Finding Software License Violations Through Binary Code Clone Detection

Armijn Hemel
gpl-violations.org
The Netherlands
armijn@gpl-violations.org

Karl Trygve Kalleberg
KolibriFX
Norway
karltk@kolibrifx.com

Rob Vermaas,
Eelco Dolstra
Delft University of Technology
The Netherlands
rob.vermaas@gmail.com,
e.dolstra@tudelft.nl

ABSTRACT

Software released in binary form frequently uses third-party packages without respecting their licensing terms. For instance, many consumer devices have firmware containing the Linux kernel, without the suppliers following the requirements of the GNU General Public License. Such license violations are often accidental, e.g., when vendors receive binary code from their suppliers with no indication of its provenance. To help find such violations, we have developed the *Binary Analysis Tool* (BAT), a system for code clone detection in binaries. Given a binary, such as a firmware image, it attempts to detect cloning of code from repositories of packages in source and binary form. We evaluate and compare the effectiveness of three of BAT's clone detection techniques: scanning for string literals, detecting similarity through data compression, and detecting similarity by computing binary deltas.

Categories and Subject Descriptors

K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*Copyrights, Licensing*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Experimentation, Legal Aspects

Keywords

Repository mining, binary analysis, code clone detection, firmware

1. INTRODUCTION

The success of free software is evident from the large and growing number of hardware devices that include free software components. Devices such as routers, televisions, set-top boxes and media players are commonly based on software such as the Linux kernel, the Samba file/print server and the BusyBox toolset [24, 1]. However, the software included in such devices is frequently distributed in a way that violates the licensing terms imposed by the original developers. For instance, the GNU General Public License (GPL) used by software such as the Linux kernel is a *copyleft* license that requires distributors of derived works to make the full source code available to recipients. This has led to numerous license violations in the past, exposing companies to significant legal risk.

While observing the license requirements appears principally a legal issue, many infringements are an *inadvertent* result of the ignorance of distributors as to the actual content of the software included in their devices. Such companies often receive software components from upstream suppliers in binary form, and thus cannot easily assess whether they contain unlicensed third-party code. For instance, many vendors of consumer-grade ADSL routers in Europe simply resell devices from suppliers in the Far East, after applying their own branding.

Thus, there is a need for downstream users to determine what third-party software, if any, is included in binary files from upstream suppliers. In this paper, we present the *Binary Analysis Tool* (BAT) that detects *code cloning* in binaries. It does so by scanning the binary for evidence that its source code included specific third-party software packages (such as the Linux kernel or BusyBox) stored in a repository. The output of the tool is a list of third-party packages that are likely used in the binary. If the repository has sufficient coverage of the corpus of free software packages, then this gives the user reasonably complete knowledge about the contents of the binary. Determining the corresponding licenses, resulting legal obligations, and whether any violations may have taken place is beyond the scope of BAT. We describe the structure of BAT, and how it decomposes "opaque" binaries such as firmwares, in Section 3.

We have implemented a number of techniques to determine potential code cloning in a binary:

- Searching for string literals that occur in the source files in the repository (Section 4).
- Using data compression as a measure of the similarity between the subject binary and binaries in the repository (Section 5). If the compressed concatenation of the subject binary and a binary in the repository is significantly smaller that the sum of the individually compressed files, then this is evidence of code cloning.

MSR'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA

- Computing binary deltas between the subject binary and binaries in the repository (Section 6). Small deltas indicate potential code cloning.

In Section 7, we evaluate the effectiveness of these methods. We discuss the limitations of our implementation and evaluation in Section 8.

2. BACKGROUND AND RELATED WORK

In this section, we motivate our work, which grew out of the first author's work in the gpl-violations.org project [24, 22], and discuss related prior work.

The Origin of GPL Violations. In the current consumer electronics industry many companies are involved in creating a single product. It has become very rare that one single company manufactures a product from start to finish. Instead, a whole chain of companies (the "supply chain") is responsible for the creation of products. The company who eventually sells the product itself often has surprisingly little involvement apart from branding. A good example to see how the market works is the website alibaba.com, where companies can order all kinds of electronic devices in bulk from a multitude of vendors in Asia and apply their own branding.

