Delft University of Technology
Software Engineering Research Group
Technical Report Series

Discovering Software License Constraints: Identifying a Binary's Sources by Tracing Build Processes

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

Report TUD-SERG-2012-010

TUD-SERG-2012-010

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

```
@techreport{BDDGH12,
  title = {Discovering Software License Constraints: Identifying a Binary's Sources by Tracing Bui
  author = {Sander van der Burg and Julius Davies and Eelco Dolstra and Daniel M. German and Armijn
  number = {TUD-SERG-2012-010},
  institution = {Software Engineering Research Group, Delft University of Technology},
  address = {Delft, The Netherlands},
  year = {2012},
  month = apr,
}
```

Copyright © 2012, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

Discovering Software License Constraints: Identifying a Binary's Sources by Tracing Build Processes

Sander van der Burg*, Julius Davies†, Eelco Dolstra*, Daniel M. German† and Armijn Hemel‡
* Delft University of Technology, Netherlands, {s.vanderburg, e.dolstra}@tudelft.nl
† Department of Computer Science, University of Victoria, Canada, {juliusd, dmg}@uvic.ca
‡ The gpl-violations.org Project & Tjaldur Software Governance Solutions, Netherlands, info@tjaldur.nl

***Abstract*—**
With the current proliferation of open source software components, intellectual property in general, and copyright law in particular, has become a critical *non-functional* requirement for software systems. A key problem in license compliance engineering is that the legal constraints on a product depend on the licenses of all sources and other artifacts used to build it. The huge size of typical dependency graphs makes it infeasible to determine these constraints manually, while mistakes can expose software distributors to litigation. In this paper we show a generic method to reverse-engineer this information from the build processes of software products by tracing system calls (e.g., open) to determine the composition graph of sources and binaries involved in build processes. Results from an exploratory case study of seven open source systems, which allowed us to discover a licensing problem in a widely used open source package, suggest our method is highly effective.

I. Introduction

Today's software developers often make use of reusable software components, libraries, and frameworks. This reuse can be observed in diverse software systems, from the tiny to the massive (e.g., *LibreOffice*), as well as in industrial, government, and academic systems. Reuse can greatly improve development efficiency, and arguably improves quality of delivered systems.

With reuse of software the question under what *license* a binary should be released becomes very important. If code is written entirely by a developer, then the developer is free to choose whatever license he wants to. However, when reusing existing third party components in a binary, the final license or licenses (if any [1], [2]) that govern this binary is determined by the licenses under which these components were released, plus the way these components are combined. Failing to comply with license terms could end in costly lawsuits from copyright holders.

To correctly determine the license of an executable, we should know three things:

- How are source files combined into a final executable (e.g. static linking, dynamic linking)?
- What licenses govern the (re)use of source files that were used to build the binary?
- How can we derive the license of the resulting binary from the build graph containing the source files?

Our goal in this paper is to provide an answer for the first question. Identifying what goes into an executable is not straightforward. An executable is created by using other executables (such as compiler or code generators), and by transforming source code and other binary components (such as libraries) into the executable.

The process of creating a binary is usually driven by a build tool, such as make, cmake, ant, etc. and it is documented in a build recipe, such as a *Makefile*. To complicate identifying the binary provenance of an executable, the build process might also use a set of configuration parameters ("configure flags") to enable or disable specific features that will affect what is ultimately included in the executable. Thus, *the way in which a binary is built may change the licensing constraints on that binary*.

The necessary information cannot, in general, be derived by analysing build recipes such as Makefiles, because they tend to have incomplete dependency information. Therefore, we *reverse-engineer the build process of the binary* by deriving its build dependency graph by tracing *system calls* made by processes during the building of a binary. These system calls reveal which files were read and written by each process, and thus allows a dependency graph to be constructed.

The structure of this paper is as follows. In Section II, we motivate the problem of determining licenses of binaries and give a real-world example of a library whose license it determined by its build process. In Section III, we argue that static analysis of build processes is infeasible, and that therefore we need a dynamic approach based on tracing of actions performed during a build. We explain our system call tracing method in Section IV. To determine the feasibility of this method we applied it to a number of open source packages (Section V). This analysis revealed a licensing problem in FFmpeg, a widely used open source component, which was subsequently confirmed by its developers. We discuss threats to the validity of our approach in Section VI. In Section VII we cover related work and Section VIII concludes this paper.

II. A Motivating Example

To motivate the need of knowing what goes into a binary we use as an example FFmpeg, which is a library to record, convert and stream audio and video, and has been used in both open source and proprietary video players. FFmpeg has been frequently involved in legal disputes[1].

Open Source components are available under various licenses, ranging from simple permissive software licenses (BSD and MIT) allowing developers to include code in their products without publishing any source code, to more strict licenses imposing complex requirements on derived works (GPLv2, GPLv3, MPL). Unfortunately it is not always easy or even possible to combine components that use different licenses. This problem is known as license mismatch [2]. Extra care needs to be taken to make sure that license conditions are properly satisfied when assembling a software product.

Incorrect use of open source software licenses opens developers up to litigation by copyright holders. A failure to comply with the license terms could void the license and make distribution of the code a copyright violation. In Germany several cases about non-compliance with the license of the Linux kernel (D-Link[2], Skype[3], Fortinet[4]) were taken successfully to court [3]. In the US there have been similar lawsuits involving BusyBox [4].

