Fine(r) grained provenance detection

Armijn Hemel
Tjaldur Software Governance Solutions
armijn@tjaldur.nl

April 14, 2016

About Armijn

- using Open Source software since 1994
- MSc Computer Science from Utrecht University (The Netherlands)
- core team `gpl-violations.org` from 2005 - May 2012
- owner Tjaldur Software Governance Solutions since May 2011
- creator of the Binary Analysis Tool (BAT) for compliance engineering of binary files

I am not a lawyer. Nothing I say is legal advice.

About Tjaldur Software Governance Solutions

- tooling (BAT, source code scanning)
- research (provenance, build tracing, audit techniques, supply chain issues)
- GPL compliance engineering
- audits (for OSADL and others)

Today's topic

Today will be about fine grained provenance scanning of source code.

This is an extension of the “Yaminabe project”, part of Linux Foundation's Long Term Support Initiative (LTSI). The Yaminabe project methods also form the basis of the OSADL license compliance audit.

The research I am presenting today was funded by Renesas.

OSADL license compliance audit

The OSADL license compliance audit (LCA) is a scoped product audit looking at various legal and technical aspects.

The LCA is limited to a single device/firmware revision combination.

Two components on the system are audited:

1. Linux kernel including external kernel modules
2. C library (glibc/uClibc), including how other components are linked to it

The audit is restricted to license compliance: detecting code theft/plagiarism, patent issues and license compatibility of user space programs are not part of the audit.

After successful completion of the audit the company gets a certificate.

Technical work done during the audit

1. license scan
2. linking analysis
3. rebuild of kernel + C library/toolchain

Focus today is on the scan bit and especially about reducing the search space and finding differences to known code.

Reducing the search space for the license scan

Even though scope is limited a full scan is still a lot of work: recent Linux kernels have over 40,000 (interesting) files.

Looking at each file would be very costly (at 1 second per file it would be 11+ hours), so we take a few shortcuts.

1. keep a database with checksums of all files in upstream `kernel.org` kernels
2. determine a checksum for each file in the kernel we audit
3. if the checksum for the file can be found in the database we trust it and ignore it
4. look in more detail at files that can't be found in the database

For the Linux kernel this typically eliminates between 95% (but usually closer to 98%) of the files and allows us to focus on the *real* problems.

Evaluation of using cryptographic checksums

Using cryptographic checksums is effective. It is fast and changing even the tiniest bit of data means that you get a completely different checksum. In newer algorithms “collisions” (where different input maps to the same checksum) are rare.

It does not say how *close* differences are and when looking at just the checksums of two different versions of a file you cannot say if they are very close or very different.

Solution: locality sensitive hashing

“Locality sensitive hashes” is a different class of hashes that allow you to tell how close a match is, by forcing collisions every now and then for data that is similar. One such hash is “TLSH” from TrendMicro.

Two TLSH hashes can be compared. When comparing TLSH outputs a number that is the *distance* between the two files: if the number is low (1 or 2) it is very close (one line, a few lines). If the score is high (400 or higher) the files are very different.

Using TLSH the question “how close is a file to upstream” can be easily answered and quantified: it can be expressed in a single number.

Steps to find closest file in a set of source code archives

1. compute SHA256 and TLSH checksums for every file in all upstream source code releases and store these in a database
2. compute SHA256 and TLSH checksums for all files in a source code archive (example: vendor kernel release)
3. discard all files from 2 that are known in 1
4. per unknown file from 3 assemble a list of candidates to compare to:
   - 4.1 file name/filepath
   - 4.2 identifiers (function names, string literals, variable names, etc.)
5. for each unknown file from 3 TLSH checksum with TLSH checksum of each candidate
6. report SHA256 checksum for file with lowest TLSH score in step 5. This is the closest match.

Use cases

Some use cases why this is very useful information to know:

- getting a feel how far upstream releases are (for upstreaming purposes)
- discovering (partial) provenance for copyright and licensing reasons

Pushing limits further: comparing code to Git

A lot of code is in Git and it might be useful to compare to see how close code is to a branch in a Git tree, all branches in a Git tree, or to all branches in a Git forest.

Use cases:

- discovering if code has already been upstreamed
- provenance tracking across repositories of patches ("did we get a patch from a repository we do not trust?")

For this we need to explode all code in a Git tree, or multiple Git trees, compute SHA256 and TLSH and possibly more information and store it out of band.

Exploding Git information

1. run `git clone` for a repository
2. find out all the revisions/commits in the Git repository using `git rev-list --all`
3. for each revision/commit do:
   - 3.1 run `git show $commit`
   - 3.2 process files in patch and filter unnecessary results (files that were removed, directories, symbolic links, duplicates, etc.) and store these in a list with interesting files
   - 3.3 for each file in the list of interesting files:
     - 3.3.1 run git show for the file
     - 3.3.2 compute SHA256 checksum and TLSH checksum for file
     - 3.3.3 store SHA256 and TLSH checksums in a database together with commit, filename and URL of the Git repository
     - 3.3.4 process each patch and compute a Git patch id using `git patch-id`. Store the resulting checksum in the database too.

