, we cite the existing link between code stability and knowldge of that code area. Since our model offers a customaizable historical weight function, we could choose this weight function according to the stability of the code. For example, we could implement an exponential weight function to a very stable code base, as older contributions should account for more of the expertise measure. Furthermore, a code base undergoing large amounts of changes

could require a logarithmic weight function, as older contributions should not affect current expertise as much as recent contributions.

REFERENCES

"Employee Turnover, Labor Lost", The Economist, July 13 2000.

"Linux".

J. Anvik, L. Hiew, et G. C. Murphy, "Who should fix this bug?" dans *Proceedings of the 28th International Conference on Software Engineering*, série ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. DOI: 10.1145/1134285.1134336. En ligne: http://doi.acm.org/10.1145/1134285.1134336

F. Armstrong, F. Khomh, et B. Adams, "Broadcast vs. unicast review technology: Does it matter?" dans *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 219–229. DOI: 10.1109/ICST.2017.27

F. T. Armstrong, F. Khomh, et B. Adams, "Broadcast vs. unicast review technology: Does it matter?" dans *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, March 2017.

B. B. Bederson, B. Shneiderman, et M. Wattenberg, "Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies", *ACM Trans. Graph.*, vol. 21, no. 4, pp. 833–854, Oct. 2002. DOI: 10.1145/571647.571649. En ligne: http://doi.acm.org/10.1145/571647.571649

P. Bhattacharya, I. Neamtiu, et M. Faloutsos, "Determining developers' expertise and role: A graph hierarchy-based approach", dans *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 11–20. DOI: 10.1109/ICSME.2014.23

C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, et P. Devanbu, "The promises and perils of mining git", dans *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, série MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. DOI: 10.1109/MSR.2009.5069475. En ligne: http://dx.doi.org/10.1109/MSR.2009.5069475

J. Corbet et G. Kroah-Hartman, "2017 linux kernel development report", The Linux Foundation, Rapp. tech., 2017.

T. Fritz, G. C. Murphy, et E. Hill, “Does a programmer’s activity indicate knowledge of code?” dans *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, série ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 341–350. DOI: 10.1145/1287624.1287673. En ligne: http://doi.acm.org/10.1145/1287624.1287673

Y. Jiang, B. Adams, et D. M. German, “Will my patch make it? and how fast? – case study on the linux kernel”, dans *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR)*, San Francisco, CA, US, May 2013, pp. 101–110.

Y. Jiang, B. Adams, F. Khomh, et D. M. German, “Tracing back the history of commits in low-tech reviewing environments”, dans *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Torino, Italy, September 2014.

D. Joseph, K.-Y. Ng, C. Koh, et S. Ang, “Turnover of information technology professionals: A narrative review, meta-analytic structural equation modeling, and model development”, *MIS Q.*, vol. 31, no. 3, pp. 547–577, Sep. 2007. En ligne: http://dl.acm.org/citation.cfm?id=2017336.2017343

D. W. McDonald et M. S. Ackerman, “Expertise recommender: A flexible recommendation system and architecture”, dans *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, série CSCW ’00. New York, NY, USA: ACM, 2000, pp. 231–240. DOI: 10.1145/358916.358994. En ligne: http://doi.acm.org/10.1145/358916.358994

T. Mens, M. Claes, P. Grosjean, et A. Serebrenik, *Studying Evolving Software Ecosystems based on Ecological Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 297–326. DOI: 10.1007/978-3-642-45398-4$_1$0. En ligne: https://doi.org/10.1007/978-3-642-45398-4_10

S. Minto et G. C. Murphy, “Recommending emergent teams”, dans *Proceedings of the Fourth International Workshop on Mining Software Repositories*, série MSR ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 5–. DOI: 10.1109/MSR.2007.27. En ligne: http://dx.doi.org/10.1109/MSR.2007.27

A. Mockus et J. D. Herbsleb, “Expertise browser: A quantitative approach to identifying expertise”, dans *Proceedings of the 24th International Conference on Software*

Engineering, série ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512. DOI: 10.1145/581339.581401. En ligne: http://doi.acm.org/10.1145/581339.581401

F. Rahman et P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship", dans *Proceedings of the 33rd International Conference on Software Engineering*, série ICSE '11. New York, NY, USA: ACM, 2011, pp. 491–500. DOI: 10.1145/1985793.1985860. En ligne: http://doi.acm.org/10.1145/1985793.1985860

P. C. Rigby, Y. C. Zhu, S. M. Donadelli, et A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at avaya", dans *Proceedings of the 38th International Conference on Software Engineering*, série ICSE '16. New York, NY, USA: ACM, 2016, pp. 1006–1016. DOI: 10.1145/2884781.2884851. En ligne: http://doi.acm.org/10.1145/2884781.2884851

C.-G. Wu, J. H. Gerlach, et C. E. Young, "An empirical analysis of open source software developers' motivations and continuance intentions", *Inf. Manage.*, vol. 44, no. 3, pp. 253–262, Avr. 2007. DOI: 10.1016/j.im.2006.12.006. En ligne: http://dx.doi.org/10.1016/j.im.2006.12.006

M. Zhou et A. Mockus, "Who will stay in the floss community? modeling participantâ's initial behavior", *Software Engineering, IEEE Transactions on*, vol. 41, pp. 82–99, 01 2015.

M. Zhou, Q. Chen, A. Mockus, et F. Wu, "On the scalability of linux kernel maintainers' work", dans *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, série ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 27–37. DOI: 10.1145/3106237.3106287. En ligne: http://doi.acm.org/10.1145/3106237.3106287