The license of our motivating example differs depending on the exact files the consuming applications depend on. As stated in FFmpeg's documentation:

> Most files in FFmpeg are under the GNU Lesser General Public License version 2.1 or later (LGPL v2.1+). [...] Some other files have MIT/X11/BSD-style licenses. In combination the LGPL v2.1+ applies to FFmpeg.
>
> Some optional parts of FFmpeg are licensed under the GNU General Public License version 2 or later (GPL v2+). See the file COPYING.GPLv2 for details. None of these parts are used by default, you have to explicitly pass --enable-gpl to configure to activate them. In this case, FFmpeg's license changes to GPL v2+.

This licensing scheme creates a potential risk for developers reusing and redistributing FFmpeg, who want to be absolutely sure there is no GPLv2+ code in the FFmpeg binary they link to. Evidence for this can be found in the the FFmpeg forums[5].

The reason why this is important is that if FFmpeg is licensed under the LGPL, it allows it to be (dynamically) linked against code under any other license (including proprietary code). But if it is licensed under the GPLv2+, it would require any code that links to it to be licensed under a GPLv2- or GPLv3-compatible license.

```
patchelf: patchelf.o
    g++ patchelf.o -o patchelf

patchelf.o: patchelf.cc
    g++ -c patchelf.cc -o patchelf.o \
      -DENABLE_FOO

install: patchelf
    install patchelf /usr/bin/
```

Figure 1. A simple Makefile

FFmpeg's source code is divided into 1427 files under the LGPLv2.1+, 89 GPLv2+, 18 MIT/X11, 2 public domain, and 7 without a license. To do proper legal diligence, an important question to answer is: if the library is built to be licensed under the LGPLv2+ (i.e. without –enable-gpl), is there any file under the GPLv2+ used during its creation (and potentially embedded into the binary)? If there is, that could potentially turn the library GPLv2+.

Given the size of components such as FFmpeg, manual analysis of the entire code is infeasible. Therefore, we need an automated method to assist in this process.

III. General Approach: Tracing Build Processes

As we saw in the previous section, it is important to be able to identify *what* source files were involved in building a binary, and *how* they were composed to form the binary.

A. Reverse-Engineering the Dependency Graph

Our goal therefore is to reverse-engineer the *build dependency graph* of a binary, which should describe how the source files were combined to form the final binary. Formally, the dependency graph is defined as $G = (V, E)$, where $V = V_t \cup V_f$, the union of *task nodes* V_t and *file nodes* V_f. A task node $v_t \in V_t$ denotes a step in the build process. (The *granularity* of a task depends on the level of detail of the reverse-engineering method; they could be Make actions, Unix processes, Ant task invocations, and so on.) It is defined as the ordered tuple $\langle id, \ldots \rangle$, where $id \in Ids$ is a symbol uniquely identifying the task, plus auxiliary information such as the name of the program(s) involved in the task or the name of the Makefile rule. A file node $v_f \in V_f$ is a tuple $\langle id, path \rangle$, where $id \in Ids$ is a symbol uniquely identifying the file node or the special value ϵ, and *path* is the file's path[6].

So how do we reverse-engineer the dependency graph for an existing product? It might seem that this is simply a matter of inspecting the build code of the product, that is, the

[1] See http://en.wikipedia.org/wiki/List_of_software_license_violations
[2] 2-6 O 224/06 (LG Frankfurt/Main)
[3] 7 O 5245/07 (LG München I)
[4] 21 O 7240/05 (LG München I)
[5] http://ffmpeg.arrozcru.org/forum/viewtopic.php?f=8&t=821

[6] We explain in Section IV-C why *path* does not uniquely identify file nodes and *id* is necessary to disambiguate them.

```
#ifdef ENABLE_FOO
#include <iostream>
#endif

int main() {
#ifdef ENABLE_FOO
    std::cout << "Hello, world!" << std::endl;
#endif
    return 0;
}
```

Figure 3. Example source code demonstrating an optional dependency

scripts and specification files responsible for automatically constructing the installable binary artifacts from the source code. For instance, for a product built using Make [5], we could just study the Makefile. As a simple running example, consider *PatchELF*, a small utility for manipulating Unix executables[7]. It consists of a single C++ source file, patchelf.cc. Building this package installs a single binary in /usr/bin/patchelf. Suppose that we want to find out what source files contributed to the build of this binary. To do this, we might look at its Makefile, shown (simplified) in Figure 1. If we only use the information included in the Makefile, following the rules in reverse order from the binary patchelf, we conclude that patchelf has a single source file: patchelf.cc. Figure 2 shows the dependency graph resulting from this analysis of the Makefile.

B. Why We Can't Use Static Analysis

In fact, the graph in Figure 2 is highly incomplete. Both manual inspection and automatic static analysis of the build code are unlikely to produce correct, complete dependency graphs. There are a number of reasons for this:

1) Makefiles are usually not complete: Makefiles do not list every dependency required. For instance, the file patchelf.cc uses several header files, such as system header files from the C++ compiler and C library, but also a header file elf.h included in the PatchELF source distribution (which was actually copied from the GNU C Library). Inspection of the build files does not reveal these dependencies; it requires a (language-dependent) analysis of the sources as well. Similarly, the linker implicitly links against various standard libraries. Also note that Makefile rules do not list all of their outputs: the install rule doesn't declare that it creates a file in /usr/bin. Therefore a static analysis of the Makefile will miss many inputs and outputs.

2) Build files are numerous, large, and hard to understand: Large projects can have thousands of build-related files, which exhibit a significant amount of code churn [6], [7]. This makes *manual* inspection infeasible; only automated methods are practical.

