A Dataset of the Activity of the `git` *Super-repository* of Linux in 2012

Daniel M. German
University of Victoria
Email: dmg@uvic.ca

Bram Adams
École Polytechnique de Montréal
Email: bram@cs.queensu.ca

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

***Abstract*—This dataset documents the activity in the public portion of the `git` *Super-repository* of the Linux kernel during 2012. In a distributed version control system, such as `git`, the *Super-repository* is the collection of all the repositories (repos) used for development. In such a *Super-repository*, some repos will be accessible only by their owners (they are private, and are located in places that are unreachable to other users) while others are available to other members of the team. The latter public repositories are used as avenues through which commits flow from one developer to another. During the last six weeks of 2011, we proceeded to automatically discover the public portion of the *Super-repository* of Linux. Then, in 2012, every 3 hrs, each of these public repositories was queried to see what new commits it had and what commits had disappeared from it using a process we call *continuous mining*. This resulted in the identification of 533,513 different commits across 451 different public repositories and how they propagated through the Linux *Super-repository*, including the repository of Linus Torvalds (i.e., the main repository of the Linux kernel). This information could help us understand how kernel contributors use `git`, how they collaborate and how commits are integrated into the Linux kernel and into the repositories of organizations that distribute the kernel.**

This dataset is at http://turingmachine.org/2015/linuxGit

I. Introduction

A team that uses a distributed version control system (D-VCS) must have at least one public repo (in this context, public means a repo that is readable to at least one more member of the team) and every developer must have at least one local repo (usually private, i.e., not readable by any other developer). We refer to the set of all repositories of a team as the *Super-repository* of the project. When the *Super-repository* includes only one public repo, a *Super-repository* acts like a centralized version control system. All the commits flow from the local repos to the public one and vice-versa, at the request of the owner of the local repository.

In practice, a team that uses a D-VCS will have one or more public repos and each developer will have one or more private repos. One of the public repos is designated as the *blessed* repo of the project, which one would find the most up-to-date branch of development. Other public repos are used to exchange commits between each other, side-passing, if necessary the *blessed* repo of a project. In general, the entire *Super-repository* will never be fully visible to anybody. Many repos will be private and live in locations that are only accessible to their owners. Other repos might be only accessible to a subset of the team (e.g. in an intranet).

Linux is a large, successful software development project. Just in 2012, we identified 4,575 developers improving it. Linus Torvalds, its leader developer also developed `git`. While `git` is becoming a popular D-VCS, it was originally built to satisfy the requirements that Linux had. Hence, one would expect Linux to be one of the projects (if not the project) that exploits the most the features of `git`. This implies that it is worthwhile to understand how the Linux kernel uses `git` and what impact `git` has in the development process of the kernel. To fully understand how `git` is used by Linux, one would need to know how repos interact with each other and how commits are moved by developers from one repo to another. Such a study faces a plethora of challenges:

- The private repos in the *Super-repository* are unavailable to others.
- `git` has no centralized logging mechanism that documents who is creating a new repo and its location.
- At any given time, there is no information that documents what repos form the public portion of the *Super-repository* of Linux. While several servers host Linux kernel repos (such as kernel.org and GitHub), many others are spread around the world on servers of different organizations.
- The *Super-repository* continuously evolves. Over time, repos are created, destroyed, and moved.
- Neither repos nor commits record information about where commits were created or which repos they have passed through. Given two different branches in two different repos, once these two repos merge the changes from the other, the two branches are indistinguishable from each other. Merge commits (if they are created) might hint to the origin of the commits they have merged, but this information is not always recorded or it might be overridden.

To overcome these challenges, we have developed a method to mine the *Super-repository* of Linux, which we call *continuousMining*. Using *continuousMining*, we mined during 2012 the public portion of this *Super-repository*. This paper documents the resulting data, including:

- The URI of 451 public repos that contributed commits to the *Super-repository*. Every 3 hours, we scanned each repo looking for changes (new commits or deleted commits). We performed 31,336 repo-scans where the repo

had changed (an average of 70 scans per repo in 2012) and retrieved the corresponding changes.