In the supply chain source code and binary artifacts are passed around, with each company possibly adding or removing binary code, without distributing the changes to the source code accordingly. Source code releases are often out of sync with binary releases, or are not even passed down the supply chain. The main reason that this happens is that many companies lack the right processes to deal with this situation. In the consumer electronics market margins are very thin. Implementing the proper processes (source code control systems, compliance officer, license checks) costs money (investments in time and expertise) in the short term and only starts paying off in the mid- to long term. Since much of the development in this industry is done with the short term in mind, it gives companies who do not implement these processes a competitive advantage over competitors who do.

Eventually this lack of processes exposes the company that is bringing the product to market to legal action from copyright holders, which could include suing for damages, or forcing a product off the market completely.

A widely publicized GPL violation is the Linksys (now Cisco) WRT54G router. This router was released in 2003 without following the conditions as outlined in the GPLv2, under which most of the software (the Linux kernel and BusyBox) that was embedded in the router is licensed. For example, there was no GPL license text included with the device. Furthermore there was no source code accompanying the device (as described in GPLv2, section 3a), nor a written offer for the source code (GPLv2, section 3b). Eventually the source code was released to the public. Linksys did not make the device itself, but bought it from CyberTAN, who based their product on a board that contained a chipset made by Broadcom. Still Linksys was held responsible.

License violations are very common. The gpl-violations.org project (founded by Harald Welte) has enforced compliance on more than 150 products, as well as solved compliance issues in more than a hundred other products directly [23]. Other publicized violations were court cases in Germany against Sitecom, D-Link and Skype. Furthermore there was legal action in the US from the Free Software Foundation against Cisco in 2008, again for Linksys products, as well as legal actions in the US by the BusyBox project against Best Buy, Samsung, JVC and others [21].

Detecting Code Clones. Thus there is a pressing need for downstream companies to be able to determine whether code received from upstream suppliers includes third-party code. That is, they need to detect whether the code contains *clones* of third-party code. (We use the term "code clone" broadly here: it not only includes the result of "cut and paste" cloning at the source level, but also inclusion of third-party packages into the final binary through proper component composition methods such as static linking.)

There has been a great deal of work on detecting clones at the source level (e.g. [9, 17, 3]). Historically this work was primarily motivated by the belief that code clones are a "bad smell" that indicates poor maintainability (which may not be true [15]). More recently, source clone detection methods have been applied to study the flow of source code through cloning between free software projects, in conjunction with license mining and classification [11]. Underlining the economic significance of the problem, there are several companies (BlackDuck, Palamida, OpenLogic, and Protecode) that market solutions that scan source code for code clones.

However, as we pointed out, tools that operate at the source level are insufficient here because downstream companies often receive only binary code, or, if they do get source code, there is no guarantee that the source code matches the binary and is complete. For instance, for a router, the upstream supplier might omit the source code of the GPL-licensed boot loader added later in the build process.

Finding Clones in Binaries. Thus, it is necessary to analyse binaries to find evidence of code cloning. This is typically done manually in an *ad hoc* fashion by experienced engineers. They search firmwares for known markers indicating compressed files or file systems in firmwares, extract these from the firmware and analyse files they found [13]. If these files are executables they are searched for known marker strings, such as print statements, debug strings, copyright strings, license strings, and so on, as well as characteristics that are unlikely to change like names and library version numbers. For instance, to detect possible violations of the GPL license on the BusyBox package, one would manually scan the binary for certain strings that occur in BusyBox.

However, such a manual approach is limited, because it only allows the user to detect clones of a small set of packages. In this paper, we therefore seek to detect cloning in binaries automatically by comparing a given binary against a large *repository* of packages. There is a limited amount of previous work on binary clone detection. Sæbjørnsen *et al.* present a scalable algorithm for detecting code clones in binary executables [20]. Their approach is general, but must be instantiated with a disassembler for every architecture. The algorithm is designed to find all possible clones in a set of binaries, including "self-clones" occurring only inside one given binary. A related approach is given by Davis et al [5]. Di Penta *et al.* [6] describe a tool that identifies licensing of Java archives (JARs) by analysing Java .class files and submitting detected package and class names to Google Code Search to identify the provenance of the code. However,

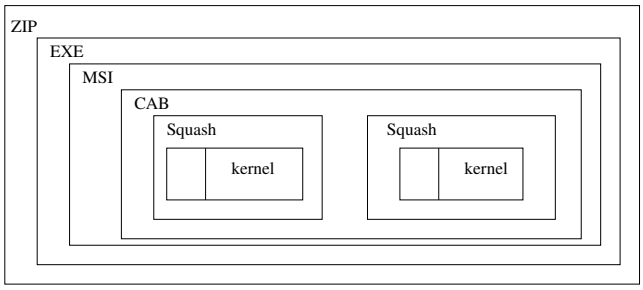

Figure 1: Structure of an NComputing L300 firmware distribution image

this method is possible due to the high-level nature of Java bytecode; machine code such as that contained in firmware images generally does not expose such information.

