Tracing Software Build Processes to Uncover License Compliance Inconsistencies

Sander van der Burg
Delft University of Technology, Netherlands
s.vanderburg@tudelft.nl

Eelco Dolstra
LogicBlox, Inc., USA
eelco.dolstra@logicblox.com

Shane McIntosh
Queen's University, Canada
mcintosh@cs.queensu.ca

Julius Davies
University of British Columbia, Canada
juliusd@cs.ubc.ca

Daniel M. German
University of Victoria, Canada
dmg@uvic.ca

Armijn Hemel
Tjaldur Software Governance Solutions, Netherlands
info@tjaldur.nl

ABSTRACT

Open Source Software (OSS) components form the basis for many software systems. While the use of OSS components accelerates development, client systems must comply with the license terms of the OSS components that they use. Failure to do so exposes client system distributors to possible litigation from copyright holders. Yet despite the importance of license compliance, tool support for license compliance assessment is lacking. In this paper, we propose an approach to construct and analyze the Concrete Build Dependency Graph (CBDG) of a software system by tracing system calls that occur at build-time. Through a case study of seven open source systems, we show that the constructed CBDGs: (1) accurately classify sources as included in or excluded from deliverables with 88%-100% precision and 98%-100% recall, and (2) can uncover license compliance inconsistencies in real software systems – two of which prompted code fixes in the CUPS and FFmpeg systems.

Categories and Subject Descriptors

K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*copyrights, licensing*

Keywords

License compliance; build systems

1. INTRODUCTION

Nowadays, software developers improve development efficiency by reusing OSS components, libraries, and frameworks. These reusable components are released under a variety of different licenses, ranging from the simple and permissive BSD- and MIT-style licenses that allow client systems to include components without publishing any source code, to the more restrictive GPL- and MPL-style licenses that impose license requirements on client systems (i.e., derived works). Due to the intricacies of OSS licenses, it may not be legally permissible to combine certain OSS components with client system code. Client systems that are released under licenses that are incompatible with the licenses of the OSS components that are used to build them are said to have a *license mismatch* problem [9].

Failure to comply with the license terms of reused external components makes the distribution of client system deliverables a potential copyright violation, opening client system distributors up to litigation by copyright holders [13, 16]. To ensure that a client system is compliant with the license terms of reused external components, one must not only understand which licenses govern the reuse these components, but also how they are combined with the source code of the client system. For example, statically linking to external components released under a GPL-style license stipulates that the source code of the client system deliverable be released under a GPL-style license as well.

In order to perform a license compliance assessment on a client system, one must know (for each deliverable being created): (1) which source files of the client system are being used; (2) which external components are being used; and (3) how the client source code and external components are being combined. We, therefore, trace and record each step taken during the build process in order to construct a *Concrete Build Dependency Graph* (CBDG), i.e., a graph that represents the actual dependencies of the client system deliverables. Since build specifications may not explicitly denote all of the client system dependencies, we construct CBDGs by tracing the operating system calls made by build tools during their execution. We annotate the source file nodes in the CBDG with their respective licenses, and analyze the CBDG to detect license compliance inconsistencies.

In order to evaluate our license compliance assessment approach, we perform a case study of seven open source systems, and address the following two research questions:

(RQ1) Does our approach accurately identify the sources that are included in constructed deliverables?
Removing the source files that appear in the CBDG truly causes changes in build behaviour in 88%-100% of cases (precision). Conversely, removing the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643013.

source files that do not appear in the CBDG does not cause changes in build behaviour in 98%-100% of cases (recall).

(RQ2) Does our approach reveal license mismatch issues?
Analysis of the constructed CBDGs reveals license mismatch issues that prompted rapid bug fixes in the FFmpeg and CUPS systems.

The main technical contributions of this paper are:

- The definition of the CBDG, which explicitly describes the steps performed when creating deliverables. Both internal (which client system source files are being used) and external dependencies (e.g., libraries) can be extracted from the CBDG.
- An approach for constructing a CBDG from traces of operating system calls made during a build execution.
- An approach to analyze a CBDG to identify license compliance inconsistencies in client systems.

Paper organization. The remainder of this paper is organized as follows. Section 2 outlines the challenges associated with license compliance assessment. Section 3 describes our CBDG-based approach to license compliance assessment. Section 4 presents the design of the empirical study that we performed to evaluate the accuracy and usefulness of our license compliance assessment approach, while Section 5 presents the results. We discuss the limitations of our approach and its empirical evaluation in Section 6. Section 7 surveys related work. Finally, Section 8 draws conclusions and discusses potential avenues for future work.

2. CHALLENGES OF LICENSE COMPLIANCE ASSESSMENT

License compliance assessment, i.e., the process of detecting inconsistencies in the license terms of an external component and its reuse by a client system, is nontrivial [8, 9, 11]. Figure 1 provides an overview of the challenges of the assessment process. We briefly describe each challenge below.

Challenge 1: Identifying included source files. OSS packages, i.e., collections of related OSS components, are composed of several source code files that often have different (potentially incompatible) licenses. In recent work, we found that 65% of the packages in the Fedora 12 distribution are heterogeneously licensed, with more than one license appearing in their source files [11].

Determining the license that takes precedence over the others in heterogeneously licensed components is often dependent on the configuration of the build system. For example, the FreeBSD kernel is normally released under the permissive BSD-2 license. However, the source code for the FreeBSD kernel includes restrictively licensed GPLv2 source files, which are configured to be excluded from the FreeBSD kernel by default. If the GPLv2 code is configured to be included in the FreeBSD kernel, its license changes from BSD-2 to GPLv2. Hence, knowing which source files are included in (and excluded from) the build process is an important challenge to overcome for license compliance assessment.

