Making Sense Of Git In A Legal Context

Armijn Hemel,[a] *Shane Coughlan* [b]

(a) Owner, Tjaldur Software Governance Solutions;
(b) OpenChain Project Director, Linux Foundation.

DOI: 10.5033/ifosslr.v9i1.121

Abstract
The Git revision control system does not enforce correctness of data but instead is reliant on correct inputs for correct outcomes. Git records potential authorship rather than copyright ownership and this means that an additional process layer is needed to ensure fidelity and accuracy of data. The core implication is that the "git blame" tool does not show potential authorship with enough granularity to allow users make clear decisions, and additional review is required to determine potential authors of code contained in any Git repository.

Keywords
Law; information technology; Free and Open Source Software; Git, Version Control

A substantial amount of open source software development is conducted using the Git revision control system[1] (hereafter "Git"). Git has had a substantial impact on the development landscape over the last 12 years, primarily through increasing the pace of development by moving from a centralized source code versioning system to a decentralized approach. This has many benefits from a technical standpoint, but it also has side effects that are may have adverse legal consequences.

This article explores some of the legal issues that may arise from the use of Git, and raises a few questions to allow thoughtful consideration regarding future enforcement or legal disputes, when information obtained from Git may play a role. This is particularly important given that at least one license compliance dispute in Germany made use of Git logs as the mechanism for establishing proof of authorship.[2] Ultimately the purpose of this article is to provide a thoughtful discussion of how systems like Git would work in a legal context, and how the information contained in Git repositories can shed light on – or create – legal questions.

Background

The Git system was initially developed by Linus Torvalds as a replacement for the proprietary BitKeeper program in 2005.[3] BitKeeper is a decentralized version control system that was used by part of the Linux kernel development community when more traditional development workflows

1 Git is free and open source software licensed under GPLv2, and can be downloaded at https://git-scm.com/.
2 *See Hellwig v. VMWare Global Inc.,* File no: 310 O 89/15, Hamburg District Court (Jul. 8, 2016); English translation available at: http://bombadil.infradead.org/~hch/vmware/Judgment_2016-07-08.pdf
3 The Linux Foundation, "10 Years of Git: An Interview with Git Creator Linus Torvalds" (April 6, 2015): http://www.linuxfoundation.org/blog/10-years-of-git-an-interview-with-git-creator-linus-torvalds/

became problematic and started suffering from scaling issues in the context of the Linux kernel.[4]

Andrew Tridgell, a Linux developer also know for his contributions to the Samba project, wanted to interface with BitKeeper and reverse engineered the BitKeeper protocol.[5] This activity was badly received by the BitKeeper developers. BitKeeper, which at that point was closed source software, could be licensed for free, if the licensed parties agreed to certain legal restrictions – including not working on competing solutions. Mr Tridgell's actions were deemed a violation of the license and the Bitkeeper developers revoked the BitKeeper license for the entire Linux kernel project.[6]

Linus Torvalds created Git in response to this challenge, and Git has subsequently become the number one choice of version control for many developers.[7] Git has also facilitated the creation of new companies offering hosting services such as GitHub and GitLab.[8]

Architecture

There are a few concepts in Git that make it different compared to other, older version control systems. While not all are unique, it should be understood that Git has distinct implementations of these ideas based on its fundamental early goal of supporting Linux kernel development.

Git Is Decentralized

Git is completely decentralized – in Git an authoritative central repository from which people receive code simply does not exist. Git creates multiple instantiations of a repository for every project it maintains, and every "clone" of a Git repository has the full history and all data as every other repository, and each repository is initially equivalent to all others. Code and metadata may freely move between any or all of these repositories. The repositories could closely map code between each other, or may diverge to follow different development paths. Any individual repository can contain a mix of commits sourced from many other repositories, with no single repository being the "authoritative" one, or being central to all commits that have been made the source of the particular project. Although a project might choose to regard one repository as the canonical one – like Linux has done with Linus Torvalds' repository – that determination can be made without reference to traditional concepts of authority such as "original source."

Different commits and the associated metadata for those commits can have different origins and a Git repository can become a "melting pot" of different commits as soon as multiple people start working on the code from that repository. As an example: since the introduction of Git into the Linux kernel development process, there have been tens of thousands of contributors. Each of these contributors has one or more Git repositories with a local copy of the Linux kernel and has made changes to one or more local copies. Changes are then sent to the repository maintained by Linus Torvalds, considered by the Linux kernel development community as the canonical repository for the Linux kernel. Some of these changes are sent to Linus directly, but others are sent via other repositories maintained by other contributors, where they might have been merged with other changes, or modified, before they are sent onwards.