License Analysis. Clone detection in itself does not point out legal issues but can only assist in pointing out possible problems. This is because legal interpretation of the licenses is not a technical issue. For example, we cannot let BAT check the physical contents of the box of the device to search for a written offer for the source code, which is one possible way to fulfill the requirements of the GPLv2 license.

Furthermore, a company could have obtained an alternative, proprietary license for a package from the copyright holders[1] which might lead to false positives if we would try to draw legal conclusions from the results of a BAT scan.

There is a substantial amount of work on analysing software licenses in open source projects. This work is not *directly* applicable to binary analysis, because binaries usually do not contain copies of the licenses that appear in the source code. However, if binary analysis reveals likely use of some open source package, tools such as FOSSology [14] or Ninka [12] can be used to determine its licenses by analysing its source code, which in turn may reveal possible license obligations upon the distributor of the binary.

Di Penta *et al.* described a method to track changes to licenses, which is important because such changes can have far-reaching consequences [7].

3. THE BINARY ANALYSIS TOOL

The Binary Analysis Tool is a modular framework that can analyse binary files such as firmwares. BAT analyses *opaque* binaries, meaning that the binaries do not necessarily have a known structure. For instance, firmwares are often a mishmash of obscure filesystems, ROM dumps, tar or cpio archives, compression layers and self-unpacking shell scripts. Figure 1 shows an example of such a binary: an installer for the firmware of an *NComputing L300* thin desktop client. It contains, within several layers of archive and installer formats, compressed SquashFS filesystems to be loaded onto the device, containing e.g. the device's Linux kernel. To extract useful information, BAT scans such input files for recognisable structures such as magic numbers, recursively unpacks and decompresses where possible, and applies code clone analyses to the resulting files.

[1]For the vast majority of GPL licensed packages this is very unlikely to happen due to distributed copyright: to relicense a work, permission from all copyright holders is needed.

It applies two types of scans to the seemingly unstructured data to extract information:

- *Unpack scans*, specifically to find and extract filesystems or compressed files that are inside a binary file.
- *Leaf scans*, scans per file that search for specific information, such as strings.

The unpack scans first search for markers indicating the start of a filesystem, archive or compressed file. If it finds a marker it tries to unpack the file at that position. If it is successful in unpacking, BAT will scan all files that were unpacked recursively. BAT supports many different archive and compression formats, such as gzip, RAR, ZIP, bzip2, SquashFS, tar and several Windows formats (CAB, InstallShield), with more being added regularly. In addition, BAT can unpack any filesystem supported by the Linux kernel by starting a Linux virtual machine to mount the filesystem image and copy its contents[2].

For every leaf file (i.e., files that are not decomposed further), the file type is extracted using the Unix file command. Leaf scans are then applied to try to find useful reverse engineering information. For instance, if the file is an ELF executable, dynamic library or object file, a scanner extracts the architecture and a list of dynamically linked libraries; if it is a Linux kernel module, a scan extracts the license that may be recorded in the license field.

Previously, there was a family of scans to detect the presence of common GPL-licensed packages by searching for selected strings. For instance, to detect the presence of BusyBox, a scan searched for the string literal "BusyBox v". These scans were limited to the "usual suspects" — packages that are the frequent subject of GPL violations, such as BusyBox, Linux, and Samba. In Section 4, we generalise over these scans by mining repositories of software packages for string literals that uniquely identify packages. In Sections 5 and 6, we generalise this further by determining the similarity between leaf files and repositories of binary packages.

4. DETECTION USING STRINGS

The first generic scanner used by BAT to find code cloning searches for *strings* inside binaries, comparing them against a database of *string literals* extracted from a large repository of source code of open source packages. This approach has several useful characteristics:

- It is simple to implement, since string literals can be extracted easily from both source and binary code.
- It does not require the packages in the repository to be compiled. The strings that end up in compiled code are generally architecture-independent (barring the occasional #ifdef, and even so it doesn't hurt if the database contains strings for other architectures than the binary's).

