Whither (Not Wither) Copyleft

Prof. Eben Moglen,[a]

(a) Columbia University; Software Freedom Law Center

DOI: 10.5033/ifosslr.v9i1.116

Abstract
This article contains an edited version of Professor Eben Moglen's speech at the SFLC Fall Conference 2016. It explores the topic of Copyleft, enforcement and community engagement from the perspective of one of the key individuals in the rise of Free and Open Source Software from interesting idea to a central pillar of the global technology industry.

Keywords
Law; information technology; Free and Open Source Software; Copyleft; enforcement

I am going to trench upon your patience just for a little while for some substantive thoughts that recent events raised for me. These are related to the global dialogue about copyright compliance and license violations. We are not and we never were copyright maximalists. We did not do what we have been doing for the past 30 years to build Free Software on the basis of the assumption that freedom required us to chase down and punish everybody who ever made a mistake or who even deliberately misused copyrighted software made for sharing.

When I began to work with Richard Stallman in 1993, GPLv2 was 18 months old. And although I had been thinking about what all of this meant for some little while, I was working on making the world safe for public key encryption, so the Free Software copyright licensing system was something of which I was only dimly aware. In the course of the first crypto wars, Richard Stallman contacted me, said he had a problem and could I help him with it. I said, "Yes. I use emacs every single day, and it will be a very long time before you exhaust your entitlement to free legal help from me." So I went and did what he needed done, and then I thought to myself, "this is the most important place for a lawyer to work right now."

"If I could just sit on Richard Stallman's email stream and have him send me what he thinks needs a lawyer – because anybody in the world who had a problem that involved freedom and computers knew one email address, and that was rms@gnu.org – pretty soon I could figure out what it was that actually needed doing." Very rapidly I realized that what needed doing was getting people to spontaneously comply with law instead of having to fight them each and every time.

Spontaneous compliance is the only conceivable way to run a legal system, I must tell you. The United States is a country with an extraordinary amount – apparently – of complaining about taxes every four years or every two. But every year, Americans pay their taxes, and they don't do it because they see crowds of people sent to jail. They do it because spontaneous compliance is the way law really works. The problem of legal engineering which presented itself to me in 1993 and the problem we are still talking about this afternoon is how to ensure spontaneous legal compliance, not how to figure out an adequate degree of coercion which will make an adequate degree of compliance at the other end.

The fundamental problem as it presented itself to me in 1993 is the problem as it still presents itself to me now. Coercion does not work if you have to do so much of it that you can't afford it. Coercion only works so long as you never lose any fight anywhere, which is why you have to keep equipping your police with bigger and bigger guns and there is always the risk that they will use them. I did not want then and I do not want now to pretend that the way that we secure compliance with copyright law with respect to Free Software is by chasing down people and making them comply. It is important every once in a while to set an example. Therefore it is important every once in a while to declare that you're in a last-resort situation, and there's nothing else that you can do but to resort to litigation.

I understand that, at the present time, there are a large number of people who are living in that expanding boundary of Free Software use and redistribution that we have all been talking about. Given where they work – the particular software they work on, the particular forms of downstream use that are most important to them – they run into infringement situations in this outer boundary area, and they therefore believe that everybody in the world doesn't get it about Free Software, and even that everybody in the world is a crook and that everybody in the world is trying to steal Free Software and make bad use of it.

What I thought was so important about Greg Kroah-Hartman and Ted T'so and the point that they recently made at the SFLC Fall Conference 2016 was this: they say that if you are sitting in the middle of the single most commercially valuable Free Software project in the world, and you have thousands of people helping you to make it, fighting with every single infringing person is not the way to win. **Converting every single person** is the way to win. Fighting can only conceivably be valuable if it is on the way to converting people. It cannot possibly stand on its own.

I have some fine clients and wonderful friends in this movement who have been getting rather angry recently. There is a lot of anger in the world, in fact, in politics. Our political movement is not the only one suffering from anger at the moment. But some of my angry friends, dear friends, friends I really care for, have come to the conclusion that they're on a *jihad* for Free Software. And I will say this after decades of work – whatever else will be the drawbacks in other areas of life – the problem in our neighborhood is that ***jihad* does not scale**.

As exemplified from discussions at the SFLC Fall Conference 2016 from the lawyers I have been friends and colleagues and occasional professional adversaries with over these decades is that in the industrial use of Free Software scale is what matters. We on our side in the community of Free Software makers have to understand that scale is what matters to us too. The problem with jihad is not that it's not virtuous or that making people obey the rules is somehow wrong. I like policemen and police forces a lot. But I know that the amount of policing necessary to produce perfect compliance is an amount of policing we can neither afford nor tolerate in the society where we live. So regrettably, I have to draw some factual conclusions to your attention:

First, if at any time in our long association over the past 23 years – this century, last century, it doesn't matter: If Richard Stallman and I had gone to court and sued a major global public company on a claim of copyright infringement that was weak enough to be thrown out of court on a motion to dismiss, we would have destroyed the GPL straightaway. If we had shown that we were prepared to risk large on coercion, even against a bad actor in our own judgement – if we had done that without adequate preparation to be sure that we won – we would have lost an example of coercion and nobody would have trusted us again.

I did sue people. It's true. Greg referred to the way in which when the Busybox developers thought they wanted to start suing and I did it for them, the results may not have been the ones they most wanted. That happens with clients all the time, particularly clients who go to court: They get something which is not quite what they wanted. But I thought that it was important then because Busybox was being embedded in everything. In the moment at which we were then living, in which the frontier of use and redistribution was expanding so rapidly, it seemed to me that it was necessary to get people's attention. I thought then, as I think now, that the people whose attention you need to get are the people who don't pick up the phone when you call them. We thought that people you can't contact, people you can't get to answer the phone, people who will never spontaneously comply – they won't even answer your mail – may be the right people to make an example of.

But on the night before we filed the Busybox cases in 2009, I chased down in Japan at 2:00a.m. the general counsel of one of the organizations we were going to sue the next day – a very large very powerful, very reputable company. I said to him, "If you give me your personal assurance that you're going to fix this problem, tomorrow you will not be sued. I will take your word for it. Nothing more." And he said yes, and I said yes. And they were not sued the following day because all we wanted was for people to pay attention and bring their engagement to the party. Even at that level, too much coercion – and we are still arguing about whether that was enough or too much – too much coercion was surely not what I wanted to apply.

Second: If when Scott and Terry and their colleagues at IBM and Hewlett-Packard first began to come to Free Software, when they first wanted to recommend it and use it and maybe even distribute it themselves or encourage other people to distribute it for them, we had criticized them for not being non-profit virtuous enough, if we had said "we are suspicious of you," let alone if we had threatened, "one step over the line buster and we will sue you" – everything else that we wanted to do would have become impossible immediately. If we had not acted as Greg and Ted said that they must act on behalf of the great project that we all love, if we had not welcomed everybody with open arms and made clear that the commercial exploitation of the software was our hope not our fear, we would have achieved absolutely nothing that really mattered to use about freedom.

Third: We spent years scrupulously getting work-for-hire disclaimers from every business and every university that employed or educated a contributor to GNU. Every time we took a right, we took a disclaimer to be sure. If there was any question that anybody needed to be contacted, we negotiated those disclaimers as long and as carefully as it took. The people who gave us work-for-hire disclaimers, they didn't "get" Free Software, I assure you. They were simply being asked to say that it wasn't work-for-hire, that some programmer who worked for them was working on a project in her or his spare time.

Suppose we hadn't gotten those disclaimers – suppose we hadn't proved to everybody that we were not trying to solicit rights on which they had a claim – if we had, for example, gone around and asked people to give us rights and software they had written while working at other companies, without

every talking to those company's lawyers. In that case not only would we have destroyed all trust, not only would we have made it absolutely impossible to achieve what we really wanted, I would have put my law license in danger.

I think that all three of those are uncontroversial propositions. But in case you're inclined to doubt any of those propositions, I have to tell you that people in my world, people in my neighborhood, people in my movement, people in many cases whom I trained, have conducted those same experiments over the last two years. The results have not been any different than I would have expected. We have created for ourselves some troubles and there are other people out there creating troubles for us.

One example is a current NSF funding solicitation for a Free Software-intended project. NSF is in fact soliciting a research funding application from a client of mine which makes Free Software. This solicitation is designed to support them. Except it isn't, because they're a GPL'ed project:

> *All projects agree to distribute all source code that has been authored while working on an NSF/BigCorp award under a BSD, Apache or other equivalent open source license. Software licenses that require as a condition of use, modification and/or distribution that the software or other software incorporated into, derived from or distributed with the software be licensed by the user to third parties for the purpose of making and/or distributing derivative works are not permitted. Licenses not appropriate thus include any version of GNU General Public License (GPL) or Lesser/Library GPL (LGPL), the Artistic License (e.g., PERL), and the Mozilla Public License.*

Don't even think of applying for research funding if you're going to make copyleft Free Software. Now if you think that that's a little much, how about this, from the same solicitation?

