Computing the license of a binary by tracing build outputs

Armijn Hemel
Tjaldur Software Governance Solutions
armijn@tjaldur.nl

May 29, 2014

About Armijn

- owner Tjaldur Software Governance Solutions
- creator of Binary Analysis Tool (BAT)

About this presentation

This presentation builds upon earlier work done by:

Eelco Dolstra, Julius Davies, Sander van der Burg, Daniel German and Armijn Hemel

Results were published as a technical report from Delft University of Technology (TUD SERG 2012-010) and currenty under review for ASE 2014.

Reworking the prototype tools is still "work in progress". Tools will eventually be part of BAT under Apache 2 license.

Today's problem

The question "What license is this binary under?" is not as straightforward as most people think and most people get it *wrong*.

Today is about the next level of analysing sources, binaries and build systems for license compliance.

Example: opkg

opkg is a package manager that is used on embedded Linux distributions.

Question: given you build opkg, what license(s) can the binary you built be distributed under?

Hint: Ohloh says opkg is GPLv2.

GPLv2? GPLv2+? GPLv3+?

(Note: this applies only to an older but widely used version of `opkg`)

`opkg` has a `COPYING` file containing the text of GPLv2.

All source code files in `opkg` are GPLv2+ **except** `libopkg/sha256.c` and `libopkg/sha256.h` which are GPLv3+!

These files are not always included, but they are most of the time. The `configure` script has a switch:

`--enable-sha256 Enable sha256sum check [default=yes]`
Correct answer: it depends and more information about the *composition* of the binary is needed.

Example: MUNGE

```
MUNGE is free software: you can redistribute it and/or
modify it under the terms of the GNU General Public
License as published by the Free Software Foundation,
either version 3 of the License, or (at your option)
any later version.  Additionally for the MUNGE library
(libmunge), you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License
as published by the Free Software Foundation, either
version 3 of the License, or (at your option) any later
version.
```

But what goes into `libmunge`? You can't tell until you actually build it!

Why source code scanning is not good enough

Static analysis (source code level) can tell you a lot, but information is vastly incomplete:

- many different types of build systems and scripts
- output from `configure` has huge influence
- environment variables set by scripts or users
- *external dependencies* might not be obviously declared, but could have a massive impact on the final license

There are many factors that can influence a build which you simply cannot find out by just scanning the source tree.

Why binary scanning is not good enough

Composition is a lot easier to find out with a binary scanner, but:

- compiler throws away a lot of very useful information (like license headers!)
- binary analysis tools like BAT work by making an educated *guess* using fingerprinting to find out what was used. There could be false positives/false negatives.

Basically by just looking at the binary you ignore a lot of very essential information that you already have access to if you have source code!

어떻게?

Source code scanning is not good enough to find out about *composition* but has a lot of information, binary scanning works with incomplete information but can find out about composition.

Best of both worlds: find out about composition and also use information from the source code.

Solution: tracing the build

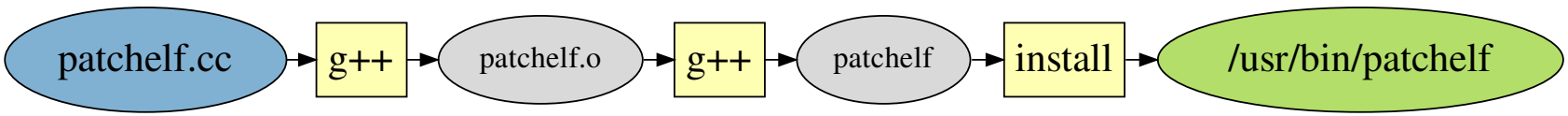
We can track system calls using `strace` and see which files are used and modified by a build process!

- no need to modify existing build tools
- (pretty much) standard package for Linux
- build system agnostic
- no special privileges needed

Trace example output

```
...
  open("patchelf.cc", O_RDONLY)
  open("/usr/include/c++/4.5.1/string",
    O_RDONLY|O_NOCTTY)
  open("elf.h", O_RDONLY)
...
```

Result: build graph



Pruning the result graph

Only tracing which files are used gives a *conservative* estimate of which files are used.

False positives (files that are opened, but not used for building the actual binary) can be pruned by using a bit more intelligence.

