Noname manuscript No.
(will be inserted by the editor)

Continuously Mining Distributed Version Control Systems: An empirical study of how Linux uses git

Daniel M German · Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Distributed version control systems (D-VCSs—such as `git` and `mercurial`) and their hosting services (such as Github and Bitbucket) have revolutionalized the way in which developers collaborate by allowing them to freely exchange and integrate code changes in a peer-to-peer fashion. However, this flexibility comes at a price: code changes are hard to track because of the proliferation of code repositories and because developers modify ("rebase") and filter ("cherry-pick") the history of these changes to streamline their integration into the repositories of other developers. As a consequence, researchers and practitioners, who typically only consider the (cleaned up) history in the official project repository, are unaware of important elements and activities in the collaborative software development. In this paper, we present a method that continuously mines all known D-VCSs of a software project to uncover the complete history of a project's development. We use this method to (1) show the divergence between the code history in the official Linux kernel repository and the complete kernel development history, and (2) to investigate the characteristics of the ecosystem of `git` repositories of the Linux kernel. Finally, we discuss how continuous mining could be adopted by current D-VCS hosting services.

Daniel German
University of Victoria
E-mail: dmg@uvic.ca

Bram Adams
Polytechnique Montréal
E-mail: bram.adams@polymtl.ca

Ahmed E. Hassan
Queen's University
E-mail: ahmed@cs.queensu.ca

1 Introduction

Distributed version control systems (D-VCSs) have taken the open source ecosystem by storm. Since initial development of D-VCSs in the early 2000s (`arch` and `monotone`) and their breakthrough for open source development in the Linux kernel development community (`bitkeeper` and `git`), D-VCSs are quickly replacing the predominant centralized version control systems (C-VCSs) like CVS and Subversion. For example, by the end of 2013 Ohloh, a directory indexing more than 500,000 open source projects, reported that 29% of the tracked projects use `git` and 5% use other D-VCSs like `mercurial` and `bazaar`, compared to 52% using C-VCS Subversion and 11% using CVS (Black Duck Inc., 2013). Similarly, the Eclipse 2012 Community Survey with 732 open source participants reported that `git` has replaced CVS as the second most popular version control system, with 23.2% (almost double the adoption of 2011), right behind Subversion, with 46.0% (Foundation, 2012).

The success of D-VCSs is due to the flexibility that they provide as well as the emergence of D-VCS hosting services like Github and Bitbucket. One of the most fundamental characteristics of a D-VCS is that it allows each developer to have her own repository and to exchange change-sets with other developers in a peer-to-peer fashion. As such, each developer can decide which code changes are worthwhile to integrate into her repository and which ones are not. Hence, code changes propagate throughout developer repositories until they reach the primary project repository (we will refer to this repository as *blessed*) containing the "official" code base. Creating and hosting a D-VCS repository has become relatively trivial through the advent of hosting services, which offer 1-click access for cloning an existing D-VCS repository and making the new clone accessible for other developers.

The ability to access the public repositories of developers in addition to the official *blessed* repository (we use the term "Super-repository" for the set of all repositories of a project) makes it possible to obtain a more complete view of development activities. For example, it might be possible to observe if work is cleaned up before it is sent to *blessed*, or if some changes are never propagated to other repositories and developers. Hence, the data in all developer repositories would provide a potential gold mine for research on development processes, collaboration, software evolution and socio-technical congruence, as well as for practitioners who want to monitor their development progress.

To understand how developers use the features of D-VCSs in practice, we studied the use of the git D-VCS in the Linux kernel project. The Linux kernel is arguably one of the largest open source software systems ever developed, with over 10,800 developers having contributed to its development since 2005 (Corbet et al, 2013). It was also at the forefront of the move towards D-VCSs. Our case study on the Linux kernel enables us to study the following research questions:

– What are the characteristics of the repositories in the Linux *Super-repository*?

– How do the repositories of a *Super-repository* interact with each other and how do commits flow across them?

However, during our study, we observed that current repository mining methods are unable to exploit the rich set of development data stored in D-VCSs. In particular, there are three major challenges that have stopped traditional mining methods from leveraging the rich data available through D-VCSs. First, **commits do not keep track of which repositories they have passed through**. This is even true for online services like Github and gitorious, which track when a merge is performed, as well as its source and destination repositories, but do not explicitly track the commits that are moved in such a request. Second, **D-VCSs recommend developers to rewrite (*rebase*) their development history, if necessary, to facilitate easier integration of their work into other repositories** (Bird et al, 2009b; Corbet, 2008a). This rewriting ranges from merging, splitting or deleting change-sets to changing their order (i.e., creating a copy and destroying the original commit). Third, **when merging a change-set into another repository, developers are also encouraged to filter (*cherry-pick*) only the interesting portions and ignore the rest**. Due to these three challenges, the actual development history of projects that use D-VCSs is permanently lost, making traditional analysis of code change history incomplete and potentially unreliable in a D-VCS repository.

In order to address our two research questions, this paper replaces the traditional "snapshot-based mining" (*snapMining*) of D-VCS repositories by a "continuous mining" (*continuousMining*) method. Whereas *snapMining* merely analyzes the list of code changes in the *blessed* D-VCS repository, *continuousMining* *continuously* crawls *all* known D-VCS repositories multiple times a day to record and analyze all new change-sets. As such, *continuousMining* not only yields more accurate development history, it also allows us to study phenomena that are impossible to study using only *blessed*, like rebasing and cherry-picking, and to study integration activities involved in moving commits throughout the repositories on their way to *blessed*. To evaluate the effectiveness of *continuousMining* (which is a prerequisite for the other two research questions), we study the following research question:

– Does *continuousMining* expose any bias in the project history recovered using *snapMining*?

The contributions of this paper are:

– A method (and its implementation) to mine D-VCSs that relies on the continuous mining of the distributed repositories. Based on this method, we also implemented a git-history tracking dashboard to improve traceability of commits in the ecosystem of Linux `git` repositories.
– We compare the use of *continuousMining* to *snapMining* and show that:
  – *continuousMining* uncovers significantly more development activity and a larger ecosystem than *snapMining*.
  – *continuousMining* can monitor the propagation of commits across repositories and identify when they arrive at *blessed*.

  - Cherry-picking and rebasing make commit identifiers (ids) insufficient to track change-sets across repositories. They also make the commit date and the committer metadata unreliable.
- An empirical study of how Linux developers use `git`, i.e., we:
  - show that one can identify different kinds of repositories in the Linux ecosystem, each of which has a different role, such as: repositories for producers of patches, for integrators, and for product lines (e.g., versions of the kernels distributed by Android, SUSE, Redhat and Ubuntu).
  - show that commits with new features take longer and visit more repositories on their way to *blessed* than commits that fix bugs.

2 Background

In this section, we introduce the nomenclature that we will use, then briefly introduce the Linux development process and how it uses `git`.

2.1 The D-VCS *Super-repository*

In C-VCSs all commits happen in a central repository. Only developers with write-access can contribute to this repository. The main development activity takes place on the so-called "trunk" of the repository (the repository's main codeline), whereas more experimental features are developed on "branches" (other codelines), before being merged back into the trunk or abandoned.

D-VCSs, such as `git`, take a radically different approach. Instead of making a working copy of the state of a repository at a particular time, the developer clones the whole history of a repository (the trunk and all branches), and develops inside her local repository. Any repository can be cloned, and, from a conceptual point of view, no repository is more important than any other. In practice, the development team will organize the repositories in a social hierarchy and at least one of the repositories will be marked as the central repository (its *blessed* repository), where the latest team-approved changes are expected to be found.

Any contributing developer aims her changes to flow from her personal repository to *blessed*, using one of three methods: by "pushing" them to the destination repository (if she has write-access–this is equivalent to committing to a centralized version control repository); by letting someone with write-access to the destination repository "pull" the changes in; or by emailing the set of commits to a person who has write access to the destination repository (D-VCSs allow exchange of commits via email with full commit history tracking).

Conceptually, we can model the collection of all repositories of a project as if it were a single repository, which we call the project's *Super-repository*. The "trunk" of this *Super-repository* is the trunk of the *blessed* repository. The trunk and branches of all public (available to all teammates) and private

(accessible only to their owner) repositories that were (recursively) cloned from the same repository are branches of the *Super-repository*. The set of all commits in the *Super-repository* is the union of all commits in each of its repos, hence each commit in the *Super-repository* exists in at least one repository in the *Super-repository*. Every time a developer pulls, pushes from another repository, or incorporates commits that she received via email, she is effectively merging one branch (in the source repository) into another one (in the destination repository). Such a merge might result in a new commit that records the merge, a so-called *merge-commit*, in contrast to the *non-merge commits*, which contain the actual code changes. Merge-commits link together branches, and correspond to integration work.

2.2 Linux and `git`

In order to study the use of D-VCSs in practice, this paper analyzes the Linux kernel, which arguably is one of the most globally distributed software systems ever developed. According to the Linux Foundation, nearly 10,000 developers from over 1,000 different companies have contributed to the Linux kernel since 2005 (Corbet et al, 2013). The requirements imposed by such a large and diverse team, working on a codebase of more than 44,000 files, has prompted many innovations in the area of distributed version control systems.

In 2002, Linux decided to start using `bitkeeper`, a commercial D-VCS system. According to the Linux Weekly News, the adoption of `bitkeeper` by the Linux development team became a success: "The rate at which patches were merged skyrocketed, the developers were happy, and the whole system ran smoothly."(Corbet, 2005). This improvement was attributed to the replacement of a workflow based on email patches with a workflow based on multiple distributed repositories that could merge their changes with minimal user intervention. In 2005, Linus Torvalds decided to stop using `bitkeeper`, and replaced it with `git` (his own implementation of a D-VCS). Because `git` was built with the Linux workflow in mind, one can expect that Linux developers exercise most of `git`'s features.

Anybody who wants to contribute to Linux is expected to clone one of its public repositories, and implement his or her contribution locally on top of such a clone. After the contributions are completed, they are sent to a mailing list for review (these emails are prepared using `git`'s email patch feature). Once the contribution is approved, it is committed by a member of the Linux development team into a repository. The newly created commit will record as its author the original creator and as its time of authorship the time when it was created in the author's repository (before it was emailed). The committer and commit date of the newly created commit will reflect who integrated this commit into the receiving repository. `git` also implements a method to trace the review and mail-patch process via sign-off fields that get added to the message of each commit. These sign-off fields usually reflect the original author

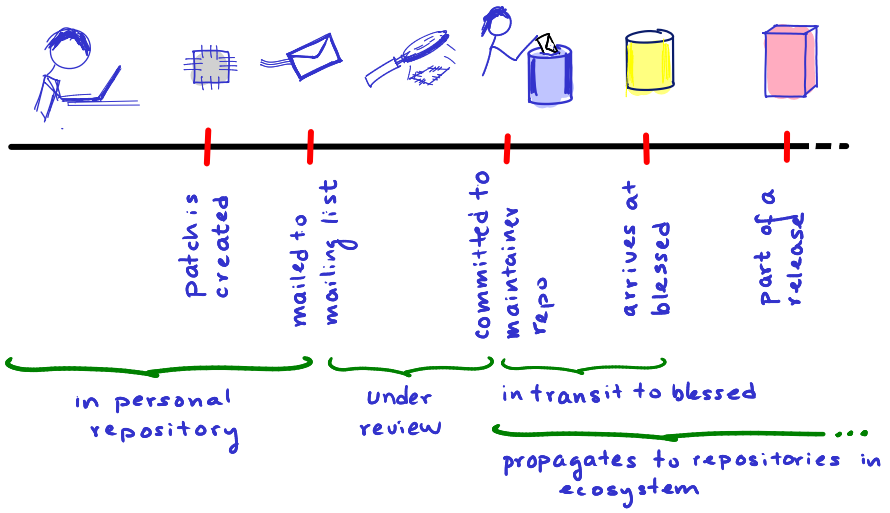


Fig. 1 Life of a patch. A patch is created in a private repository. After that, the patch is sent to one of the kernel mailing lists where the patch might be picked up by a module maintainer, who signs it off and commits it to her public repository. From there, the patch (inside a commit) starts to be propagated to other repositories, eventually reaching *blessed*. This paper concentrates on the propagation of a patch in the ecosystem of public repositories, once the patch has been accepted by a module maintainer.

of a commit, the module maintainer who first committed to her repository, and other individuals involved in the review.

Once commits are in the `git` repository of the module maintainer, they start to propagate (via push and pulls) to other repositories, until eventually, they reach *blessed* (the repository of Linus Torvalds). Some commits might not go through a review process. In that case, they move from their source repository to *blessed* via push and pull operations between individual repositories. Such commits might still include a sign-off field with the name of the person that first committed the change. This person will be recorded as both the author and committer of the commit. Figure 1 depicts the life of a patch. Since personal repositories are only accessible by their owners, we concentrate on the movements of commits in the ecosystem of public repositories.

2.3 Challenges Introduced by D-VCSs

In order to understand how software projects like the Linux kernel use a D-VCS like `git`, we first need to address the three major challenges introduced by D-VCSs: (Bird et al, 2009b): (1) once a repository is merged into another, the history does not contain information about the location from where the commits originated, except for any optional commit log information stored in merge-commits (if these merge commits were created); developers can alter

the history of commits in their repository by (2) moving commits around (rebasing) or (3) filtering the commits (cherry-picking). We first discuss these challenges, and in the next Section presents our method to resolve them.