4 *Id.*
5 *Id.*
6 InfoWorld, "After controversy, Torvalds begins work on 'git'" (April 19, 2005): https://www.infoworld.com/article/2669670/operating-systems/after-controversy--torvalds-begins-work-on--git-.html
7 The Eclipse Foundation, "Eclipse Community Survey Results" (June 23, 2014): https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/
8 Readers interested in a more detailed examination of how revision information is stored using Git, and some of the hazards that Git presents when attempting to mine its data, might wish to review the work of Professor Daniel German at the University of Victoria, e.g., Bird et al., "The promises and perils of mining Git" 6th IEEE International Working Conference on Mining Software Repositories (May 17, 2009): http://ieeexplore.ieee.org/document/5069475/#full-text-section

The proviso here is that the maintainers of Linux may or may not accept a particular suggested change, and there may be some challenges in having an accepted upstream version of code that propagates to versions of the code in other repositories. To a large extent the canonical version of the Linux kernel is clearly signposted by the key moderators, but it is still possible that other parties may maintain versions of the code outside of their direct influence.

Git Stores Content, Not Differences

Git stores the entire content of a file, and not just the differences between versions of a file. Every revision of a file is added to the Git store in its entirety and assigned a unique identifier, called in Git an index, based on its contents: files that have the same content will have the same index in Git.

Code that flows between Git repositories, or within Git repositories, are called commits. These commits can be imported (pulled) from other repositories, along with metadata about the particular change that that commit represents. For efficiency, the data regarding a commit might be sent to a repository in the form of a patch (or a collection of patches), but the data itself will be stored in Git in its entirety (after the patch has been applied), and therefore can also be retrieved in its entirety.

The benefit of this architecture is that any two arbitrary files in a Git repository can be compared to each other in their entirety. It is important to know that any particular piece of metadata will only be associated with the specific commit that introduced a change, and not the content: the same file can be committed multiple times with different commit messages but will be stored just once.

Dissection of the Information Contained in a Git Commit

Data in a Git commit can be split into two parts:

1. Metadata, such as information about the date of the commit, the author, a commit message, etc.

2. Actual file data.

Showing A Git Commit

There are various ways to show information regarding Git commits. To show the history of a file, or an entire repository, the "`git log`" command can be used. To show data specific to one individual commit, the "`git show`" command can be used. The result is that if a party wants to see the entire history of changes/commits made to a file or entire repository, they would use the `git log` command, whereas if they were looking for more granular information, such as when/how/by whom a specific change/commit was made, they would use the `git show` command.

Because Git stores the entire content of a file, and not individual changes relative to a canonical file, it is possible to adjust how data is displayed in Git. Such information is (re)computed as required by the particular request made to provide particular information. This architecture makes it possible to show more data, or less data, or data in different formats, depending upon the particular diff algorithm used and the options selected for that algorithm. This allows for maximum flexibility in selecting the particular type of information to be extracted, and to what parts of the repository that information relates.

Metadata In A Git Commit

The metadata of a Git commit falls into two distinct categories:

1. Machine generated information inserted by tools within Git
2. User generated information inserted by the creator of a commit

Machine generated content

Some of the fields that are machine-populated by Git could be important in identifying copyright ownership and authorship:

1. `commit id`: a unique id identifying a particular commit
2. `author`: e-mail address and name of the party who committed code to Git
3. `author date`: date that the change was committed into Git
4. `committer`: email address and name of the person that committed the change to the repository
5. `commit date`: date the commit was added to the repository
6. `Git commit message`: a message from the person committing the change. This is entered by the committer, but it is stored with other automatically machine-generated content, rather than the user-generated data.

This machine-generated information can be retrieved from Git by using the commands "git log," to retrieve all metadata regarding a project, or "git show," to retrieve specific metadata regarding a particular commit.