Early success: FFmpeg

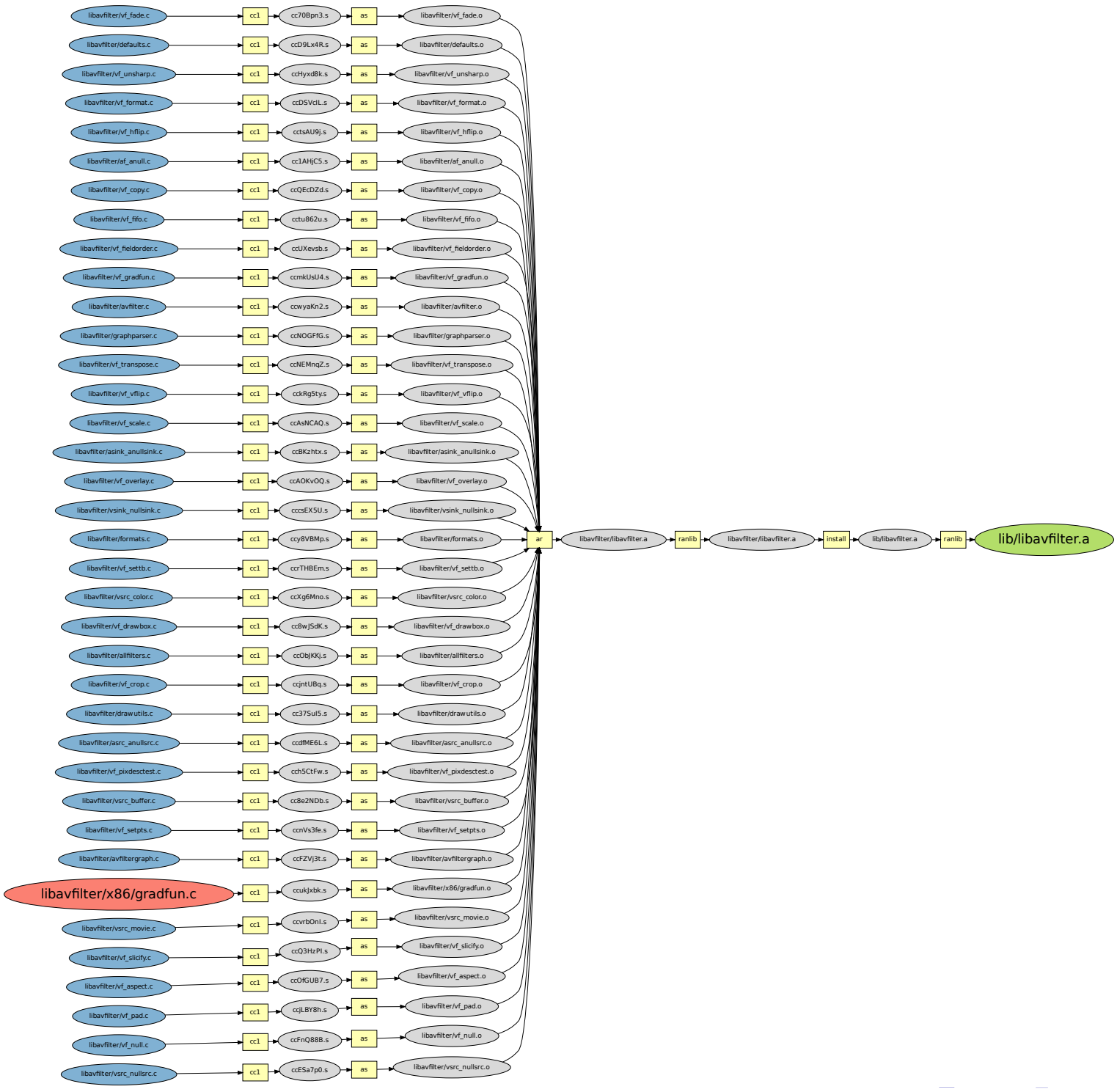
FFmpeg is a mix of GPLv2+ and LGPLv2.1+ licensed code. The configure script has an option to only use the LGPLv2.1+ sources for a build.

With our approach we found that some GPLv2+ code was *always* included in libavfilter.

The offending code was in libavfilter/x86/gradfun.c, licensed under the GPLv2+.

This was not trivial to find out from the FFmpeg build scripts.

FFmpeg fixed it within hours after being informed.