Challenge 2: Identifying used external components. While external components that are required by the client

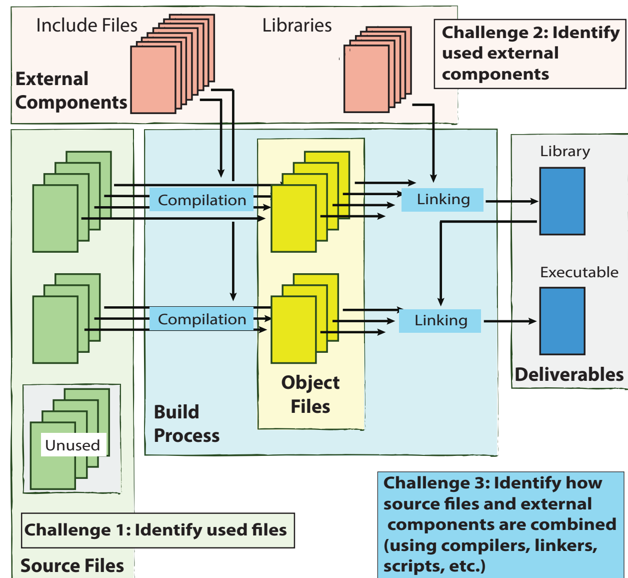


Figure 1: Challenges of license compliance assessment.

system are typically listed in product documentation, this documentation is often: (1) incomplete, i.e., missing dependencies, such as those recursively used to create the external components that are used by the client system, or (2) out of date, i.e., incorrect dependencies, such as those that are no longer used. For example, in recent work, we identified an error in the build system of the PHP package in Fedora 12, where a deliverable was mistakenly being linked to a GPLv2 licensed component [11]. Linking to this GPLv2 component caused the GPLv2 license to take precedence over the PHP license for several PHP deliverables. The build system was later corrected to link the PHP deliverables with an external component that is compatible with the PHP license. Indeed, the external components used to assemble client system deliverables impose restrictions on the license that the client system can be released under.

Challenge 3: Identify how source files and external components are combined. The method by which client source code is combined with external components also has an impact on the license constraints that apply to the client system [2, 9]. For example, external components that are released under the GPL license transitively apply the GPL license to client system deliverables that statically link with them, i.e., client system deliverable must also be distributed under the GPL license if a GPL component is statically linked with it; however, if instead of using static linking components are connected using RPC or a Web API, then the resulting client system might not be required to be licensed under the GPL [7].

2.1 Addressing the Challenges of License Compliance Assessment

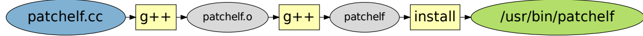
The challenges of license compliance assessment listed above highlight that the legal constraints that are imposed on the distribution of client systems depend not only upon which internal source files and external components are used (challenges 1 and 2), but also upon how these sources are combined to produce client system deliverables (challenge 3).

```
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/
```

(a) The (simplified) Makefile.



(b) The build dependency graph extracted from the Makefile.

Figure 2: The *PatchELF* Makefile and corresponding dependency graph.

The *build system* of a software project specifies this process of translating sources into deliverables. When executed, *build tools* orchestrate the order-dependent execution of compilers and other tools in order to correctly produce deliverables according to build specifications.

While it may seem natural that one could apply static analysis to the build system to address these challenges, this approach is fraught with peril. First, there are several build technologies abound, such as: (1) the countless variants of make; (2) Ant, Maven, and Gradle for Java systems; (3) language-specific scriptable formalisms such as Python's Setuptools or Ruby's Rake; and (4) abstraction-based build technologies like GNU Autotools or Perl's MakeMaker that allow developers to express build dependencies using abstractions, and use platform and configuration specifics to generate appropriate low-level build specifications.

Furthermore, a static analysis of the build system will likely produce an incomplete picture of the concrete client system dependencies. Consider, for instance, the *PatchELF* utility for manipulating Unix executables.[1] *PatchELF* consists of a single C++ source file, patchelf.cc. Figure 2 shows a simplified version of the build system of *PatchELF* (specified using make [6]). Figure 2a shows the build specification that describes how patchelf.cc is compiled into patchelf.o, linked into patchelf, and installed into /usr/bin/patchelf. Static analysis of Figure 2a will produce a *build dependency graph* (Figure 2b), i.e., a directed acyclic graph describing: (1) the sources, deliverables, and intermediate files (as circular nodes), (2) the commands that update deliverables or intermediate files (as square nodes), and (3) the ordering requirements for these commands (as edges). While Figure 2b provides useful information, it does not address the license compliance assessment challenges:

- **Header file nodes are missing (Challenge 1)** – The patchelf.cc source file includes elf.h, yet since the build system does not express this relationship, it is missing from Figure 2b. This relationship is especially important for license compliance assessment in this case, since the elf.h header file that was actually copied from the GNU C library.

- **External component nodes are missing (Challenges 2 and 3)** – The link command that produces the patchelf deliverable does not list automatically linked system libraries, which also have licensing implications.

- **Phony targets may mask the build step that was performed (Challenge 3)** – The install target does not denote that it generates /usr/bin/patchelf deliverable. Hence, the link between the /usr/bin/patchelf deliverable and the patchelf intermediate file would need to be inferred.

A static analysis of build systems for license compliance assessment would therefore not only need to process the various types of build specifications, but would also need a context-specific understanding of the source code (to uncover header file dependencies) and the executed commands (to uncover implicitly linked libraries and build rule side effects). To avoid the pitfalls of static analysis, we use dynamic analysis of build behaviour to perform license compliance assessment. Rather than constructing a build dependency graph from the build specifications, our approach constructs a *Concrete Build Dependency Graph* (CBDG) that contains all of the information that is missing in Figure 2b. Moreover, our CBDG-based approach is technology-agnostic, and can be applied to build systems specified using any technology.

3. LICENSE COMPLIANCE ASSESSMENT APPROACH

In order to detect the license compliance inconsistencies, we construct and analyze the CBDG of a software system. As shown in Figure 3, the process is divided into four steps. We describe each step below.