An (early) example of the retrieval of all metadata regarding a project can be seen from the Linux kernel:

```
commit 1db7fc75a410d9a15cbc58a9b073a688669c6d42
Author: akpm@osdl.org <akpm@osdl.org>
AuthorDate: Sat Apr 16 15:24:02 2005 -0700
Commit: Linus Torvalds <torvalds@ppc970.osdl.org>
CommitDate: Sat Apr 16 15:24:02 2005 -0700
```

This was retrieved using the following command:

```
git log --pretty=fuller[9]
```

The commit identifier, in the form of a hexadecimal string, is located after the word "commit" on the first line of output. The author recorded by git in the Author field is akpm@osdl.org (Andrew Morton, at that time affiliated with OSDL.org – the Open Source Development Labs, predecessor to the Linux Foundation) and the commit was pushed to the repository and then immediately committed by Linus Torvalds (this is a largely automated step).

9 "`pretty=fuller`" is a command that request a git log which prints the information about the commit in the "fuller" format, which displays author, authordate, commiter, and commitdate information. *See* https://git-scm.com/docs/git-log

Commit id

The commit identifier, or commit id, uniquely identifies each commit and its associated metadata in Git. Information regarding the commits can be retrieved from Git using this identifier.

Author

The author field records the e-mail address and/or the name of the contributor who submitted a change for inclusion. This data is set by the person submitting a change into a repository. This field is not meant for recording copyright ownership or authorship in the legal sense, as it literally is only a name and an e-mail address associated with a submitter. The actual copyright authorship might not be with the submitter, but with a different individual or individuals, and the copyright ownership might be with those different individual or individuals or to an entity to whom those individuals owe an obligation to assign, such as an employer. Copyright statements are recorded in the code itself and sometimes in the Git commit log message, as explained in more detail below.

Git only allows a single value for the author field. If there are more authors involved in writing the code (which is not uncommon) then this field in Git will not correctly reflect authorship. As a result, some development teams have used a variety of approaches to work around this issue. In the Linux kernel, the Git commit message, as described below, is sometimes used to record that there are more than one author, but in a project like Netfilter the names of the authors and their associated e-mail addresses are concatenated and put into the Author field, for example:

```
commit 2cfbd9f565e91356679bdee3f1e9b3133a9d14ad

Author:     Patrick McHardyHarald Welte
<kaber@trash.netlaforge@gnumonks.org>

AuthorDate: Sat Apr 22 02:08:12 2006 +0000

Commit:     Patrick McHardyHarald Welte
<kaber@trash.netlaforge@gnumonks.org>

CommitDate: Sat Apr 22 02:08:12 2006 +0000
```

A drawback of concatenating names is that it becomes a bit harder to process programmatically to accurately extract authorship information. What is important to note is that for some projects, such as the Linux kernel, the Author field does not necessarily reflect who actually wrote or contributed the code and it should not be exclusively or even primarily relied upon.

AuthorDate

When a commit is pushed to a repository, the local date on the machine where the commit was pushed is used to set the `AuthorDate` field. Note that pushing a commit to a repository does not mean that commit is actually committed to that repository, , as commitment is a separate step. Separating the time of push and the time of commit is a design choice by Git to allow people to work on code without having a network connection available. The `AuthorDate` field value is not set to the date/time of a (central) repository, and it does not reflect the actual date when the particular change was written by the author. This is because in the distributed nature of Git, such a setting would make little sense.

There are examples where this date is set incorrectly such as the following commit in the Linux Kernel:

```
commit 12ca45fea91cfbb09df828bea958b47348caee6d

Author:     Daniel Vetter <daniel.vetter@ffwll.ch>
```

```
AuthorDate: Sat Apr 25 10:08:26 2037 +0200
Commit:     Eric Anholt <eric@anholt.net>
CommitDate: Mon Nov 30 09:44:23 2009 -0800
```

Note that the `AuthorDate` is set to 2037, a date 20 years in the future, even though the code was added to Linus' kernel repository in 2009. In this case, the individual entering the change incorrectly entered the `AuthorDate` data – likely as the result of a typographical error – demonstrating that `AuthorDate` data can not always be relied upon for accuracy.

Commit

The commit field has the name and email address of the person that committed the change to a repository.

CommitDate

Similar to `AuthorDate`, the `CommitDate` is set to time on the local machine of the committer. If both `AuthorDate` and `CommitDate` are set properly (the machine of the author and committer are both synchronized with a reliable time source like NTP) then they will either have the same value, or `CommitDate` will be later than `AuthorDate`. Because patches could be in a repository for a long time before they are pulled into another repository the difference between the two values could be anything from less than a second to several years.