3) Dependencies might be the result of many other build processes: If the binary is created using files that are not part of the project itself (such as external dependencies like libraries) this process has to repeated again (recursively) to be able to determine which files were used to create such files.

4) Code generators might make it difficult to follow the build process: For example, a build might use a parser generator (such as yacc). Therefore one might need to do static analysis of the source for files of the parser (*.y files) and know the type of dependencies required by the generated code. This is complicated if the source code has ad-hoc code-generators. In such case the output of them would have to be analysed to determine what is needed to build them.

5) Configuration options and compilation flags: Most build-configuration tools (Autoconf/Automake, cmake, qmake, etcetera), allow customization of the build process by analyzing the local environment where the system is being built: what libraries are installed, what is their location, what features should be enabled and disabled. In some cases, this build-configuration process might determine which of two alternative libraries will be used (e.g. one is already installed in the target system). In essence, this process would require the execution of the build configuration process to determine the actual instance of the Makefile to be used. Such a Makefile might further contain compilation flags that would require the dynamic analysis of the creation of the final binary. This kind of conditionality is widely used in open source systems, where the person building a product might enable or disable various options, usually depending on which libraries are already installed in the system where the product is expected to run.

For example, assume that the source code of patchelf.cc is as shown in Figure 3, and we attempt static analysis to determine any other dependencies that it might have. However, the compiler flag -DENABLE_FOO indicates that the compiler should enable the preprocessor define ENABLE_FOO, and therefore, any static analysis would require that this be taken into consideration. This directive might include or exclude dependencies; in the example, enabling it causes a dependency on the header file iostream.

6) There are many build systems: Many different build systems are in use, such as Ant, Maven, countless variants of GNU Make, language-specific scriptable formalisms such as Python's Setuptools, and code generation layers above other build systems such as the GNU Autotools and Perl's MakeMaker. In practice, a static analysis tool would be limited to a few of them at most; and build systems that use general-purpose languages (such as Autoconf scripts or Python Setuptools) for specifying build processes are practically impossible to analyse.

C. Dynamic Analysis through Tracing

Given all these issues, the only effective way to determine the source provenance of a binary that we can hope for is to perform *dynamic analysis* of the build process. That is,

[7] http://nixos.org/patchelf.html

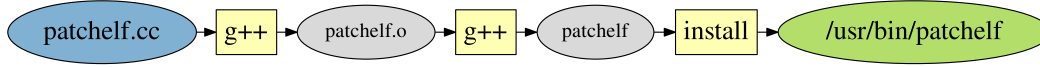


Figure 2. Hypothetical dependency graph derived from an analysis of the Makefile in Figure 1. Make actions are depicted as yellow rectangles and files as ovals; input files are blue, and final binaries are in green; and temporarily created files are in grey.

given concrete executions of the build systems that produce the binary under scrutiny and its dependencies, we wish to *instrument* those executions to discover the exact set of files that were used during the build, and the manner in which they were used (e.g., used by a compiler, a linker, or during the installation or configuration steps). The result of such an instrumented build is a *trace* of files used to create the binary. This trace can be annotated with licensing information of each of the nodes, and further analysed to do license auditing of the binary.

Previous approaches to tracing have instrumented specific build tools such as GCC and the Unix linker [8] or used debug instrumentation already present in GNU Make [9]. These methods are limited to specific tools or require correct dependency information in Makefiles. In the next section, we present a generic method intended to work for *all* tools used during a build.

IV. Method: Tracing the Build Process through System Calls

A tracing of the build process seeks to discover the *inputs* and the *outputs* of each step in a build process in order to produce a dependency graph. That is, the instrumentation of the build process must log which files are *read* and *written* by each build step, respectively. Therefore, if we can determine all file system accesses made by a build process, we will be able to produce the dependency graph of the binary being built.

A. Overview

In this section, we describe one implementation of this idea: namely, by logging all *system calls* made during the build. System calls are calls made by user processes to the operating system kernel; these include operations such as creating or opening a file. Thus, logging these calls allows us to discover the inputs and outputs of build steps at the granularity of operating system processes or threads. Thus, in essence, we determine the data flow graph between the programs invoked during the build. A nice aspect of this approach is that the necessary instrumentation is already present in most conventional operating systems, such as Linux, FreeBSD, Mac OS X and Windows.

A major assumption this approach makes is that if a file is opened, it is used to create the binary (either directly–it becomes part of the binary, such as source code file) or indirectly (it is used to create the binary–such as configuration file). Not all files opened will have an impact in the licensing of the binary that is created, but we presume that because they *might* have an impact, it is necessary to evaluate how each file is used, and if the specific use has any licensing implications for the final binary.

Consider, for instance, tracing the system calls made during the execution of the command make install with the Makefile shown in Figure 1. On Linux, this can be done using the command strace[8], which executes a program and logs all its system calls. Thus, strace -f make install prints on standard error a log of all system calls issued by make and its child processes. (The flag -f causes strace to follow fork() system calls, i.e., to trace child processes.) Make first spawns a process running gcc to build patchelf.o. Since this process creates patchelf.o, it will at some point issue the system call

```
open("patchelf.o",
  O_RDWR|O_CREAT|O_TRUNC, 0666)
```

Likewise, because it must read the source file and all included headers, the trace will show system calls such as

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

Continuing the trace of Make, we see that the linker opens patchelf.o for reading (as well as libraries and object files such as /usr/lib/libc.so, /usr/lib/libc_nonshared.a and /usr/lib/crtn.o) and creates ./patchelf, while finally the install target opens ./patchelf for reading and creates /usr/bin/patchelf. Thus we can derive a dependency graph for the final, installed binary /usr/bin/patchelf. The nodes in this graph are files and the processes that read, create or modify files. In the graph for PatchELF, there will therefore be a path from the nodes patchelf.cc and elf.h to /usr/bin/patchelf, revealing the dependencies of the latter.