- We identified 533,513 commits that were created in 2012 (485,027 non-merges and 48,486 merges). To put them into context, if one were to mine, at the end of 2012, the *blessed* repo of Linux (*blessed* for short), one would have found only 64,029 (8.3%) of them. The remaining 91.7% of commits did not reached *blessed*.
- The 533k commits were authored by 4,575 different individuals (using 5,541 different email addresses) and committed by 1,058 different individuals (using 1,172 different email addresses).
- The 533k commits contained 135,532 different patches.
- We identified 56 million commit propagations. A commit propagation is an event in which a commit is seen for the first time in a repo or disappears from it.

II. Description of the Data

In Linux, the repo of Linus Torvalds is the *blessed* repo of the kernel development. His repo serves as the destination where commits are expected to flow. However, his repo only tracks the successful code. Features that are being currently developed, or that do not make it to the kernel will never be seen in *blessed*. The main reason is that the repos around *blessed* serve the same purposes as branches in a centralized version control system: developers do their work in their local repos (their own personal branches) and only when it is ready, it starts to move towards *blessed*.

Blessed is only writable by Linus Torvalds. In Linux, commits move from the personal repos of their creators to *blessed* using a combination of email patches, pushes and pulls. A typical commit will be created in a personal repo, then emailed as a patch to a person responsible for integration (`git` keeps track of the metadata of the commit). Another alternative is for the creator to push her changes to her public repo (e.g., in github) from where the integrator can pull the changes into her private repo. This integrator will repeat the process: she will push the commit (along many others) to her public repo, and issue a pull-request to the next integrator (in the path to *blessed*). If the commit is deemed worth it, it will eventually reach *blessed* (Linus will pull the changes from the integrator public repo into this own private repo, and then push these changes to *blessed*).

Unfortunately, *blessed* contains no information about these interactions between repos. The only trace of these interactions are merge-commits (i.e., commits that combines the work of one or more branches into another branch), but not every merge results in a merge commit (e.g., fast-forward commits) and sometimes the log of the merge commits does not document that the commit performed a merge. Even if there is a merge commit, this merge commit only documents an actual merge operation, and does not record every single repo in the path from its creation to *blessed* (this path usually consists of fast-forward commits).

To fully understand how Linux uses `git`, we need to mine the entire ecosystem of repos. However, we will never have access to the private repos of developers. Fortunately, because the kernel uses pull-requests to move commits between integrators, almost every developer has have a publicly accessible repo where she can share commits with other developers. If we were to mine all these public repos of the team, we can get a better picture of how integration is done in the kernel. Furthermore, as mentioned before, once a commit moves from one repo to another, we cannot tell in which repo it appeared first, hence the propagation of commits between repos needs to be dealt with afterwards.

To address these issues, we have developed a method of mining D-VCS repos called *continuousMining*. *continuousMining* queries repos ar a certain frequency to identify commits that have appeared or disappeared since the last query. The details of the method, including its implementation, can be found in [2]. In that paper, we demonstrated that *continuousMining* is superior to querying the *blessed* repo at one time, since it is capable of observing code as it moves across repos, documenting when commits are rebased (a common operation in the kernel) and recording code that is not yet in the *blessed* repo (see Peril 4 in [1]).

To our knowledge, research on Linux has always used a single snapshot of the *blessed* repo of Linux. GhTorrent [3] continuously mines Github looking for new changes, but it does not record propagation of commits (when a commit appears in a repo, whether the commit was seen before in a different repo, and when). Hence, we are the first to document how commits propagate in a D-VCS.

III. Methodology: How the Data was Gathered

We started *continuousMining* on the Linux kernel in Nov 2011. By January 1st, 2012 we were mining 262 repos. This number grew to 530 by the end of the year. Every 3 hrs, for each of these repos: 1) we would synchronize our copy of the repo; 2) create a log of all commits (in all branches of the repo) and compare it to the previous iteration's log; 3) label any commit that was not in the previous iteration as "New", and every commit in the previous log no longer in the current one as "Deleted". [2] documents this process in detail. This dataset concentrates on commits created during 2012 (but also contains older commits).