Git Commit Message

In the Git metadata there can also be entered some user-generated content, in the the "commit message" field. Although any information which the committer wishes to be associated with the commit can be entered in this field, it is considered good coding practice to have a description of the change, and other information that will be useful when revisiting code at a later date, so the log including that commit message can act as documentation or background information for any party analysing the code.

There is no restriction on what can be included in the git commit message, and it may include purported copyright ownership or authorship statements. It should be noted that for the Linux kernel, almost no copyright statements have been input into the git commit message logs recorded in the last 12.5 years, but there are large number of authorship statements in those logs (which may or may not reflect "authorship" in the sense that that term is used in international copyright law).

Tags

The Linux kernel logs contain more than the technical background of a specific commit; they may also contain information about a contributor who reported an issue, links to bug tracking systems, links to e-mail discussions, names of possible co-authors, and so on. The most important of these tags is the so called "`Signed-off-by`" tag,[10] or the "developer certificate of origin"[11]

Developer Certificate Of Origin/Signed-Off-By

One mechanism put in place by the Linux kernel community and subsequently adopted by many other communities is the "developer certificate of origin" or DCO. The DCO is a social contract that

10 https://ltsi.linuxfoundation.org/developers/signed-process
11 http://developercertificate.org/

acts as a safeguard to prevent problematic code (for example, code that is proprietary, that does not have appropriate license permissions to be contributed to the project, etc.) from being added to the Linux kernel. The DCO works by having authors and maintainers (either of which could be the "committer" of the code) "signing off" on the code, using one or more lines indicating that any particular change has been "`Signed-off-by`" and containing the names of each individual that has signed off on the commit that is being entered into Git, for example:

```
Signed-off-by: Linus Torvalds <torvalds@osdl.org>
```

The "`Signed-off-by`" tag in Git is part of the the Git log message, meaning it is typed in by a human. The "`Signed-off-by`" tag is not machine-generated or machine-validated, and therefore accuracy of its contents are not enforced by the Git tool. This means that there can some variability, including spelling mistakes or the use of different punctuation. When the authors of this article researched the Linux kernel Git log no less than forty four different variations (excluding case differences) of the "`Signed-off-by`" tag were found. Spelling variations make it more difficult to process and validate Git commit metadata using tools.

Tags Possibly Indicating Copyright Authorship

In the Linux kernel several tags similar to "`Signed-off-by`" were found that could possibly indicate authorship information. Typographic errors are as they appear in the logs:

```
author
co-authored-by
origionally-authored-by
written-by
also-written-by
patch-by
patch-updated-by
eventually-typed-in-by
coded-by
typing-done-by
original-code-by
original-coded-by
```

There were also numerous tags where the precise intent was less obvious:

```
based-in-part-on-patch-by
based-on-a-patch-by
based-on-code-by
based-on-original-patch-by
based-on-patch-by
based-on-work-by
includes-changes-by
initial-patch-by
initial-work-by
```

```
original-patch-by
patch-inspired-by
patch-dusted-off-by
patch-inspired-by
reworked by
derived-from-code-by
improved-by
modified-by
neatening-by
```

Two quite interesting tags from a copyright perspective are "`generated-by`" and "`generated by`," which are used for commits related to Coccinelle, an automated source code checker used by the Linux kernel developer community to discover defects and generate patches. It is unclear at the moment who would be the actual author of a patch automatically generated by Coccinelle: the person who wrote the original code prior to the patch, the person who wrote the tool that automatically generated the patch, or the person who wrote the specification that was used by the tool to generate the patch to be generated, and then ran the tool and submitted the patch for inclusion in the original code or a person who ran the tool to generate a patch based on a specification made by someone else.

Ambiguous Tags

Some developers have invented their own tags where it is unclear if they are attempting to provide an authorship reference:

```
wordsmithing-by
credits-to
reported-and-helped-by
inspired-by
```

There are also tags where it is not clear what was intended:

```
based-on-the-original-screenplay-by
meh'ed-by
based-on-the-true-story-by
duh-by
hallelujah-expressed-by
toasted-by
catched-by-and-rightfully-ranted-at-by
```

The Linux kernel log has several more tags that are probably worth exploring in an authorship context. What is important to know is that it is not always clear what sort of tag might be used by an author or a committer as an indicator of who is the author of the change from a copyright perspective. Because the ability of authors and committers to "roll their own" tags when they make entries into the Git tool, it might be worth looking at adding tooling support, creating templates, reducing the flexibility of creating new or unusual tags, or reinforcing best practices in tagging changes, in order to address issues related to clarity of copyright authorship and ownership for Linux, and other projects, moving forward.