3.1 License Identification

We first identify the licenses that govern each of the source files of a client system. To do so, we use *Ninka* [10] – an open source license identification tool that analyses sentences in source code comments.[2] Ninka is capable of identifying more than 120 different licenses. Performing this step produces a mapping of source files to licenses.

3.2 Build Trace Collection

In order to address the challenges of license compliance assessment outlined in Section 2, the CBDG must describe which client sources and external components are used, and how they are combined to create the client system deliverables. To achieve this, we record which files are read and written by each step in the build process. Specifically, we trace the *operating system calls*, i.e., calls made by user processes to the operating system kernel, that are made while the build is executing. These system calls provide user-space processes with access to system resources like filesystems. The necessary tools for collecting system call traces are already available for most conventional operating systems, such as Linux, FreeBSD, Mac OS X, and Windows.

To illustrate why system call traces will be useful for license compliance assessment, we use the example of tracing the system calls made when executing make install using the *PatchELF* Makefile shown in Figure 2a. On Linux, this can be done using the strace command[3] as follows: strace -f make

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

[2] http://github.com/dmgerman/ninka

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

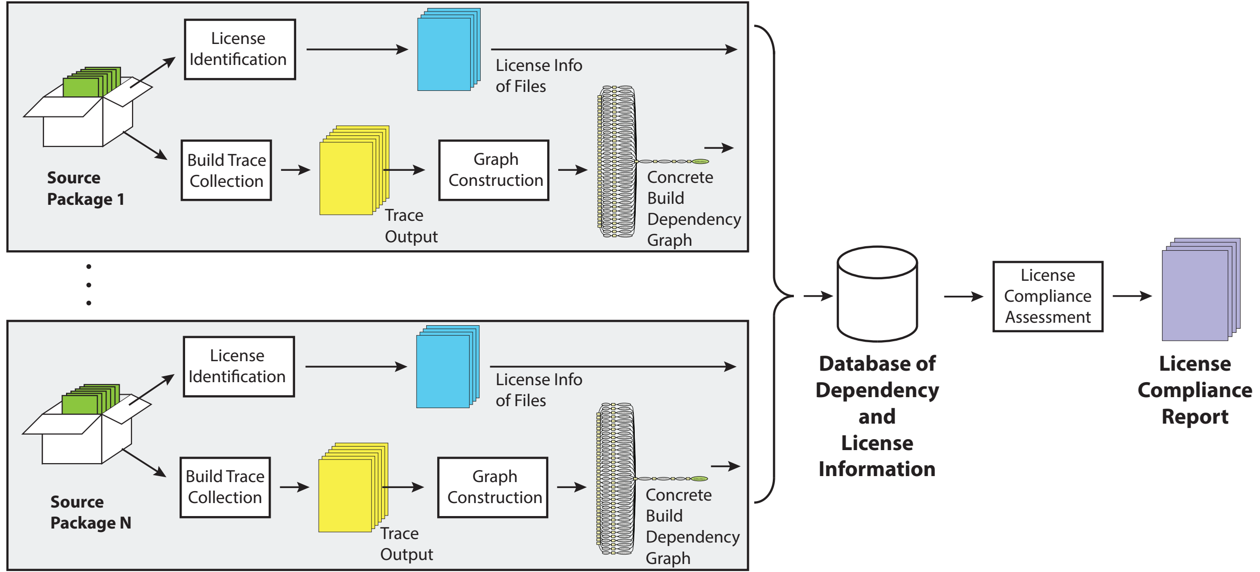


Figure 3: An overview of our license compliance assessment approach.

install. The -f flag instructs the strace command to record system calls made by child processes of the make install command as well. The make install command will first spawn a process to run g++ in order to compile patchelf.o. Since this process will write to the patchelf.o file, it will at some point issue the system call:

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

Furthermore, because g++ must read the patchelf.cc source file and all of the headers that it includes, the trace will include 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)
```

After compiling, the linker needs to read patchelf.o, as well as the libraries and object files, such as /usr/lib/libc.so, /usr/lib/libc_non-shared.a, and /usr/lib/crtn.o) in order to write patchelf. Finally, the install command will read ./patchelf and write /usr/bin/patchelf. Thus, the system call trace contains all of the necessary information to derive the CBDG for the /usr/bin/patchelf deliverable.

Narrowing the scope of traced system calls. Since large projects have complex build systems [14] that can take hours to build [12], the output of strace can be extremely large. For performance reasons, we select only the following types of system calls for CBDG construction:

- System calls that take file name arguments, e.g., open(), rename() and execve().
- System calls related to process management, e.g., vfork(), execve(), and vfork().

3.3 Graph Construction

We parse the collected system call traces of build processes in order to construct the CBDG. Formally, the CBDG is defined as $CBDG = (V, E)$, where $V = V_t \cup V_f$, the union of *task nodes* V_t (i.e., steps in the build process) and *file nodes* V_f (i.e., the files that are read by or created during the build process). The granularity of a task node can vary depending on the level of detail required, e.g., V_t could be the set of build commands or operating system processes. Each task $t \in V_t$ is defined as an ordered tuple $\langle tid, \ldots \rangle$, where $tid \in Tids$ is a symbol that uniquely identifies t. Additional information can also be included in the task tuple (e.g., the name of the program(s) invoked by the task). A file node $f \in V_f$ is also represented using a tuple: $\langle fid, path, license \rangle$, where $fid \in Fids$ is a symbol that uniquely identifies f, *path* is the file's path, and *license* is the license that we identified for f using Ninka (*cf.* Section 3.1).

Each task $t \in V_t$ represents a process that was executed during a build. Information about each t is stored in a $\langle tid, program, args \rangle$ tuple, where *program* is the path of the last program executed by the task (i.e., the last program 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.

Our CBDG constructor assembles the CBDG by reading each build trace line by line. When a new process is created in the trace, a corresponding task node t is added to V_t. When a task t reads a file f in the trace, we add f to V_f (if it did not already exist), and an edge from f to t. Similarly, when a process opens a file for writing in the trace, we add a file node f to V_f, and an edge from the task node to f.

Implementation details. There are several intricacies of system call traces that must be carefully handled when constructing CBDGs:

- **Relative paths must be normalized with respect to the current working directory (*cwd*) of the calling process.** Thus, we must keep track of the current working directory by processing chdir() calls. Since the *cwd* is inherited by child processes, our CBDG constructor must also keep track of parent/child process

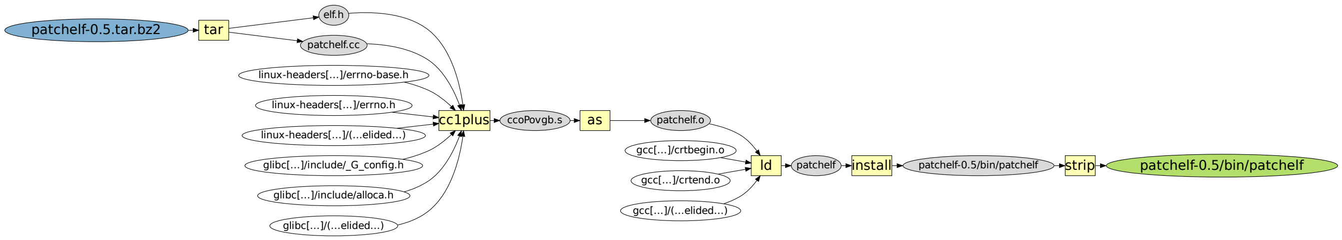


Figure 4: The *PatchELF* dependency graph. External dependencies are depicted in white.

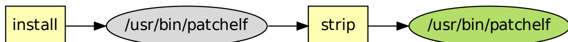
relations, and propagate the *cwd* of the parent to the child when it encounters a fork operation.

- **Process IDs (*PIDs*) may be reused while performing a build.** In fact, this occurs frequently in the build traces of large packages because PIDs are by default limited to 32,768 in Linux. Since CBDG analysis needs to distinguish between different processes that had the same PID, PIDs are not used as unique TIDs in the CBDG. Instead, a unique TID is generated every time a process creation operation is encountered.

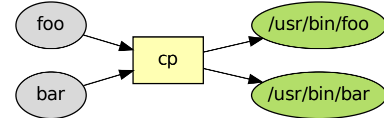
- **While we do need to track system libraries that are dynamically linked to client system deliverables, CBDGs do not need to contain dynamic libraries that are loaded by the tools used during the build execution.** This is because the loading of a dynamic library by a build tool rarely has license compliance implications. We observed that after the dynamic loader has opened all shared libraries, it issues an arch_prctl() system call on x86_64 Linux systems to set the 64-bit base for the FS segment register that is internally used by glibc. Thus, to filter away system calls made due to the loading of a dynamic library, we ignore any open() system calls performed between a call to execve() and arch_prctl().

- **Files may be written more than once during a single build.** Hence, a file's path may not uniquely identify an instance of a file in the CBDG. For example, a deliverable can be written by first linking client code with external components, and then later post-processed by a command like strip, which removes debugging symbols. Represented naively, this scenario would yield the following subgraph:

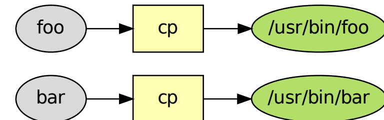
Cycles in the CBDG muddle the data flow, especially if there are multiple processes that update a file – the order that these processes occur in is impossible to discern. Therefore, we create a new node for each modification that a file undergoes. File nodes are disambiguated by tagging them with the ID of the task that created them as shown below:



- **Handling coarse-grained processes.** System call tracing identifies inputs and outputs at the granularity of Unix processes. An underlying assumption is that every process is part of at most one conceptual build step. However, this is not always the case. For example, consider the command cp foo bar /usr/bin/. This command installs two files in the /usr/bin directory. Although conceptually, these are two build steps, since they are executed by a single process, the following subgraph is generated:



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 handle such *coarse-grained* processes that may perform multiple build steps at once (e.g., cp and install) on a case-by-case basis by adjusting the CBDG as follows. First, we scan the CBDG for the set of known coarse-grained processes that have more than one output file, 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 pair of corresponding input and output files, e.g.,



Note that both cp tasks in this graph correspond to the same process in the trace.

Example: The PatchELF CBDG. As with most Unix packages, building the *PatchELF* package starts with unpacking the source code distribution (i.e., patchelf-0.5.tar.bz2), running its configure script, running make, and finally running make install. Figure 4 shows the CBDG resulting from analyzing the trace of a build of this package. Comparing Figure 4 with Figure 2b (i.e., the graph constructed using static analysis) shows that the CBDG exposes dependencies that patchelf.cc not only has on elf.h, but also on numerous header files from glibc and the Linux kernel. Similarly, the linker loads several object files and libraries from glibc and GCC that were invisible to the static analysis. The CBDG also reveals that the installed version of /usr/bin/patchelf is rewritten by strip.

Table 1: An overview of the studied systems.