For each repo, we needed to determine who its committers were. This was mostly a manual process. When we observed a new commit in a repo R, if the commit was not from a known committer to R, then there were two possibilities: R had a new committer (we searched the Web and mailing lists for evidence of this), or the commit originated in another repo S. If it originated in S, then either we were already mining S or not. To determine if S was a known repo, we looked at the every known repo T that the committer was allowed to write to. If the commit was found also in T during the same time window, then $S = T$ and we assumed that the commit originated in S and had propagated from S to R in the last three hours. Otherwise we tried to find S (which was not currently known) using any merge information in the repo, searching the mailing list and using the Web. We feel confident that for every commit

that reached blessed in 2012, we have properly documented its origin repo. This process is documented in [2].

IV. Data Schema

Name	Description
Commits	Metadata of commits.
Logs	Log message of commits
FilesMod	Metadata of files modified in commits.
CommitsBlessed	Commits found in *blessed* at the end of 2012.
Commits2012	Commits committed in 2012.
Merges	Commits that merged one or more branches.
Repos	Repositories mined.
Owner	Committers of a given repo.
Aliases	Unified names of developers and the email addresses they use (active in 2012).
PathToBlessed	The path in the DAG of *blessed* that describes how a commit reached *blessed*.
RepoProp	Propagation of commits: when a commit appears or disappears from a given repo.

TABLE I
Tables in the schema and their description.

The main entities of this dataset and their relationships can be described as follows: there are *developers* who have created *commits* in their public *repos*. A given *developer* (identified by his or her *uniname*) has one or more *email addresses* to identify oneself as the committer or author of a given *commit*. *Commits* propagate through *repos* from their *repo* of origin. Finally, *commits* flow from their *repo* of origin to other repos and ultimately into *blessed* via *merges*. A subset of *commits* have found their way to *blessed* (by the end of 2012–we call them *commitsBlessed*). Any *repo* has zero or more documented *committers* (we call them *owners*) who are the only developers who can commit to it. The schema of the database, depicted in Figure 1 documents these entities and relationships and Table I describes the purpose of each table.

A. Propagations

The table *repoprop* is composed of attributes *cid*, *repo*, *seen*, *op* (either 'N' or 'D') and *origin*. If *origin* is false a commit *cid* is either added (*op* 'N') or deleted (*op* 'D') from a given *repo* at date/time *seen*. If *origin* is true then this was the very first scan of a repo (*op* is 'N' for all the commits in this scan). Repos were scanned every 3 hrs. We choose a period of 3 hrs because it was long enough to complete a scan (most scans took between one and two hours, depending on network traffic) and most repos didn't change during this period. Because we could not stop activity in the repos during the scan, but a scan was not an atomic operation. In few instances new commits had propagated to more than one repo between scans, and we resolved this manually (see [2] for details).

B. Path to Blessed

The information collected in *repoprop* is similar to the propagation of a disease. We know where a commit originated, and, at every snapshot (at 3hrs intervals) we know other repos that also had the commit (or that deleted it). However, when a new repos receives it, we do not know for certain which repo it received it from. Given a set R consisting of two or more repos that had the commit c at snapshop t_i and a new repo $S \notin R$ that has c as t_{i+1}, S could have received c for any repo in R. For this reason, we combined information from *repoprop* and the directed acyclic graph (DAG) of the commits found in *blessed*. Because we know when commits arrive to a repo, we can convert the DAG into a tree, where each node (commit) has only one successor and only one merge into *blessed*. This tree is documented in table *PathToBlessed*: for any commit *cid*, its successor is *mnext*; its next successor merge is *mnextmerge* and it is eventually merged into blessed at commit *mcidlinus* on date *mwhen* (*mnextmerge* and *mcidlinus* can be the same). If the commit was committed directly into blessed (by Linus), *mcidlinus* is null. Figure 2 illustrates the use of this information. It shows the tree of commits that were merged into blessed at merge commit 5ede3ceb7b2c2843e153a1803edbdc8c56655950.