File Data

As mentioned earlier each file committed to Git is stored in an "index." The file data in Git can be shown using the "`git show`" command. This data can be shown in a variety of ways, such as the whole file, or as a difference to another file in Git.

Showing Differences Between Files As A Patch

The default in Git is to display differences between files as a patch in a unified format, called "unified diff."[12] The concept of the unified diff format has been in use for decades and is a standard way to describe differences between files, and how to transform one file into a new version of the file, or another form of the file. Unified diff does not convey any information about copyright authorship – either of the original file or the differences between it and another file – but instead provides a recipe describing how to transform one file into another.
Git implements several diff algorithms (minimal, patience and the default 'myers' diff algorithm[13]). These algorithms all create patches in a different way. For example, the 'minimal' diff algorithm tries to compute the smallest difference possible. The other algorithms create different patches, but the end result of applying a patch is always the same. When examining a patch it is not just the patch that should be looked at in isolation; a few things need to be considered as well – such as which algorithm was used, as well as the state of the file before and after the change.

An example from the Linux kernel git extracted from the Git log looks like this:

```
diff --git a/arch/ppc/kernel/pci.c b/arch/ppc/kernel/pci.c
index 98f94b6..47a1530 100644
--- a/arch/ppc/kernel/pci.c
+++ b/arch/ppc/kernel/pci.c
@@ -1432,7 +1432,7 @@ pci_bus_to_hose(int bus)
        return NULL;
 }

-void*
+void __iomem *
 pci_bus_io_base(unsigned int bus)
 {
        struct pci_controller *hose;
```

The first line of the patch in unified diff format describes which command was run and which files were involved in creating the diff, where it should be noted that the first parts of the path of each file ("a" and "b") were inserted automatically and the "--git" flag is specific to Git and not available in the standard versions of "diff"

The next line describes the Git index before and after the patch, also called the "start index" and "end index". These are separated by '..'.. Also included are the file system permissions of the file (100644), which are irrelevant from a copyright perspective. This line is also one that is specific to Git.

12 http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html
13 The default algorithm is the "Myers" diff. The "minimal" diff tries to create smaller patches, while the "patience" algorithm tries to create better readable patches.

The next two lines describe the “before” and “after” files, similar to the first line of the patch.

The rest of the patch describes where changes should be made and how they should be made. Lines starting with “@@” describe the position in the original file where the change should be made. Every time a line starts with “@@” it indicates the start of what is called a “patch hunk”. Each patch contains one or more patch hunks. In this particular patch the change should be made at line 1432 of the original file, and the patch touches 7 lines in the original file. After the change the code will be at the exact same position and no lines will have been added or removed.

Lines that indicate change start with either a '-' or a '+'. Other lines are left as is. These context lines default to 3 lines before the lines that need to be changed and 3 lines after the change (this is not always possible, for example at the start or end of the file) and they serve two purposes: to make it easier for humans to see the patch in context, but also to allow programs such as “patch” (a program that applies patch files to source code) to apply patches in a fuzzy way when the offsets don't entirely match, so it can use the context to find out where the patch should be applied. For example in the above patch if in the original file the lines starting at line 1432 do not match with the context lines in the patch, but it would match with another line, then the patch program could apply it there.

Optionally there could be more lines in a patch generated by Git, such as for example when a file has been moved, copied or deleted. Also in a git patch several diffs can be concatenated, if patches from multiple Git repositories are applied. This is unique to Git.

Tracking Changes Of Code

In the context of conducting a legal review of software, it may be important to know who added a piece of code and when they did so. The information from the Git commit metadata described above can provide some information about who inserted code into the system, and possibly on which date such an activity occurred, but it is less useful at answering questions related to how much code a contributor has added and how much of that code is still contained in any individual repository. It also does not provide any assurances about whether the person inserting the code was the author of that code, in a copyright sense, or who or what entity owns the copyrights in that code.

Similar to Subversion or CVS, Git comes with a “blaming” (or “praising”) tool called “git blame” that shows who last changed each line of code and in which commit the code was last altered. While this is useful, it does not tell the full story of provenance.