> *Awardees may file patent applications, providing that they grant to BigCorp a non-exclusive, worldwide, royalty-free, sub-licensable license to all intellectual property rights in any inventions or works of authorship resulting from research conducted under the joint award.*

So, as it turns out, not only can you patent some software here but all your intellectual property rights – that is including your copyright since it's all works of authorship – will be non-exclusively licensed to Big Corp. I have changed Big Corp's name to protect the theoretically innocent.

This is a current DARPA funding solicitation also for a project that makes Free Software:

> *The program will emphasize creating and leveraging open source technology and architecture. Intellectual property rights asserted by proposers are strongly encouraged to be aligned with non- viral open source regimes. Exceptions for proprietary technology will be considered only in compelling cases. Make sure to carefully document and explain these reasons in submitted proposals.*

Once again, you are strongly urged to make wonderful open source software under this award. Don't think of using copyleft. We don't want you to. So have to put a special explanation in the grant request, which is of course equivalent to "thanks but no thanks." This I must tell you: if you want to talk about curing cancer, cure this for me. This is more dangerous than all the copyright infringement by accident or deliberation occurring out there in the Free Software world right now. This will make copyleft wither away.

Because throughout the research infrastructure in this wonderful great country of ours, **if copyleft is not allowed, then a whole generation** of the most talented people we work with **will come to the conclusion –** before they get their BA, before they get their doctorate, or before they decide to go and do something in industry – they will already have concluded **that there is something wrong with copyleft and you shouldn't use it**. I don't know any way to sue this out of existence. I don't know any way to deal with this militarily. This is a diplomatic challenge. This is a diplomatic challenge that requires lawyers who know how to do this work, which is not done by lawyers who sue people.

It is not about coercion. It is not even about encouraging people to convert. It's about reversing a problem that we have partially brought on ourselves and which other people are taking advantage of "bigly," if you ask me. This is where the limits of counseling meet the limits of coercion: the real answer is that you have to have a great big ecosystem and everybody has to believe in it. Or else you have to have as many lobbyists as BigCorp, and they have to be spread all over the research infrastructure, assuring copyleft's future. So what I want to say about all of this is that we are now at a turning point.

The good news of today is that this turning point should carry us all from the stages of fear and compliance to the stages of engagement and leadership. We are now actually ready. I don't mean ready plus or minus three years or ready plus or minus the regulators of fintech. I mean we are ready now with, SPDX and OpenChain and better tooling and Debian machine-readable copyright files that read on everything that everybody really uses. We are ready to begin to reduce the costs of compliance and lowering the costs of finding how to comply, to a level which really will allow us to do what Greg and Ted were talking about: country-by-country and commercial environment-by-commercial environment all around the world, making things just work.

I remember how much Nokia admired Apple for the just-works zen of it all. I agree with Jeremiah Foster at GENIVI that it is awfully good that we got their Maemo development off the floor and into things like cars, because it was wonderful stuff. I'm not going to tell stories now about how hard it was to try to get Nokia not to fly into the side of the mountain with that stuff back in 2010. It was a sad experience. But what we have now is the opportunity to avoid all the evolutionary dead ends that ever beset us. We have an opportunity to put this Free Software where we want it, which is everywhere, and to make it do what we want, which is to spread freedom.

We're not in a place where the difficulty is how do we get enough ammunition to force everybody to comply. We don't need ammunition. We need diplomacy. We need skill. We need to work together better. We need to understand how that working together purposively brings us to the point where everyone is not afraid of FOSS anymore and we are not worried about their complying anymore. We are just all engaging and leading the task of making Free Software. But I have to convince a lot of people of that, and not all of them are on the so-called other side. That process is going to be a complicated one It's going to take a couple of years.

We have some backing up to do and some moving forward to do at the same time. Although anarchism is good at moving in many directions simultaneously, it is not always good at understanding where it has to back up and where it has to move forward. But this will make us. Because the long-term threats to copyleft are not to be found in people who aren't doing it quite right. The long-term threats to copyleft are not to be found in the idea that too many people are getting away with too much and we have to go and get on our motorcycles and run them down and pull them over to the side of the road and give them a ticket. That's simply not the model that is

relevant right now and not everybody fully understands that.

So from my point of view, the purpose of today – with blockchain, and thinking about what the lawyering we've all done for decades means, and the purpose of talking to the clients about what they really need – is to make the point that we are not going to war to save the GPL. That's not where we are right now. We're not even going to war to save copyleft right now. We are certainly not going to war to save any projects right now. That's just destroying the village in order to save it and we've never been that kind of lawyers. We're not going to become that kind of lawyers.

What we do have is a real problem in deciding how to make copyleft relevant forever. There are a lot of smart people in this room who in their quiet moments face-to-face with me or with other people here have been known to say, "You know, I think copyleft might be becoming irrelevant now. "It was good. It put some principles deep in everybody's minds. It gave everybody a real sense about what our aspirations are. But from an operational point of view, we don't need it anymore." I fear that copyleft's most powerful supporters have helped to bring people to that conclusion. The purpose of today – even before news reached me from the outer world – the purpose of today was to say that's also not where we are. Where we are is: copyleft is a great idea that changed the world. It needs refreshment now in order to appeal to a younger generation of people who write programs for sharing.

In order to make Copyleft appeal to those people who write programs for sharing, we need to make it simpler to use, quicker to understand, and better at doing all the jobs it's supposed to do. We need to refrain from going unnecessarily to war. The lessons that we learned over the last quarter century are still good: That way won't work. I agree with the people who have suggested that if a campaign of coercive compliance is carried just a moment too far, willingness to use copyleft among the rational businesses of the world will decline to a point which is dangerous to freedom, because I do believe that copyleft is important to freedom. Indeed, I think it's crucial to freedom. Indeed, that's what I was taught by the greatest computer programmer I've ever known. So my point here is let's not get confused. This is not war time. This is diplomacy time.

Skill counts. Agility counts. Discretion counts. Long credibility counts. Ammunition? Ammunition is worthless because wherever we fire it, we work everywhere and it's only going to hit us. My thanks to Richard Matthew Stallman. He invented the world I live in. Years ago, Larry Lessig said that Richard Stallman had invented the twenty-first century. And I said, well, that may or may not be true, but any twenty-first century Richard Stallman did not invent is a twenty-first century I won't consider it safe to live in. And that's still true. To my comrade, to my client, to my friend Richard Stallman: my deepest and most determined thanks. There is nothing, nothing in the world, that could ever divide us as much as we have been brought together by the dream that we have shared and that we continue to give our lives to. It could not have happened without one man's thinking.

Conclusion

At Red Hat, there used to be – back in the old days before the Progress Energy Tower and all the wonderful things that have followed from Red Hat's commercial success, back when it was just barely not Bob Young's and fully Matthew Szulik's – there used to be up on the wall in the reception area a painted motto. It said "Every revolution begins as an idea in one man's mind," which is a quotation from Ralph Waldo Emerson. And deep in the American grain – as deep in the American grain as Ralph Waldo Emerson himself – is Richard Stallman, whose dream it was that made the

revolution I'm still trying to kick down the road towards some finish line or other I won't live to see.

To him, to you, to all of us – to the people who have made this stuff, to the people who have shared the stuff, to the people who have rolled up the barbed wire and carried it away so we could all just do the work and not have to worry about it – to my friends, to my clients, to the lawyers who have inspired me to teach them, my deepest and most unending gratitude.

About the author

Professor Eben Moglen *began building software as a professional programmer at age 13. He worked as a designer of advanced computer programming languages at IBM from 1979 to 1985. In 1991, he represented Philip Zimmerman, the developer of PGP ("Pretty Good Privacy"), threatened with prosecution by the US government for making strong encryption Free Software that everyone could use. In 1993, he joined forces with Richard M Stallman to provide world-class legal representation and expertise to the Free Software movement. With RMS, he conceived, wrote, and created a public process for discussion and adoption of GPLv3, the current version of the world's most widely-used Free Software license.*

In addition to his work with Free Software developers, Professor Moglen has advised major IT companies and national governments around the world. In 2010, he testified before the European Commission on the FOSS consequences of Oracle Corporation's acquisition of Sun Microsystems, and before the US Congress on Internet privacy and consumer protection. He has appeared numerous times on software- and privacy-related issues as amicus curiae before the US Supreme Court.