Performing a trace on just one package reveals the direct dependencies of that package, e.g., files such as patchelf.cc and /usr/lib/libc_nonshared.a. Since the latter is a binary file that gets linked into the patchelf program, for the purposes of license analysis, we would like to know what source files were involved in building libc_nonshared.a. For example, if this static library contains a file distributed under a non-permissive copyleft license, then this can have legal consequences for the patchelf binary.

Therefore, we can perform the same trace analysis on external dependencies of PatchELF, such as Glibc (the package that contains lib_nonshared.a). The result will be a

[8] http://strace.sourceforge.net/

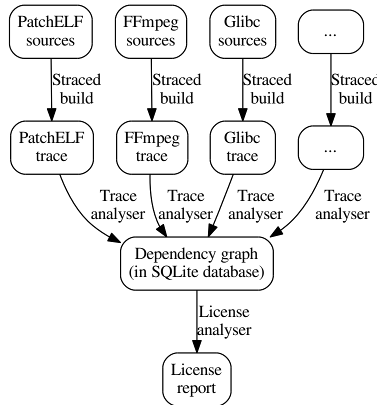


Figure 4. Overview of the system call tracing approach. Nodes denote artifacts, and edges denote the tools that create them.

large dependency graph showing the relationships *between* packages. For instance, in this graph, some source files of Glibc have paths to /usr/lib/libc_nonshared.a, and from there, to /usr/bin/patchelf.

Figure 4 summarises our approach. For each project we're interested in and its dependencies, we perform a complete build of the package. This can typically be done fully automatically by using package management systems such as RPM [10]; all we need to do is to trace the package build with strace. A *trace analyser* is then applied to each trace to discover the files read and written by each process. This information is stored in a database, which, in essence, contains a dependency graph showing how each file in the system was created. This dependency graph can then be used by license analysis tools (which we do not develop in this paper). For instance, for a given binary, such a tool could use the dependency graph to identify the sources that contributed to the binary, determine each source file's license using a license extraction tool such as Ninka [11], and compute the license on the binary.

We now discuss in detail how we produce a system call trace, how we analyse the trace to produce the dependency graph, and how we can use the dependency graph to answer queries about binaries.

B. Producing the Trace

Producing a system call trace is straight-forward: on Linux, we essentially just run strace -f /path/to/builder, where builder is the script or program that builds the package.

For performance reasons, we only log a subset of system calls. These are the following:

- All system calls that take path arguments, e.g., open(), rename() and execve().
- System calls related to process management. As we shall see below, it's necessary to know the relationships between processes (i.e. what the parent process of a child is), so we need to trace forks. On Unix systems, forking is the fundamental operation for creating a new process: it clones the current process. The child process can then run execve() to load a different program within its address space; the underlying system calls responsible for this are clone() and vfork().

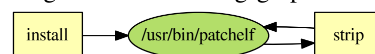
C. Producing the Build Graph

The trace analyser reads the output of strace to reverse-engineer the build dependency graph. This graph is defined as in Section III-A, where each task represents a process executed during the build. Task nodes are extended to store information about processes: they are now a tuple $\langle id, program, args \rangle$, where $program$ is the path of the last program executed by the task (i.e., the last binary loaded into the process's address space by execve(), or the program inherited from the parent) and $args$ is the sequence of command-line arguments.

The analyser constructs the graph G by reading the trace, line by line. When a new process is created, a corresponding task node is added to V_t. When a process opens a file for reading, we add a file node v_f to V_f (if it didn't already exist), and an edge from v_f to the task node. Likewise, when a process opens a file for writing, we add a file node v_f to V_f, and an edge from the task node to v_f.

There are some tricky aspects to system call traces that need to be taken into account by the analyser:

- System calls can use relative paths. These need to be absolutised with respect to the current working directory (*cwd*) of the calling process. As this requires us to know the *cwd* of each process, we need to process chdir() calls. Since the *cwd* is inherited by child processes, the analyser must also keep track of parent/child relations, and propagate the *cwd* of the parent to the child when it encounters a fork operation.
- Process IDs (*PIDs*) can wrap around; in fact, this is quite likely in large packages because PIDs are by default limited to 32,768 in Linux. Therefore the analyser needs to distinguish between different processes with the same PID. For this reason, task IDs are not equal to PIDs; rather, a unique task ID is generated in the graph every time a process creation is encountered in the trace.
- A special problem is the representation of files that are written *multiple times*, i.e., are created or opened for writing by several different processes. This is quite common. For instance, it is usual in the make install step to remove debug information from binaries by running the strip command. This means that after linking, the binary patchelf is first copied from the temporary build directory to /usr/bin/patchelf by the command install, and then read and recreated by strip. Represented naively, this gives the following graph:



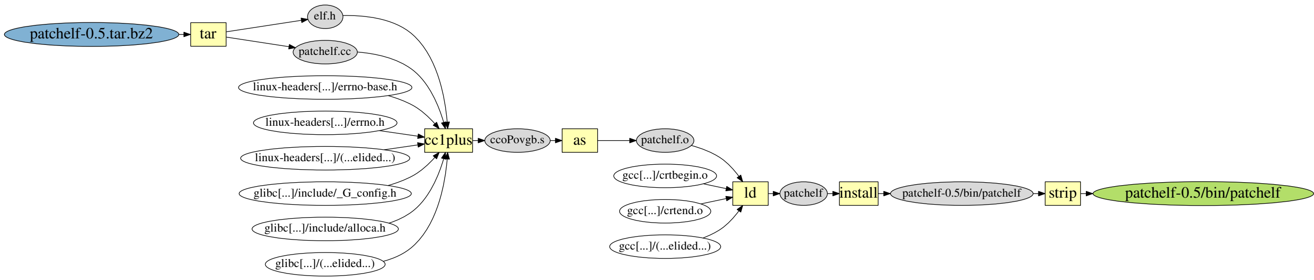
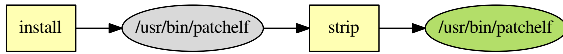


Figure 5. PatchELF dependency graph. External dependencies are depicted in white.

Such cycles in the graph make the data flow hard to follow, especially if there are multiple processes that update a file: the order is impossible to discern from the graph. Therefore, each modification of a file creates a *new* node in the graph to represent the file, e.g.,



These nodes are disambiguated in the graph by tagging them with the task ID that created them.
Note that this approach allows "dead" writes to a file to be removed from the graph, analogous to the discovery of dead variables in data flow analysis. For instance, in the build script echo foo $>$ out; echo bar $>$ out, the second process recreates out without reading the file created by the first process. Thus, there is no edge in the graph exiting the first out node, and its task is not equal to the creator of the out node, so it can be safely removed from the graph. This is an instance of the notion developed in [12] that many concepts in memory management in the programming languages domain can be carried over to the domain of the storage of software on disk.

- We are currently not interested in the dynamic library dependencies of tools used during the build, because these generally don't have license implications. For instance, every program executed during the build opens the GNU C library, libc.so.6 (almost every program running under Linux uses it), and therefore we are not interested in these. However, we *do* care if the *linker* is the one opening a library when it's creating a binary (during a separate, later call to open().) So to simplify the graph, we omit these dependencies. To detect calls to open() made due to dynamic linking, we use the following method: we observed that after the dynamic linker has opened all shared libraries, it issues an arch_prctl() system call. Thus, we ignore any opens done between a call to execve() and arch_prctl().

Let us return to the PatchELF example described in Section III. Building this package has several automated steps beyond running make install. As with most Unix packages, building the PatchELF package starts with unpacking the source code distribution (a file patchelf-0.5.tar.bz2), running its configure script, running make and finally running make install.

Figure 5 shows the dependency graph resulting from analysing the trace of a build of this package. It is instructive to compare this dependency graph with the "naive" one in Figure 2. The system call trace reveals that compiling patchelf.cc causes a dependency not only on elf.h, but also on numerous header files from Glibc and the Linux kernel. Likewise, the linker brings in several object files and libraries from Glibc and GCC. (Many header and library dependencies that are detected by system call tracing have been omitted from the figure for space reasons.) The analysis also reveals that the compilation of patchelf.cc is actually done by two processes, which communicate through an intermediate temporary assembler file, ccoPovgb.s; and that the installed executable is rewritten by strip.

In our implementation, the analyser stores the resulting graph in a SQLite database. As sketched in Figure 4, other tools can use this database to answer various queries about a binary. Multiple traces can be dumped into the same database, creating a (possibly disconnected) graph consisting of the output files of each package and the processes and files that created them. This allows inter-package dependencies to be analysed.

D. Using the Build Graph

Once we have the full dependency graph for a binary, we can perform various queries. The most obvious query is to ask for the list of source files that contributed to a binary $x \in V_f$. Answering this question is not entirely trivial. It might seem that the sources are the files in the graph that contributed to the binary and were not generated themselves, i.e., they are source vertices in V_f from which there is a path to x:

$$\{v_f \in V_f \mid \deg^-(v_f) = 0 \land \exists \text{ a path in } G \text{ from } v_f \text{ to } x\}$$

But as Figure 5 shows, this doesn't yield the files in which we're interested: the "source files" returned are source distributions such as patchelf-0.5.tar.bz2, from which a tar

task node produces files such as patchelf.cc. (Additional "sources" in this graph are the header files, object files and libraries in Glibc, GCC and the kernel. However, if these packages are traced and added to the dependency graph, then these files are no longer sources; instead, we get more source distributions such as glibc-2.12.2.tar.bz2.) While this answer is strictly speaking correct – patchelf-0.5.tar.bz2 *is* the sole source of the package – it is not very precise: we want to identify *specific* files such as patchelf.cc.

To get the desired results, we simply transform the full dependency graph into one from which task nodes such as tar and its dependencies have been removed. For PatchELF, this yields a set

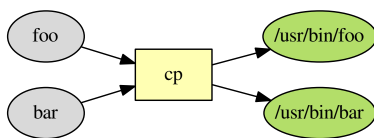
$$\{\mathsf{patchelf.cc}, \mathsf{elf.h}, \mathsf{errno\text{-}base.h}, \mathsf{errno.h}, \ldots\}$$

On this set, we can apply Ninka to determine the licenses governing the source files.

More interesting queries can take into account *how* files are composed. For instance, given a binary, we can ask for all the source files that are *statically linked* into the binary. Here the graph traversal doesn't follow dynamic library inputs to ld nodes. It does follow other task nodes, so for instance bison nodes are traversed into their .y sources.