[2]We do the mount in a VM, rather than on the host machine, for several reasons. The host kernel may not support the necessary filesystems (in fact, may not be a Linux machine). Also, some firmwares use modified versions of filesystems such as SquashFS that require patches to the kernel; BAT can build and use a patched kernel on the fly. Finally, loopback mounts require root privileges, while VM executions using QEMU/KVM do not.

- Strings tend to be *stable*, i.e., many strings will be the same across versions and refactoring operations, and they are not affected by changes to the compiler.

Extracting Strings from Source. The strings database is created as follows. Given a repository of source distributions of software packages, an extraction tool unpacks each of these and looks for C or C++ source files. For the firmware domain, C and C++ are essentially the only languages that we need to support, with the exception of shell scripts that can be detected using regular source code clone detection techniques.

Each of the source files is then scanned for string tokens. In our current prototype implementation, we do this using a fairly *ad hoc* Unix sed script to strip comments from the source and a regular expression to extract the string literals. In the future, we intend to use a proper lexer or parser. However, the current approach has the advantage of being easy to adapt to other languages; indeed, it is hard to write a parser that works properly on all language variants that can be expected in a large corpus of source code [9], so a pragmatic extraction method may well be more effective. Furthermore, parsing C and C++ source code typically requires the preprocessor to be applied, which in turn requires knowledge of the header search path and compiler variables set by the package's build system. (For instance, we would have to analyse the package's Makefiles.) This makes precise automatic string extraction on large repositories difficult. A parser based on island grammars [18] should address this problem.

Each string literal is stored in the database as a tuple $(string, packageName, version, sourceFile)$. Because of the way the scanner looks for strings (discussed below), multiline string literals are broken into their constituent lines and stored separately, with newline characters removed.

For our evaluation (Section 7), we obtained all 23,896 source RPM packages from Fedora releases 5, 9, 11 and 14 (released between March 2006 and November 2010). From these, we extracted and unpacked all files with extension tar.gz or tar.bz2 – 16,085 in total. From 1,728,718 C and C++ source files we extracted 42,238,120 string literals. The resulting SQLite database is 13 GiB in size. Our prototype extraction tool took 101 hours to complete, but we expect that this can be optimised significantly.

Detecting Strings in Binaries. Given a binary file x, the scanner first extracts strings by calling the GNU strings tool, which looks for sequences of printable ASCII characters followed by an unprintable character (e.g., most in the range 0x00–0x1f). Note that because strings in the C language are terminated with a 0 byte (which is unprintable), string literals are never merged by the strings tool. The scanner then looks up all detected strings in the database to determine if they occur in the source code of any package. For each string, it makes note of the matching tuples.

Note that looking for printable characters means that no string literal containing an unprintable character (e.g., "foo-bar\0xff") is detected. An improved implementation would search the strings database for every byte sequence in the file (skipping any sequence with a prefix that doesn't match a prefix in the database).

Ranking Method. Given a set of tuples that match the binary, we must then determine whether we can conclude probable cloning, and present conclusions to the user. For instance, if we find a large number of strings that only occur in package foo in the repository, it is reasonable to conclude that the binary contains a clone of foo. On the other hand, very short strings, or longer strings that occur in many packages[3], are not strong evidence of cloning. Thus, for each package that contains at least one string literal found in the binary, we compute a *score* that represents the strength of the evidence that a clone of that package occurs in the binary. BAT then presents the detected packages, ordered by descending score, up to a cut-off value.

The basic scoring metric used is the sum of the length of the strings that match a package. Intuitively, a package p with a score of n denotes that the package matches n characters worth of strings in the binary. However, the contribution of a string to the score must be adjusted to take non-uniqueness into account. Therefore, the contribution of a string s to the score of a package p is

$$\frac{\text{length}(s)}{\alpha^{|\text{pkgs}(s)|-1}}$$

where $\text{pkgs}(s)$ is the set of packages in the repository that contain the string s. The constant $\alpha > 1$ causes strings that happen frequently to have a rapidly diminishing contribution to the score. (BAT uses $\alpha = 5$.) Note that the granularity of scoring is at the package level, rather than at the package version level: a string that occurs in multiple versions of package p counts as a single hit for p.