Package	Version	Domain	Size (SLOC)†	Files
Aterm	2.5	Program transformation	21.5k	133
Opkg	0.1.8	Software package management	22.2k	132
Bash	4.1	Command shell	115.7k	1,114
CUPS	1.4.6	Printer management	142.8k	1,398
Xalan	2.7.1	XML processing	176.0k	1,334
OpenSSL	1.0.0d	Digital communication security	308.0k	2,105
FFmpeg	0.7-rc1	Multimedia processing	442.2k	2,267

† Non-comment, non-whitespace lines of code calculated using SLOCCount (http://www.dwheeler.com/sloccount/).

3.4 License Compliance Assessment

We then use the constructed CBDG to perform license compliance assessment for a given deliverable. The process is divided into three steps:

Step 1: Identify the files $f \subset V_f$ that are included in a deliverable d. We do so using the following logic:

$$\{f \in V_f \mid \deg^-(f) = 0 \wedge \exists \text{ a path in } G \text{ from } f \text{ to } d\}$$

Note that if this logic is applied to a raw CBDG, the list of source files will include source distribution files (e.g., patchelf-0.5.tar.bz2) and omit the source files that were contained within it (e.g., patchelf.cc). While this set of source files is technically correct (i.e., patchelf-0.5.tar.bz2 is the sole source of the *PatchELF* package), it is not useful for license compliance assessment.

To address this, we further filter task nodes such as tar (and its edges) out of the CBDG. Applying the search logic to the filtered *PatchELF* CBDG this yields the following set of source files:

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

Step 2: Identify the licenses of the files $f \subset V_f$ that are included in deliverable d. We used Ninka to determine the licenses of each source file identified by Step 1. We identify the licenses that govern the external components (which we did not have source code for) by examining product documentation. This license information is denoted in the file node tuples in the CBDG. Hence, the CBDG nodes selected by Step 1 also contain their license details.

Step 3: Identify how external components are combined with client deliverables. Finally, we detect how each external component is combined with client deliverables by scanning the process nodes on the path connecting an external source with a client deliverable. For instance, some lawyers consider that statically linking external components imposes more restrictive legal constraints than dynamic linking does.[4] Hence, we can flag the source files of external components that are statically or dynamically linked into client deliverables for further analysis.

4. EMPIRICAL STUDY DESIGN

We performed an empirical study to evaluate the accuracy and usefulness of our license compliance assessment approach. To structure the study, we formulate the following two research questions:

[4]For example, Larry Rosen shares this view [15]; however, the Free Software Foundation considers static and dynamic linking to be equivalent [7].

(RQ1) Does our approach accurately identify the sources that are included in constructed deliverables?
Before the results of our license compliance assessment approach can be validated, we must first ensure that the generated CBDG accurately reflects the build processes of the studied systems.

(RQ2) Does our approach reveal license mismatch issues?
While license compliance assessment using the CBDG may work in theory, it is not clear whether license compliance inconsistencies can actually be uncovered using it. Hence, we set out to detect real-world licensing issues using our approach.

4.1 Studied Systems

We study seven open source systems in order to address our research questions. We study systems of various sizes and domains to combat potential bias in our results. Table 1 provides an overview of the studied systems.

Aterm enables creation and manipulation of Annotated TERMs in C source code. Opkg is a lightweight package manager used to download and install OpenWrt packages. Bash is the GNU implementation of the Bourne Again SHell (BASH) – a Unix command shell. CUPS is a common system for interfacing with printers developed by Apple Inc. for Unix. Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. OpenSSL is a toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. FFmpeg is a package used to record, convert, and stream audio and video.

5. EMPIRICAL STUDY RESULTS

In this section, we present the results of our empirical study with respect to our two research questions. For each research question, we first present our approach to addressing the question, then present the results that we observe in the studied systems, and finally, discuss the broader implications of our findings.

(RQ1) Does our approach accurately identify the sources that are included in constructed deliverables?

Approach. We evaluate how accurately the constructed CBDGs classify files as being included in or excluded from the build process. A file is classified as included if it appears in the CBDG, otherwise it is classified as excluded.

Table 2: Results of RQ1: CBDGs provide very high recall and precision.

Package	Files					False Negatives	False Positives	Recall	Precision
	All	Source files	Excluded	Included	% Included				
Aterm	133	117	60	57	49%	1	0	98%	100%
Opkg	132	106	9	97	92%	1	3	99%	97%
Bash	1,111	1,086	806	280	26%	0	34	100%	88%
CUPS	1,398	1,079	213	866	80%	12	0	99%	100%
Xalan	1,334	1,334	379	955	72%	0	4	100%	99%
OpenSSL	2,105	2,027	1,180	847	42%	0	5	100%	99%
FFmpeg	2,267	2,239	986	1,253	56%	0	7	100%	99%

We use the build output and status to establish the ground truth for our evaluation by executing a clean build after removing each source file in the studied systems. We refer to these builds as the *removed builds*. If the removed build for a source file X executes without error, then the removal of X had no impact on the build output, and X is recorded as excluded from the build process. On the other hand, if the removed build for file X has an error, then X had an impact on the build output, and we mark X as being included in the build process.

We measure the accuracy of a CBDG by comparing the ground truth for each file to the files that appear in the CBDG for each studied system. We use four metrics to measure the accuracy of a CBDG:

False negatives – Files that are excluded from the CBDG, but actually have an impact on the build process.

False positives – Files that are included in the CBDG, but do not have an impact on the build process.

Recall – The proportion of the files that have an impact on the build process that are included in the CBDG.

Precision – The proportion of false positives in the CBDG.

Results. CBDGs contain very few false negatives, and hence have very high recall. Table 2 shows that we only observe a total of 14 false negative in the seven studied systems. We manually investigate the root cause of these false negatives by inspecting the build specifications.

We find that files often appear as false negatives in the CBDG due to errors in the build specifications. For example, the build specifications of Aterm, Opkg, and CUPS check for the existence of these files, yet they were never read (or written) by any of the processes spawned during the build.