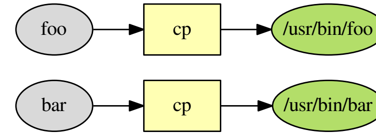
E. Coarse-grained Processes

Because system call tracing identifies inputs and outputs at the granularity of Unix processes, the main assumption underlying this approach to tracing is that every process is part of at most one conceptual build step. However, this is not always the case. Consider the command cp foo bar /usr/bin/. This command installs *two* files in /usr/bin. Conceptually, these are two build steps. But since they are executed in a single process, we get the following dependency graph:



This is undesirable, because foo and its sources now appear as dependencies of /usr/bin/bar, even if the two programs are otherwise unrelated. We call these processes *coarse-grained*.

In cases such as cp (and similar commands such as install), we can fix this problem on an *ad hoc* basis by rewriting the graph as follows. We scan the dependency graph for cp task nodes (and other tasks of which we know they may be coarse-grained) that have more than one output, and each output file has the same base name as an input file. We then replace this task node with a *set* of task nodes, one for each corresponding pair of inputs and outputs, e.g.



Note that both cp tasks in this graph correspond to the same process in the trace. Since coarse-grained processes are a threat to the validity of our approach, we investigate how often they occur in the next section.

V. Evaluation

The goal of this evaluation is to serve as a demonstration of the correctness of our method, and second, its usefulness.

A. Correctness

We performed an experiment to determine how often our method produces false negatives (**RQ1**) and false positives (**RQ2**) on a number of open source packages, including FFmpeg, patchelf and a number of other open source packages, including Apache Xalan, a Java based XSLT library using the Apache Ant build system. A *false negative* occurs when a file is a dependency of a binary (i.e., necessary to build the binary) but does not appear in the graph as such. Conversely, a *false positive* occurs when a file appears as a dependency of a binary in the graph, but is not actually used to build the binary.

To measure this in an objective manner, we used the following method. First, for a given binary, we use our tracer to determine the dependency graph. This gives us a list of source files from the package's distribution that were supposedly used in the build, and a list of source files that were not. Then:

- To determine false negatives (**RQ1**), for each presumptively unused file, we rebuild the package, where we *delete* the file after the configure step. If the package then fails to build or gives different output, the file apparently *is* used, so it is flagged as a false negative. If the package builds the same, then it is a true negative.
- To determine false positives (**RQ2**), for each assumed used file, we likewise rebuild the package after deleting the file. If the package still builds correctly, then the file apparently is not used by the build, so it is a false positive. Otherwise it is a true positive.

We used some optimisations in our experiment. For instance, for **RQ1**, we can delete all unused files at the same time. If the package still builds correctly, then there are no false negatives.

Table I shows the results of our experiment to determine false negatives and positives. For each package in our evaluation, it shows the number of apparently unused and used files (i.e. non-dependencies and dependencies), the number of failed builds (i.e., false negatives), the number of successful builds (i.e., false negatives), and the recall and precision.

Package	Files "unused"	Files "used"	False negatives	False positives	Recall	Precision
patchelf	19	3	0	1	1.00	0.67
aterm	60	57	1	0	0.98	1.00
ffmpeg	986	1253	0	7	1.00	0.99
opkg	9	97	1	3	0.99	0.97
openssl	1180	847	0	5	1.00	0.99
bash	806	280	0	34	1.00	0.88
xalan	379	955	0	4	1.00	1.00

Table I
RQ1 & RQ2: FALSE NEGATIVES AND POSITIVES

As the table shows, the number of false negatives reported by this method is very low, and in fact, none of them are really false negatives: the files *are* used during the build, but not to build the binary. In the aterm package, we had to include a source file called terms-dict, used by a test case which was not compiled in the regular build process. Due to strictness of the Makefile an error was raised. The package opkg requires a file called libopkg_test.c to build a test case in the test suite, which was not present in the resulting executable.

The number of reported false positives is also low. Most of these false positives occured because a particular file was not actually used in the build (e.g. a documentation file) and the build system did not explicitly check for its presence (e.g. this could happen by wildcards in a build configuration file). For example, for bash the number of false positives was higher than the other packages, because it contains a number of documentation files and gettext translation files, which did not cause an error in the build process if they were removed.

One interesting mistaken false positive report came from patchelf. A false positive was flagged because the package uses the elf.h header file, which is also present in the GNU C Library, so the package compiles successfully even if it is deleted from patchelf.

B. Usefulness

In [13] we analysed the Fedora 12 distribution for the to understand how license auditing is performed. One of the challenges we faced was determining which files are used to create a binary. This is particularly important in systems that include files under different licenses that do not allow their composition (for example the same system contains a file under the BSD-4 clauses license and a file under the GPLv2, which cannot be legally combined to form a binary).

From that study we have selected three projects with files that appear incompatible with the rest of the system:

- FFmpeg is a library to record, convert and stream audio and video, which we have introduced in Section II. This package may be covered under different licenses depending on which build time options are used.
- cups is a printing system developed by Apple Inc. for Unix. The library is licensed under a permissive license (the CUPS License). Version 1.4.6 contained 3 files (backend/*scsi*) under a old BSD-4 clauses license which is not compatible with the CUPS License. For this system (and the next two) we would like to know if these files are being used to build the binaries of the product in a way that is in conflict with their license.
- bash is a Unix command shell written for the GNU project. Recent versions have been licensed under the GPLv3+, but it contains a file under the old BSD-4 clauses (examples/loadables/getconf.c). We analyzed version 4.1.