There are many instances to be found in the Linux kernel repository where only a part of the code on a line was changed and the code in question is a mix of contributions from various contributors. The “git blame” tool will only show the name of the last contributor to change any single line, regardless of what that change was, or if it is copyrightable. To uncover the rest of the provenance information of any particular line of code, a more thorough search has to be undertaken. This includes review of how much code from any single contributor has been maintained between multiple revisions, and if any particular line of code that was added or modified can count as a copyrightable work. The existing Git tools alone cannot provide this information.

To illustrate this: there are situations where the contributor who last altered a line is not the contributor who wrote the code in question. An example from the Linux kernel are the so called “kernel janitors” who clean up code so it conforms to coding standards that should be adhered to, such as right indentation, the number of columns that are used, the removal of excessive white space, and so on. This work merely consists of rearranging pre-existing code to make it conformant with the coding standard. In such situations the output of “git blame” obscures the real author and instead returns the janitor's name when queried.

One example from the Linux kernel where this is the case is the following commit:

```
commit 2029cc2c84fb1169c80c6cf6fc375f11194ed8b5
Author: Patrick McHardy <kaber@trash.net>
Date:   Mon Jan 21 00:26:41 2008 -0800

    [VLAN]: checkpatch cleanups

    Checkpatch cleanups, consisting mainly of overly long lines
and
    missing spaces.

    Signed-off-by: Patrick McHardy <kaber@trash.net>
    Signed-off-by: David S. Miller <davem@davemloft.net>
```

The commit message indicates that the changes were made by running the "checkpatch" tool, which is a tool used by the Linux kernel developers to find code that doesn't adhere to certain stylistic conventions.

The patch itself confirms that for some files the only changes are the addition of missing spaces for purpose of making code look "cleaner" but which have no effect on the functionality of the code itself:

```
diff --git a/net/8021q/vlanproc.h b/net/8021q/vlanproc.h
index f908ee3..da542ca 100644
--- a/net/8021q/vlanproc.h
+++ b/net/8021q/vlanproc.h
@@ -4,16 +4,15 @@
 #ifdef CONFIG_PROC_FS
 int vlan_proc_init(void);
 int vlan_proc_rem_dev(struct net_device *vlandev);
-int vlan_proc_add_dev (struct net_device *vlandev);
-void vlan_proc_cleanup (void);
+int vlan_proc_add_dev(struct net_device *vlandev);
+void vlan_proc_cleanup(void);

 #else /* No CONFIG_PROC_FS */

 #define vlan_proc_init()        (0)
-#define vlan_proc_cleanup()     do {} while(0)
-#define vlan_proc_add_dev(dev) ({(void)(dev), 0;})
-#define vlan_proc_rem_dev(dev) ({(void)(dev), 0;})
-
```

```
+#define vlan_proc_cleanup()    do {} while (0)
+#define vlan_proc_add_dev(dev) ({(void)(dev), 0; })
+#define vlan_proc_rem_dev(dev) ({(void)(dev), 0; })
 #endif


 #endif /* !(__BEN_VLAN_PROC_INC__) */
```

Changes in the patch are for example the removal of whitespace after “add_dev” and “proc_cleanup”.

Git blame shows that the lines changed by this patch have been modified by Patrick McHardy[14]:

```
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  1) #ifndef
__BEN_VLAN_PROC_INC__
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  2) #define
__BEN_VLAN_PROC_INC__
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  3)
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  4) #ifdef
CONFIG_PROC_FS
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  5) int
vlan_proc_init(void);
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  6) int
vlan_proc_rem_dev(struct net_device *vlandev);
2029cc2c (Patrick McHardy 2008-01-21 00:26:41 -0800  7) int
vlan_proc_add_dev(struct net_device *vlandev);
2029cc2c (Patrick McHardy 2008-01-21 00:26:41 -0800  8) void
vlan_proc_cleanup(void);
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700  9)
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 10) #else /*
No CONFIG_PROC_FS */
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 11)
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 12) #define
vlan_proc_init()        (0)
2029cc2c (Patrick McHardy 2008-01-21 00:26:41 -0800 13) #define
vlan_proc_cleanup()     do {} while (0)
2029cc2c (Patrick McHardy 2008-01-21 00:26:41 -0800 14) #define
vlan_proc_add_dev(dev)  ({(void)(dev), 0; })
2029cc2c (Patrick McHardy 2008-01-21 00:26:41 -0800 15) #define
vlan_proc_rem_dev(dev)  ({(void)(dev), 0; })
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 16) #endif
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 17)
^1da177e (Linus Torvalds  2005-04-16 15:20:36 -0700 18) #endif
/* !(__BEN_VLAN_PROC_INC__) */
```

Each line in the git blame starts with the commit id (short form format) where the line was last

14 This code can also be browsed online at https://github.com/torvalds/linux/blame/2029cc2c84fb1169c80c6cf6fc375f11194ed8b5/net/8021q/vlanproc.h

modified, the name of the person modifying it, the date from the Git commit, plus the content of the line. The lines starting with "2029cc2c" were introduced by this particular patch.