libavfilter/vf_fade.c
cc1
cc70Bpn3.s
as
libavfilter/vf_fade.o
libavfilter/defaults.c
ccD9Lx4R.s
libavfilter/defaults.o
libavfilter/vf_unsharp.c
ccHyxd8k.s
libavfilter/vf_unsharp.o
libavfilter/vf_format.c
ccDSVclL.s
libavfilter/vf_format.o
libavfilter/vf_hflip.c
cctsAU9j.s
libavfilter/vf_hflip.o
libavfilter/af_anull.c
cc1AHjC5.s
libavfilter/af_anull.o
libavfilter/vf_copy.c
ccQEcDZd.s
libavfilter/vf_copy.o
libavfilter/vf_fifo.c
cctu862u.s
libavfilter/vf_fifo.o
libavfilter/vf_fieldorder.c
ccUXevsb.s
libavfilter/vf_fieldorder.o
libavfilter/vf_gradfun.c
ccmkUsU4.s
libavfilter/vf_gradfun.o
libavfilter/avfilter.c
ccwyaKn2.s
libavfilter/avfilter.o
libavfilter/graphparser.c
ccNOGFfG.s
libavfilter/graphparser.o
libavfilter/vf_transpose.c
ccNEMnqZ.s
libavfilter/vf_transpose.o
libavfilter/vf_vflip.c
cckRg5ty.s
libavfilter/vf_vflip.o
libavfilter/vf_scale.c
ccAsNCAQ.s
libavfilter/vf_scale.o
libavfilter/asink_anullsink.c
ccBKzhtx.s
libavfilter/asink_anullsink.o
libavfilter/vf_overlay.c
ccAOKvOQ.s
libavfilter/vf_overlay.o
libavfilter/vsink_nullsink.c
cccsEX5U.s
libavfilter/vsink_nullsink.o
libavfilter/formats.c
ccy8VBMp.s
libavfilter/formats.o
libavfilter/vf_settb.c
ccrTHBEm.s
libavfilter/vf_settb.o
libavfilter/vsrc_color.c
ccXg6Mno.s
libavfilter/vsrc_color.o
libavfilter/vf_drawbox.c
cc8wJSdK.s
libavfilter/vf_drawbox.o
libavfilter/allfilters.c
ccObJKKj.s
libavfilter/allfilters.o
libavfilter/vf_crop.c
ccjntUBq.s
libavfilter/vf_crop.o
libavfilter/drawutils.c
cc37SuI5.s
libavfilter/drawutils.o
libavfilter/asrc_anullsrc.c
ccdfME6L.s
libavfilter/asrc_anullsrc.o
libavfilter/vf_pixdesctest.c
cch5CtFw.s
libavfilter/vf_pixdesctest.o
libavfilter/vsrc_buffer.c
cc8e2NDb.s
libavfilter/vsrc_buffer.o
libavfilter/vf_setpts.c
ccnVs3fe.s
libavfilter/vf_setpts.o
libavfilter/avfiltergraph.c
ccFZVj3t.s
libavfilter/avfiltergraph.o
libavfilter/x86/gradfun.c
ccukJxbk.s
libavfilter/x86/gradfun.o
libavfilter/vsrc_movie.c
ccvrbOnl.s
libavfilter/vsrc_movie.o
libavfilter/vf_slicify.c
ccQ3HzPI.s
libavfilter/vf_slicify.o
libavfilter/vf_aspect.c
ccOfGUB7.s
libavfilter/vf_aspect.o
libavfilter/vf_pad.c
ccjLBY8h.s
libavfilter/vf_pad.o
libavfilter/vf_null.c
ccFnQ88B.s
libavfilter/vf_null.o
libavfilter/vsrc_nullsrc.c
ccESa7p0.s
libavfilter/vsrc_nullsrc.o
ar
libavfilter/libavfilter.a
ranlib
libavfilter/libavfilter.a
install
lib/libavfilter.a
ranlib
lib/libavfilter.a

Drawback: information overload and performance hit

`strace` can be *very* verbose and log files (the way I generate them to get enough information) easily can become a few GiB of data.

Tracing also has a significant performance hit so it is not something you want to do every single build.

Drawback: not all build systems are suitable

Not all build systems are suitable to be traced, since they open all files in a directory (Java).

In practice I have limited it to C/C++ programs on Linux/*BSD but sometimes build systems can interfere. Example: BusyBox build scripts open all files in the directory.

Worst case: all files in the directory tree are seen as inputs (which is the same as the best that source code scanners do now).

Propagating information through the graph

By adding information to the graph interesting patterns can become visible:

- security: does this binary include a file with a known security defect?
- license information: which licenses are combined in the binary?

A license calculus

By adding licensing information interaction between licenses becomes more visible.

Using simple "license math" and rewrite rules we could possibly compute a license of a binary!

$BSD2 \rightarrow permissive$
$MIT \rightarrow permissive$
$GPLv2 + GPLv2orlater = GPLv2$
$GPLv2orlater + GPLv3 = GPLv3$
$GPLv2 + GPLv3 = error$
$permissive + GPLv3orlater = GPLv3orlater$
...

Some interactions might not be possible to compute or detect automatically, but I think we can get very far, which is better than what we have today.

Problem: quality of licensing information

The method described only works if license information in files is of good quality. Often licensing information is complete crap.

I use FOSSology and Ninka, which only agree in about 65% of files I have scanned with these two scanners because license scanning is *hard*.

Current status

Rough prototype exists, database with licensing information exists, but I need time to tie everything together.

Q&A and discussion