On the other hand, the number of false positives was slightly larger. We again perform an inspection of the build specifications in order to determine the root cause for these false positives. The inspection revealed that these false positive files are used only if they were present. If the false positive files were not present, no error was issued, but the output of the build is changed. For example, in Bash, the number of false positives was higher than the other packages because Bash contains various localization files (i.e., .po files) that translate the Bash deliverables into many different languages. The build system of Bash processes any .po files that are present.

> CBDGs can accurately determine whether a source file is included in or excluded from the build process with a recall of 98%-100% and precision of 88%-100%.

Discussion. Although beyond the scope of this paper, the results of our false negative analysis suggest that CBDGs may also be useful for detecting errors in build specifications. If we use the CBDG as the ground truth to evaluate the removed build results (the inverse of the experiment performed above), we could detect errors in the build specifications of studied systems. We expand upon this and other uses of the CBDG in Section 8.

(RQ2) Does our approach reveal license mismatch issues?

Approach. We aim to detect inconsistencies between the license of the client deliverables and the external components that are used to create it. A license of a deliverable is said to be inconsistent with an external component that is used to create it if the license of any file in an external component contains terms that cannot be satisfied by the terms of the license of the deliverable.

In recent work, we analyzed the Fedora 12 distribution to understand how license auditing is performed [11]. One of the challenges we faced was determining which files are used to create a deliverable. This is particularly important in systems that include files under different licenses that do not allow for their composition. For example, one deliverable that contains a file released under the BSD-4 license and another file released under the GPLv2 license. This type of inconsistency can be detected using the CBDG. Any file that is a predecessor to a deliverable in the CBDG must have a license that is compatible with the license of the deliverable.

Specifically, our license compliance assessment is performed using the following five steps:

1. Extract the licenses of each file in each of the studied systems using Ninka.
2. Identify the declared license of the client deliverables by examining product documentation (i.e. the license of the deliverables as stated by the authors of the software).
3. Trace the build of each system and generate its CBDG.
4. Annotate the CBDG file nodes with license information.
5. Traverse the CBDG to identify the sources that are used to create the client deliverables.
6. Mark client deliverables that contain sources that are released under incompatible licenses as inconsistencies.

In our prior study [11], we identified three systems that appear to contain license incompatibilities.

FFmpeg – The license of the FFmpeg deliverables varies depending on the configuration flags that are specified prior to executing the build. By default, its deliverables are released under the LGPLv2+ license. However, when configured using the appropriate flags, the deliverables are released under the GPLv2+ license.

CUPS – The CUPS deliverables are licensed under the permissive CUPS license. The studied version (1.4.6) contains three files (backend/*scsi*) that are released under the BSD-4 license, which is not compatible with the CUPS License. It is important to know whether any of these three files are being combined with CUPS licensed files during the build process.

Bash – The Bash deliverables are licensed under the GPLv3+. However, there is a source file (examples/loadables/getconf.c) that is released under the BSD-4 license, which is incompatible with the GPLv3+. Again, we would like to know if this file is combined with the GPLv3+ licensed files during the build process.

We perform a deeper analysis of these three systems using our approach to detect if there are truly inconsistencies that appear in the constructed deliverables. We identify inconsistencies using our license compliance assessment approach (*cf.* Section 3).

Results. We uncovered two LGPLv2+ vs GPLv2+ license compliance inconsistencies in ffmpeg. By default, FFmpeg is expected to create libraries and binary programs to be released under the LGPLv2+ license. Through CBDG analysis, we found two inconsistencies, i.e., two files that are released under the GPLv2+ license that were being used during the default build process. First, libpostproc/postprocess.h is a header file that is include throughout the FFmpeg codebase. However, this header only contains definitions of constants. Since constants are likely not copyrightable (they are likely considered facts and not creative expression), we believe this is a low risk violation.

However, the second inconsistency in libavfilter/x86/gradfun.c contains source code (inlined assembly) that was being compiled into the libavfilter library and redistributed with the FFmpeg deliverables. By combining libavfilter/x86/gradfun.c (licensed under the GPLv2+) with the LGPLv2+ code requires libavfilter to also be released under the GPLv2+ license. The CBDG of libavfilter is shown in Figure 5, and is a high risk violation.

To confirm that the libavfilter/x86/gradfun.c inconsistency was of serious concern, we contacted the FFmpeg development team. 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 libavfilter/x86/gradfun.c were contacted to arrange relicensing of this file under the LGPLv2.1+.[5] Three days later, this file's license was changed to LGPLv2+, which is compatible with the LGPLv2.1+.[6]

We identified three files that caused license compliance inconsistencies in CUPS. Specifically, backend/(scsi|scsi-iris|scsi-linux).c were released under the BSD-4 license, and were incompatible with the permissive CUPS license. According to the CBDG, these files were used to create the backend/scsi deliverable, which is distributed with the CUPS deliverables. To confirm this inconsistency, we filed a bug in the CUPS issue tracker.[7] One day later, the bug was resolved by removing the offending files, since SCSI devices were no longer supported.

One license compliance inconsistency was detected in the Bash system. The examples/loadables/getconf.c file is released under the BSD-4 license, which is incompatible with the other files that are released under the GPLv3+ license. According to the CBDG, this file is compiled into a program called examples/loadables/getconf. Since this file is only an sample program that is not part of a distributed during product installation, this does not raise license compliance concerns.

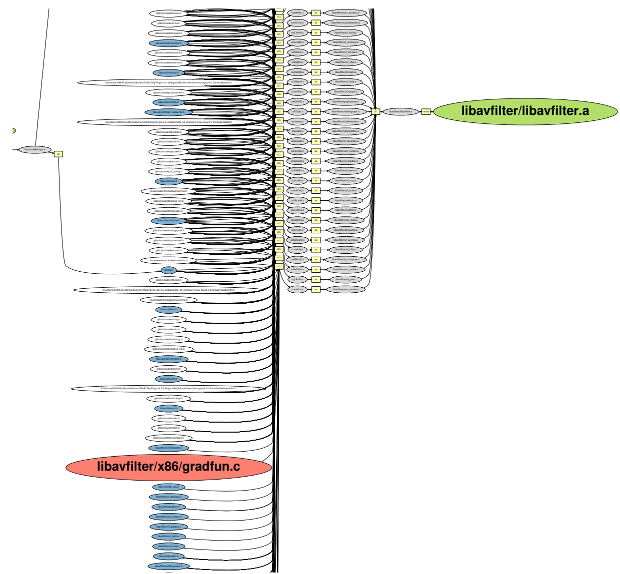


Figure 5: Excerpt of the CBDG of libavfilter. The library libavfilter is licensed under the LGPLv2.1+, but required libavfilter/x86/gradfun.c (depicted in red) licensed under the GPLv2+. We notified the FFmpeg team, and they quickly corrected this inconsistency.

> Our CBDG-based approach identifies several license compliance inconsistencies in the studied systems. Two of these inconsistencies generated prompt responses (in the form of code fixes) from the FFmpeg and CUPS development teams.