2.3.1 No Traceability of Commits across Repositories

Paradoxically, D-VCSs assume an self-centred view of the world. A repository does not keep track of other repositories in the *Super-repository*; and a repository does not keep track of the commits that occurred in other repositories and have not propagated to it[1]. Commits receive a unique identifier upon creation using a hash function (`git` uses SHA-1) of the contents of the commit, i.e., its metadata[2], patch and parent commits. Since the slightest change to the metadata would alter the SHA1, the commit cannot accumulate any information about the repositories through which it has passed on its way to *blessed*. Only merge-commits (if these commits are created) might contain, as part of their commit message, information about the address (URI) of the branch being merged, but this depends on the discipline of the developer doing the merge.

2.3.2 Commits are Moved Around

Rebasing is the act of moving commits from their current location (following an older commit) to the new head (newest commit) of their branch (Chacon, 2009). Developers use rebasing to make their changes work with the newest changes committed in the parent repository (e.g., *blessed*), and to make their repository easier to integrate into other repositories. Because rebasing changes the ancestors of a commit, the SHA1 of the commit itself changes as well (even if the rest of the metadata remains the same) and this results in a new commit ID for the same change, losing traceability with the old commit.

2.3.3 Commits are Filtered

Cherry-picking is a process that allows a developer to select one or more parts from a larger change-set in the source repository and only apply those in the destination. As with rebasing, cherry-picking a commit will change the commit's ID because the ancestors of the destination are likely different than in the original repository. Also, the dropped parts of a change-set in the source repository might be discarded by dropping the branch in which they reside. Again, possibly important development history disappears, making it impossible to retrieve a complete history from *blessed*.

[1] Even services on top of D-VCSs, like Github, do not provide a way to know the set of all commits in a *Super-repository*, i.e., the commits that have already arrived to *blessed* and those that are still in other repositories.

[2] The metadata consists of the time during which the commit was first committed (authorship date), the name of the author, the time when it was last committed (commit date), the committer, and the commit message.

3 Continuously Observing Events as They Happen

To overcome the three challenges described in Section 2.3, we need to be able to observe, over time, the commits as they are created in their original repository, the propagation of commits across repositories and, potentially, whether commits are deleted or rebased.

Traditionally, researchers have mined only the *blessed* repository of a project that uses a D-VCS, i.e., the final version of accepted commits. We will refer to this as the "snapshot method" (*snapMining*). Unfortunately, this method is not able to see how commits move across repositories, and misses activities that are not propagated to *blessed*. Also, the history of the interactions between the various development repositories is lost.

If D-VCSs had a central logging feature[3], we could simply exploit such logs. However, even hosting services for D-VCSs do not provide full tracking of commits. For example, Github only tracks cloned repositories if the cloning operation is done inside Github's web interface too, and Github tracks the origin of commits in a merge only if such a merge was the result of a pull-request. An in-depth explanation of this lack of traceability can be found in Gousios et al (2014) and Kalliamvakou et al (2014).

The alternative is to continuously query each repository in a *Super-repository*, discovering their new commits and those commits that these repositories no longer contain. Such continuous querying will allow us to know when and where commits are created, and how they propagate to other repositories. Along the way, new repositories in the *Super-repository* will be discovered. We have developed a method to achieve this, which we call: **continuous mining of distributed version control repositories**, and we refer to it as ***continuousMining***.

Using *continuousMining*, it is possible to retrieve: a) the repositories that compose the *Super-repository*, and for each of them, its owners (the developers that have commit privilege to it), and its branches; and b) the set of all commits in the repository, and for each commit, the moment at which it appears and/or is deleted in each repository in the *Super-repository*. *ContinuousMining* is composed of three steps: 1) *Initializing the set of repositories*, 2) *Crawling process*, and 3) *Reviewing commits and discovering new repositories.* These steps are depicted in Algorithms 1 and 2, while their output is described in Table 1.

3.1 Initializing the set of repositories

At the very beginning of the mining process, the *Super-repository* (i.e., the set of known repositories R) will contain, at the very least, the blessed repository of a project and any other known and accessible repository used by the project. We found that projects typically publish the locations of their main repositories. Services like Bitbucket, Github and Gitorious also cross-reference

[3] `bitkeeper` is the only D-VCS that optionally supports centralized logging.

Table 1 The output of our method for *continuousMining*.

Name	Description
R	set of known public repositories
$Owners$	set of tuples $\langle person, repo \rangle$ indicating that $person$ can commit in $repo$
C	set of commits in *Super-repository*
$Prop$	set of tuples $\langle c, r, t \rangle$ showing that commit c has propagated to repo r at time t
Del	set of tuples $\langle c, r, t \rangle$ showing that commit c is deleted from a repo r at time t
B	set of tuples $\langle b, c, r, t \rangle$ showing that branch b in repo r had commit c as its head at time t

hosted repositories that have been cloned from each other (as long as these clones are created within these services).

3.2 Crawling process

This step is fully automated. Every fixed time interval τ, each repository r in the *Super-repository* is queried for changes (part I of Algorithm 1). If r has changed, we record its new commits (including the metadata and patch of the commit), which commits were deleted, and the list of branches that each repository currently has. We timestamp this information with the time at which we updated the repository.

A major challenge is that this process is not atomic, i.e., we cannot stop people from updating the various repositories while we perform a scan. Therefore, we have to deal with two issues: a) commits that have been propagated between scans, where we do not find the new commit in its true origin, but in a repository to which it has been propagated; and b) finding commits that originated in new repositories that are not yet in our list of repositories. For this, we keep a list of the *owners* of every repository. The owner of a repository is the name of the person that can commit to that repository. A repository might have more than one owner. We deal with variations in names by first unifying the names of developers and their email addresses using a method similar to Bird et al (2006).

If a new commit c appears in repo r, but $committer(c)$ is not an owner of r, then we mark the commit for review (see part II of Algorithm 1).

3.3 Reviewing commits and discovering new repositories

As described above, the crawling process finds, at every iteration, commits that need to be reviewed. This reviewing process is described in Algorithm 2 and it is semi-automated (usually conducted once a day). It is composed of two phases. The first phase is automated: for each commit c to review, we automatically verify if the commit is created by the owner of the repository (we

Algorithm 1: Crawling process

input : τ an interval of time between scans of the repositories

output: R set of known repositories in *Super-repository*
output: C set of all commits in *Super-repository*
output: *Prop* set of tuples each showing when a commit is observed arriving at a repository
output: *Del* set of tuples each showing when a commit is deleted from a repository
output: B set of tuples showing a branch, the repository where it is observed, and its head
output: *ToReview* set of tuples containing commits that are likely created in a repository not in *Super-repository* yet and that must be manually reviewed

$C \leftarrow Prop \leftarrow Del \leftarrow Branches \leftarrow Owners \leftarrow \emptyset$;
$R \leftarrow$ known repositories in *Super-repository* (at least *blessed*);

Make a local clone of every repo in R;

repeat *at intervals* τ
1 **for** *every repository* r *in* R **do**
Synchronize the local copy of r ;
if r *changed since last scan* **then**
$t \leftarrow$ current time;
$new \leftarrow$ commits not previously seen in r ;
$Del \leftarrow Del \cup$ commits deleted in r;
$B \leftarrow B \cup$ new branches in r;
for *every commit* c *in new* **do**
$Prop \leftarrow Prop \cup \langle c, r, t \rangle$;
for *every commit* c *in new but not in* C **do**
if $\langle committer(c), r \rangle \notin Owners$ **then**
11 $ToReview \leftarrow ToReview \cup \langle c, r, t \rangle$
else
$C \leftarrow C \cup c$

might have updated the list of $Owners$ of that repository since the time this commit was added to $ToReview$—see part α of Algorithm 2). Otherwise, we verify if the same commit was found in the same iteration or in the immediately next one in a repository r_1 such that the $committer(c)$ is one of the $Owners(r_1)$ (see part β of Algorithm 2). In that case, we tag the commit as coming from r_1 and remove it from $ToReview$.

The second phase is manual (although we have developed scripts to help us with it). For every commit, we first inspect the commits that follow it, and if there is a merge, and the merge is by a different committer, then we manually inspect this merge to find if it refers to an unknown repository r_2. If r_2 is unknown, we clone r_2, process it for the first time, identify its owners, and, if c is found in r_2 and the committer of c is an owner of r_2, then we mark c as having originated in r_2 (see part γ of Algorithm 2). Otherwise, we scan the mailing lists and web sites of the project to determine if the $committer(r)$ should be added as an owner of r (part δ of Algorithm 2). Although we have developed tools to help with this phase, it requires manual intervention. Fortunately, we

Algorithm 2: Reviewing commits and discovery of new repositories.

input : *ToReview* set of tuples containing commits that are likely created in a repository not in *Super-repository* yet and that must be manually reviewed

output: Updates R, $Owners$, C and $ToReview$ –see Table 1.

for *every tuple* $\langle c, r, t\rangle \in ToReview$ **do** // automated process

α **if** $committer(c) \in owner(r)$ **then**

$ToReview \leftarrow ToReview - \{\langle c, r, t\rangle\}$;

β **else if** $\langle c, r_1, t_1\rangle \in Prop$ *such that* $t_1 - t < I$ *and* $\langle committer(c), r_1\rangle \in Owners$ **then**

set r_1 as the true origin of c;

$C \leftarrow C \cup c$;

$ToReview \leftarrow ToReview - \{\langle c, r, t\rangle\}$;

for *every tuple* $\langle c, r, t\rangle \in ToReview$ **do** // req. manual intervention

γ **if** c *is followed by a merge and the merge refers to an unknown repo* r_2 **then**

Make a local clone of r_2;

Process r_2 using section A of $Mining$;

$Owners \leftarrow Owners \cup owners(r_2)$;

if $committer(c)$ *is one of* $owners(r_2)$ *and* c *is seen in* r_2 **then**

set r_2 as the true origin of c;

$C \leftarrow C \cup c$;

$ToReview \leftarrow ToReview - \{\langle c, r, t\rangle\}$;

$R \leftarrow R \cup r_2$;

δ **else if** *there is other evidence that* $committer(c)$ *is an owner of* r *or* $committer(c)$ *made many commits to* r *that are never followed by somebody else's merges* **then**

$C \leftarrow C \cup c$;

$Owners \leftarrow Owners \cup \langle committer(c), r\rangle$;

$ToReview \leftarrow ToReview - \{\langle c, r, t\rangle\}$;

expect that (1) a short interval τ minimizes propagation before new commits are found and that (2) over time the number of newly discovered repositories will decrease.

We have implemented *continuousMining* for `git` as a set of perl scripts with a relational database (postgresql) as its backend. We have run it continuously on Linux since November 2011. Since then, we have identified 2.3 million different commits from 635 repositories (as of Sept 2013).

4 Empirical Study

In this section, we present an empirical study of how Linux uses `git` as a concrete case study of how D-VCSs are used in practice. Through this study, we achieve two goals: first, we demonstrate that *continuousMining* is effective at providing a richer picture on the use of `git` than *snapMining*, and second, we present an in-depth study of how `git` is used by Linux developers.

Thus, we ground our empirical study on the following three research questions:

- **RQ1.** Does *continuousMining* expose any bias in the project history recovered using *snapMining*?
- **RQ2.** What are the characteristics of the repositories in the Linux *Super-repository*?
- **RQ3.** How do the repositories of a *Super-repository* interact with each other and how do commits flow across them?

4.1 Setup

The goal of this empirical study is to compare the information recovered using the two mining methods: *snapMining* and *continuousMining*. The subject of this study are the commit activities of the Linux kernel developers during 2012.

4.1.1 snapMining

As the *blessed* repository, we use the `git` repository of Linus Torvalds. We made a clone of his repository on Jan 2, 2013. We extracted the metadata, and the patch of all commits performed in 2012.

4.1.2 continuousMining

We started mining the Linux *Super-repository* in Nov 9, 2011. The mining performed during 2011 was used to fine-tune the implementation of *continuousMining*, to gather a substantial part of all repositories, and to determine the mining interval τ.

4.1.3 Initializing the set of repositories

We seeded our list of repositories with those found in git.kernel.org on Nov 9, 2011 (193 repositories). We settled for a mining interval τ of 3 hrs, since a) it took approximately 2 hrs to complete a mining iteration, and b) only 1.4% of commits were propagated in such a time window. Considering that most public repositories of the Linux kernel are updated a median of once every 12 days, and very few were updated daily, it is unlikely that reducing τ would find many commits that are added to a public repository, and then deleted before a scan in that repo is performed. A shorter τ would have reduced the number of iterations where a commit had propagated before it was first discovered but this effect was already very small.

4.1.4 Crawling process

The growth of the number of crawled repositories can be seen in Figure 2. By Jan. 1, 2012 we were crawling 262 repositories and by the end of 2012 we were crawling 530 repositories. However, only 479 of these 530 repositories had

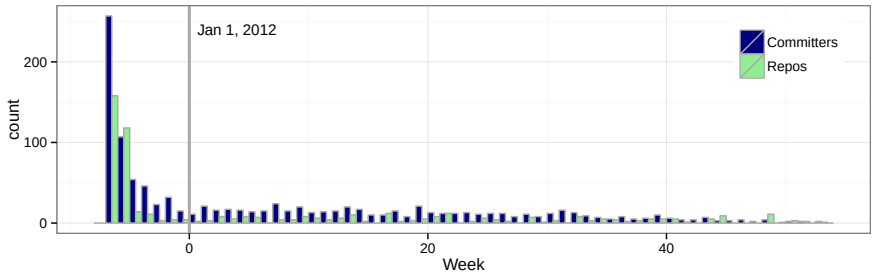


Fig. 2 Number of new repositories and committers discovered per week. Week 1 corresponds to the first week of 2012. The spike on the left hand side corresponds to the final weeks of 2011, the period during which we were discovering the majority of the repositories in the ecosystem.

at least one new commit during 2012. We completed 32,945 repository-crawls that found at least one new commit.