However, there is a serious problem with this metric: it fails to take into account *internal cloning* in the repository. This is the phenomenon that some packages in the repository are copied verbatim into the source trees of other packages. For instance, there are 27 packages in our repository (such as Firefox) that contain a copy of SQLite, and 5 that contain a copy of libxml2. String matches against such cloned packages must not produce a drastically lowered score, but instead should be counted as a single hit against the "original" package (e.g., SQLite).

The best approach to deal with this would be to detect source code clones in the repository and eliminate cloned strings from the database. Since this is expensive, our current implementation uses the following heuristic to detect wholesale internal cloning, i.e., the common case where a package source tree is included largely unmodified in another tree: it groups together all matches of a string s in different packages that are in *identically named source files*, and consideres these internal clones. For example, the string internal error: inflate stream corrupt occurs in four different packages in the repository, each time in a file named gzread.c. We then assign the string s to precisely one of the matching packages (which therefore sees its score increase by $\text{length}(s)$).

While any of the matching packages will do — since they all contain the cloned code, they are equally likely to be true positives — we attempt to minimise the number of packages in the resulting ranking for the user's convenience, using the

[3]The string literal that occurs in the most packages is the printf format string %s (3495 packages). The most common strings consisting of a single word are version (1749 packages), name (1682), help (1510) and unknown (1437). The most common strings consisting of a sentence are Out of memory (586) and out of memory (580).

Score	Package	# Unique
114573.95	subversion	2901
61089.06	libxml2	44
45354.69	openssl	2066
43808.14	db (sqlite)	381
6890.76	glibc	147
2013.99	*cadaver* (neon)	0
392.00	**wireshark**	6
342.01	*php* (sqlite)	2
340.94	*vtk* (sqlite)	0
256.74	zlib	0
253.75	**ruby**	13
179.23	*thunderbird* (expat)	0
174.27	**Subcommander**	2
117.01	apr	0

Table 1: Potential clones in svn ranked by evidence, as detected by scanning for strings

following iterative process. Given the strings that are the result of internal cloning, we compute the potential score increase that each package p would receive if every string that matches p were to be assigned to it. However, we do not give a score increase to some p if there is a p' that has more *unique* string matches (i.e., strings that occur exclusively in p'); the idea is that there is exclusive evidence for cloning of p' that p lacks, so we do not need to bother the user with p. We then pick the package p with the highest potential score increase and assign all matching strings to it, which are then removed. This process is repeated until no strings are left.

If there are multiple packages with the same potential increase (or within 10% of the winner), we attempt to eliminate potential internal clones using the following simple heuristic: if, given packages p_1 and p_2, p_1 contains fewer unique strings than p_2, then we assume that p_2 contains a clone of p_1, and we eliminate p_2 from consideration. For instance, the package zlib has only 760 strings in total against gtkwave's 11,731 (of which 245 are shared), so we conclude that gtkwave contains a clone of zlib, rather than the other way around.

Example. Table 1 shows the ranking produced by BAT's string analysis on svn, a statically linked binary (6,967,056 bytes large) of the Subversion source revision control system, against the string database containing the four Fedora releases. (As we discuss in Section 7, we use statically linked binaries for evaluation because the actual code clones are known.) The table lists the total score for the package — recall that this essentially measures the number of matching characters — and the number of strings that are unique to the package. We use a cut-off value of 100 for the score, which produces good recall while maintaining acceptable precision and was determined by running BAT on a set of known binaries.

The bold-faced package names denote false positives. In the case of wireshark and ruby, this was because those packages in the repository contain string literals equal to the names of *symbols* in packages actually used by the binary. For instance, wireshark contains the string literal originatorSignatureValue, which is also the name of a variable in openssl. The Subcommander package was detected because it contains strings copied from the version of Subversion used to build our binary. These strings also occur in the Subversion source tree, but as multi-line string literals that are not yet handled by our extractor.

Package names in italics are instances of internal cloning within the repository that are not correctly resolved to the "original" package (with the name of the original package in parentheses). For instance, svn is linked against sqlite, but BAT assigns these strings to the php and vtk packages, which contain copies of sqlite. Also, while db (Berkeley DB) is used directly by svn, its source tree also contains a copy of sqlite. We consider these packages true positives, since they do contain the linked code, but they are suboptimal to the user.

All libraries used to link the svn binary are represented in Table 1, a 100% recall rate. (This follows from our knowledge of the linker invocation that created the svn binary.) The precision is 79%.