Professor Moglen earned his PhD in History and his law degree at Yale University. After law school, he clerked for Judge Edward Weinfeld of the United States District Court in New York City and for Justice Thurgood Marshall of the United States Supreme Court. He has taught at Columbia Law School since 1987 and has held visiting appointments at Harvard University, Tel Aviv University, and the University of Virginia. In 2003, he was given the Electronic Frontier Foundation's Pioneer Award for efforts on behalf of freedom in the electronic society. Professor Moglen is admitted to practice in the State of New York and before the United States Supreme Court.

The FOSSology Project: 10 Years Of License Scanning

Michael C. Jaeger,[a] Oliver Fendt,[a] Robert Gobeille,[b] Maximilian Huber,[c] Johannes Najjar,[c] Kate Stewart,[d] Steffen Weber,[c] and Andreas Würl [c]

(a) Siemens AG, Corporate Technology (b) Freelance Consultant
(c) TNG technology Consulting GmbH (d) The Linux Foundation

DOI: 10.5033/ifosslr.v9i1.123

Abstract
FOSSology is an open source project developing a Web server application and a toolkit for open source license compliance. As a toolkit it allows performing license copyright and export control scans from the command line. The FOSSology Web application provides a database and Web UI for implementing a compliance workflow.
The FOSSology project published the first version of its software in December 2007. Given this ten year anniversary of license scanning this article presents the motivation for building and using FOSSology its history and its status as of today. Because SPDX represents the *de facto* standard for exchanging license and copyright information about software packages an introduction to FOSSology's support for exporting and importing SPDX documents is also presented.

Keywords
Free and Open Source Software, License Scanning, Compliance Tools, SPDX, OSS Analysis

Introduction

The use of software is granted under a specific license. Open source software, like proprietary software, has conditions that must be complied with. In absence of a license, the software must be treated as all rights reserved, and not distributed further. As a result, understanding the license is key to being able to determine what one is allowed to do with the software.

Open source software is licensed using open source compliant licenses. The Open Source Initiative provides a definition of open source[1] and an open source compliant license has to comply with all parts of this definition. However, the list maintained by OSI is only a small set of the licenses currently in use of open source projects.

Over the past few decades a lot of different Open Source compliant licenses were drafted by different authors. Today we are confronted with the so called "OSS license proliferation" problem. More than 1000 Open Source compliant licenses are "in use", some of them differ only in a few words, while

1 Open Source Initiative: The Open Source Definition https://opensource.org/osd - 2017

others pursue a totally diverse interests. The SPDX working group collects main open source licenses and as of today, this effort has identified more than 340.[2] While a few licenses are very popular, meaning that many open source projects apply them to their work, as can be seen by usage statistics,[3] some other licenses are just published and applied by individual organizations to their software. Examples of popular licenses are the GNU General Public License 2.0 or the MIT License; examples of organization specific licenses are the Apple Public Source License 2.0 or the Microsoft Public License. Very briefly, authors and organizations have created all these different licenses for multiple reasons, among which:

- Authors of open source software have particular intentions for the use of their open source software;
- Commercial organizations strive to protect their commercial interests; or
- Non-profit organizations strive to protect or promote the use and adoption of open source software.
- Given the three points above, individuals or organizations have authored updated versions of their licenses, adding to the number of existing texts with even more new texts.

This article does not intend to compare or discuss all the different licenses. Rather it points to another challenge that results from the high number of existing license texts: Assuming the redistribution of an open source software component, regardless if it is as part of a commercial product or as part of a new open source project, this step requires the determination of the exact text of the applicable license for multiple reasons:

- Some licenses request providing the license text along with the redistribution of the software component.
- Some licenses express particular conditions when exercising the granted right of redistribution.
- Some conditions of some licenses are not compatible with conditions of other licenses. In this case combining two components licensed with incompatible licensing conditions between them is not possible.

As a result of the explanations given above, the first step of redistributing open source software is to determine the exact license text. However, realistically though, because each open source project tends to borrow from others, a mix of licenses tends to be present in most open source software components. When there are tens of thousands of files that make up a modern software package, it becomes a significant amount of work to properly respect the licenses. Therefore, the challenge is not only the great number of existing license texts, but also to cover the fact that many open source components show multiple open source licenses applying for some parts of the component.

In addition, a third challenge arises: authors of open source software do not use, in many cases at least, a standardized form of licensing. While licenses, such as the GPL versions, have standardized headers for source files to express a common way of licensing, many authors have found individual ways of referring to a license, sometimes using prosaic language. Thus, license statements which refer to a common license text can be either not unambiguously pointing to a particular license or are just hard to identify as a licensing statement.

2 SPDX Workgroup - a Linux Foundation Collaborative Project: SPDX License List https://spdx.org/licenses/ - 2016
3 Github.com: Open source license usage on GitHub.com https://blog.github.com/2015-03-09-open-source-license-usage-on-github-com/ - 2015

In summary, we have three different challenges for finding exactly the applicable license texts for use when redistributing open source software:

- A high number of licenses exist (OSS license proliferation);
- Multiple licenses can be found in a single open source component;
- Authors sometimes do not unambiguously refer to a particular license including its text.

Software tools exist to cover these challenges: license scanning software searches in open source software code for known license texts, licensing expressions and license relevant statements. One of these tools is provided by the FOSSology project. The FOSSology software is designed to determine the licensing condition of open source components. FOSSology was first published in 2007 by a group of Hewlett-Packard (HP) engineers, which is about 10 years ago. Therefore, it is now a good time for updating what has happened with FOSSology and its status as of today.

This article is organized as follows: the next section introduces the FOSSology project and gives a brief overview of its history. A subsequent section explains some of the technology used in FOSSology. Another section provides an overview of FOSSology and SPDX and the last section concludes this article.

Project History

FOSSology was first published by engineers from HP. An early first version of the software existed inside the company. Before FOSSology came in to being, an HP software engineer, Glen Foster, wrote some tools to perform license scanning. The focus was on scanning Linux distributions released with HP products. At that time Linux distributions were already large portions of open source software. Thus, a scan tool with the capability to scan large archives was the focus from the beginning. A first version of the software consisted of individual shell scripts. Subsequently, those scripts evolved into C language and compiled executables for speed. Then, the C code was enhanced to make it more capable for extension with future license texts and more licensing statements. This resulted in the original Nomos license scanner. FOSSology combined Nomos with a license categorization concept named buckets: users could define buckets based on detected licenses. With this approach, software was scanned with individual file focus. At the same time the software provided a large number of static HTML files for reporting.

In a subsequent effort, Robert Gobeille became involved by leading a project to speed this process up. The basic approach was the reuse of scans: Files that had been scanned already would not show a different licensing when scanned again. By including a database, the software avoided rescanning a large percentage of the files in a distribution. Another point was the reporting, which was at first, the standard output of the executables. By creating a plugin Web interface served by a Web server, dynamic and configurable reporting could be easily added.

After this setup had been established, the project came to realize that the value of the FOSSology project was in free use for all organizations, while plans to productize it seemed unlikely. Rather, the idea emerged that its development could be leveraged by making it an open source project. Thus FOSSology was born. One of the visions for FOSSology was to make it an open source data mining system, not being limited to scanning for license texts and statements, hence the name FOSS + ology - the study of Free and Open Source Software. Therefore FOSSology implemented an architecture of pluggable agents that can be composed into a pipeline of tasks applied to an uploaded open source component. After FOSSology was made open source, it was not only limited to providing the community with a license scanner (at that time called “license detector”). In 2008, the FOSSology

project identified the potential in providing the analysis data for a public repository of software and license metadata at FOSSology.org.[4]

FOSSology 2

In 2012, the project released version 2 of FOSSology. A new installation package structure reorganised the software project. Furthermore, an architectural change was made with the implementation of a new scheduler which orchestrates the different scanning agents. This change also helped to design and run the agents more independently. While such changes did not bring new features to the users, the new architecture provided a more extensible structure for the FOSSology project. This followed the overall vision that FOSSology represents an analysis and reporting framework where agents as modules can be combined into a workflow running on OSS components.

Another improvement introduced in later versions of the FOSSology 2 era was the introduction of data access objects (DAO). On the first hand, the DAOs helped making database access more systematic. But with the different report formats, the DAOs also ensured consistency between the different outputs: rather than each reporting agent implementing its own query logic, all agents could call the same functions to query the database, for example, for found licenses in the uploaded OSS component.

With version 2, FOSSology evolved into a multi user Web application that covered two main trends of licensing open source software: Not only did open source software become more and more popular and awareness about license compliance increased, but also the licensing showed more forms of individual statements. Further additions in the FOSSology 2 era were about organizing uploads with tags and the ability to correct findings brought up by the scanners. FOSSology turned from a server based scan tool to a Web application for users to upload, analyse and organize OSS components for their licensing conditions.