4.1.5 Reviewing commits and discovering new repositories

This step automatically fixed the origin of 1.4% of the commits (7,714). As described in Section 3, we incorrectly attributed these commits during *continuousMining*, because pulling changes from repositories is not transactional. Hence some commits were copied between repositories while we were doing our crawling. We manually processed 0.7% of the commits (3,925) that had originated in previously unknown repositories. Frequently, a scan would result in many new commits that were resolved at once. For example, on April 20, 2012 we had to manually process 852 commits, but all of them came from the same unknown repository. The output of this step included the list of owners of each repository. Figure 2 shows the growing number of committers that were identified. We estimate that the amount of work invested to do this manual analysis was approximately 50 hours during the first two weeks of mining, and 2-3 hours per week after that.

By January 2012 we were confident that we were mining most of the active and publicly accessible repositories in the Linux *Super-repository*, and we were tracking commits from the first public repository in which they were seen. As with *snapMining*, for each commit we recorded its metadata and any associated patches.

We observed that the same patch would appear in the *Super-repository* with different commit ids. For that reason, we computed a patch id using the following method: first, using '`git log --patch`' we extract the entire patch of the commit (it might contain changes to one or more files); second, we remove all index lines (index lines are used internally by `git` to indicate the SHA1 identifier of the exact files the patch applies to) and, from each hunk, the line numbers where the hunk is to be applied. For example, the patch:

```
index 6936e0a..f748cc8 100644
--- a/drivers/pnp/driver.c
+++ b/drivers/pnp/driver.c
@@ -197,6 +197,11 @@ static int pnp_bus_freeze(struct device *dev)
        return __pnp_bus_suspend(dev, PMSG_FREEZE);
 }
+static int pnp_bus_poweroff(struct device *dev)
...
```

becomes:

```
--- a/drivers/pnp/driver.c
+++ b/drivers/pnp/driver.c
@@-@@ static int pnp_bus_freeze(struct device *dev)
        return __pnp_bus_suspend(dev, PMSG_FREEZE);
 }
+static int pnp_bus_poweroff(struct device *dev)
...
```

Finally, we compute the SHA1 checksum of the resulting patch (all changes to all files in the commit). Patch ids allow us to uniquely identify a contribution (patch), even when it is committed multiple times and its commit id changes (for example due to rebasing or cherry picking). A merge commit usually does not contain a patch, unless there was a conflict in the branches that it merges. In the latter case, the merge commit's patch contains the changes needed to resolve the conflict. Hence, a commit might have zero or one patches associated with it.

4.2 Comparing the Information Recovered using *continuousMining* and *snapMining*

Table 2 Basic statistics performed on the commits from 2012, based on the two mining methods.

Metric	***cont.***	***snap***	**Ratio *cont./snap***
Active repositories	479	1	479
Unique commits	533,513	64,029	8.3
Unique non-merge commits	485,027	58,953	8.2
Unique merge commits	48,486	5,076	9.5
Unique patches (patch ids)	135,352	58,355	2.3
Unique author email addresses	5,646	3,434	1.6
Unique authors	4,575	2,883	1.6
Unique committer email addresses	1,185	283	4.2
Unique committers	1,058	245	4.3

For each commit, we record its metadata, the repository where it is created, whether it is a merge or a non-merge, its associated

patch and its patch id. The basic statistics of the mining processes of *snapMining* and *continuousMining* are reported in Table 2. As can be seen, **using *continuousMining* we observe 8.3 times more commits, and 2.3 times more patches in the *Super-repository* than in *blessed*.** Hence, a patch is found in an average of 3.6 commits (135,352 patches across 485,027 non-merge commits). The additional number of patches in *Super-repository* compared to *blessed* implies that there is a large amount of contributions that are not reflected in *blessed*. These contributions could be (1) rejected patches, (2) patches that were still in transit by the end of 2012 (end of our mining), or (3) development that is never expected to reach blessed (such as product line-type customizations). Rejected commits often keep on wandering around in at least one repository, making them hard to distinguish from commits that are still in transit. Between January 1st and Dec. 15th, 2013, only 931 (1.2%) of these patches had arrived at *blessed*.

With respect to the development team, **the number of different authors and committers in the *Super-repository* was, respectively, 1.6 and 4.3 times larger than those discovered using *snapMining*.** These individuals collaborate using 479 different repositories. There is a low percentage of committers who managed to have a patch accepted into *blessed* (23.2%). This implies that **76.8% of the committers observed in the *Super-repository* are not visible in *blessed*.** However, we observed that, of the 479 repositories, 360 (75.1%) had at least one commit contributed to *blessed*.

continuousMining uncovers significantly more activity and a larger ecosystem of developers than *snapMining*.

5 Results

The following subsections present the results for our research questions. For each research question, we discuss its motivation, the concrete findings in the Linux kernel project as well as the new opportunities for research that are opened up. While RQ1 focuses on bias introduced by *snapMining*, RQ2 and RQ3 highlight the richness of the data produced by *continuousMining*, since it enables new types of questions and analyses.

RQ1. Does *continuousMining* expose any bias in the project history recovered using *snapMining*?

Motivation

It is expected that by mining more repositories one would recover not just more complete data, but also a more accurate picture of the development process than by mining *blessed* only. However, one important question is if this

additional data exposes bias (i.e., significant differences) in the information recovered from *blessed* using *snapMining* compared to the information recovered using *continuousMining*? Such a bias might give an incorrect view of the development process. Our data collection showed that there were 2.3 times more new patches, and 8.3 times more commits in the *Super-repository* than in *blessed* (see Section 4.2). Without rebasing, each patch would correspond to exactly one commit. Hence, rebasing seems to be common, which means that the metadata of commits (commit message, author, committer, dates of authorship and commit) might change across different incarnations (commits) of the same patch. In other words, what is observable in *blessed* might be significantly different than what is actually happening.

Even though *blessed* provides an accurate account of which commits have been accepted into a project, its view of the development history is reactive: a commit only becomes visible in *blessed* once it has been integrated (Jiang et al, 2013), even though the commit might have been around in the *Super-repository* for months. For example, if one analyzes on a certain day the number of commits per week for each of the past weeks, one obtains a distribution D_1. If that night, the maintainer of *blessed* merges 1,000 new commits[4], and the next morning one generates a new distribution (D_2) of the number of commits per week for each of the past weeks, the two distributions can be substantially different, with certain periods in the past suddenly exhibiting a large number of commits. *continuousMining* on the other hand enables pro-active tracking of a commit (with respect to *blessed*) from the moment that it enters its first repository.

Findings:

RQ1.1 Regarding the relationship between commit ids and patches

At the beginning of our research we naively expected that every commit contained a different patch. However, when we discovered that the number of commits in the *Super-repository* was 8.3 more than in *blessed* we hypothesized it was because the same patch was occurring in different commits. As described in Section 4.2, the number of commits in the repo was 533,513, but we only observed 135,352 different patches. For this reason we decided to look at the distribution of the number of commits in which a given patch appears. The number is heavily skewed, with an average of 3.7 commits per patch in the *Super-repository* and 55% had one commit per patch. Table 3 shows the comparison between patches and commits seen in *blessed* and the *Super-repository*. Note that 482 patches appeared in at least two commits in *blessed*—we verified every one of these commits and they contained non-trivial patches. We believe that this is the result of cherry-picking: the same patch is integrated into different development branches (each resulting commit will have a different commit id) and when the second branch is merged into *blessed*, `git`

[4] During 2012, there were 19 days where Linus merged at least 1,000 commits on the same day.

recognizes that the patch has already been applied (when the first branch was merged), and ignores the patch in the second commit. Although more work is needed to understand what kind of patches undergoes such cherry-picking, the implication of this result is that **in a *Super-repository*, the commit id cannot be used to uniquely identify patches as they move across repositories (and their branches)**.

Observation #1*a*: In a *Super-repository*, the same patch can appear in different commits, making commit ids insufficient to track patches in the *Super-repository*.

Table 3 Comparison between patches and commits in *blessed* and the *Super-repository*.

	Blessed	**Super-repository**
Number of Non-merge commits	58,953	485,027
Number of patches	58,356	135,532
Ratio of patches to non-merges	98.9%	27.9%
Ratio of non-merges committed in 2012 that reached *blessed*		12.1%
Ratio of patches that reached *blessed*		43.1%

Since we track commits from their origin, we know when a commit in *blessed* contains a patch that was previously seen in another commit (that didn't reach *blessed*), hence we are in the position to measure how many commits in *blessed* have different commit metadata than the earlier commit version. If we assume that the original metadata is correct, the different commit metadata could introduce bias in the data that is mined from *blessed* using *snapMining*. Table 4 shows these commits, classified by the observed type of change in metadata. **37.9% of the non-merge commits found in *blessed* were later copies of an earlier commit that contains the same patch**. This means that more than one third of commits were re-commits of the same change, and hence, had their date of commit changed. For these commits, **the median difference in time between the first version of the commit and the one that arrived in *blessed* is 14.2 days**. In other words, for 18.9% of the commits in *blessed*, the actual commit date is skewed by at least 14 days. Furthermore, in 11.4% of the changes in *blessed*, the authorship date was changed (a median difference of 10.9 days). Our results show empirically that **commit date and even author date metadata in *blessed* is biased**, which means that relying on the metadata stored in *blessed* likely gives a partially-incorrect representation of what really happened.

RQ1.2 Regarding rebasing

Of these metadata changes, we suspected that basic rebasing of commits might be one of the most common causes. As discussed in Section 2.3.2, simple

Table 4 Breakdown of changes in the metadata of patches for commits in *blessed* that are copies of earlier commits containing those patches. Committer and author names have been unified to account for changes in email addresses.

	Count	Proportion
Date of commit	22,326	37.9%
Committer	7,330	12.4%
Date of authorship	6,725	11.4%
Author	203	0.3%
Commit message	13,323	22.6%
Commit message summary (subset of commit message)	2,634	4.5%

rebasing[5] occurs during a synchronization with another repository (`git pull`) when a locally-developed codeline is detached from its location and re-attached after the latest pulled commit in the same branch. We identify simple rebasing as follows: if during a scan of a repository, a commit is deleted from this repository, and, in the same scan, a new commit appears in the same repository with the same patch id as the deleted one, we mark this new commit as a simple rebase of the deleted commit[6].

In the *Super-repository*, 20% (27,856) of patches had been simply rebased at least once. This rebasing has the side effect that the commit-date of the pre-rebase commit is lost and the new commit has the commit-date of the pull. Such re-committing helps explain why sometimes developers appear to commit many changes in a very short period of time. For example, we observed that, during 2012, in thirteen occasions a developer committed more than 100 changes in one minute, whereas what actually happened was the automatic simple rebasing of those changes (during a `git pull`). Other common reasons for rebasing are modifying the commit message to improve the commit summary or to add *tested-by*, and *signed-off-by* fields. The large proportion of changes in metadata between the first commit containing a patch, and the commit that arrives at *blessed* with the same patch shows that, at least **in Linux, rebasing is a common activity that introduces substantial bias in the data that is mined from *blessed* using *snapMining***.

Observation #1*b*: *continuousMining* exposes substantial bias in the commit's date and committer name as recorded by *snapMining*.

RQ1.3 Regarding the arrival of commits at *blessed*

Commits need to move from the repositories where they are first seen to *blessed*. We wanted to explore any patterns in the arrival of the commits to *blessed*, To do so, we looked at the rate at which they arrived at *blessed* over time. Figure 3 shows the number of commits authored (green) and committed

[5] See *The Basic Rebase* in `http://git-scm.com/book/ch3-6.html`.

[6] Simple rebasing is usually performed automatically during a `git pull` operation with the option `--rebase`.

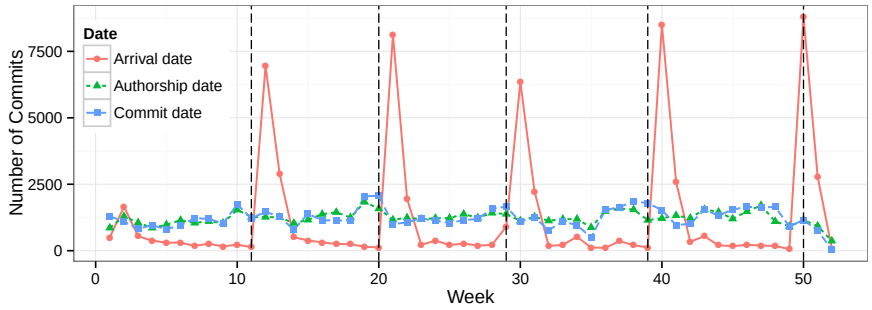


Fig. 3 Number of commits per week based on three criteria: authorship date, commit date, and date-of-arrival at *blessed* (only includes commits committed in 2012). Both authorship and commit date have a fairly uniform distribution; however, the date of arrival at *blessed* peaks on the week of a release or right after—which are depicted as vertical dashed lines.

(blue) per week in the *Super-repository*, and compares them to the number of commits arriving (red) into *blessed* per week. The overall authoring and committing of commits in the *Super-repository* is more or less uniformly distributed over the year, only slightly peaking around releases (vertical dashed lines). However, if we look at when these commits arrived at *blessed*, we notice that most of them are merged in the so-called *merge window*, i.e., the two weeks after a major release. After these two weeks, only bug fixes are permitted[7]. This pattern is not surprising: it similar to that of multi-branch development (using centralized version control systems), in which developers continue with their work (e.g. new features) while their completed work is being tested and integrated into a current release. The implication of this result is that the many repositories of Linux act as branches where new features and bug fixes are constantly being developed. Once these work-items are completed, they move into *blessed* in a well defined pattern that starts with the release of a version (and the beginning of the integration of the next one). The significant difference between centralized and distributed repositories in this regard is that in the centralized repo all the activity is visible at once, while in the distributed one, only the commits that have reached *blessed* are visible (unless one mines the rest of the repositories, as we have done in this research).