5. DETECTION USING COMPRESSION

The second generic scanner in BAT attempts to detect clones using data compression. Data compression has been used in many clustering and classification tasks, such as to find similarity between natural languages, animal genomes, works of literature and music [4], as well as to classify files as malware (such as viruses) automatically [2].

The idea is that if the concatenation of the subject binary (such as a firmware) and a binary from a repository of precompiled open source packages compresses better than those two files individually, then this is evidence of redundancy between the two, and therefore cloning. That is, given two binaries x and y and a compression function C, if $|C(xy)|$ is substantially smaller than $|C(x)| + |C(y)|$ (where xy denotes the concatenation of files x and y, and $|s|$ is the length of a string in bytes), then there likely is cloning. So if we want to discover clones in some binary x, we compute the compressibility of xy for all binaries y (typically, static or dynamic libraries) in a repository of pre-compiled open source packages. The obvious downside of this approach is that it is architecture-dependent and requires the packages in the repository to be built from source. However, pre-built binaries of all common open source packages for many architectures, including the ubiquitous MIPS and ARM, are readily available from the repositories of Linux distributions such as Debian GNU/Linux.

So how do we define "substantially smaller"? When using data compression to find correlations between files, it is customary to use the *normalised compression distance* [4] as a metric, defined as

$$\mathrm{ncd}(x,y) = \frac{|C(xy)| - \min(|C(x)|\,,\ |C(y)|)}{\max(|C(x)|\,,\ |C(y)|)}$$

However, this metric is inappropriate here because we are not measuring the *similarity* between binaries x and y, but rather how much of y appears in x. (Consider the case where x contains all of y, but is also much larger. In that case, the NCD will be quite low.) Therefore, BAT uses essentially the following asymmetric metric:

$$\mathrm{reuse}_c(x,y) = \frac{|C(x)| + |C(y)| - |C(xy)|}{|C(y)|}$$

This measures the "improvement" of compressing the concatenation over compressing the files separately, expressed

$\text{reuse}_c(\mathsf{svn}, p)$	Package p
0.945	libsqlite3.a
0.899	libexpat.a
0.868	libdb.a
0.842	**libdb_cxx.a**
0.839	libz.a
0.823	libxml2.a
0.772	libneon.a
0.765	libapr-1.a
0.694	libcrypto.a
0.675	libssl.a
0.441	libpthread.a
0.435	libc.a
0.314	libaprutil-1.a
0.255	libm.a

Table 2: Potential clones in svn as detected using compression

as a fraction of the size of the compression of y. Values close to 1 indicate much cloning, i.e., most of y appears in x, while values close to 0 indicate little or no cloning. (Due to infelicities in the compression algorithm, the value can be slightly below 0 or above 1.)

For example, suppose that we want to discover whether the binary svn contains part or all of the library libsqlite3.a. Using the command xz -9 (an implementation of the Lempel-Ziv-Markov chain algorithm) as the compression function, we obtain $|C(\mathsf{svn})| = 2{,}563{,}804$, $|C(\mathsf{libsqlite3.a})| = 252{,}872$, and $|C(\mathsf{svn\ libsqlite3.a})| = 2{,}576{,}616$. Thus, the compression of the concatenation is 240,060 bytes shorter than the concatenation of the individually compressed files. Then $\text{reuse}_c(\mathsf{svn}, \mathsf{libsqlite.a}) = 240{,}060/252{,}872 \approx .95$. This indicates that svn almost certainly contains a clone of libsqlite.a.

The catch in the metric is that it may not work very well if y is very small (e.g., a library of a few kilobytes). Then the compression of y may not be able to exploit much redundancy, while the compression of the concatenation $C(xy)$ can reconstruct most of y from the code in the x. This effect is somewhat mitigated by the fact that code tends to be unique [10]. Nevertheless, during the evaluation (Section 7) we found it necessary to account for this effect by giving small binaries a penalty:

$$\text{reuse}_c(x, y) = \frac{|C(x)| + |C(y)| - |C(xy)| - \beta}{|C(y)|}$$

where β is the minimum number of bytes with which $C(xy)$ should beat $C(x) + C(y)$ to register any reuse.