A major change that users actually noticed was the reworked file contents view for reviewing license findings including the highlighting of text areas of license relevant statements or licensing headers in files. What sounds straight forward turned out to be a complicated programming problem: license headers or license relevant statements are usually put into comment sections of source code. At the same matching license expressions using, for example, regular expressions required cleaning the text from comment sections. Otherwise matching text areas would have been compromised by these. However, for highlighting the matched text area in the Web UI, the file contents are displayed including comment sections. Highlighted text areas would shift because of comment sections previously omitted for the matching. As such, recalculation of the exact text position was necessary. As an additional challenge, source code files can contain multiple licensing statements or headers scattered across multiple locations in a file which require a comprehensive approach to recalculation. In the end this was worth the effort, as it turned out that highlighting license relevant text areas greatly helps to quickly identify and classify license relevant parts of the file on screen.

Also n this period, a first version of SPDX generation was published as an external module by the University of Omaha Nebraska.[5] SPDX is a specification for expressing metadata about software packages including licensing information.[6] Leveraging the modular structure of FOSSology, the SPDX output was implemented as just another reporting agent. A first version of the software generated SPDX 1.1 compliant documents while a subsequent evolution of the generation plugin provided SPDX 2.0 conforming documents as output format.

4 Robert Gobeille: The FOSSology project - MSR '08 Proceedings of the 2008 international working conference on Mining software repositories
5 Matt Germonprez, Gary O'Neall, Sameer Ahmed: Tooling up for SPDX - Open Compliance Summit 2013
6 SPDX Workgroup - a Linux Foundation Collaborative Project: SPDX License List - https://spdx.org/sites/cpstandard/files/pages/files/spdxversion2.1.pdf - 2016

In the version 2 era, the project was also transferred to the Linux Foundation as a collaborative project. A discussion about this step was coincidently taking place at the time of the splitting of the HP company into two organizations.[7] Eventually the Linux Foundation offered to host the FOSSology project where it fits into the Open Compliance Program.[8]

FOSSology 3

FOSSology 3 introduced a new license scanner Monk as a new feature. This scanner finds license texts faster than the Nomos agent. Both the matching and the difference between stored license text and found text is highlighted which helps the user to quickly identify the license. Additionally, the keywords used by Nomos are highlighted in the same text view. These visual hints help in the license decision process where the results can be managed in FOSSology: Differences to reference license texts from the FOSSology database are clearly shown to the user. Another feature introduced with version 3 was the editing capability of copyright phrases found by FOSSology. This is important since there is no rule as to how to indicate copyright ownership and there is a variety of different ways that copyright ownership may be expressed. Although the implemented functionality to extract copyright notices is striving to extract only the relevant information, it is sometimes necessary to postprocess the results, mainly to remove formatting characters.

New JavaScript frameworks like jQuery and jQuery Datatables modernized the client look and feel while refactoring on the server side, such as dependency injection increased testability. FOSSology continued with technical improvements with more use of jQueryUi for a better client experience and the implementation of PHP templating using the Twig library.

Another open source project for license scanning, Ninka[9] has been integrated using a wrapper. The main idea was to have three license scan approaches in FOSSology to allow for more adaptive scanning as well as the ability to conclude licensing based on the results of these. With these license scanners integrated, another agent was added for more automation in the workflow: a decider agent allows for defining rules such as reusing decisions from other packages or automatic decisions if all scanners determine the same license. For decisions based on certain text phrases, another scan agent speeds up the process: users can select distinct licensing statements found in a particular file and apply a rule for the entire upload to alter, confirm or remove a particular license.

Another feature was a refactored SPDX 2.0 RDF file generation. Release 3.1 extended the output formats for the SPDX tag-value notation in addtion to RDF/xml. Release 3.2 added the ability to import SPDX documents from other FOSSology instances or even other software tools. Furthermore, a word processor document report was added in Fossology 3.2, which contains not only licensing information, but also summarises analysis decisions as well as scan findings. And finally, an added JSON output format increases the possibilities to export results for other applications.

Another feature area implemented in version 3.2 of FOSSology is license obligation and risk management. This feature allows for defining obligations and risks and associating them to licenses. When a report is generated, all the obligations and risks of the licenses in effect (the concluded licenses) are generated in the report, given that an administrator of FOSSology has assigned the obligations and risks to the licenses. This especially helps to efficiently deliver a component license analysis without subsequent manual editing steps.

7 Hewlett-Packard Co.: HP To Separate Into Two New Industry-Leading Public Companies, Press Release, October 6th 2014

8 The Linux Foundation: Open Complicance Program – A Linux Foundation Initiative https://compliance.linuxfoundation.org

9 German, Daniel M.; Di Penta, Massimiliano and Davies, Julius : Understanding and auditing the licensing of open source software distributions. In Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, pp. 84-93. IEEE, 2010.

Last but not least, FOSSology 3 added features that reduce the time needed for component analysis and scanner corrections by reusing information from previously analysed uploads. For example, when scanning new versions of software, the analysis can be limited to the differences compared to an older version. In fact, this reuse is not only limited to conclusions or corrections of licenses on a file basis: also identified custom text passages in previous uploads can be taken over to new uploads. With this feature the manual correction time of a newer version of a software component is minimized to the actual differences in licensing only.

Technology

FOSSology is a derivative of a LAMP application. LAMP is an acronym that denotes applications that run in Linux, use the Apache Web server, build on MySQL as a database and provide a PHP-based Web UI. In FOSSology, a PostgreSQL database server is used instead of MySQL. Because of its dependencies on the Linux APIs and libraries, FOSSology cannot be easily ported to the Windows or Mac OS X platforms. However, virtual machines or docker-based builds make its use on these platforms possible today.

Database Approach

Since scanning for licenses in open source components yields large amounts of data, the use of a database is a required. PostgreSQL is available on most Linux distributions and represents a mature dependency, while allowing for portability of the FOSSology software.

In the first days of FOSSology, the reference schema was stored in the so called GoldDb. Schema changes were managed via a centralized implementation in `lib/php/libschema.php`. However, some operations cannot be represented as schema updates for an existing database. Therefore, additional steps for migration of data are required during upgrade to a new release. This support is very important as FOSSology users create a growing database of scanned source code files which should be maintained with new versions of the software. The script `install/fossinit.php` executes the correct `install/db/dbmigrate*` files depending on the release that is stored in the database and ends up in a well defined state.

While some queries would work well with other database management systems, some specialized queries rely on PostgreSQL, e.g. recursively computing full path names. The performance gain of executing the logic in the database instead of PHP justifies the dependence upon the database technology. An OR-mapper is not (yet) used, due to the large number of complex, highly optimized queries.

PHP Stack

FOSSology prior to version 2.6 did not use any PHP frameworks. The first use is found in release 2.6 which is, strictly stated, not a minor release, because it changed how PHP dependencies were integrated by using the composer package manager. Composer allows for managing libraries and their (transitive) dependencies. The dependency manager for PHP manages updates from the previous releases of dependencies, and also if the system cannot connect to the Internet. This technology change was required due to the end of life of the formerly used PEAR channel.

The transition to a modern and standardized PHP application is an ongoing process with many different aspects. The first aspect is improving testability of new components. Since PHP is used as Web frontend, the structure is continuously improved to ensure a MVC like paradigm. HTML rendering is migrated from PHP print statements to twig templates. The previously mentioned DAO objects have helped to improve security by using an abstraction for database configuration. Then, a

re-factoring aimed at separating logic from presentation and persistence layer code was started. In this presentation and persistence layer, code was replaced with open source components where possible. Most of the required refactoring has been applied from version 2.6 through version 3.1.

License Scanning

Nomos is the main license scanner in FOSSology and it is based on regular expressions. As indicated above, the text formatting and programming language specific comment characters, such as '//' (or "/*", ";;", "REM", "%" and similar variations) present a challenge for regular expressions. To circumvent this problem, Nomos uses short seed expressions to identify regions of interest. It normalizes a portion of the scanned file in the vicinity and then scans for larger snippets. After the list of matching snippets is established, Nomos determines their positions in the scanned file and the snippets are mapped to license findings.

License findings are either positive matches to known licenses with their version, or unknown licenses in the style of a known license. This design guarantees a low false negative rate, as license relevant portions of a file are identified even if the license text is not yet known in FOSSology. Currently, Nomos holds more than 3000 snippets that map to more than 650 licenses.

Apart from the regular agent mode, Nomos can be run in the one-shot analysis mode. Here a single file can be uploaded and is scanned on the fly. If FOSSology is installed, Nomos can also be called from the command line and the output can be directed to standard out for plain text processing of scan results.

One structural disadvantage of matching license relevant text findings with regular expressions is the lack of an ability to detect manipulated license text. While this topic is may be interesting from a legal perspective, custom variants of popular license texts are a problem for tool-based license scanning. One example of this problem is the use of the MIT license and the addition of one or more sentences with extra conditions. A regular expression based approach would consequently identify the MIT license, which is a classic example of a permissive license and would possibly not find any "not-so-permissive" custom additions.