Table 5 shows the number of non-merge commits per release, for each merge window in 2012. This table shows that the number of commits that Linus Torvalds accepts during the merge window is significantly larger than during the bug-fixing window, while the total number of commits per release is relatively constant. **The arrival of commits at *blessed* shows a well-defined periodicity around releases, i.e., the Linux "merge window".**

Observation #1c: *continuousMining* can monitor the propagation of commits across repositories and identify when such commits arrive at *blessed*.

[7] `http://www.kernel.org/doc/Documentation/development-process/2.Process`

Table 5 Number of commits for the merge and bug-fixing windows of each release, and the percentage of commits that contain "fix" or "bug" (capitalization ignored) at least once in its entire commit message (denoted as *Msg.*) or their one-line summary (denoted as *Sum.*).

Rel	Merge	Incl. fix/bug		Bug-Fix.	Incl. fix/bug		Total
	Window	Msg.	Sum.	Window	Msg.	Sum.	
3.2	9,459	24.6%	15.8%	1,955	57.3%	42.9%	11,415
3.3	9,858	26.5%	17.5%	1,969	57.5%	41.2%	11,828
3.4	10,068	22.5%	13.3%	1,702	60.1%	42.1%	11,771
3.5	9,160	23.6%	15.1%	1,968	56.4%	41.9%	11,129
3.6	11,083	25.2%	15.4%	1,920	56.5%	42.4%	13,004
3.7	11,737	24.1%	15.4%				

RQ1.4 Regarding commits that arrive during the bug-fixing window

As we have seen, Linus Torvalds designates a bug-fixing window: any non-merge that arrives during this window is expected to be a bug-fix. This gives an opportunity to evaluate one common method to identify bug fixes. Researchers—such as Mockus and Votta (2000); Hassan (2008)—often assume that commits with the strings "bug" or "fix" in their commit message are defect fixing commits. Table 5 shows for every window (both merge and bug-fixing) the number of commits that contain the strings "bug" or "fix" in the summary (the one-liner that describes the commit), or the detailed commit message—the average length of the commit message in Linux was 10.25 lines. The summary is part of the commit message.

During the bug fixing windows only 2 in every 5 commits contain the strings "bug" or "fix" in their one-line summary, and 3 out of 5 in the detailed description of the commit message. Hence, those heuristics seem to miss a significant number of bug fixes.

We inspected the commit messages and summaries during the bug-fixing windows, and in many of these bug-fixes the commit message does not state explicitly that they fixed a bug (and by consequence neither does their commit summary), instead the commits mention the action that the developer took to fix it, e.g.: "md: Avoid write invalid address if read_seqretry returned true...", "md/raid5: Make sure we clear R5_Discard when discard is finished..." and "md: make sure everything is freed when dm-raid stops an array..".

However, the commit messages of commits that merge these fixes into *blessed* often contain *fix*. For example, the three commits mentioned above were merged together with the commit message "Merge tag 'md-3.7-fixes' of git://neil.brown.name/md ... Fixed a recently introduced deadlock". Other examples of such merge-commit messages are: "Pull SELinux fixes from ...", "Merge tag 'gfs2-fixes' of ...", "This batch of changes is mostly clean ups and small bug fixes." and "Pull MIPS fixes from ...". This implies that **the commit message of the merge that incorporates commits into *blessed* can be an important source of information when trying to identify bug fixes**. In fact, 88% of merges in the bug-fixing window contained "bug" or "fix" in their commit message (compared to only 48% in the merge window).

Because it is possible to identify the merge that merges a commit into *blessed*, it is possible to identify bug-fixing commits with information extracted from *blessed* using *snapMining*.

Observation #1*d*: *continuousMining* provides a pro-active view of a project's development history (with respect to the information stored in *snapMining*) and uncover patterns of development otherwise invisible.

Opportunities:

To understand a development process in full, one needs to use *continuousMining*, since it can uncover patterns of top-level integration that are otherwise lost in the recorded history of *blessed*. Furthermore, *continuousMining* provides the opportunity to explain some of the large differences in activity between the entire *Super-repository* and *blessed* (Table 5).

We observed that the number of accepted commits (including the number of bug-fixing commits) seems more or less stable for each release, with around 10,000 commits in the merge-window, and 1,900 commits in the bug-fixing window. Hence, it seems that the project has a kind of constant development capacity, however it is not clear why this is the case and if this is a long-term phenomenon. *ContinuousMining* of the *Super-repository* would provide researchers with development information that can help understand such an interesting phenomenon.

In addition, an advantage of identifying the arrival time of a commit in *blessed* is that one can tag with a high level of confidence the commits that arrive during the bug-fixing window as bug-fixes. For instance, during the bug-fixing windows, we found that not all commits in the bug-fixing window contain the words *fix* or *bug* (see Table 5). One could exploit this information to improve current models for automated identification of bug-fixing commits (e.g., Tian et al (2012)) and to identify potential for bias, in a way similar to that found in bug databases—as reported by Herzig et al (2013).

Another advantage of monitoring the arrival of commits at *blessed* is that we can categorize bug-fixing commits into those that are discovering during the development of the new-features (those accepted during the merge-window and that contain "bug" or fix") and those that are created to fix a bug that are found after integration has taken place. We know that it is important to discover defects as soon as possible. With this data we can add a new dimension to studies of defect-fixing activities that separates bugs-fixes into those two categories.

One of the most important lessons that we have uncovered in this research question is that the commit-date reflects the last time a patch was committed, and at least in Linux, 37.9% of commits in *blessed* have been recommitted (22.6% had their commit message reworked too). This means that, unless this phenomenon is taken into account, studies should not rely on this date. As shown in Table 4, these change in metadata of a patch from its first instance

to the one that arrives at *blessed* could be a potential threat to validity to studies that rely on the commit or authorship time, or the name of the committer. Especially the bias in author data is worrisome, since many studies have considered this data to be much more reliable than commit data. By tracing commits throughout the *Super-repository*, one can understand what prompts developers to change commit metadata like author names in the first place. The fact that commits change their id makes it difficult to map to their origin in the mailing list. A shift in dates could also increase the difficulty to map the bug fix to the bug-report. On top of that, it is not clear if commits in Linux are entangledKawrykow and Robillard (2011)

Rebasing and cherry-picking in D-VCSs are activities that have not been studied in detail. How does rebasing help integration? What prompts somebody to cherry-pick a particular change?

As we have observed, commits in *blessed* are successful ones. They have arrived at *blessed* from other repositories where they are expected to be tested and chosen for integration. When we study only the *blessed* repository of Linux, we are only studying commits that have gone through this vetting process. The entire *Super-repository* is similar to an enormous centralized repository where development happens in multiples branches (each branch in each repository would be equivalent to one branch in this centralized repo). Concentrating in *blessed* is similar to only studying the main integration branch of a software project (not even the older releases of Linux are found in *blessed*–as we will discuss in RQ2). Development in Linux is probably not that different from development in a large organization that does multi-branch development, as those described in Shihab et al (2012) and bird12. Future research should compare these two development models.

We can conclude that the answer to "**RQ1.** Does *continuousMining* expose any bias in the project history recovered using *snapMining*?" is **Yes**.

RQ2. What are the characteristics of the repositories in the Linux *Super-repository*?

Motivation:

Up until now, we treated the *Super-repository* as a collection of "equal" repositories, each of them serving up commits to *blessed*. However, the extremely low percentage (23.2%) of committers having commits accepted into *blessed* suggests that not all commits are intended to be sent towards *blessed*. Some repositories belong to automatic integration servers like linux-next[8], which "is intended to be a gathering point for the patches which are planned to be merged in the next development cycle" (Corbet, 2008b). Yet, other repositories contain experimental or niche features that do not benefit everyone.

[8] `git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next`

ContinuousMining allows us to study the different types of repositories and their development activities, and the interactions between them.

Findings:

RQ2.1 Regarding Activity

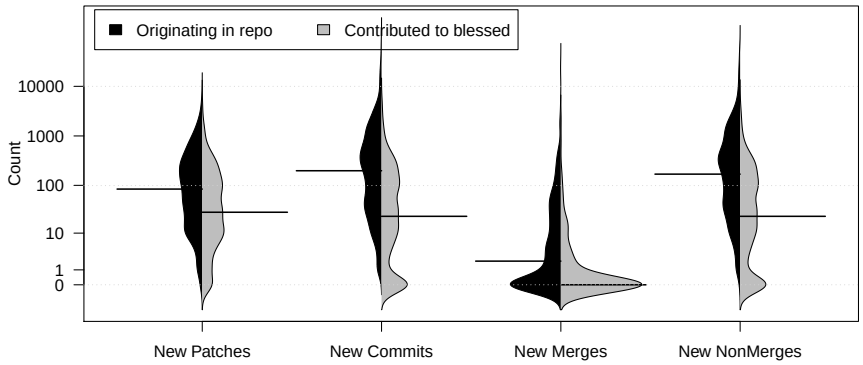


Fig. 4 These bean plots show the distribution of the number of different patches and number of different types of commits that originated in each of the 459 repositories, and how many of them were contributed to *blessed*. The left side (in black) is the total count for each type, while the right side (in grey) is the size of the subset that were contributed to *blessed*. The horizontal lines are the median values for each set.

We identified 459 different repositories that contributed at least one commit to the *Super-repository* during 2012. We observed a very wide variance in both their activity, and the way they contributed to blessed. To illustrate their differences both in terms of activity and their contributions to *blessed* we first identified how many patches and commits originated in each repository. For patches, we only considered patches that we first observe in 2012. For each patch that was first seen in the *Super-repository* in 2012, we record the repository where it first appeared. We also label each of these patches as contributed to blessed or not. We do the same for commits: we record the first repository where a commit is seen, and whether this commit is contributed to *blessed*. The Figure 4 shows using bean plots the distribution of the number of patches and commits originating from each of the repositories. Each sides of bean plot corresponds to a density distribution of the number of repositories have a given count. The left hand side of each bean plot (the black section) is the total number of patches or commits (we also show the distributions for the two types of commits: merges and non-merges). The right hand side (grey) is the number of them that were contributed to *blessed*. The horizontal lines represent the median number. For instance, the median number of patches that originated in a repository was 84, but only a median of 28 of these patches

reached *blessed*. This implies that most repositories had a good proportion of patches that did not reached *blessed*. The same applies to the number of commits (median of 198 new commits per repository; median of 23 contributed. Also, it can be seen that many repositories never contribute commits to *blessed*.

When commits are divided into merges and non-merges, it can be observed that many repositories never created any merges (41% of repositories had no merges, and in general a median of 2 merges per repository). It also shows that some repositories never contributed non-merges, implying that these repositories' activity never reached *blessed*.

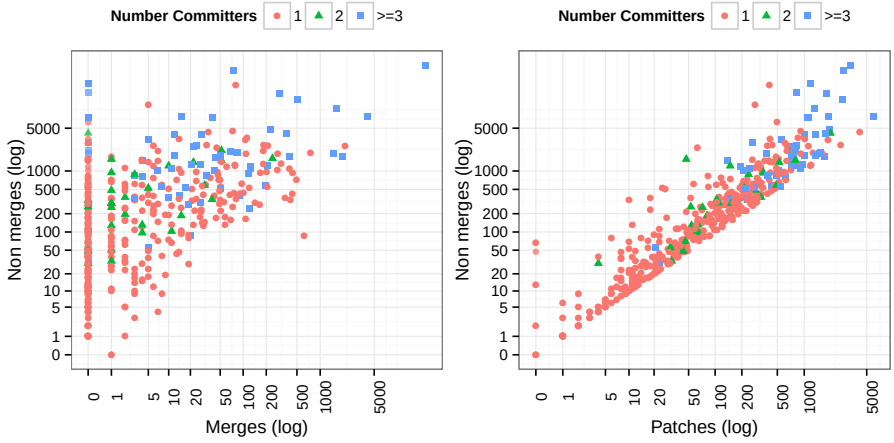


Fig. 5 The scatterplot on the left shows for each repository its number of non-merges and merges, while the one on the right compares non-merges to patches.

Figure 5 allows to explore the relationship between merges and non-merges, and non-merges and patches (a patch can exist in many different non-merges). As can be seen, there is a large proportion of repositories that do not have merges, which implies that they do not perform any integration work. Regarding patches to non-merges, we can observe that many repositories, regardless of their size, create new non-merges from previous ones (the result of rebasing or cherry-picking).

Observation #2*a*: The activity in the repositories varies widely, and in most repositories non-merges heavily outnumber merges. Many repositories do not have any merge commits, implying that they do not integrate other repositories' work.

RQ2.2 The Roles of Repositories

One would expect that most commits end in *blessed*. However, that is not the case. The right hand side of the bean plots in Figure 4 show the distributions

of the contributions to *blessed*. One noteworthy result is that **22% of the repositories did not contribute a single commit to *blessed*** (99 repositories). To further explore the contributions of repositories to *blessed*, and their role in the ecosystem, we computed, for each repository, the following metrics:

- **Ratio of patches contributed to *blessed***. Number of patches created in the repository divided by the number of patches that made it to *blessed*. This ratio gives us an indication of whether patches created in this repository are intended for *blessed* or not.
- **Ratio of patches to non-merge commits**. If this ratio is low, this implies that the repository is rebasing patches (from itself or somewhere else). A ratio of 1 implies that every patch is found in exactly one commit.
- **Ratio of patches to merge commits**. The lower this number, the more integration that the repository is doing. When this metric is undefined (the number of merge commits for a repository is zero), this repository is not doing any integration work.