Discussion. The fact that a file with a non-compatible license is used during the creation of a deliverable does not necessarily imply a license violation. One needs to understand how such a file is combined with the other source code. This information is explicitly stated in the CBDG. We expect that CBDGs can assist in the process of license compliance assessment by removing files that are not included in the build process, and by showing the processes that read each of the files and which deliverables that they are contained in.

[5] http://web.archiveorange.com/archive/v/4q4BhMcVOjgXyfN3wyhB

[6] http://FFmpeg.org/pipermail/FFmpeg-cvslog/2011-July/039224.html

[7] http://CUPS.org/str.php?L3509

6. THREATS TO VALIDITY

We now discuss the threats to the validity of our analysis.

6.1 Construct Validity

Coarse-grained processes, i.e. those processes that perform several build steps, yield dependency subgraphs that do not accurately portray concrete dependencies. Thus, files may be falsely reported as dependencies, which can in turn lead to false positives in our license compliance assessment report. While we decompose many known-to-be coarse-grained processes, those that are unknown may introduce noise in our assessment report. However, our approach for handling coarse-grained processes is conservative, i.e., coarse-grained processes trigger an overestimation of the set of source files used to produce deliverables, but it will not underestimate it.

Processes may access files that do not have an impact on the deliverables. Consider, for instance, if a C compiler opened all header files in its search path regardless of whether they were included in the source file(s) being compiled. Although we have not encountered such cases, we cannot rule out the possibility. Nonetheless, we conservatively mark such files as being part of the build process, since they may have an impact on the deliverables, and hence, the role that they play in the build process should be assessed for potential license compliance implications.

We assume that the external components that are used by the studied systems have correctly document the licenses that they are released under. However, due to the heterogeneity of licenses used by source files that make up these components [11], the reported licenses may not be correct. We are actively expanding the scope of our CBDG to include source file nodes for external components by collecting additional system call traces from their build processes. Using the expanded CBDG, our license compliance assessment would apply directly to the licenses listed in the internal and external source code files that compose client system deliverables.

The CBDG constructor does not trace file descriptors and inter-process communication (e.g. through pipes). For instance, the CBDG constructor fails to detect that the patch task in the command cat foo.patch | patch bar.c depends on foo.patch. The CBDG constructor only detects that the patch task reads and (re)creates bar.c.

6.2 Internal Validity

We assume that the build systems of the studied systems are correct, whereas defective build systems may fail nondeterministically. Such build system defects could introduce noise in the results of RQ1, i.e., a build execution may fail due to a defect in the build system rather than because of the file that we have temporarily removed. However, a manual analysis of a sample of injected build failures suggests that they are indeed caused by the removed files.

6.3 External Validity

We focus our evaluation on seven open source systems, which threatens the generalizability of our case study results. However, we studied a variety of systems from different domains to combat potential bias. Nonetheless, additional replication studies are needed.

The build systems of the studied systems generate (or maintain) make specifications, which may bias our case study results towards such technologies. Nonetheless, our approach is agnostic of the underlying build system, operating on system call traces, which can theoretically be extracted from any build system.

Our approach requires access to the source code of a system and all of its dependencies in order to completely assess potential license implications. Such a constraint may limit the usefulness of our approach to identifying license compliance inconsistencies in the reuse of open source packages. On the other hand, software distributions that can be completely deployed from source code are not uncommon nowadays (e.g., Gentoo Linux[8] or NixOS [5]).

7. RELATED WORK

We now discuss the related work with respect to build system analysis and licence compliance assessment.

7.1 Build System Analysis

Prior work has shown that the build system contains plenty of information that can be leveraged for other purposes. For example, Tu and Godfrey show that information from the build system can be used to compose a "build-time architectural view" [18], which they create by hand by inspecting the build documentation of a system. We consider our work an extension or theirs, since the CBDG is a build-time architectural view. Moreover, we provide a method to create and analyze CBDGs automatically.

Other work has constructed build dependency graphs to assist in build system maintenance. Through dynamic analysis of make debugging output, Adams *et al.* develop the MAKAO tool to visualize and reason about build dependencies specified in Makefiles [1]. Tamrawi *et al.* propose a technique for verifying Makefile behaviour by constructing symbolic dependency graphs using static analysis [17]. Although our approach is also based on a build dependency graph (i.e., CBDGs), our goal is to detect license compliance inconsistencies rather than support build maintenance. Hence, our approach is based on a more concrete instance of a build dependency graph that contains dependencies that are often omitted from the build specifications.

Others have also used operating system call traces to examine concrete build system interactions. Perhaps the most related instance is that of "Build Audit" [3], a tool that produces a report of the processes invoked by and files involved in the build process. Coetzee *et al.* also use operating system call tracing to reliably derive dependencies for accelerating slow builds by optimally parallelizing them [4]. However, these tracing tools and techniques do not retain the information in a graph – a necessary precondition for license compliance assessment. Although these tracing techniques will reveal which files were read and written during the build, this only addresses two of the three challenges of license compliance analysis (*cf.* Section 2). These traces alone do not describe how client code and external components are combined, which is critical for determining the licensing constraints that are imposed by the external components.

7.2 License Compliance Assessment

The reuse of open source in both commercial and other open source systems has created a need for license compliance assessment support. Von Willebrand and Partanen

[8]http://www.gentoo.org

describe the difficulties of licensing compliance assessment, stating that "it is not uncommon for FOSS packages to contain code that causes them to pose potential or clear risks when redistributing them" [21]. Combining components and files governed by different licenses makes this process difficult. Bain leads a group of lawyers in the creation of a document that explains the different ways of combining components governed by different licenses and the legal implications [2]. German and Hassan documented the ways in which open source systems address the combination of components with different, potentially incompatible licenses [9]. German and di Penta have also proposed a method to perform license compliance assessment of Java applications [8]. They discuss three main challenges: provenance discovery, license identification, and architectural analysis. Only the first two challenges have been addressed.