For handling this case the agent named Monk was introduced into FOSSology. This agent considers the reference license text collection from the FOSSology database. Originally these texts were added to FOSSology to allow the user to review the original license texts in the UI. The Monk agent uses these texts to compare with the found text in the files of the uploaded software component. Technically, Monk tokenizes the license reference texts and the text found in a file by space or line break characters. Also common comment characters are filtered. Then, Monk computes the Jaccard text similarity index and adds a weighting to the computed index. The weighting assigns longer text matches with less similarity greater weight than shorter matches with 100% similarity. This is necessary because some longer license text includes shorter license text. If the weighting was not added, the shorter 100% match would always be preferred over longer, but not exact matches.

The obvious disadvantage of Monk is that it recognizes only those licenses which are part of the FOSSology license database. In this way, both the Nomos and Monk agents complement each other: Nomos also detects unknown licensing statements or license texts, however, with less precision. At the same time Monk can give very precise detection results for all known licenses.

FOSSology and SPDX

As mentioned earlier, SPDX represents the de-facto standard for expressing metadata about software packages. SPDX stands for "Software Data Package Exchange" and describes an initiative by the

Linux Foundation to set standards for communicating the components and their licensing.[10] As part of the SPDX effort, the specification also defines a comprehensive list of licenses with standard identifiers and specifications for report formats (cf. SPDX License List). Since FOSSology's main functionality is license scanning, supporting SPDX as a report format represents a natural step. Also the license identification uses the SPDX standard identifiers where present.

Since release 3.0, as mentioned above, FOSSology has had the ability to export SPDX 2.0 reports. Since the generated output was already SPDX 2.1 compatible, it is now also labelled to be, the more up to date, version 2.1.

The two output formats from the SPDX definition are supported. The RDF/xml (a Resource Description Format developed by the W3C) and tag-value. The tag-value format is a more human readable output and is similar to the debian-copyright format as used for Debian packages.[11] These reports represent the result of the scanning and analysis in both machine and (in the case of tag-value) human readable format. Scan results are expressed using the "LicenseInfoInFile" tag, while the analysis result is written using the "LicenseConcluded" tag.

Because the implementation of SPDX report generation uses a template library, FOSSology can also generate the well known debian-copyright files. The major difference between the SPDX tag-value format and debian-copyright is that debian-copyright aggregates files by found (or concluded) licenses while SPDX maintains a listing for each individual file. As such, SPDX documents could be converted to debian-copyright files but not vice versa.

Importing SPDX Documents

In 2017, many tools in the area of license compliance were able to write SPDX documents. Since SPDX format is machine readable it is an obvious idea, to implement importing functionality as well. However, to our knowledge, no license scanning tools (The Open Source project Eclipse SW360 can import an SPDX document to generate license documentation for products) were available in 2017 to read or import SPDX formats. This functionality serves two main use cases:

- If a party receives an SPDX document, how would the receiving party review this document? What would be required is a view where the file or directory structure is shown along with the imported SPDX (licensing) metadata similar to reviewing license scan results provided by the agents.

- If a user requires analyses of a software component, maybe the analysis results of an older version of this component would be available for reuse. Existing analysis results could be available to the public to continue working with for future versions of a software component. Importing existing analysis results helps by reducing effort when analysing new versions of a software component.

Since 2017 FOSSology has been able to import SPDX documents notated in RDF/xml to cover these two use cases. In the same manner as with agent scan results, users can use FOSSology as a tool to verify the information present in an SPDX document when applying it to an uploaded software component. After importing, the necessary workflow is simply the verification of a scan result.

Since the analysis work of a licensing situation can be very time consuming, reuse of existing analyses represents an important capability to reduce effort and avoid duplicate work. FOSSology

10 SPDX Workgroup - a Linux Foundation Collaborative Project: SPDX License List - https://spdx.org/sites/cpstandard/files/pages/files/spdxversion2.1.pdf - 2016

11 Debian Project: Machine-readable debian/copyright file https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/ - 2017

servers can exchange analysis data between each other and FOSSology can exchange analysis information with other license scanning tools, allowing for general reuse between tools.

Conclusion

FOSSology helps to bring clarity to open source licensing, and also supports the adoption of open source software while respecting the intentions of the authors expressed through their licensing. FOSSology itself is licensed under the GPL-2.0 and hosted by the Linux Foundation. Therefore, it matches the slogan "Open Source Compliance with Open Source Tools".

OSS license compliance tooling shall be available to all, including universities, individuals, OSS projects and companies. It should not be the privilege of larger organisations or companies, which can afford to purchase licenses for commercial tools. Since the source code of FOSSology is available, it can be analysed and - if desired - be improved. FOSSology provides full transparency, which improves confidence within the context of license compliance work.

FOSSology has now existed for more a decade. During this time, FOSSology has undergone major renovations in its architecture to keep pace with common technical evolution. It has been improved in the relevant areas of OSS license analysis, such as more precise review functionality, more scanning and detection functionality, automation of conclusions, data exchange using the de-facto standard SPDX and a more modern UI.

FOSSology implements precision, enables workflow and allows its users to review, approve, and correct the results the agents have produced. All these capabilities are required for achieving OSS license compliance.

Although licensing found in OSS components is still heterogeneous and sometimes is expressed in very special ways, its standardization is underway, for example, the Reuse project as proposed by the Free Software Foundation Europe.[12] FOSSology will follow this trend by further automating the license recognition of these standards so that the effort required for manual review is reduced while keeping the high precision and certainty of its license recognition.

Acknowledgement

The authors would like to especially thank Paul Guttmann for his support.

About the authors

Oliver Fendt *has more than 16 years experience in open source software, its license conditions and how to comply to the different licenses. During this time he kicked off several initiatives, among these initiatives are the open source project SW360 and the sponsoring of considerable contributions to the open source project FOSSology. He has developed different trainings about open source software and how to achieve license compliance and has given OSS compliance trainings since 2008.*

Robert Gobeille *is the creator of FOSSology and the original project leader. He works currently in projects with nexB.*

12 Free Software Foundation Europe e.V. (FSFE): REUSE Initiative https://reuse.software - 2017

Maximilian Huber *is a consultant at TNG Technology Consulting Max spends most of the time to develop and support the Linux Foundation project FOSSology and the Eclipse incubator SW360.*

Michael C. Jaeger *is one of the maintainers for the FOSSology project and SW360 (available on Github), both in the area of OSS handling w.r.t. license compliance and component management. At Siemens Corporate Technology in Munich, Germany, Michael works in several roles as project lead, software architect, trainer and consultant for distributed systems, server applications and their development with open source software.*

Johannes Najjar *is a Senior Consultant at TNG Technology Consulting GmbH. He has a background in high energy physics and currently focusses on IOT and Cloud Computing.*

Kate Stewart *is a Senior Director of Strategic Programs at the Linux Foundation responsible for a portfolio of open source projects and standards. With almost 30 years of experience in the software industry, she has held a variety of roles and worked as a developer in Canada, Australia and the US. For the last 20 years she has managed software development teams in the US, Canada, UK, India and China, and focused on delivery of open source based products from Freescale, Canonical & Linaro.*

Steffen Weber *is a software developer with background in algebra and numerics. High ranking in algorithmic competitions favors the focus switch to IT after the PhD in mathematics. Since 2013 he worked as full time developer for projects in certain languages with different frameworks.*

Andreas Würl *has worked for more than seven years as an IT consultant at TNG. His main focus is participating in and improving the agile software development process mainly with sustainable design and architecture. He practises a variety of programming and confguration languages and enjoys contributing to open source software in his free time.*

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.*

Bitcoin: an open source currency and more

Carlo Piana [a]

(a) Founder, Array; attorney (Avvocato) in Milano

DOI: 10.5033/ifosslr.v9i1.120

Abstract
The emergence of a cryptocurrency in the digital domain went unnoticed for years until the general press started to cover Bitcoin's more than tenfold raise in price in the space of a few months in 2017. Earlier on, technical and legal discussion revolved around one of its fundamental building blocks, but Bitcoin is more than the blockchain or an investment object. It is a revolutionary open source artefact that is bound to change the way we consider currency, as well as a proof-of-concept for how parts of international trade could disfranchise banks and other institutions as payment service providers.

Keywords
Law; information technology; Free and Open Source Software; currency; cryptocurrency; blockchain; fintech

During the last part of 2017 Bitcoin has achieved relevance in mainstream media. The debate is focused on "is it worth buying?"; "is it a scam?"; "is it a currency or commodity?"; "is it safe?". These are all legitimate questions but they miss an important trait of Bitcoin: it is open source, peer-to-peer, standard-based distributed technology.