Figure 6 uses these metrics to plot all the repositories. The X-axis corresponds to the total number of commits produced by the repository (a measure of its activity), while the Y-axis is the ratio of patches contributed to *blessed*. Using the median of each axis, the plot is divided into four quadrants. Each point is annotated in three ways: a) whether it is an official repository (*is the repo hosted on kernel.org?*), depicted by its shape; b) the ratio of patches to non-merge commits (*how much rebasing is there?*), depicted as its size (smaller means more rebasing); and c) the ratio of patches to merge commits (*how much integration is there?*), depicted as its colour (a warmer colour–closer to red–means more integration, while grey means that the repository had no merges).

The repositories in the top two quadrants have most of their patches contributed to *blessed*, i.e., they are **contributors** of patches. 15% of the repositories had all their patches contributed to *blessed*. The lower part of the right bottom quadrant contains active repositories that rarely contribute to *blessed*, hence they are **consumers** of commits from *blessed*. These repositories are summarized in Table 6, which shows the most active repositories with at most a 5% ratio of patches contributed to *blessed*. All of these repositories are product lines of Linux. For example, the Android-msm[9] repository hosted by Google contributed only 10 patches to *blessed*, yet it had a total activity of 11,922 commits.

The repositories depicted with a colour closer to blue represent those with a larger patch-to-non-merge-commit ratio, corresponding to repositories dedicated primarily to produce code (they are **producers** of patches), while more red colours imply integration work (they are mainly **integrators**). Note how, as the repository increases its churn (in terms of number of commits—horizontal axis), it moves from being a producer to being an integrator. We

[9] `https://android.googlesource.com/kernel/msm`

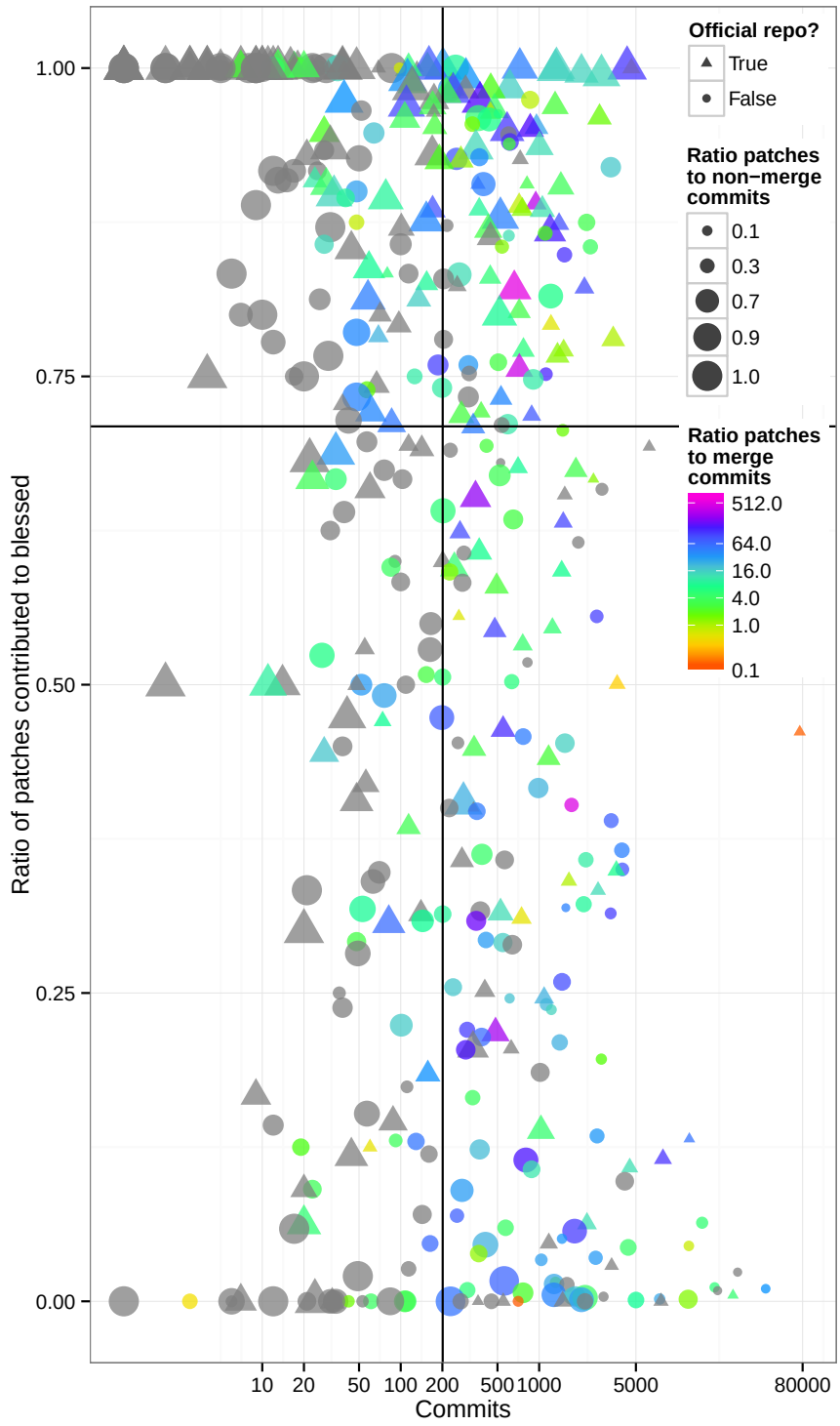


Fig. 6 Scatterplot showing, for each repository, the number of commits in the repository versus the ratio of contributed patches to *blessed*. The size of the points represents the ratio of patches to non-merge commits, and the colour represents the ratio of patches to merge commits (grey indicates that the repository had no merge commit).

Table 6 Most active repositories with a ratio of contributed patches of at most 5%. All these repositories are product-lines of Linux or older releases still being maintained (e.g., linux-stable).

Address	#New Commits	#New Patch.	#Contr. Patch.	Prop. Patches Contr.
linaro.org/.../andygreen/kernel-tilt	43,290	2743	28	1.0%
ubuntu.com/.../**ubuntu-precise**	27,141	1184	28	2.3%
kernel.org/.../rt/**linux-stable-rt**	25,129	407	2	0.4%
ubuntu.com/.../**ubuntu-quantal**	19,459	809	7	0.8%
linaro.org/.../linux-tb-ux500	18,508	1765	20	1.1%
linaro.org/../linux-linaro-tracking	12,058	1291	58	4.4%
android.googlesource.com/.../**msm**	11,922	5911	10	0.1%
kernel.org/.../stable/**linux-stable**	7,591	1116	0	0.0%
github.com/vstehle/linux	7,421	1079	2	0.1%
denx.de/linux-denx	5,005	1915	2	0.1%
opensuse.org/kernel	4,399	1716	75	4.3%

can observe a large number of repositories in grey, which didn't have a single merge. These repositories are pure **producers** of patches.

Observation #2*b*: Repositories serve different purposes. They can be divided into **contributors**—those that send most of their commits to *blessed*—and **consumers**—those that take commits from the *Super-repository* but hardly contribute to *blessed*. Contributors can be further divided into **producers** of patches and **integrators** of patches.

RQ2.3 On committers and authors

In terms of persons who committed to a repository, 79.7% of the repositories have only one committer (and 87.1% at most two). This person is usually responsible for integrating the work of various authors (the median ratio of authors to committers is 10). If only one person commits to a repository, this implies that **in Linux, commits are moved from one repository to another via email patches and pull operations** that are performed by the owner of the destination repository. Hence, **most repositories are controlled by one person who has the ultimate power to decide what makes it into that repository**. This is best illustrated by the *blessed* repository, where only Linus Torvalds can commit to it. Many repositories (10% of the total) had only one author and one committer. These repositories never integrated somebody else's work (not even via email patches from other developers). Similarly, most committers only work on one repository: 82% of them (865) only committed to one repository, and a further another 10% to two. The maximum was one person who committed to was nine repositories.

Observation #2*c*: Most of the repositories are personal, i.e., only their owner can commit to them; and most committers commit to only one repository.

We identified three groups of contributors that performed significant activity, but did not contribute any commits to blessed:

- **Integration-testers** are people like Stephen Rothwell, who maintain the linux-next integration repository (most of his commits are created automatically as part of the integration testing he is responsible for).
- **Experimentors** are people offering less mainstream versions of the kernel, such as G. Roeck who is the maintainer of Linux Staging (stand-alone drivers and filesystems that are not ready to be merged into *blessed*) and T. Gleixner who maintains Linux RT (real-time version of the kernel).
- **Product-line maintainers** are those who are customizing the code in *blessed* for specific uses in product-line repositories (such as those employed by Android, Ubuntu and Linaro).

It is important to mention that in the case of integration-testers and experimentors, their repositories were usually personal. This means that we can propagate the committers' roles to their repository, i.e., **we can also classify repositories into whether they do integration testing or experimental work**. Product-line maintainers typically work in repositories that are shared among many developers.

On the other hand the most active developers that contributed to *blessed* can be divided into two groups (non-mutually exclusive): **producers** and **integrators**.

- **Producers** dedicate their efforts to create patches that end in *blessed*. Examples of producers are H. Verkuil and H. Hartleys (both maintainers of various drivers).
- **Integrators** are those that merge code on its way to *blessed*, and are responsible for the majority of the merges in *blessed*. In this category, we find Linus Torvalds (maintainer of *blessed*), and A. Bergman and O. Johannson (maintainers of the arm port of the kernel).

Because most committers commit to one repo only (see Observation 2c above), there is, for the majority of contributors, a one-to-one relationship between the role of the contributor and the role of the repository where he/she commits. Some developers, however, play both roles, and maintain both producer and consumer repos, most notably Greg Kroah-Hartman (arguably the second most important of contributor to the kernel), who maintains 7 producer repositories, including the USB and tty subsystems and one consumer, the older versions of the kernel (known as stable). We have added a note regarding this.

Observation #2d: Contributors to the kernel can be classified into different roles. Some are similar to those of repositories (such as **producers** and **integrators**). Other roles such as **integration testers**, **experimenters** and **product-line maintainers** are not visible in *blessed*.

5.4 Official versus Non-Official Repositories

We observed that official repositories behaved differently than non-official. Official repositories are located at *kernel.org*. In Figure 6, we depicted official and non-official repositories (triangles correspond to official repositories, and circles to non-official). One aspect in which they appear different was whether they contributed to *blessed* or not. To evaluate this hypothesis we computed the ratio of patches that originated in a repo and ended in *blessed*, compared to the total number of patches that originated in that repository.

As it can be seen in Figure 7, the official repositories are more likely to contribute patches to *blessed* than non-official (median of 85% vs 50%). This difference is statistically significant (Kolmogorov-Smirnov test with $D = 0.29$ and $p << 0.01$). Also many non-official repos tend not to contribute to *blessed*, and vice-versa (many official repositories contribute most of their patches). The most active of the non-official are usually maintained by organizations that distribute bundled versions of Linux (such as Android, Linaro, SUSE, and Ubuntu). These repositories seem to be using `git` as a way to maintain a customized version of Linux on top of *blessed* (their own product line of the kernel). Note that their patches could have been contributed via mailing lists, however it is well-known that a project like Android has had difficulties getting its changes integrated into *blessed* (Kroah-Hartman, 2010).

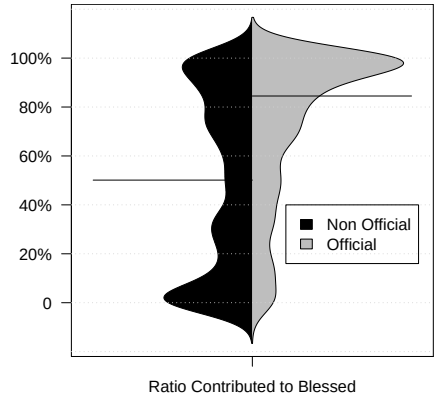


Fig. 7 Bean plot comparing the density of distributions of the ratio of patches contributed to *blessed* of official and non official repositories.

We can also observe that **there are some repositories in the official server that do not contribute to *blessed***. Four of these official repositories that do not contribute to Linux are Linux-stable-rt (the stable version of Real Time linux), Linux-stable (the stable version of the last released version of the kernel–meant to receive only bug-fixes); and the 3.2 (bwh/linux-3.2.y) and 2.6 (linux-2.6.32.y-drm33.z) versions of the kernel that still receive bug-fixes. Hence we can conclude that in Linux, **old releases are maintained**

in different `git` repositories, rather than the more traditional process of creating a branch in *blessed*. This explains why *blessed* does not have any branches.

Observation #2*e*: Official repositories are more likely to contribute their patches to *blessed*. Also, *blessed* does not host any branches; instead, other repositories are used for this purpose.

Opportunities:

Different repositories have different goals, each participating in its own way as part of the *Super-repository*. Because most repositories are owned by only one person, there is a symmetry in the role of the repository and its owner. Future work should concentrate on trying to understand the roles of repositories and their owners and how they work together. Future work should also look at the way that `git` appears to facilitate the work of those that package Linux as a product (e.g., Linux, Ubuntu, TI and Linaro) and the role that these stakeholders play in the ecosystem of repositories.

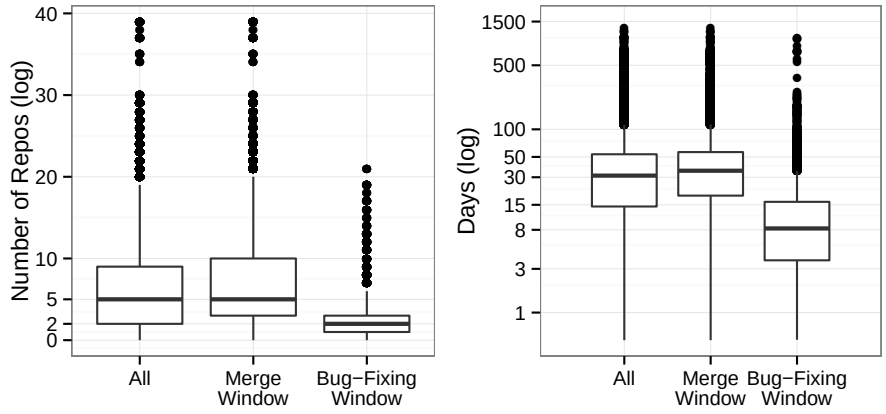
RQ3. How do the repositories of a *Super-repository* interact with each other and how do commits flow across them?