A crucial consideration in the selection of the compression function C is that the size of its sliding window must be large enough to hold both files at the same time. Otherwise, the first file is no longer completely visible when the compressor reaches the second file, and it cannot exploit redundancy between the two. For instance, the compressor bzip2 fails this criterion, because even at the highest compression settings its window is limited to 900 KiB. By contrast, xz -9 uses a window size of several hundred megabytes, which is sufficient for the binaries in our evaluation.

Table 2 shows the ranking produced by BAT's compression analysis on the svn binary, ordered by descending reuse_c value, with $\beta = 1000$. The corpus of binaries against which the binary is compared is a set of 57 static libraries, including those used to build svn. The table shows all matches above the cut-off value of 0.1. (Again, this threshold was determined from a set of binaries as to yield good recall and acceptable precision.) All non-clones have values between -0.402 and 0.08. The only false positive is libdb_cxx.a, which is nearly identical to libdb.a but for the addition of some C++ wrappers. Total computation time took 148 seconds on an Intel Xeon E5620 with 32 GiB RAM.

Of course, detecting the presence of the very same libraries used to build a program is not very exciting, and the high recall and precision are not surprising. In Section 7, we use a corpus of libraries from a different Linux distribution to see whether the technique works across different versions of dependencies and compilers.

6. DETECTION USING BINARY DELTAS

The third and final generic scanner in BAT is essentially a variant of the data compression method: it detects code cloning by computing *binary deltas*. To detect whether a binary x contains a clone of a binary y from our binary repository, we compute the delta (or *diff* or *patch*) from x to y; that is, a list of instructions such as byte copy operations that reconstructs y from x. If this delta d is sufficiently smaller than y itself, then this is because large parts of y appear in x, and indicates cloning.

This method has the same limitations as compression: it requires a repository of pre-compiled binaries to check against, is architecture-dependent, and is significantly more computationally intensive than string scanning.

We use the following metric to measure potential cloning of a binary y in a binary x, as identified by binary delta computation:

$$\text{reuse}_d(x, y) = 1 - \frac{|D(x, y)|}{|D(\epsilon, y)|}$$

where $D(x, y)$ is a function that generates a binary delta between x and y, and ϵ denotes the empty (0-byte) string. That is, it measures how much better the patch generator performs given x than when given an empty string, with the limit 1 denoting much apparent cloning (because y can be completely reconstructed out of pieces of x), and 0 presumably indicating little cloning (because the contents of x do not give the generator anything more to "work with" than ϵ). Note that we must compare $|D(x, y)|$ to $|D(\epsilon, y)|$ rather than $|y|$ because delta generators typically apply a compression step to the list of delta operations. An optimal patch generator should not produce values below 0 (that is, perform worse given x than given ϵ as the base file), but practical generators often do.

For example, to see whether the binary svn contains part or all of the library libsqlite3.a, and using bsdiff (discussed below) as the delta generator, we find that $|D(\mathsf{svn}, \mathsf{libsqlite3.a})| = 26{,}130$ and $|D(\epsilon, \mathsf{libsqlite3.a})| = 261{,}138$. We therefore conclude that svn is likely to contain part of libsqlite3.a, since $\text{reuse}_d(\mathsf{svn}, \mathsf{libsqlite3.a}) \approx 0.90$.

There are many binary delta algorithms, such as xdelta (an implementation of the VCDIFF format defined in RFC 3284 [16]) and bsdiff [19]. The xdelta program computes a sequence of three different operations to generate the target file: adding a sequence of bytes specified in the patch, copying a sequence of bytes from the base file, and repeating a byte constant a number of times. This algorithm performs well on inputs such as text, but not so well on executable

$\text{reuse}_d(\text{svn}, p)$	Package p
0.900	libsqlite3.a
0.854	libexpat.a
0.821	libz.a
0.750	libdb.a
0.714	**libdb_cxx.a**
0.684	libxml2.a
0.615	libapr-1.a
0.572	libneon.a
0.479	libcrypto.a
0.455	libssl.a
0.281	libc.a
0.176	libm.a
0.137	libpthread.a
0.077	libaprutil-1.a

Table 3: Potential clones in svn as detected using binary deltas

code. This is because minor changes to the source code of a program or its build environment can cause numerous changes in the resulting binary: for instance, the addition of a single statement can cause all subsequent offsets in the program to change. Because BAT compares binaries that may differ in versions, applied patches, compilers used, and so on, we would expect this algorithm to perform poorly for code clone detection.