The degrees of freedom in the Bitcoin phenomenon certainly goes beyond the software but the software plays an important role. The fact that Bitcoin is not controlled by any party but instead requires the implicit consensus of users is of utmost importance. It is possible that Bitcoin will never be used as a large-scale distributed currency. However, one fundamental component has definitely already attracted the attention of many, and it is the **blockchain**. Apart from its use as a component of crypto-currencies the concept of distributed ledger based on blockchain has become a highly regarded object of large investment by the technology industry.

The blockchain is important and it has many interesting applications in areas like "fintech" but focusing solely on this overlooks other interesting aspects of the emergence of Bitcoin.

Bitcoin in two simple words: ledger and blockchain

I have read many descriptions of Bitcoin that avoided tackling the technology behind it. These descriptions left the odd feeling that most authors had not made a serious effort to understand the underlying foundation. At the same time, there are so many intricacies that any attempt to explain it can easily get lost in the details and become unhelpful to the non-technical reader. I attempted to

strike a balance in the most relevant economic blog in Italy and – thankfully – the reception of the article by both inexperienced and experts readers was quite enthusiastic.[1] This small achievement makes me sufficiently bold to venture into a similar explanation in this article.

Bitcoin does not exist in the physical world. It is a **unit of account** in a transactional accounting system, otherwise known as a "**ledger**". In a ledger, a party enters records of active (income) and passive (expenses) movements. How much "money" an account holds depends on everything that has been credited to it, less everything that has been spent from it. All movement (after the first one, bear with me for a while, I will explain) have a debited account and a credited account – actually one *or more* of them. When all the credit is spent, the account will show "zero" unless it receives credit and will not be permitted to spend more. The system only permits fully funded expenditures.

This is similar to running a bank account. Imagine the scenario where we have Alpha Bank logging an expenditure on Alice's account, and Beta Bank logging an expenditure in Bob's account. Alice will have a corresponding reduction in her balance, Bob will have a corresponding increase in his, minus any wiring charges applied by the originating or destination bank. This transaction requires two banks that mutually recognize their wiring instructions as money. Each bank gives credit to the other if the transaction is genuine. If Alice's account is not sufficiently funded, Alpha Bank will pay for the transaction regardless, so if bank authorising the transfer is actually opening credit to Alice. The banking system has regulatory requirements and laws that create this trust, lead by an overarching central bank and a system that ensures the balance of all transactions is credited to each bank and that all operations are reflected by the banks' balance sheets.

You trust the compliance and reliability of the regulatory framework between two banks. It requires that banks can only loan within certain limits and under certain conditions the state guarantees the deposits in case of bankruptcy of a bank. In other words, the system and the law create **trust**. Trust about the fact that an entry in an accounting system accrues value that can be later spent to buy goods or services, without any physical object of intrinsic value to vouch for that transaction.

Now, let us remove the banks from the picture and consider the foundation of Bitcoin. In a situation without banks, who guarantees that Alice has money, that she has not spent it, that the transaction credits value that can actually be spent, and that double spending is not allowed? In this simplified framework we have two possibly reciprocally unknown peers, we lack the traditional mechanisms of facilitating and assuring a transaction. How can we operate without a trusted third party? The solution lies in a complex system of peer-to-peer software and algorithms conceived to provide public trust because all transactions are public and apparent to everybody. In this scenario there is no trusted party as with traditional transactions. Instead trust is provided by having many public "eyes" making forgery computationally impossible.

The foundation of Bitcoin or similar systems without trusted or known peers is the **blockchain**. As suggested by the name, the blockchain is a chain of blocks which are continuously created and which contain the **ledger**. The latest block and all predecessors contain (nearly) all transactions that have ever occurred. The ledger is not a seamless log, it is a database made of chained blocks of text that are widely replicated and distributed. These blocks are created in a peer-to-peer network that is **public and open** both because everybody can *read* it (by downloading the entire blockchain from one or more peers), *and* because anybody can contribute to *creating* it. By downloading and reviewing an updated copy of the blockchain any peer can safely tell the balance of every account by checking all inbound and outbound transactions.

1 The blog was published by Econopoly, which is the avenue of publication of external writers of the largest Italian economy newspaper, Il Sole 24Ore. The article attracted 400,000 reads only in its first day of publication (a Sunday). http://www.econopoly.ilsole24ore.com/2017/12/17/bitcoin-bolla-o-souffle/ [Italian]

How the blockchain is generated (in Bitcoin)

On average every ten minutes a block is produced and distributed. The block is linked to and depends on the previous one. The main payload of each block is a list of (un)**validated transactions**. As soon as the block containing them is accepted as part of the blockchain, those transactions become validated and final.

"Validated" means that the transaction originates from a sufficiently funded "account" and has been signed with a valid unique private key. Given these conditions it shall prevail against any subsequent conflicting transaction, resolving cases of double expenditure. Trust that the transaction is irrevocable is therefore achieved only when it is integrated in the blockchain.

But **who creates those blocks**? Interestingly in Bitcoin and similar blockchain-based technology this can be anybody who has invested sufficient resources to "mine" the next block in the chain. This is where things get really interesting.

The activity of creating blocks is called "**mining**". Mining reflects the idea that – as with gold – blocks are figuratively dug out of the ground. The creation of the blocks is an expensive and rewarded task in order to ensure that incentives to take over the blockchain for nefarious reasons are counterbalanced by the effort it would require to overspend the rest of the network and accomplish such result. The more computing power needed to create a block, the higher – by a factor of many millions – the collective effort put into being the first one to publish a new block. In order to be *half sure* (50% chance) to create a specific new block you would have to provide at least *half* of this collective computing power.

This race makes it highly unlikely that somebody would take over the process as too many parties have a competing interest.

Who holds the checked flag to tell that you have been successful? "Nobody" and "everybody" are both acceptable answers. To be accepted your proposed block must meet two basic conditions: to be formally impeccable (including containing only valid transaction) and to show proof to have solved a **mathematical puzzle**. In other words, you must show a **proof-of-work**, a decision made not by a person but by the protocol itself, which is designed to rapidly resolve any potential dissent.

The puzzle, the proof-of-work, is a kind of treasure hunt, where any solution brings you to the next puzzle, and you can start solving the next puzzle only after the previous one has been completed, so no significant head start can be achieved by any one party.

The block is formed of pure text, arranged in a pre-defined way. One of the parts that must appear in the block is **a string that uniquely identifies** the previous block. This string is mathematically calculated using a **public, open algorithm** called a "hashing algorithm".[2] The hashing algorithm, if applied to a block, irrespective of the length of the originating block, gives a fixed-length hexadecimal[3] string called a "hashing footprint" or simply **hash**. It is computationally easy to calculate the hash from the originating block. Because this is a deterministic algorithm anyone with the same originating block will obtain the same hash. It is almost impossible that two different blocks could originate the same hash if the hash is sufficiently long (the possible combinations are 16^{64}).

The algorithm is designed so even inconspicuous variation in the originating block will generate a significantly dissimilar hash. An example could provide a more graphical explanation:

The quick brown fox jumps over the lazy dog

2 Bitcoin uses SHA-256, a hashing algorithm contributed by the NSA.
3 it contains numbers from 0 to 9 and letters from a to f

Generates the following hash:

c03905fcdab297513a620ec81ed46ca44ddb62d41cbbd83eb4a5a3592be26a69

By changing the capitalization of the first letter of "The" the hash changes to:

1153a4080f1fcb04425aa0b841c2b14606fe6df25d9076d2a1face2d5af57129

It is impossible to mathematically calculate how the originating block must be changed to obtain a hash with a given content. The only way to obtain such information is to "*brute force*" the result.

Let us suppose that a participant must find a hash with two consecutive examples of the letter "a". She would probably attempt a few random hashes hoping to stumble into a valid outcome. In the example above I was successful in just two attempts. Let us make it harder, by asking that the matching string must be *at the beginning* of the hash. I can count how many possible combinations I have, knowing that only one valid combination among them: $16^2 = 256$ possible combinations. If the stakes are higher, the odds of winning must be lower. This is achieved by increasing the number of digits in the "winning" combination.

The challenge in Bitcoin mining is finding a block which has a hash lower than a certain value and therefore has with a certain number of leading zeros. Currently, this number is 16, which computes to 1 in $16^{16} = 1{,}844 * 10^{19}$ combinations, or *one in eighteen trillion trillion* combinations.

There is more. A miner must find a block that generates a sufficiently low hash, that is well formed to contain only valid actual transactions *and* that **contains the hash of the previous block**. That means that the process begins only after the previous block has been published, providing a average time to solve the puzzle limited to **10 minutes**.

This is hard-coded in the software. The protocol is self-adjusting, increasing the difficulty as soon as the blocks start to be generated at a faster pace. As such the protocol is conceived to resist both an anticipated increasing success (with more invested resources) and to Moore's law (computing power becomes cheaper and more available over time).