Motivation:

As we described for RQ1.3, commits (and the patches that they contain) need to move from their repository of origin to *blessed*. Since *continuous-Mining* monitors all repositories, it allows us to follow these commits (and their patches) as they propagate from their repository of origin throughout the *Super-repository*. The pattern of propagation of commits can uncover interesting development activities and practices about the integration process of a project, especially involving the hidden committers and types of repositories identified in RQ2.

Findings:

The median number of repositories visited by a commit is 5. Figure 8(a) shows the distribution of the number of repositories visited before *blessed* across all commits, as well as breaks this distribution down across merge and bug-fixing commits. This breakdown shows that bug-fixing window commits (i.e., bug fixes) only pass through a median of 2 repositories instead of 5. A Mann-Whitney test shows that this difference is statistically significant with $p \ll 0.01$. Of the intermediate repositories through which commits passed, some serve as integration hubs that merge thousands of commits, while others (like linux-next), help to test the commits before they are propagated further.



(a) By Repositories (b) By Days

Fig. 8 Number of repositories through which a non-merge commit propagates before it reaches *blessed*, and its latency (the difference between the time it is authored, and its arrival at *blessed*). Bug-fixes travel through fewer repositories and arrive faster at *blessed* than merge-window commits.

Figure 9 illustrates the propagation flow of commits. It shows 3,599 patches (i.e., commits) that Torvalds merged on one day of the merge window (December 12th, 2012—we only show one day because the graph would be too large otherwise), and the paths that these commits took to arrive at *blessed*. A yellow node represents repositories that produced the commits, and a green node integrators—those that merged somebody else's commits— but might have also produced commits. Of course, we could only show (and study) the flow of commits to public repositories, since the developers' private repositories are not visible. However, unless a patch is committed directly to *blessed* by Linus Torvalds (he committed 4.3% of patches this way, i.e., 2,530), commits have to move across one or more public repositories before they arrive at *blessed* (see Figure 1 on page 6).

Since most commits are not directly applied to *blessed*, there is a latency between the time at which they are first authored and the time at which they arrive to *blessed*. This latency is shown in Figure 8(b). The distribution of the latency between the commits accepted in the bug-fixing window (median of 7.5 days) and the merge-window (median of 34.2 days) is statistically different (Kolmogorov-Smirnov test with $D = 0.54$ and $p \ll 0.01$). In other words, during the merge-window, contributing repositories follow a hierarchical-like structure, with *blessed* as the root. This structure is flatter during the bug-fixing window in order to reduce the latency of the patch—and fix bugs faster.

Observation #3: Commits with new features take longer and visit more repositories in their way to *blessed* than bug-fixing commits. They visit a median of 5 repositories versus 2, and have a median latency of 34.2 days versus 7.5.

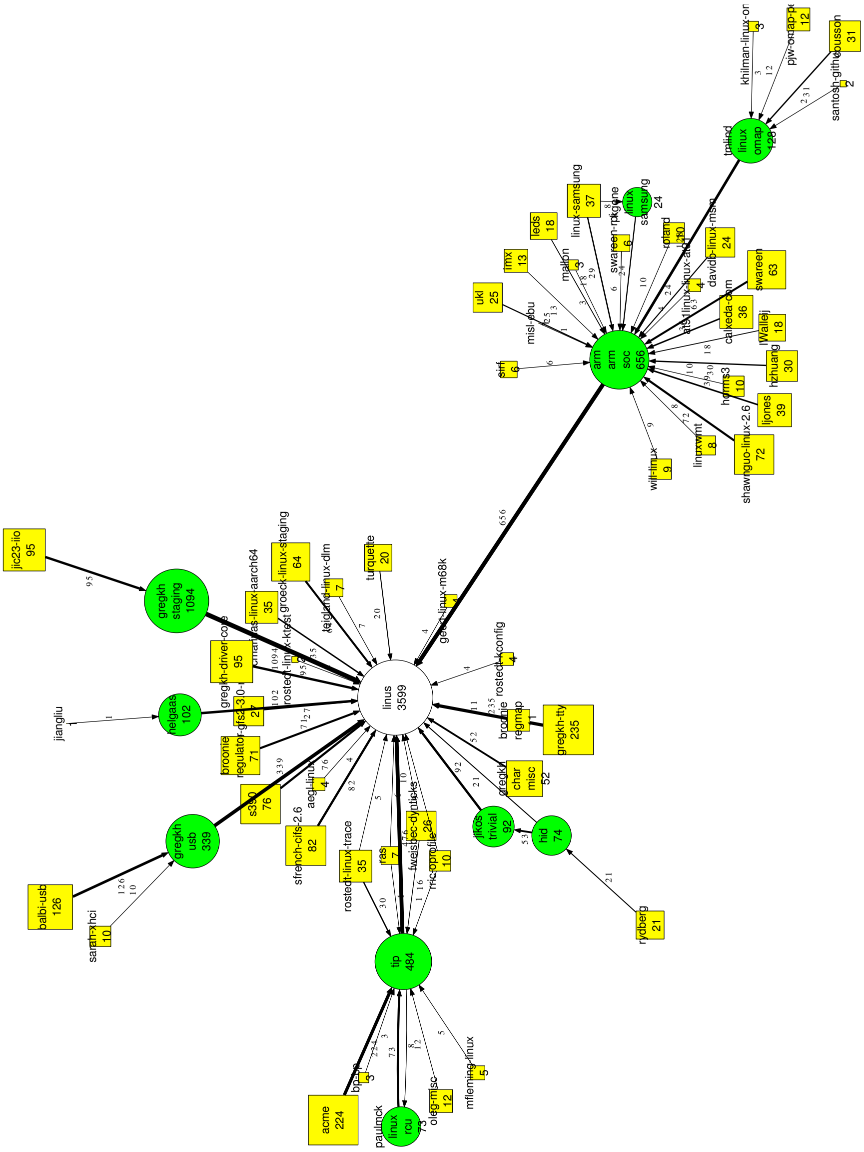


Fig. 9 This graph shows the repositories that commits visited on their way to *blessed* (Linus Torvalds' repository—depicted in white) that were merged into *blessed* on December 12th, 2013 (the busiest day of the year and part of a merge window). Yellow repositories only produced commits, while green ones merge other repositories. The edges are annotated with the number of commits that were merged in this path, and the repository indicates how many commits it contributed.

Opportunities:

Comparing the commits as they move across repositories will provide a new view on the integration effort that goes into developing Linux. Without *continuousMining*, such integration needs are impossible to measure or analyze. Furthermore, *continuousMining* allows us to study the different development processes followed for bug-fixing commits and merge-window commits.

6 *Hydraladder*, a dashboard to Track Commits across the Linux SuperRepository

As we have demonstrated (see *Observation* #1*a* on page 17), the commit ids of patches often change as they move from one repository to another. Within a *Super-repository*, the tracking of commits and the patches that they contain is a feature that neither `git` nor `git` hosting repositories support. For instance, **it is not possible to know, for a given patch (or a commit), the repositories where it has been, and where it has been deleted from.**

Older versions of the kernel that are still being maintained and product-line repositories (those that take commits from *blessed* in order to customize the kernel to their needs, such as Android, Linaro and Ubuntu) are particularly affected since they tend to cherry-pick changes as they see fit. As such, the resulting rebased commits have no link anymore with the original commits, making it impossible for product-line repositories to trace back the origin of the commits in their products, and for kernel developers to understand the impact of their changes on product-line projects.

In April 2013, we implemented and deployed *Hydraladder*, which is a dashboard that displays, for a given commit in the Linux *Super-repository*, the repositories where this commit has been, and its path to *blessed*. *Hydraladder* is located at `http://hydraladder.turingmachine.org/`. During the Linux Foundation Collaboration Summit 2013, we introduced this service to some Linux developers, who suggested two use-cases. The first was tracking commits that had arrived in *blessed* without first arriving in linux-next (i.e., which had not been integration-tested), and the second was cross-referencing of commits.

The first use-case is intended to help identify commits that might not have been thoroughly tested. Linux-next is used to test commits and to verify their integration before they arrive at *blessed* (Chapman (2011)). Every commit is expected to pass through Linux-next before it is integrated into *blessed*, in order to increase its chances of successful integration. Since we know when a commit arrives in any repository, this report is trivial to create thanks to the information retrieved with *continuousMining*. As commits are seen in *blessed*, a report is automatically generated listing the commits that have not been seen in Linux-next but have arrived at *blessed* (Linus Torvalds' repository).

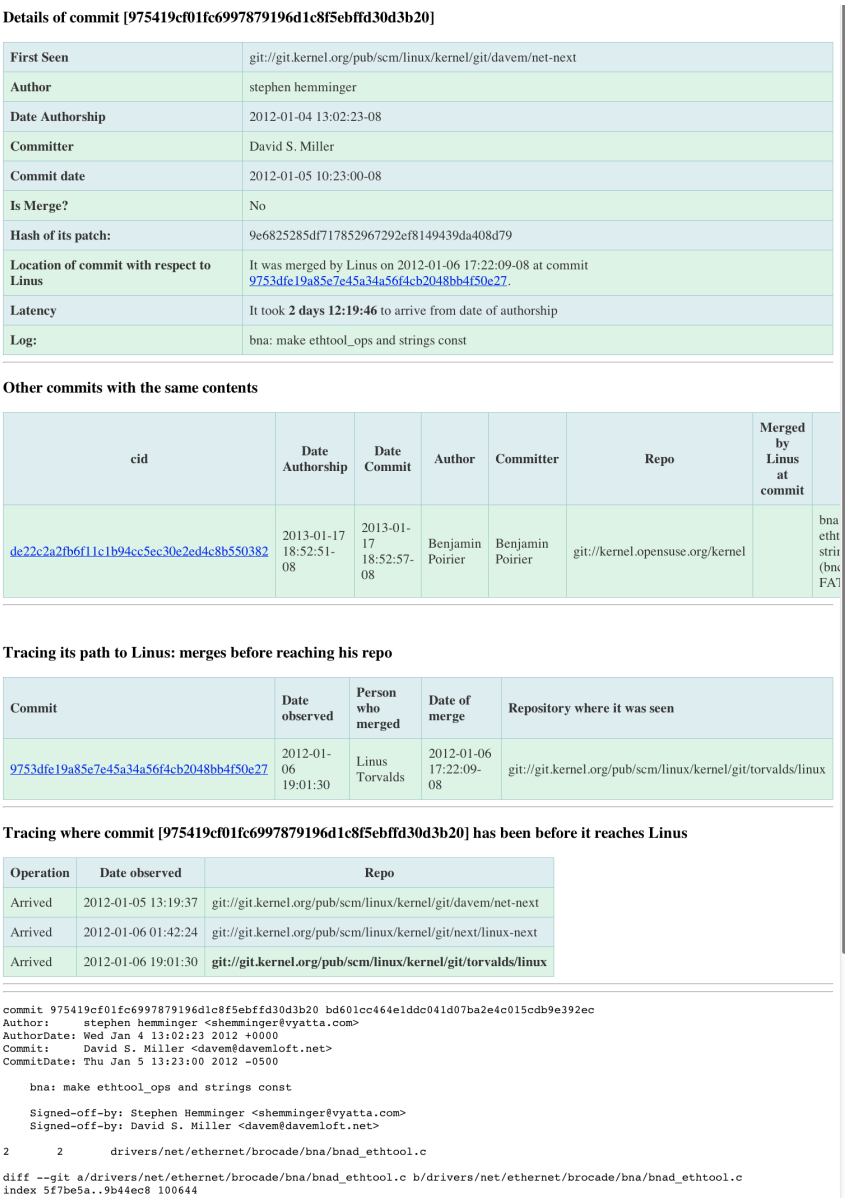
The second use case—cross-referencing of commits—has been shown to be a time consuming issue (Dhaliwal et al (2012)). Cross-referencing of commits can be divided into two types: a) cross-referencing commits that contain the

```
commit 70cb8bb0d365f0bc8b20fa67347caf9598a4674e
Author:     Kees Cook <keescook@chromium.org>
AuthorDate: Wed Jun 5 11:47:18 2013 -0700
Commit:     Luis Henriques <luis.henriques@canonical.com>
CommitDate: Fri Jun 14 10:18:47 2013 +0100

    x86: Fix typo in kexec register clearing

    commit c8a22d19dd238ede87aa0ac4f7dbea8da039b9c1 upstream.

    Fixes a typo in register clearing code. Thanks to PaX Team for fixing
    this originally, and James Troup for pointing it out.
[...]
commit 073028356cb7dd59b95a7a6be564946976118165
Author:     Kees Cook <keescook@chromium.org>
AuthorDate: Wed Jun 5 11:47:18 2013 -0700
Commit:     Steve Conklin <sconklin@canonical.com>
CommitDate: Thu Jun 20 13:36:00 2013 -0500

    x86: Fix typo in kexec register clearing

    BugLink: http://bugs.launchpad.net/bugs/1193029

    commit c8a22d19dd238ede87aa0ac4f7dbea8da039b9c1 upstream.

    Fixes a typo in register clearing code. Thanks to PaX Team for fixing
    this originally, and James Troup for pointing it out.
[...]
commit c8a22d19dd238ede87aa0ac4f7dbea8da039b9c1
Author:     Kees Cook <keescook@chromium.org>
AuthorDate: Wed Jun 5 11:47:18 2013 -0700
Commit:     H. Peter Anvin <hpa@linux.intel.com>
CommitDate: Wed Jun 12 15:16:18 2013 -0700

    x86: Fix typo in kexec register clearing

    Fixes a typo in register clearing code. Thanks to PaX Team for fixing
    this originally, and James Troup for pointing it out.
[...]
```

Fig. 10 Example of how Linux developers cross-reference commits across repositories. The top two commits, first seen in two Ubuntu repositories, reference the bottom commit (which was the only one integrated into *blessed*). The three commits contain the same patch.

same patch; and b) cross-referencing commits that are mentioned in the message of a commit. Cross-referencing commits that contain the same patch is trivial, since—thanks to the information retrieved with *continuousMining*—we keep a hash of the patch of every commit. For the second kind of cross-referencing of commits, we scan the message of every commit for commit ids and link those commit ids that contain the same patch. An example of this cross-referencing is shown in Figure 10, where two commits in the Ubuntu repositories reference a commit that was integrated into *blessed*. In this particular case, *Hydraladder* is capable of discovering these relationships automatically, since these commits contain exactly the same patch.