To determine the license of the files in the projects we used Ninka. We traced the build of each system, and identified the files used. We then compared the licenses of these files, to try to identify any licensing inconsistency. [9]

1) FFmpeg: By default, FFmpeg is expected to create libraries and binary programs under the LGPLv2+. However, we discovered that two files under the GPLv2+ were being used during its compilation: libpostproc/postprocess.h and x86/gradfun.c. The former is an include file and contains only constants. The latter, however, is source code, and it is used to create the library libavfilter. In other words, the library libavfilter is licensed under the LGPLv2.1+, but required libavfilter/x86/gradfun.c, licensed under the GPLv2+. We manually inspected the Makefiles, which were very complex. It was not trivial to determine if, how, and why, it was included.

We therefore looked at the library, where we found the symbol corresponding to the function located inside x86/gradfun.c, which gave us confidence that our method was correct. With this information we emailed the developers of FFmpeg. They responded within hours. The code in question was an optimized version of another function. A developer quickly prepared a patch to disable its use, and the original authors of x86/gradfun.c were contacted to arrange relicensing of the file under the LGPLv2.1+[10]. Three days later, this file's license was changed to GPLv2+[11]

2) cups: The dependency graph of cups shows that the three files in question: backend/(scsi|scsi-iris|scsi-linux).c were used to create a binary called /backend/scsi. Coincidentally these files were recently removed in response to Bug #3509.

3) bash: The file in question, examples/loadables/getconf.c is not used while building the library. This is consistent with the directory where it is located (examples).

Although the goal of this paper is not to provide a system capable of correctly investigating license issues, we have shown that the information provided by the build graph

[9]Full-size renderings of graphs can be found at http://www.st.ewi.tudelft.nl/~dolstra/icse-2012/, with problematic files showing as red nodes.
[10]http://web.archiveorange.com/archive/v/4q4BhMcVOjgXyfN3wyhB
[11]http://ffmpeg.org/pipermail/ffmpeg-cvslog/2011-July/039224.html

tracer is sufficient to highlight potential licensing inconsistencies. These inconsistencies could be solved automatically by developing a complementary license calculus built on top of the build tracer.

VI. Threats to Validity

In this section, we discuss threats to the validity of our work. It is worth noting that apart from bugs in the kernel or in our analysis tool, system call tracing is guaranteed to determine all files accessed during a build, and nothing more. In that sense there is no risk of false positives or negatives. The real question, however, is whether it can reconstruct a meaningful dependency graph: a simple list of files read and written during a build is not necessarily useful. For instance, a source file might be read during a build but not actually used in the creation of (some of) the outputs.

Thus, with respect to external validity, as noted in Section IV-E, the main threat is that coarse-grained processes (i.e. those that perform many or all build steps in a single process) yield dependency graphs that do not accurately portray the "logical" dependency graph. Thus, files may be falsely reported as dependencies. This can in turn lead to false positives in tools that use the graph; e.g., in a license analyser, it can cause false warnings of potential licensing problems. Note however that this is a conservative approach: it may over-estimate the set of source files used to produce a binary, but that is safer than accidentally omitting source files.

Therefore, our approach may not work well with coarse-grained build systems such as Ant, which tends to perform many build steps internally (although Ant can also be instructed to fork the Java compiler, which circumvents this problem). In many cases, users can still get usable dependency graphs by instructing the build tool to build a single target at a time; we then get traces that only contain the files accessed for each particular target. However, in Unix based environments (where most open source components are used) most build systems are process based, which can be analysed without much trouble.

A further threat to external validity would be processes that access files unnecessarily. (One might imagine a C compiler that does an open() on all header files in its search path.) However, we have not encountered such cases. Finally, system call tracing is not portable and differs per system. However, most widely used operating systems support it in some variant.

With respect to the internal validity in our evaluation of **RQ1** and **RQ2**, where we used file deletion as an objective measure to verify if a file is or is not a dependency, a limitation is that complex build processes can fail even if files that are unnecessary to a specific target are missing. For instance, in a large project, the build system might build a library in one directory that is subsequently *not* used anywhere else.

Construct validity is threatened by limitations in our prototype that can cause file accesses to be missed. Most importantly, the trace analyser does not trace file descriptors and inter-process communication (e.g. through pipes) yet. For instance, it fails to see that the task patch in the command cat foo.patch | patch bar.c has a dependency on foo.patch; it only sees that patch reads and recreates bar.c. Our prototype also lacks support for some uncommonly-used file-related Linux system calls, such as the *at() family of calls. Implementing the necessary support seems straightforward.

The conclusion validity of our work is mainly limited by the limited size of our evaluation. Although the size is limited, many open source packages use common build tooling and very similar deployment processes, which should not be too difficult to analyse with our tooling. The main challenge lies in supporting uncommon open source packages, which deviate from these common patterns.

VII. Related Work

Tu and Godfrey argue for a build-time architectural view in their foundational paper [14]. They study three systems, *Perl*, *JNI*, and *GCC*, to show how build systems can exhibit highly complex architectures. In this paper we use the same abstraction level, files and system processes, to understand structural and compositional properties of built binaries. Our dependency graphs are based on Tu and Godfrey's modelling language for describing build architectures, but we streamline the language to efficiently communicate the information we are interested in. The systems we study in this exploratory study are much smaller and simpler than those in Tu and Godfrey, and in this respect, we contribute an interesting corollary: even "trivial" systems, e.g. patchelf, Figure 5, a single binary built from a single source, can exhibit a surprisingly complex and nuanced build architecture. Complex builds may be the rule, rather than the exception, even in small systems.

