December 2020

Finding out how close source code files are to files in the Git version control system.

Armijn Hemel

Finding out how close source code files are to files in the Git version control system.

Introduction

A lot of popular software is developed using a version control system. Historically systems such as RCS, CVS and Subversion were used, but a lot of developers have moved to Git[i] . These systems have a lot of information available about the history of a file. When software is distributed, it is often distributed without this history information. In some situations it is important to find out how close a certain piece of software is to any given version in a version control system, for example for assessing copyright, security research or other provenance issues.

Tags

Git, software engineering, security, defect discovery, tlsh

Detailed description

For various reasons it can be important to find provenance of a piece of software. One example is to see how much a vendor's source code tree is deviating from an upstream open source project and potentially missing out on important fixes, or duplicating already existing work (example: fixing bugs that have already been fixed in another tree, branch or version).

For open source software this can be efficiently automated, because even though open source software can be modified, modifications tend to be fairly small and there are no radical departures from the initial code base: research has shown that for popular software like the Linux kernel typically 98% to 99% of the source code is reused unmodified and that only a handful of files are really changed[ii]. The other files that are not in the upstream version are a mix of existing files that have been (very slightly) modified, or new files that have been added.

For the modified files it is important to find out how much they actually deviate from the version, or perhaps any version that is upstream, for the new files it is important to find out if they might have been based on existing files.

Development in Git is distributed by design. It is not mandatory to send changes back to the “main” repository and after a repository has been “forked” the code will typically start to deviate. Developers of consumer electronics devices often “cherry pick” changes from several repositories, or create their own (internal) repository that incorporate changes from several repositories, which might or might not send changes back to the “main” repository. For example, a company developing a Linux based

product might start with a Linux kernel repository obtained from a chipset manufacturer, that used the main Linux kernel repository, then backported fixes from the long term stable kernel, added their own code and changes that have not yet been incorporated into the main repository, and add some other code from third party repositories (for example: device drivers). The company developing the Linux based product might add their own changes or improvements. In a typical workflow for the Linux kernel there are easily 5 or more repositories involved that can be regarded as “upstream”.

Finding the closest match for a single file

When trying to find the provenance of a file that was obtained from Git (and which might, or might not, have been modified afterwards) it might not be good enough to check a single branch in a Git repository, or all branches (“tree”) in a Git repository. It might be necessary to check many branches in possibly many trees (“forest”) to find the closest match for a file.

To find the closest repository and revision a combination of exact and fuzzy matching can be used. For exact matching a standard cryptographic checksum like SHA256, SHA1 or MD5 can be used. For fuzzy matching a “locality sensitive hash” such as TLSH[iii] can be used.

The process first analyses all commits to every branch in several Git trees. For each commit the change (in “patch” form) will be retrieved. The patch will be analyzed to see which files were changed (only additions of files, or modifications of existing files, while property changes or deletions will be ignored). Then the file will be checked out for that revision and at least two checksums will be computed namely a regular checksum (SHA256/SHA1/MD5/etc.) and a locality sensistive hash (TLSH/other locality sensitive hash) and stored in a database with meta information such as but not limited to filename, full path, revision and name of the repository.

When a file needs to be researched the following steps are taken: first checksums (SHA256/SHA1/MD5/etc.) are computed and looked up in the database. If there are one or more matches, the matches will be reported.

For files for which no match could be found in the database the locality sensitive hash (like TLSH) is computed. This hash is then compared to the locality sensitive hashes stored in the database to find the closest match. Using some metainformation (path, filename) the list of hashes to compare with can be reduced. The metadata of the match that is closest will then be reported.

Steps for finding the closest match for a single file

To create the database with files added in each commit perform the following steps:

1. use “git clone” to create a local copy of a git repository (for example: Linux kernel)
2. use “git tag” to find out all the tags (branches) that exist in the git repository. Optionally these can be filtered to remove unwanted tags or branches. Alternatively a list of branches or tags that need to be analysed can be specified.
3. for each tag from step 2 find all revisions/commits using “git rev-list” and store these in a list.

Optionally commits that represent “merges” can be ignored in this step.

4. for each commit in the list from step 3
    a) run “git show” with the commit as parameter
    b) extract names of files in the patch (if any) and discard results that are not useful, such as files that were removed, additions/removals/modifications of directories, modification of permissions of files, symbolic links, etcetera.
    c) for each file from the previous step do the following:
        i. run “git checkout -f” with as parameters the Git commit hash and the name of the file to get the state of the file as it was at the time after the patches from the commit have been applied, for example:

            $ git checkout 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2 -f Makefile

            to checkout the file “Makefile” from revision 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
        ii. compute SHA256/SHA1/MD5/etc. as well as locality sensitive hash (like TLSH) for the file that was checked out in the previous step
        iii. store checksums in a database, together with metadata like the URL of the Git repository, commit id of the file, file name and pathname and possibly other metainformation

Then when an archive is analyzed perform the following steps:

1. compute checksums (SHA256/SHA1/MD5) and possibly locality sensitive hash (TLSH) for each file in the archive
2. search database to see if any of the checksums computed in step 1 can be found in the database and report these together with Git URL, commit information and possibly any other metainformation stored in the database.
3. for files for which there were no matches do the following:
    a) compute locality sensitive hash (if not already done in step 1)
    b) compare locality sensitive hash to locality sensitive hashes in the database to find the closest match and report this together with Git URL, commit information and possibly any other metainformation stored in the database. As an optimization the list of candidate files that needs to be compared can be shortened by only looking at files with the same file name.

i Git distributed version control sytem - https://git-scm.com/
ii WCRE 2012 proceedings: Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices - http://www.computer.org/csdl/proceedings/wcre/2012/4891/00/4891a357-abs.html
iii Trend Micro Locality Sensitive Hash - https://github.com/trendmicro/tlsh