For a given commit C, *Hydraladder* shows a) other commits with the same contents (patch), b) commits that include C's commit id in their commit message, and c) commits that C lists in its message. Figure 11 shows a sample output of *Hydraladder*.



Details of commit [975419cf01fc6997879196d1c8f5ebffd30d3b20]

First Seen	git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next
Author	stephen hemminger
Date Authorship	2012-01-04 13:02:23-08
Committer	David S. Miller
Commit date	2012-01-05 10:23:00-08
Is Merge?	No
Hash of its patch:	9e6825285df717852967292ef8149439da408d79
Location of commit with respect to Linus	It was merged by Linus on 2012-01-06 17:22:09-08 at commit 9753dfe19a85e7e45a34a56f4cb2048bb4f50e27.
Latency	It took **2 days 12:19:46** to arrive from date of authorship
Log:	bna: make ethtool_ops and strings const

Other commits with the same contents

cid	Date Authorship	Date Commit	Author	Committer	Repo	Merged by Linus at commit	
de22c2a2fb6f11c1b94cc5ec30e2ed4c8b550382	2013-01-17 18:52:51-08	2013-01-17 18:52:57-08	Benjamin Poirier	Benjamin Poirier	git://kernel.opensuse.org/kernel		bna etht strir (bn FA

Tracing its path to Linus: merges before reaching his repo

Commit	Date observed	Person who merged	Date of merge	Repository where it was seen
9753dfe19a85e7e45a34a56f4cb2048bb4f50e27	2012-01-06 19:01:30	Linus Torvalds	2012-01-06 17:22:09-08	git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux

Tracing where commit [975419cf01fc6997879196d1c8f5ebffd30d3b20] has been before it reaches Linus

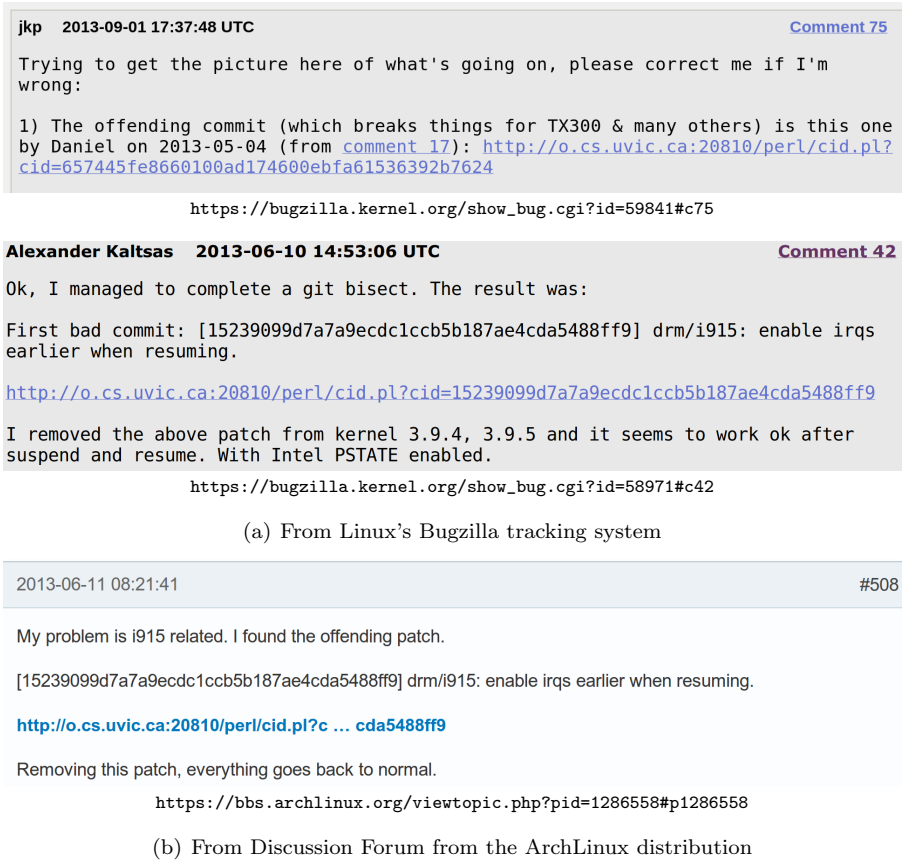
Operation	Date observed	Repo
Arrived	2012-01-05 13:19:37	git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next
Arrived	2012-01-06 01:42:24	git://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next
Arrived	2012-01-06 19:01:30	**git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux**

```
commit 975419cf01fc6997879196d1c8f5ebffd30d3b20 bd601cc464e1ddc041d07ba2e4c015cdb9e392ec
Author:     stephen hemminger <shemminger@vyatta.com>
AuthorDate: Wed Jan 4 13:02:23 2012 +0000
Commit:     David S. Miller <davem@davemloft.net>
CommitDate: Thu Jan 5 13:23:00 2012 -0500

    bna: make ethtool_ops and strings const

    Signed-off-by: Stephen Hemminger <shemminger@vyatta.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>

2       2       drivers/net/ethernet/brocade/bna/bnad_ethtool.c

diff --git a/drivers/net/ethernet/brocade/bna/bnad_ethtool.c b/drivers/net/ethernet/brocade/bna/bnad_ethtool.c
index 5f7be5a..9b44ec8 100644
```

Fig. 11 Sample of the output of *Hydraladder*.

Hydraladder has been running since April 2013. As of September 2013, it tracks commits in 633 repositories, has seen 2.3 million different commit ids, and followed 109 million commit-propagations. Figure 12 demonstrates that *Hydraladder* (as url `http:/o.cs.uvic.ca:20810/`) is being used by actual Linux developers to cross-reference commits across repositories, and to com-

plement their comments in the bug tracking system (`bugzilla.kernel.org`). A full user study of *Hydraladder* is outside the scope of this paper.



(a) From Linux's Bugzilla tracking system

(b) From Discussion Forum from the ArchLinux distribution

Fig. 12 Three examples of Linux developers using *Hydraladder*.

7 Further Discussion and Future Work

This section discusses findings and insights from our study regarding `git`, *continuousMining*, Linux and D-VCSs.

7.1 Regarding `git`

As the popularity of `git` continues to grow, it is important to understand how `git` is used, along with its advantages and disadvantages.

The distributed nature of version control systems like `git` implies a lack of a central authority. This feature becomes a two-edged sword: on the one hand,

it eases the participation of outsiders in the development, and facilitates—as is done in the Linux kernel—the creation of product lines by third parties. On the other hand, development lacks traceability (even for "official" members of the team). This lack of traceability has two important outcomes.

First, it is impossible to know what clones have been made of a repository (and by extension what clones have been made of clones). Cloning a repository is a feature available to any person with read access to the repository, contrary to C-VCSs, which can be configured in such a way that users can read and commit changes, but cannot copy the repository itself.

Second, at any given time, it is practically impossible to know what is the set of all the commits in the *Super-repository* of a project. There will always be repositories on the computers of developers that are only readable to their owners (such as their laptops), hence, there will always be commits that are only known to their authors. Even in public repositories, `git` does not currently support any centralized logging features. While this does not seem to be an issue for open source developers, it might be to commercial users.

Hosting services for `git` are providing a pro-active logging mechanism if, and only if, the operations are performed via their interface. However, this logging and cross-referencing is currently incomplete, and it is hard to force people to do all their D-VCS manipulations through a specific interface.

Alternatively, `git` can be enhanced by optionally adding centralized logging features. Any time a developer performs an operation, a log would be created in a centralized location. This log could be lightweight (storing only metadata) or comprehensive (storing also the patches). Such a log would allow an organization to trace development as it happens. `bitkeeper`, a commercial D-VCS, is already supporting this feature. However, such a feature impacts developers in different ways, such as their ability to work offline or to freely experiment without anyone seeing their mistakes. Future research should look at a compromise between ease-of-use and the ability for organizations to manage who can clone their code-bases (and potentially remotely disable access to them). If this work would succeed, it could obviate the need for *continuousMining*.

7.2 Regarding *continuousMining*

The information stored in D-VCSs changes over time, either because it is occluded by new information (e.g., we don't know where a commit originated because, after a while, every repository has such a commit), and because information is altered by its users (e.g., commits are rebased). Until recently, researchers have assumed that information in many software repositories was immutable (emails are not changed once they are archived, and commits could not be altered once they were performed) and almost instantaneous (e.g., an email is archived seconds after it is distributed by a mailing list, and a commit is recorded the moment that it is created). However, D-VCSs demonstrate

that this is not always the case. Researchers who are trying to understand how these development tools are used should take into consideration these issues.

As discussed above, traceability in D-VCSs can be addressed in two ways: a pro-active manner, in which the D-VCSs implement a centralized logging mechanism, or in a re-active manner, in which activity is identified and logged after it has occurred.

ContinuousMining is a re-active logging mechanism. It tries to discover changes *after* they have occurred by scanning the changes present in each repository. One of the challenges of *continuousMining* is that, in general, some repositories might be readable only by their owners (e.g., those hosted on laptops). Similarly, *continuousMining* has to discover the locations of the repositories. These challenges can be mitigated, for example if a team is always expected to use a given location for its repositories—such as Github or Gitorious). Private repositories could be easily scanned if they are hosted in a networked file system (a requirement imposed by tools such as Crystal that help identify and prevent conflicts in merges (Brun et al, 2011)). However, these problems cannot be completely avoided, especially when we cannot change the behaviour of the subjects of our research. For instance, to study the use of `git` by the Linux kernel, we cannot use any of these methods. The repositories of Linux are spread around the world, in a multitude of servers, and it will be impossible to instrument the tools that they use to gather information logs of their activities.

Note that we do not consider *continuousMining* a permanent solution to the need of centralized logging features in `git`. Rather, we believe that it is a temporary solution that helps address this problem, especially for researchers who are interested in studying how `git` is used in practice. As the results herein demonstrate, the use of *continuousMining* can be valuable both to researchers and practitioners.

Linux is an extreme case, as not many software projects have the same size or number of developers as Linux. It is possible that in small projects with a handful of developers, `git` is used as a C-VCS, and hence, the difference in the information recovered between *continuousMining* and *snapMining* will be minimal or non-existent. However, we would not know if this is the case unless we use *continuousMining*. If some projects use `git` as a C-VCS, then it is worth exploring why. Therefore, future research should look at how different projects use `git`.

7.3 Regarding Linux

Linux is a project where thousands of developers from many different organizations collaborate together. Linux is also customized by many different organizations to reach the consumer (such as Android, Ubuntu, Linaro and Busybox). By being the backbone through which code flows between participants, `git` plays a crucial role in this ecosystem.

The flow of code in Linux is analogous to the flow of sand in an hourglass, as depicted in Figure 7.3. At the center of the hourglass is Linus Torvalds, who is the maintainer of *blessed*. He decides what makes it into the official Linux release. The contributors to Linux create patches that eventually reach him. After a patch has arrived at Linus, it is then propagated to the product-line repositories where the kernel might be further customized.

We have observed that sometimes code will also flow from the bottom of the hourglass to the top. In that case, the product-line maintainers will author patches that are sent to integrators who commit these patches to their repositories, and send them to *blessed*. Sometimes, product line repositories will cherry-pick commits from producers before they arrive in *blessed*. However, in general, commits will flow in a hierarchical graph from producers of patches to *blessed* via integrators, and from *blessed* to product lines via product line maintainers. Linux is a rich ecosystem with many participants each taking different roles. This ecosystem deserves to be studied in more detail in future research.

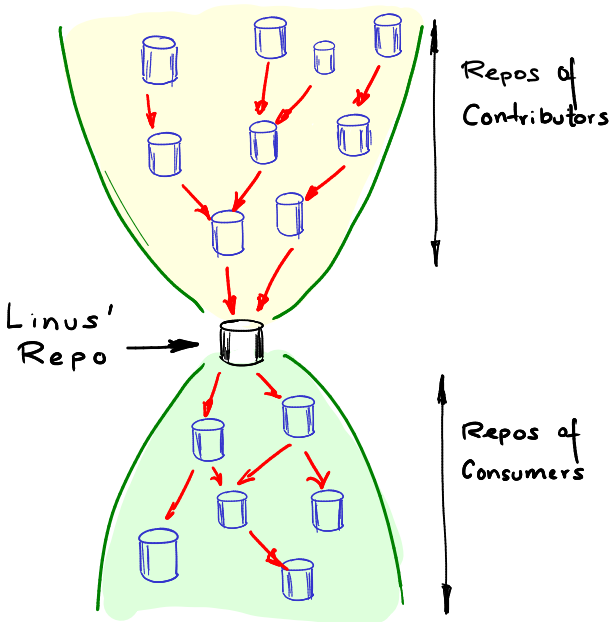


Fig. 13 The flow of contributions in Linux is similar to an hourglass. Contributions flow until they reach Linus Torvalds' repository. After that, they flow to product-line repositories.