Tuunanen et al. used a tracing approach to discover the dependency graph for C-based projects as part of *ASLA* (Automated Software License Analyzer) [8], [15]. They modified the C compiler gcc, the linker ld and the archive builder ar to log information about the actual inputs and outputs used during the build. The resulting dependency graph is used by ASLA to compute license information for binaries and detect potential licensing conflicts. However, instrumenting specific tools has several limitations compared to system call tracing. First, adding the required instrumentation code to programs could be time consuming (finding the "right" places to instrument the code); second, many tools would need to be instrumented: compilers, linkers, assemblers, etc. Finally, a package might contain or build a code generator and use it to generate part of itself. Clearly, such tools cannot be instrumented in advance.

Adams *et al.* analyse debug traces of GNU Make to reverse-engineer large build systems [9]. Their study approaches the build from a developer's point of view, with an aim to support build-system maintenance, refactoring, and comprehension. Adams *et al.* generate and analyse dependency graphs from extracted trace information, similar to us. However, we must emphasize the different aim of our study: Adams *et al.* reverse-engineer the semantic *source architecture* of the build to help developers gain a global picture. We reverse-engineer the logical *binary architecture* to help developers and other stakeholders answer questions about composition and provenance of binary artifacts. Adams *et al.* strive to understand the build system in and of itself, whereas we exploit the build system to acquire specific information.

The main use case for this specific information is copyright analysis of software systems. Disinterring the myriad ways by which "fixed works" of original authorship are embedded, compiled, transformed, and blended into other fixed works is crucial for understanding copyright [16]. A study by German *et al.* showed this information is important to owners and resellers who need to understand the copyright of their systems [13]. German *et al.* studied package metadata in Fedora to find license problems. In their study they showed that Fedora's current package-depends-on-package level of granularity, while helpful, is ultimately insufficient for understanding license interactions in a large open source compilation such as Fedora. Only a file-depends-on-file level of granularity can work, and this study directly addresses that problem.

Build Audit [17] is a tool to extract and archive useful information from a build process by tracing system calls. Although it is capable of showing the invoked processes and involved files, it cannot show the interaction between processes, which we need. Similarly, in [18], [19] strace or ptrace techniques are used to derive dependencies from system calls for parallelizing builds and to diagnose configuration bugs.

VIII. Conclusion

In this paper we have proposed reverse-engineering the dependency graph of binaries by tracing build processes – specifically, we showed that we can do so by tracing *system calls* issued during builds. Our proof-of-concept prototype implementation suggests that system call tracing is a reliable method to detect dependencies, though it may over-estimate the set of dependencies. This is the essential foundation upon which we want to build a license analysis tool.

Acknowledgements: We wish to thank Tim Engelhardt of JBB Rechtsanwälte for his assistance, the FFmpeg developers for their feedback, Karl Trygve Kalleberg for many comments and discussions on license calculi, and Bram Adams for pointing out related work.

References

[1] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th working conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, May 2011, pp. 63–72.

[2] D. M. German and A. E. Hassan, "License integration patterns: Addressing license mismatches in component-based development," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 188–198.

[3] ifrOSS, "GPL court decisions in Europe," http://www.ifross.org/ifross_html/links_en.html#Urteile, 2004–2009.

[4] Software Freedom Law Center, "BusyBox Developers Agree To End GPL Lawsuit Against Verizon," http://www.softwarefreedom.org/news/2008/mar/17/busybox-verizon/, 2008.

[5] S. I. Feldman, "Make—a program for maintaining computer programs," *Software—Practice and Experience*, vol. 9, no. 4, pp. 255–65, 1979.

[6] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, May 2011, pp. 141–150.

[7] M. de Jonge, "Build-level components," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 588–600, Jul. 2005.

[8] T. Tuunanen, J. Koskinen, and T. Kärkkäinen, "Automated software license analysis," *Automated Software Engineering*, vol. 16, pp. 455–490, December 2009.

[9] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *ICSM*. IEEE, 2007, pp. 114–123.

[10] E. Foster-Johnson, *Red Hat RPM Guide*. John Wiley & Sons, 2003.

[11] D. M. German, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, p. 437–446.

[12] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society, May 2004, pp. 583–592.

[13] D. M. Germán, M. D. Penta, and J. Davies, "Understanding and auditing the licensing of open source software distributions," in *ICPC*. IEEE Computer Society, 2010, pp. 84–93.

[14] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *ICSM*, 2001, pp. 398–407.

[15] T. Tuunanen, J. Koskinen, and T. Kärkkäinen, "Retrieving open source software licenses," in *Open Source Systems, IFIP Working Group 2.13 Foundation on Open Source Software*, ser. IFIP, E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, Eds., vol. 203. Springer, Jun. 2006, pp. 35–46.

[16] L. Rosen, *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall, 2004.

[17] Build Audit, "Build Audit," http://buildaudit.sourceforge.net, December 6, 2004.

[18] M. Attariyan and J. Flinn, "Using causality to diagnose configuration bugs," in *2008 USENIX Annual Technical Conference (USENIX '08)*. USENIX, 2008.

[19] D. Coetzee, A. Bhaskar, and G. Necula, "apmake: A reliable parallel build manager," in *2011 USENIX Annual Technical Conference (USENIX '11)*. USENIX, 2011.

TUD-SERG-2012-010
ISSN 1872-5392