A (well) rewarded effort

Why should one invest the relevant resources required to solve such a difficult puzzle? This activity is **well rewarded**. The reward consists in an amount of Bitcoins and is how every past and future Bitcoin enters the system.

During 2017 Bitcoin has jumped over USD 10,000 per unit and is swiftly moving towards a valuation of around 20,000. Therefore the reward to obtain new coins is huge. Who arbitrates ownership? Since there is no tribunal, no central bank or other authority, the system is **self-governing**.[4] The first to achieve a result publishes the block. The result is swiftly propagated to all nodes. All mining nodes will then decide in a matter of milliseconds that it is time to move onto the next block. In the unlikely, but not impossible, case that two miners publish their own block simultaneously the blockchain spawns into two different **branches**. This means that nodes start receiving two different blocks for the next few places. Again, the software dictates that the longest chain wins, and since the pace will inevitably be different, as the branch with more computing power attached will outpace the other, eventually the weaker branch will die off because all the blocks in

4 "Code is Law here", literally. This is a quote by Lawrence Lessig, *Code is Law – On Liberty in Cyberspace*, Harward Magazine 2000. https://harvardmagazine.com/2000/01/code-is-law-html

the losing branch will be unable to spend their reward. This state of uncertainty has been experienced for up to an hour in the past (six blocks).

When a brand new block is created it carries freshly minted, or mined, Bitcoins. The miner will have associated her account to the Bitcoin and in the process will have generated a private key permitting her to spend the Bitcoins. This is via generating an outbound transaction with that block known as "entry point". The private key is the only enabler of this transaction. While the transaction is not authorised by a third party anybody can see where the transaction comes from and valid holder of the corresponding secret key, and this is all it is required to computationally assert trust.

The recipient of the transaction will have increased credit, and will use her own private key to make all subsequent transactions related to this and other credit. All these transactions are made available to all miners, and these miners will collect and place them into their candidate blocks.

The amount of awarded Bitcoins **halves** at given intervals, therefore it will come a time when the generated Bitcoin will be below the minimum amount of Bitcoin that can be spent (one hundredth of a millionth of a Bitcoin, currently). As the Bitcoin yielding curve is logarithmic, there will ever be **21 million** usable Bitcoins.

As soon as we will approach the upper limit, what would be the reward, as the newly minted Bitcoins will only be issued in ever-smaller fractions? What would compensate the effort of making new blocks, and make sure nobody is in a position to game the system as soon as the proof-of-work will be less demanding?

The reward is not only in the minted Bitcoin. There is an (optional) reward consisting of a **fee** that the parties in a transaction offer to those who publish the transaction in their blocks. The higher the fee the more likely it is that the transaction will make the ledger. A fee-generating transaction is (naturally) prioritized over the non fee-generating ones. Therefore, even in the future, a sufficient incentive not to meddle with the blockchain growth process should be guaranteed.

Who has invented it?

The crypto-currency christened "Bitcoin" was allegedly conceived by Satoshi Nakamoto – a pseudonym with no known author (or even authors) – who delivered the concept and the first iteration of the open source software tools. "Satoshi" published an academic paper to describe the working of the crypto-currency based on a distributed ledger, peer-to-peer network operating a blockchain and a private-public key pair system.[5]

Ththe system can be summarised as follows:

> *The steps to run the network are as follows:*
>
> 1. *New transactions are broadcast to all nodes.*
> 2. *Each node collects new transactions into a block.*
> 3. *Each node works on finding a difficult proof-of-work for its block.*
> 4. *When a node finds a proof-of-work, it broadcasts the block to all nodes.*
> 5. *Nodes accept the block only if all transactions in it are valid and not already spent.*
> 6. *Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.*

5 The paper is available at https://bitcoin.org/bitcoin.pdf

Is it Currency or is it a commodity?

A commodity is a kind of good which is traded by its quantity and quality, not as an individual item. Currently, Bitcoin is traded as if it was a commodity, but this state of affairs is unusual because a commodity has an intrinsic value and is useful per se. Some commodities are consumable, which means that they naturally perish and must be consumed by a "best-before" date, leading to stock-keeping incurring a seizable fraction of their value. This is the case with oil, gas, frozen orange juice, barley and coffee. Other commodities are not naturally consumable and are traded over longer time periods, even indefinitely, since they are more easily stored.

Bitcoin as a commodity and as a high-yielding tradable object is therefore somewhat unusual. It may be regarded a perversion of the original system intent, because by its nature Bitcoin should have a *transaction* value rather than a value of its own. The success of Bitcoin is also one of its most striking current shortcomings. It is very difficult to use a currency whose value floats between wide margins day over day. This volatility impedes an important characteristic for a currency: to express an intermediate value by which, in an economic system, players can exchange currency against goods or services, allowing the recipient of the currency to consistently trade it with an equivalent value in goods or services. All the values on the market can therefore be coherently expressed in one single unit of measure that permits ta calculation of all relative "prices" (e.g., how many man/hours work is worth a car, how many movie tickets can I buy by selling a loaf of bread, etc.).

One of the basic functions of a currency is normally understood to be a reasonably stable **price system**. This is not always strictly true, with an example being that in Europe several hundred million people at once started using a totally new currency called the euro in a nearly zero-inflation zone. Many people continued making a mental conversion between the new currency to the old one, because their experience had stratified over the years, while the new unit did not hold much meaning for them. Nonetheless, the new currency was adopted before a new set of relative prices had sunk in older people's habits (some still make this conversion, which after nearly twenty years has very little meaning).

Therefore, it is not *essential* for a currency to represent such a social reference system, and this aspect of Bitcoin is not without precedent. An absolute role is also not played by another commonly recognised function: that of **accumulating value** for later expenditure. This important function is not well served in hyperinflation situations (e.g. the one currently experienced in Venezuela or Zimbabwe), when prices may significantly within the day. Of course even in these situations there is no doubt that the national currency is still a currency, albeit it cannot be kept for very long without losing its presumedvalue.

What makes a currency a currency (even an open source one)?

Bitcoin, as many other open source revolutions (Internet, Free and open source software, open content, open data, etc.) forces us to rethink what we know about economics from the perspective of openness and lack of control. It suggests that the time is right to reassess some about currency: that there cannot be currency without a legal tender emitted by a central bank. At least theoretically it has been demonstrated that an anarchic, uncontrolled, distributed payment system with a currency of its own *is possible*.

This is not the first time the concept of "currency" has undergone a re-thinking process. In ancient times, it was thought that a coin held worth because it had an intrinsic value, that of the materials it was forged from. Even back then, this was only half-true, as the implicit function of the coins was to represent an easily-accumulated, stored, transported and exchange token at conventional value in a shared prices system. The coining material had value but the value of an object still depended on the

perception of the parties involved in each transaction. Accepting the coin did not indicate interest in using the metal but rather to further exchange it against another item. This implied that the important value was not in the meal but rather in what it could accomplish as coinage.

At the end of the day, we can safely state that any kind of currency, from its users' perspective, is an **implicit contract**. When we buy something, the buyer and seller settle for a price attached to the currency as a value which is measured against all other values. Even when a price is fixed or imposed, individuals can still decide whether to trade for that price or not. The buyer and seller know the quantity of currency required to buy a certain quantity of goods or services. The worth of the currency emains implicit. If the price is not sufficient, the deal is not made.

A Dollar is worth a Dollar. An euro is worth an euro. If by magic everybody had ten times more Dollars, and prices, obligation or debt was equally increased, everyone's wealth would remain perfectly static.

The fact that a currency is **legal tender** is often cited as a reason why Bitcoin is not a currency. However, this holds dubious merit. "Legal tender" means that one cannot refuse payment of a debt made by offering that the assigned currency. Conversely, nobody can be forced to accept currency which is not legal tender unless settlement with that particular currency had previously been agreed upon. None the less, in certain countries, especially those suffering from high inflation, sometimes a parallel market expressed in a foreign and more stable currency appears, something called "**dollarization**" due to the frequent use of the US Dollar for such parallel trade. Often this dollarization is illegal, and even importing foreign currency is or outright illegal, or subject to tight control.

But using foreign currency is not illegal per se and a foreign currency can be chosen by the parties in a transaction to settle the dues originating from their relationship. In Italy, for example, the Civil code was issued during WWII addressed a period of strong autarchy imposed by a fascist regimebut conversely was quite liberal regarding the use of foreign currency. It allowed full address of obligations which are expressed in currency "which is not legal tender within the State". The debtor could offer to pay the equivalent of the chosen currency in the national legal tender at the exchange rate at the time when the debt is due (art. 1278). However, this legal option can be originally excluded by the parties, and this exclusion holds as long as the currency in which the obligation is denominated can be easily obtained. There is no reference to the fact that an alternative currency must be legal tender some country (the law does not mention "foreign", but just "not legal tender in the State") so the question of what happens outside the domestic jurisdiction in terms of legal tender is in fact irrelevant. The result is that currency which has no legal value can be treated as holding the same value as the legal tender. This holds true in general, as an obligation in foreign currency is treated as a monetary one, not as a barter, as would happen if it was a commodity.