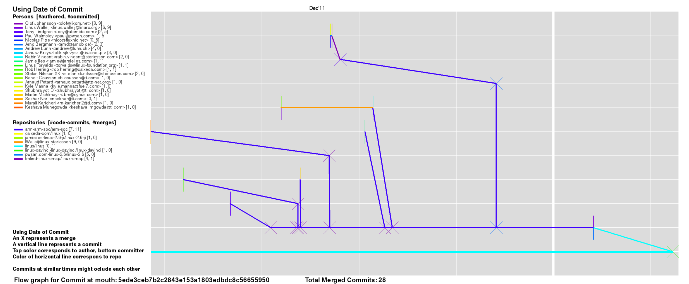

Fig. 2. Tree formed by the 28 commits merged into blessed at merge-commit 5ede3ceb7b2c2843e153a1803edbdc8c56655950. Each horizontal line represents a repo, Points marked with X are merges.

C. Patches in Commits

We found that as commits move throughout the *Super-repository*, they change commit-id [2]. This is because re-basing and changing the metadata of a commit are frequent operations used by Linux developers. For this reason we extracted the patch of every commit (`git log -patch`) and removed line-number context information; then we computed its hash, which we call the *codecontents* of the commit. While we observed 485k different non-merge commits in 2012, there were only 135k different patches. On average, the same patch appears in 2.3 different non-merge commits.

V. Threats to Validity

This dataset documents events that happened in 2012. While it contains events before 2012, these are incomplete. We used the period before 2012 to debug and calibrate our algorithms. For example, it contains repos that were inactive during 2012, and some propagations between repos before 2012.

Unfortunately some parts of this dataset are not reproducible. Once commits have propagated between two repos, it can be impossible to know where the commit originated. If a commit is rebased, it is very likely that the original commit is lost forever (unless the predecessor commit propagated to another repo; yet, it would be hard to know if this commit was the source of the rebased commit).