Using exploded Git information

1. compute SHA256 checksums for all files in an archive
2. filter out all known files using SHA256 checksums
3. compute TLSH checksums for all unknown files
4. per unknown file assemble a list of candidates using:
   - 4.1 file name/filepath
   - 4.2 (optionally) identifiers like function names, string literals, variable names, etc
5. for each unknown file compare TLSH checksum to TLSH checksums of the candidates.
6. report Git repository URL and Git revision for file with lowest score in step 5.

Trust and priorities

In Git land on a technical level all repositories are equal, but on a social/project level they are not.

In the Linux kernel the repositories “linux” (Linus Torvalds) and “linux-stable” are very central, but a random repository from a developer on GitHub is not.

Some repositories might be trusted, some untrusted and it is good to see if code is in a trusted repository (when it is OK to use) and when only in an untrusted repository (when you don’t want to use it).

Trust and priorities differ per person.

Finding cherry picked patches

In some cases it is possible to find out if a file is actually a base version plus a single patch that has been cherry picked from a commit:

1. use `git show` to show the contents of a file in a repository and write it to a temporary location
2. use `git diff` to compute differences in both directions between an unknown file and the file from step 1
3. use `git patch-id` to compute patch ids for both patches
4. lookup patch ids from step 3 to see if it is known patch and report Git URL and revision

Results (1)

For Renesas a few scans were done to find out how close one of their new products (R-Car, 3rd generation) is to mainstream.

The product used a Linux kernel 4.3 release candidate as a base version and was then further developed, with patches being upstreamed through various repositories and maintainers.

A database was built with data from 6 repositories from kernel.org, including Linus' repository, linux-stable and a few repositories of known Renesas contributors/contractors. Each was assigned a priority (Linus' repository the most important, then linux-stable, etc.).

Several scans were done to see how close a certain revision was to the collective data in the Git trees, where each time one more Git tree was marked as "untrusted".

```
SCANNING 40434 files
588 FILES NOT FOUND IN DATABASE
COMPUTING AND COMPARING TLSH OF FILES NOT
  FOUND IN DATABASE

CLOSEST REVISION FOR drivers/spi/spi-sh-msiof.c IS
84a73014d86fd660822a20c032625e3afe99ca58 FROM
git://git.kernel.org/pub/.../torvalds/linux.git
WITH DISTANCE 25

CLOSEST REVISION FOR drivers/gpu/drm/rcar-du/rcar_du_crtc.
266c73b77706f2d05b4a3e70a5bb702ed35431d6 FROM
git://git.kernel.org/pub/.../torvalds/linux.git
WITH DISTANCE 50

...
```

```
=======================================
               SUMMARY
=======================================
FILES SCANNED: 40434
FILES FOUND IN UPSTREAM RELEASE: 39846
FILES NOT FOUND IN UPSTREAM RELEASE: 588
TOTAL DISTANCE: 5721
IDENTICAL FILES IN GIT: 660
NOT MATCHED IN GIT: 6
UNDETERMINED IN GIT: 0
0-60: 52
61-150: 7
over 150: 6
```

(whoops, slight bug there? ;-))

Results (2)

The more repositories were disabled, the bigger the difference (higher TLSH score). This was expected, as some patches were still travelling on their way to mainline according to Renesas.

From Git alone it was not possible to see in which direction patches are flowing and if the repositories are a “source” (patches were pulled from it), or a “sink” (patches were pushed to it) and if the patches were already on their way to mainline, or if they could be an evolutionary dead end.

Alternative scanning methods

The above described method compares files to *all* files in a Git forest at once, completely ignoring "time": if files are removed from newer versions the above method will still happily report them.

Sometimes it is better to compare single Git tags (that are pinned to a certain revision in time) in repositories. A script written for this showed higher TLSH scores, confirming that the Renesas product is an amalgation of various versions and various repositories, as expected.

Recently Renesas tagged a new version, based on a newer Linux kernel version (around kernel 4.5) and scans were repeated. Although there were many more files the scores were *dramatically* lower.

```
comparing tags rcar-3.0.0 and v4.3-rc1
FILES SCANNED: 40393
TOTAL DISTANCE: 154704
IDENTICAL FILES: 37161
0-60: 2684
61-150: 268
over 150: 95

...

comparing tags rcar-3.2.0 and v4.5
FILES SCANNED: 41392
TOTAL DISTANCE: 110693
IDENTICAL FILES: 38654
0-60: 2401
61-150: 139
over 150: 65
```

Future work

It is hoped that these methods can help define a standardized method to see how well vendors are doing tracking upstream, or upstreaming code.

There is still a lot of research that we can do:

- filtering useless data (MIPS data is not relevant in ARM context) – coarse grained support implemented
- weighted reporting/finding hotspots (for example in some parts of the kernel you don't want big changes)
- automatically discovering areas ripe for refactoring: if everyone changes it the code might be in need of changing

Availability of data and scripts

The BAT database (available for purchase from Tjaldur Software Governance Solutions) already comes with TLSH hashes for more than 20 million files. The BAT database generation script has TLSH support.

Scripts to explode information from Git repositories will be made open under the Apache 2 license. A test database with data for the Linux kernel will be made available to interested parties.

Scripts to scan and compare using both SHA256 and TLSH will be made available under the Apache 2 license.

It will be incorporated in my scanning services.

Q&A

Contact

- `armijn@tjaldur.nl`
- `http://www.tjaldur.nl/`
- BAT: `http://www.binaryanalysis.org/`