It is therefore reasonable to conclude that Bitcoin was born as a system to provide a generally available, all-purpose payment system to transfer value between parties irrespective of the underlying obligation, which is ultimately the role of currency. It is, if used properly, money, in a manner that is not dissimilar to foreign currency.

Conclusion

Money is a fundamental element of a complex society. A complex society tends to have money and a denominated price system even in the absence or against the intervention of a State. It follows that money is what makes trade possible, trade is what makes a liberal society thrive, and therefore money is an item of critical importance. Controlling money is a way by which governments can help or hinder their citizens. Having a currency that can be used outside the banking and financial systems is an option that cannot be disregarded lightly or labeled dismissively as a "black economy."

Bitcoin is the first software-defined currency with a complete system of its own. Despite its many shortcomings, such as the environmental cost of making and maintaining it or the already discussed volatility, or the relative uncertainty of when a transaction is final, or the associated fraud causing significant losses, Bitcoin has been used and it is accepted in limited but not irrelevant cases for small to very large transactions as was originally intended. At times, it is used to find a workaround for payments in national tragedies, lack of democracy, nefarious governments, and it allows people to keep some space from illiberal constraints of dictatorship.

Bitcoin is open source and it inherently fits into the broader ecosystem of open technologies and solutions. It is based on publicly available, open standards and infrastructure suchas the Internet. It something that until a few decades ago would have been unthinkable. The question is where it will go next.

About the author

Carlo Piana *is an Italian attorney admitted to the Milan Bar. It has founded Array, a new concept law firm (Array is an array) which specializes in Information Technology Law and Free and Open Source Software in particular. External General Counsel of the Free Software Foundation Europe, is a member of the Editoria Committee of IFOSSLR and a member of the Council of the Legal Network, the initiative sponsored by FSFE to discuss under the Chatham Rule the bleeding edge of openness-related legal issues.*

DOI: 10.5033/ifosslr.v9i1.120

The Bid by OpenChain to Transform The Supply Chain

Shane Coughlan,[a]

(a) OpenChain Project Director, Linux Foundation.

DOI: 10.5033/ifosslr.v9i1.122

Abstract
OpenChain aims to increase open source compliance in the supply chain. This issue, which many initially dismiss as a legal concern or as low priority, is inherently tied to ensuring that open source is as useful as possible with as little friction as possible. In a nutshell, because open source is about the use of third party code, compliance is the nexus of where equality of access, safety of use and reduction of risk can be found. OpenChain is built to increase trust between organizations to accomplish this.[1]

Keywords
Law; information technology; Free and Open Source Software; Supply Chain; Compliance

Today many companies understand open source and act as major supporters of open source development. However, addressing open source license compliance in a systematic, industry-wide manner has proven to be a somewhat elusive challenge. The global IT market has not yet seen a significant reduction in the number of open source compliance issues discoverable in areas like consumer electronics over the last decade.

The majority of compliance issues originate in the midst of sharing multiple hardware and software components between numerous entities. The global supply chain is long and the participants are simultaneously intertwined and disparate. It is perfectly possible to have companies making hardware, companies making software and companies doing both collaborating around a relatively small component. The results in terms of products are often outstanding but the challenge of keeping track of everything is substantial.

Compliance Is A Process Challenge That Spans Multiple Organizations

Open source presents a specific challenge in the global supply chain. This is not because open source is inherently complex but rather due to the varying degree of exposure and domain knowledge that companies possess. By way of example, a company developing a small component that requires a device driver may have staff entirely unfamiliar with open source. One mistake, one misunderstanding, and one component deployed in dozens of devices can present an issue. Most compliance challenges arise from mistakes. Few, if any, originate with intent.

1 This is an edited version of an article originally appeared on OpenSource.com

Ultimately solving open source compliance challenges involves solving open source compliance in the supply chain. This is no small task: there are thousands of companies across dozens of national borders using numerous languages in play. The solution lies beyond the realm of inter-company negotiation. To address open source compliance challenges the global supply chain must align behind certain shared approaches.

Because No Single Company Makes A Finished Device, No Single Company Can Solve Compliance Challenges

Awareness of this fact and the provision of a practical solution are two different matters. It takes time for ideas and suggested approaches to percolate and mature. It took input from lawyers and managers and developers and political scientists. It took, in short, a while for the ingenuity of the human community to bounce ideas back and forth until a simple, clear approach could be found.

The Best Solutions Are Often The Simplest, With The Lowest Barriers To Entry

The OpenChain Project formally launched in October 2016 and is hosted by The Linux Foundation. It originated in discussions that occurred three years earlier and continued at an increasing pace until a formal project was born. The basic idea was simple: identify key recommended processes for effective open source management. The goal was equally clear: reduce bottlenecks and risk when using third-party code to make open source license compliance simple and consistent across the supply chain. The key intention was to pull things together in a manner that balanced comprehensiveness, broad applicability, and real-world usability.

OpenChain Is Intended To Make Open Source License Compliance More Predictable, Understandable And Efficient For The Software Supply Chain

The OpenChain Project is trying to build and disseminate an industry standard for license compliance. It is designed to be the foundation for open source compliance in the supply chain. Engagement and adoption is simple, free and supported by a vibrant community backed by leading multinationals across multiple sectors.

There are three interconnected parts to the OpenChain Project. A Specification that defines the core requirements of a quality compliance program. A Conformance method that helps organizations display adherence to these requirements. A Curriculum to provide basic open source processes and best practices.

A Simple Specification That Explains The Key Requirements Of A Quality Compliance Program

The core of the OpenChain Project is the Specification. This identifies a series of processes designed to help organizations of any size to address open source compliance issues effectively. The main goal of organizations using the OpenChain Specification is to become conformant. This means that their organization must meet the requirements of a certain version of the OpenChain Specification. A conformant organization can advertise this fact on their website and promotional material, helping to ensure that potential suppliers and customers understand and can trust their approach to open source compliance.

A Clear And Free Way To Check Conformance With The Specification

OpenChain Conformance can be checked via a free online self-certification questionnaire provided by the OpenChain Project. This is the quickest, easiest and most effective way to check and confirm

adherence to the OpenChain Specification. There is also a manual conformance document available for organizations whose process requires a paper review or disallows web-based submissions. Both the online and the manual conformance can be completed at a pace decided by the conforming organization and both methods remain private until a submission is completed.

A Curriculum To Support Conformance And With Broader Questions Of Training And Processes

The OpenChain Curriculum helps organizations meet certain aspects of the OpenChain Specification. It provides a generic, refined and clear example of an open source compliance training program that can either be used directly or incorporated into existing training programs. The knowledge it contains can also be applied to adjusting or adopting various processes for managing open source inside an organization. The OpenChain Curriculum is available with very few restrictions to ensure organizations can use it in as many ways as possible. To accomplish this it is licensed as Creative Commons – Zero[2] (CC-0), effectively public domain, so remixing or sharing it freely for any purpose is possible.

Community and Support

The OpenChain Project provides what we believe to be a compelling approach to making open source compliance more consistent and more effective across multiple market segments. However, good ideas need implementation, and in the context of open source this inevitably hinges on the creation of a supporting community. The OpenChain Project at the time of writing has twelve Platinum Members that support its development and adoption: Adobe, ARM, Cisco, GitHub, Harman, Hitachi, HPE, Qualcomm, Siemens, Toyota, Western Digital and Wind River. It also has a growing community of almost 200 participants on the main mailing list.

At its core the OpenChain Project is about providing a simple, clear method of building trust between organizations that rely on each other to share code and create products. Any organization that is OpenChain Conformant is aligning behind key requirements that their peers agree are required in a quality compliance program. This is about confirming overarching processes and policies, while allowing the specifics of each process and policy to be crafted by each organization to suit its specific needs.

Conclusion

The OpenChain Specification is ready for adoption by any organization that creates, uses or distributes free and open source code. The online conformance is free of charge, the mailing list and Work Team calls are open to everyone. Arguably, this is the first time a single, unifying approach to addressing the challenge of open source compliance in the supply chain exists.

References

- About the OpenChain Project: https://www.openchainproject.org/quick-start
- About the OpenChain Specification: https://www.openchainproject.org/spec
- About OpenChain Conformance: https://www.openchainproject.org/conformance

2 https://creativecommons.org/publicdomain/zero/1.0/

- About the OpenChain Curriculum: https://www.openchainproject.org/curriculum

About the author

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.*