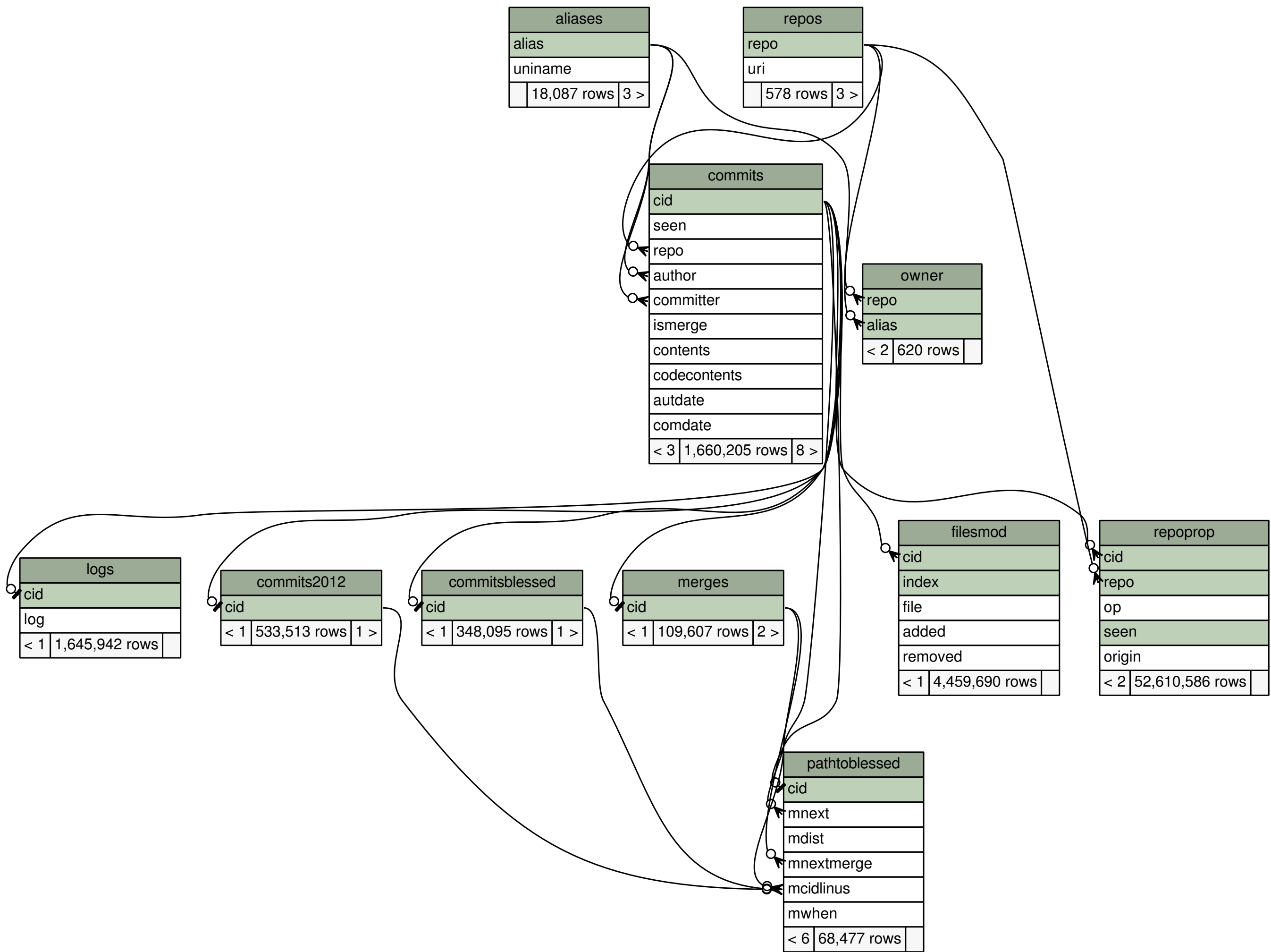

Fig. 1. Schema of the database. Shaded fields are the primary key of the table. To facilitate understanding, we have added foreign key constraints. The numbers under a table schema correspond to the number of tuples in it. The database contains 578 repositories, but only 451 contributed at least one new commit to the Linux *Super-repository* during 2012.

With regards to the data collection, we have done our best to manually verify it. We do not claim that we uncovered all the public repos that were active in 2012. We manually verified the source of all commits that reached *blessed* and were created in 2012. The unification of developers who committed or authored a commit in 2012 was done manually. We did not unify committers before (in that case, their *uniname* is null).

The window of 3hrs between the scan of a repo could have been too long. It is possible that, in between two scans, a public repo could have been updated—e.g., a commit is added— and in the next update such a commit has been deleted. In such a situation, the commit will not be recorded by us. We believe that although possible, such cases are unlikely, especially because the average time between updates of a repo was 5 days. When a commit propagated from its repo of origin to another repo between scans (i.e., the new commit is found in two new repos) we manually looked at the commit to determine its true origin.

Our dataset records all repos that we knew about and had access too. In some cases, we knew of the existence of repos but were not able to reach them (e.g., they were behind a firewall). Because we concentrated on repos that produced commits to *blessed* (as described above, we were diligent to find the source of commits that reached *blessed*) it is possible that we missed some repos that are never contributed back to the kernel (such as those of organizations that distribute linux versions) or those who had work in progress.

REFERENCES

[1] Christian Bird, Peter C Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *MSR '09: Proc. of the 6th Int. Working Conf. on Mining Software Repositories*, pages 1–10, 2009.

[2] Daniel M. German, Bram Adams, and Ahmed E Hassan. Continuously mining the use of distributed version control systems: an empirical study of how Linux uses git. *Journal of Empirical Software Engineering*, To appear.

[3] Georgios Gousios. The GHTorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR*, pages 233–236, 2013.