The "git blame" output before this patch was applied shows the following result:

```
^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  1) #ifndef
__BEN_VLAN_PROC_INC__

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  2) #define
__BEN_VLAN_PROC_INC__

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  3)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  4) #ifdef
CONFIG_PROC_FS

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  5) int
vlan_proc_init(void);

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  6) int
vlan_proc_rem_dev(struct net_device *vlandev);

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  7) int
vlan_proc_add_dev (struct net_device *vlandev);

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  8) void
vlan_proc_cleanup (void);

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700  9)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 10) #else /* No
CONFIG_PROC_FS */

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 11)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 12) #define
vlan_proc_init() (0)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 13) #define
vlan_proc_cleanup()      do {} while(0)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 14) #define
vlan_proc_add_dev(dev)   ({(void)(dev), 0;})

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 15) #define
vlan_proc_rem_dev(dev)   ({(void)(dev), 0;})

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 16)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 17) #endif

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 18)

^1da177e (Linus Torvalds 2005-04-16 15:20:36 -0700 19) #endif /* !
(__BEN_VLAN_PROC_INC__) */
```

As can be seen the code is almost identical, apart from the whitespace changes. Although the changes introduced by Patrick McHardy are only a single whitespace, the output of Git blame might give the impression that the entire line was written by him.

The extent to which such differences confirm or challenge copyright authorship – and thus the ability to use this information to make a legal claim for license enforcement – is a matter of legal interpretation. However, the potential value is clear, and granular methods of exploring data are the subject both of ongoing research and emerging tools such as cregit.[15]

15 Edge, Jake, "Token-based authorship information from Git," LWN.net (Apr. 31, 2016) https://lwn.net/Articles/698425/

Recommendations

To address some of the shortcomings that exist in Git when it comes to researching copyright in source code stored in Git, it would be useful if some or all of the below were explored:

1. Adoption of standardized tags (either supported in Git directly, or as part of a "social contract") that allow entry of copyright ownership, copyright authorship and copyright date information to reduce ambiguity around other tags that might be used to indicate this data. In addition the reduction of the ability in Git to "create your own" tags to prevent confusion about what data is being entered and to allow more automation of the extraction of information based on consistent input data.
2. Functionalities in Git or when using Git that would allow more detailed forensics regarding contributions, at a sub-line level. Tools like cregit are a first step in this direction.

Conclusion

This paper has provided a brief tour of the types of authorship data obtainable from the Git system. The key takeaway is that the Git revision control system does not enforce correctness of data but instead is reliant on correct inputs for correct outcomes. Git records potential authorship rather than copyright ownership and its core "git blame" tool does not show potential authorship with enough granularity to be regarded as a canonical authority or a single source of truth. Therefore, improvements to the Git tool, or enforcement of greater discipline in Git tool users in tagging and data-entry, would be required to truly use Git as an authority for providing a verifiable record of copyright authorship and ownership in legal proceedings is likely required, and courts should be cautious in relying upon Git outputs as dispositive on questions of copyright ownership or authorship. Additional review and an additional process layers would be helpful to ensure fidelity and accuracy of data and to accurately determine potential authors of code contained in any Git repository.

About the authors

Armijn Hemel *is an active researcher of and internationally recognized expert in open source license compliance and supply chain management. He studied computer science at Utrecht University in The Netherlands, where he pioneered reproducible builds with NixOS. In the past he served on the board of NLUUG and was a member of the core team of gpl-violations.org. Currently he is a board member at NixOS Foundation.*

Shane Coughlan *is an expert in communication, security and business development. His professional accomplishments include spearheading the licensing team that elevated Open Invention Network into the largest patent non-aggression community in history, establishing the leading professional network of Open Source legal experts and aligning stakeholders to launch both the first law journal and the first law book dedicated to Open Source. He currently leads the OpenChain community as Project Director.*