Tuunanen *et al.* used a tracing approach to discover the dependency graph for C-based projects as part of *ASLA* (Automated Software License Analyzer) [19, 20]. 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. 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, and so on. Finally, a package might contain or build a code generator and use it to generate part of itself. On the other hand, the system call tracing approach we propose does not require instrumentation of the build tools and can be applied to any build environment.

The main use case for this specific information is license compliance analysis of software systems. Understanding the myriad of ways that "fixed works" of original authorship are embedded, compiled, transformed, and blended into other fixed works is crucial for understanding licenses and copyrights [15]. A study by German *et al.* showed this information is important to owners and distributors who need to understand the copyrights of their systems [11]. They 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.

8. CONCLUSIONS

Modern software development relies heavily on the reuse of (open source) software components, libraries, and frameworks. However, the plethora of (potentially incompatible) licenses that these components are distributed under impose legal constraints on client systems. Indeed, license compliance is a critical nonfunctional requirement for software systems that rely on such external components.

To assist in license compliance assessment, we propose an approach that constructs and analyzes Concrete Build Dependency Graphs (CBDGs) for license compliance inconsistencies. These CBDGs not only identify the source files and external components that are included in client deliverables, but also describes how they are combined to produce deliverables – a detail that has critical legal implications.

To evaluate our CBDG-based license compliance assessment approach, we perform an empirical study on seven open source systems. We make the following observations:

- The CBDGs that were constructed from the studied systems can accurately classify files that have an impact on the client deliverables with a recall of 98%-100% and a precision of 88%-100%.
- Our analysis uncovers several license compliance inconsistencies – two of which prompted rapidly fixed bugs in the FFmpeg and CUPS systems.

Future work. Although this paper focuses on license compliance assessment, the CBDG is a generic data source that can be used for a variety of other tasks. We have shown that the CBDG is an accurate source of concrete build execution data. In future work, we plan to use this data to verify the correctness of build specifications. Dependencies that appear in the CBDG, but are not expressed in the build specifications may lead to incorrect build system behaviour. Adams *et al.* have shown that incorrect build behaviour can lead to defects that are frustrating and difficult to diagnose [1]. In prior work, we have shown that incorrect build behaviour can even impact end-users if it permeates through to software releases [14]. Hence, we believe that verification of build systems is a natural next step for CBDG analysis.

An automatic license analysis of a CBDG would not only flag potential inconsistencies/violations, but would also list the legal requirements that the licenses being used impose on the licensor. In order to automatically analyze CBDGs for license compliance, it is necessary to have a calculus that is aware of the requirements that each license has, and, given a task node in a graph, it can make an assessment of the legal requirements that such input files create on the resulting file. This calculus cannot be universal: different jurisdictions have different views on what a license requirement might mean, and some organizations might be more sensitive to legal risks that others. Thus, such a calculus would need to be configurable by its user.

Due to the dynamic nature of the CBDG, it can only detect licensing inconsistencies in a single build configuration. Many modern software systems support several configurations, which may vary in terms of licensing implications. More powerful static analysis techniques for build systems could lead to more powerful license compliance assessment tools capable of examining several configurations at once.

9. ACKNOWLEDGEMENTS

We would like 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.

This research was partially supported by the Natural Science and Engineering Research Council of Canada (NSERC).

10. REFERENCES

[1] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter. Design recovery and maintenance of build systems. In *ICSM*, pages 114–123. IEEE, 2007.

[2] M. Bain. Software interactions and the GNU general public license. *International Free and Open Source Software Law Review*, 2(2):165–180, 2010.

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

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

[5] E. Dolstra, A. Löh, and N. Pierron. NixOS: A purely functional linux distribution. *J. Funct. Program.*, 20(5-6):577–615, Nov. 2010.

[6] S. I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, 1979.

[7] Free Software Foundation. Various licenses and comments about them. https://www.gnu.org/licenses/license-list.html, 2014.

[8] D. German and M. Di Penta. A method for open source license compliance of java applications. *Software, IEEE*, 29(3):58–63, May 2012.

[9] 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*, ICSE '09, pages 188–198, Washington, DC, USA, 2009. IEEE Computer Society.

[10] 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*, ASE '10, page 437–446, New York, NY, USA, 2010. ACM.

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

[12] A. E. Hassan and K. Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the 21st International Conference on Automated Software Engineering*, ASE '06, pages 189–198. ACM, November 2006.

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

[14] 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*, ICSE '11, pages 141–150, New York, NY, USA, May 2011. ACM.

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

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

[17] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 650–660, 2012.

[18] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *17th IEEE International Conference on Software Maintenance*, ICSM '01, pages 398–407, 2001.

[19] T. Tuunanen, J. Koskinen, and T. Kärkkäinen. Automated software license analysis. *Automated Software Engineering*, 16:455–490, December 2009.

[20] T. Tuunanen, J. Koskinen, and T. Kärkkäinen. Retrieving open source software licenses. In E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, editors, *Open Source Systems, IFIP Working Group 2.13 Foundation on Open Source Software*, volume 203 of *IFIP*, pages 35–46. Springer, June 2006.

[21] M. von Willebranda and M.-P. Partanenb. Package review as a part of free and open source software compliance. *International Free and Open Source Software Law Review*, 2(1):39–60, 2010.