7.4 The D-VCSs paradox

One of the great benefits of D-VCSs is providing version-control features to anybody who wants to participate in the development, without having to be approved and given an account (as it is the case with C-VCSs). Hence, **D-VCSs democratize the participation in software development, and open the door for anybody to participate.** The only requirement is that it should be possible to clone and pull changes from at least one repository in the system. Cloning a D-VCS is equivalent to the creation of a branch in a C-VCS.

However, the paradox is that, while **in C-VCSs anybody with an account can commit changes to the repository**, and hence contribute to the development, **in D-VCSs the committers to *blessed* and other repositories that feed into it decide what contributions are committed into *blessed***. In other words, in D-VCSs anybody is able to create their own patches with the help of version control features, but only a handful of developers control what makes it into the official product. **`git` provides the freedom to participate in the development, but `git` does not provide the freedom to contribute.**

In Linux, more than 79.7% of repositories have only one committer. The hierarchical organization of these committers will control what code is merged into *blessed*. At the center of this is Linus Torvalds' repository. Anybody can clone Linus' repository, and add features to it, but Linus will ultimately decide what he integrates into Linux. **D-VCSs allow universal participation, but also allow ultimate centralization of the integration process.**

8 Threats to Validity

With respect to construct validity, our empirical study depends on the accuracy of *continuousMining* and its implementation. The algorithms and their implementation were fine-tuned and debugged during two months (November to December 2011), when we spent a considerable amount of time manually verifying the accuracy of the recorded data. We have unified 5,962 email addresses into 4,766 individuals (and manually verified them) to make sure that we did not over-count developers.

Regarding internal validity, our study naïvely ignores how commits (and their patches) evolve. It is very likely that many of the patches that do not arrive at *blessed* are previous versions of the patches that do. In-depth analysis should link all related commits together to reconstruct the correct lifecycle and path of a commit. However, we leave such detailed analysis for future work.

Another aspect that might affect internal validity is that we compared the activity in *blessed* against the activity in the entire *Super-repository*, even though some of this activity will never make it to *blessed*. The 25% of repositories that never contributed to *blessed* (pure consumers) do not represent part of the ecosystem of repositories that create the Linux kernel (as found in Linus

Torvalds' repository). However, one can argue that because these repositories are closer to what users know as Linux, the activity of these repositories also represents an important part of the activity of the Linux kernel development team. Hence, the commits in the entire *Super-repository* reflect better the activity of the team than the commits found only in *blessed*. The activity of consumers (including the development of their own features) and how they integrate features from blessed is an important area that needs to be studied.

It is also important to mention that our method is only capable of observing activity in public repositories that we know about. There will always be repositories (including many personal ones) that cannot be accessed (Bird et al, 2009b). For example, we know the addresses of some corporate Linux repositories that are behind firewalls, and hence we cannot reach them. For this reason, we believe that we are underestimating the size of the *Super-repository* of Linux and its activity. Nonetheless, our findings regarding the differences between the data recorded in *blessed* and the *Super-repository* of Linux should be considered as a starting-point for further research. In particular, we believe that studies that try to understand the development practices of Linux should look into these differences.

The main threat to external validity is that Linux is perhaps the largest project that has ever used a D-VCS (Barr et al, 2012). Therefore, what we observed in Linux might not be happening in smaller projects (e.g., in projects using `git` as a centralized repository). Similarly, some of the information that we have observed might only be created in `git` and not in other D-VCSs. For this reason, studies in other projects are needed. We are currently mining the history of five more projects, including one that uses `mercurial`, in an attempt to address this issue.

To minimize the effects of these threats we will publish–and make publicly available–the data that we have gathered using *continuousMining*[10].

9 Related Work

Bird et al. (Bird et al, 2009b) analyzed the `git` D-VCS technology from the perspective of mining software repositories researchers. They identified 9 promises and 7 perils of the then brand new D-VCS technology. `git` promised a larger and richer set of development data, and was able to distinguish between patch authors and committers. However, a large amount of data was only recorded in the private logs of developers (impossible to obtain), and the three challenges discussed in this paper were also identified as major perils. Whereas these perils were explored conceptually by Bird et al., *continuousMining* is able to quantify the actual impact of these perils, which indeed turn out to be a threat.

The domain of software integration is the closest related to this paper. Brun et al. (Brun et al, 2011) studied merge conflicts, i.e., errors happening

[10] Please contact the first author for information regarding access to this huge amount of data.

when changes in one branch are merged into another branch. In particular, they note that integration is risky because the person merging a commit is not the developer herself, usually many commits are merged at once, and merging typically happens a long time after development. Some of these factors might explain why rebasing and cherry-picking are performed so heavily in the Linux kernel project. Shihab et al. (Shihab et al, 2012) established an empirical link between the post-release quality of two proprietary projects and the branching structure of their repository. Especially misalignment between branching and the organizational structure plays a major role in software quality. This is why collaborative open source projects like the Linux kernel tend to opt for D-VCSs. Finally, Bird et al. (Bird and Zimmermann, 2012) propose techniques to improve the speed with which changes move through the branches of a repository. These techniques have been applied primarily on the branches of *blessed*, however, given the scale of the *Super-repository* of a project like the Linux kernel, similar techniques could be applied there as well. Note that the software integration phase typically follows the patch reviewing process. This process has been studied in depth, in particular to identify the kinds of patches that tend to be accepted (Baysal et al, 2012; Jiang et al, 2013; Rigby et al, 2008; Weissgerber et al, 2008). Similar studies could be performed on software integration.

The act of obtaining finer-grained development history by using *continuousMining* instead of *snapMining* has strong similarities with the use of IDE interaction histories like the edit history of specific lines in a file or the order of opening files. Zou et al. (Zou and Godfrey, 2006) used an Eclipse plugin to monitor the files viewed during maintenance. Other researchers built more powerful tools like SpyWare (Robbes and Lanza, 2007), or used the interaction histories provided by Mylyn or the Eclipse Usage Data Collector to study the impact on software quality of specific developer interactions (Zhang et al, 2012), predict future bugs (Lee et al, 2011), or measure how developers resume interrupted tasks (Parnin and Rugaber, 2011). Developer actions inside an IDE represent the ultimate level of mining, since they even capture uncommitted changes.

Similar to bias caused by rebasing and cherry-picking, bias has also been studied in other domains, like bug reports. Antoniol et al. (Antoniol et al, 2008) found that many bug reports do not report bugs, but enhancements or other issues. This tag bias skews models towards subsystems with proportionally more bug reports. Bird et al. (Bird et al, 2009a) studied linkage bias. The higher the severity of a bug, the lower the chance that a bug report is linked to the source code files that fixed the bug. Similarly, bug reports written by experienced developers are more likely to be linked to source code than those by less experienced developers. This linkage bias skews the models towards more severe bug reports written by more experienced developers. Even perfectly-linked bug reports are biased towards experience (Nguyen et al, 2010), hence noise-resistant models and noise removal techniques have been built to deal with this bias (Kim et al, 2011).

Hydraladder was motivated by the ability of `bitkeeper` to do centralized logging. `bitkeeper` can be deployed such that any commit to any repository of a *Super-repository* is automatically logged to a centralized location. This logging records the metadata and the patch of a commit, at the moment it is created. *Hydraladder*, in contrast, builds this information by regularly querying repositories to try to discover new commits that happened since the last query. `bitkeeper`'s approach is more powerful, since the client connects to the server at every event, while *continuousMining* (and by extension *Hydraladder*) has to be able to query the repositories (something that is not always possible) to read their histories, and will never be as comprehensive as `bitkeeper`'s logging. Unless `git` supports centralized logging like `bitkeeper` (either in its core implementation or by instrumenting it), methods that reconstruct events by analyzing history logs (such as *continuousMining*) are the only current alternative to do so, although they will always provide only a partial view of the activity of the entire *Super-repository*.

10 Conclusions

D-VCSs are becoming more prevalent. Their working model promotes the creation of satellite repositories where developers can work on features without having to propagate their change-sets to a centralized repository to get the benefits of version control (as C-VCSs do).

Mining the *blessed* repository of a project will always show only a subset of the activity of its developers, with possibly biased metadata. We have demonstrated that by using *continuousMining* (i.e., periodically mining the repositories that are used by the development team), we can uncover new activities that were before invisible. We expect that this new information will foster new research that looks into the benefits and potential drawbacks of using D-VCSs. In particular, we have demonstrated that Linux is a rich ecosystem that spawns beyond what is observable in Linus Torvalds' repository, a view that we consolidated into an hourglass model. We also expect that other researchers will be able for the first time to look into these phenomena, in order to help us understand the keys towards successful development and integration of source code changes in large-scale open source projects like the Linux kernel.

References

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proc. of the 2008 Conf. of the Center for Advanced Studies on Collaborative research: meeting of minds (CASCON), pp 23:304–23:318

Barr ET, Bird C, Rigby PC, Hindle A, German DM, Devanbu P (2012) Cohesive and isolated development with branches. In: Proc. of the 15th intl. conf. on Fundamental Approaches to Software Engineering (FASE), pp 316–331

Baysal O, Holmes R, Godfrey MW (2012) Mining usage data and development artifacts. In: Proc. of the 9th IEEE working conf. on Mining Software Repositories (MSR), pp 98–107

Bird C, Zimmermann T (2012) Assessing the value of branches with what-if analysis. In: Proc. of the ACM SIGSOFT 20th intl. symp. on the Foundations of Software Engineering (FSE), pp 45:1–45:11

Bird C, Gourley A, Devanbu PT, Gertz M, Swaminathan A (2006) Mining email social networks. In: MSR, pp 137–143

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009a) Fair and balanced?: bias in bug-fix datasets. In: Proc. of the the 7th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE), pp 121–130

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009b) The promises and perils of mining git. In: MSR '09: Proc. of the 6th Int. Working Conf. on Mining Software Repositories, pp 1–10

Black Duck Inc (2013) Tools: Compare Repositories. `http://www.ohloh.net/repositories/compare`

Brun Y, Holmes R, Ernst MD, Notkin D (2011) Proactive detection of collaboration conflicts. In: Proc. of Foundations of Software Engineering (FSE), pp 168–178

Chacon S (2009) Pro Git. Apress

Chapman D (2011) A Guide To The Kernel Development Process. `http://www.linuxfoundation.org/content/1-guide-kernel-development-process`

Corbet J (2005) The kernel and BitKeeper part ways. `http://lwn.net/Articles/130746/`

Corbet J (2008a) How to participate in the linux community. `http://ldn.linuxfoundation.org/book/how-participate-linux-community`

Corbet J (2008b) Linux-Next and Patch Management Process. `http://lwn.net/Articles/269120/`

Corbet J, Kroah-Hartman G, McPherson A (2013) Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. `http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013`

Dhaliwal T, Khomh F, Zou Y, Hassan AE (2012) Recovering commit dependencies for selective code integration in software product lines. In: ICSM, pp 202–211

Foundation E (2012) Eclipse community survey. `http://www.eclipse.org/org/press-release/20120608_eclipsesurvey2012.php`

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp 345–355

Hassan AE (2008) Automated classification of change messages in open source projects. In: SAC, pp 837–841

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: 35th International Conference on Software Engineering, ICSE '13, pp 392–401

Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast?: case study on the linux kernel. In: MSR, pp 101–110

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 92–101

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: ICSE '11: Proceedings of the 33th International Conference On Software Engineering, pp 351–360

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proc. of the 33rd Intl. Conf. on Software Engineering (ICSE), pp 481–490

Kroah-Hartman G (2010) Android and the linux kernel community. `http://www.kroah.com/log/linux/android-kernel-problems.html`

Lee T, Nam J, Han D, Kim S, In HP (2011) Micro interaction metrics for defect prediction. In: Proc. of the 19th ACM SIGSOFT symp. and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE), pp 311–321

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: ICSM, pp 120–130

Nguyen T, Adams B, Hassan AE (2010) A case study of bias in bug-fix datasets. In: Proc. of the 17th Working Conf. on Reverse Engineering (WCRE), pp 259–268

Parnin C, Rugaber S (2011) Resumption strategies for interrupted programming tasks. Software Quality Control 19(1):5–34

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the apache server. In: ICSE '08: Proc. of the 30th Int. Conf. on Soft. Eng., pp 541–550

Robbes R, Lanza M (2007) Characterizing and understanding development sessions. In: Proc. of the 15th IEEE Intl. Conf. on Program Comprehension (ICPC), pp 155–166

Shihab E, Bird C, Zimmermann T (2012) The effect of branching strategies on software quality. In: Proc. of the Intl. Symp. on Empirical Software Engineering and Measurement (ESEM), pp 301–310

Tian Y, Lawall J, Lo D (2012) Identifying linux bug fixing patches. In: Proc. of the 2012 Intl. Conf. on Software Engineering (ICSE), pp 386–396

Weissgerber P, Neu D, Diehl S (2008) Small patches get in! In: Proc. of the intl. working conf. on Mining Software Repositories (MSR), pp 67–76

Zhang F, Khomh F, Zou Y, Hassan AE (2012) An empirical study of the effect of file editing patterns on software quality. In: Proc. of the 19th Working Conf. on Reverse Engineering (WCRE), pp 456–465

Zou L, Godfrey MW (2006) An industrial case study of program artifacts viewed during maintenance tasks. In: Proc. of the 13th Working Conf. on Reverse Engineering (WCRE), pp 71–82