The bsdiff delta generator, on the other hand, deals with this variation in executable code by allowing for slight difference between byte sequences in the base and target files. In essence, it searches for sequences of bytes that are equal between the base and the target, and then "grows" these regions by adding bytes from the base to the start or end of the sequence so long as half the bytes are equal. It then stores the arithmetic difference between the bytes in the base and the corresponding bytes in the target, e.g., $(0, 0, 0, 6, 0, 5)$. A least half these bytes will be 0, and the remaining bytes are often small and highly repetitive due to the nature of changes such as offset shifts in executables. The resulting sequences are therefore compressed using bzip2.

Table 3 shows the potential clones discovered in svn by this method using bsdiff as the delta generator, on the same corpus as in Section 5. The results are similar to those for the compression method in Table 2. Delta computation took 204 seconds. The cut-off value used was 0.01; most non-clones have values between -0.48 and -0.01.

7. EVALUATION

In this section we describe the results of a preliminary investigation of the effectiveness of the three clone detection methods described above. The evaluation seeks to answer the following research questions:

- **RQ1:** *Is the string search method effective in identifying code cloning in binaries?*
- **RQ2:** *Is the compression method effective in identifying code cloning in binaries?*
- **RQ3:** *Is the binary delta method effective in identifying code cloning in binaries?*

To answer these questions, it is necessary to apply the clone detection methods to binaries to determine the *precision* — the fraction of the reported clones that are actually correct — and the *recall* — the fraction of the actual clones that are reported. Given an opaque third-party binary such as a firmware, it is usually not hard (just tedious) for an experienced engineer to determine whether reported clones are correct, by manual inspection of the binary and the evidence for cloning. However, due to the closed-source nature of such binaries, we do not have reliable information about the actual clones used to produce the binary. Thus, we can determine precision but not recall.

Binary	tp	fp	fn	Recall	Precision
busybox	6	2	0	1.00	0.75
ffmpeg	6	3	6	0.50	0.67
vim	3	0	0	1.00	1.00
qgit	13	9	8	0.62	0.59
yasm	4	0	0	1.00	1.00
speexenc	2	0	1	0.67	1.00
pdftops	5	0	0	1.00	1.00
gd2topng	4	0	2	0.67	1.00
w3m	4	0	1	0.80	1.00
openssh	4	4	0	1.00	0.50
Average				0.83	0.85

Table 4: Precision and recall of the string method on the known binaries

Clone Detection in Known Binaries.

To address this problem, we apply the clone detectors to a set of binaries with known instances of code cloning. Specifically, we created a set of statically linked binaries of common open source packages. From the invocation of the linker in the build log of the binary, we can determine all libraries used in the binary. To build the static binaries, we used the Nix package manager [8], which has a declarative build language that makes it easy to build variants of compositions of software packages, such as static builds or builds that use a different C compiler. We can then determine precision and recall in a variety of scenarios.

To address RQ1, we show in Table 4 the result of applying the string scanner to a set of 10 statically linked binaries, using the strings database created from the four Fedora releases described in Section 4. Analysing *why* false positives or negatives occured leads to the following lessons:

- It is important, in future work, to improve detection of internal code cloning in the repository, because many false positives and negatives are caused by misattribution of strings due to internal cloning. For instance, if the string analysis fails to detect internal cloning, then a string can drop below the reporting threshold.
- Many false positives are due to symbols in the binary matching string literals in other packages, as we already noted in Section 4. In the future, we may mitigate this effect by 1) stripping all relocation symbols off ELF executables; and 2) mining the repository for symbols. Symbols in themselves are not useful for clone detection in binaries, because they cannot distinguish between a definition and a use of a function or variable. But we can use knowledge of their existence to disregard the intersection of string literals and symbol names in the string analysis.

Binary	tp	fp	fn	Recall	Precision
busybox	5	0	0	1.00	1.00
ffmpeg	9	0	7	0.56	1.00
vim	4	0	2	0.67	1.00
qgit	23	9	6	0.79	0.72
yasm	2	0	2	0.50	1.00
speexenc	5	0	1	0.83	1.00
pdftops	4	0	2	0.67	1.00
gd2topng	6	0	3	0.67	1.00
w3m	6	9	3	0.67	0.40
openssh	6	0	1	0.86	1.00
Average				0.72	0.91

Table 5: Precision and recall of the compression method on the known binaries

Binary	tp	fp	fn	Recall	Precision
busybox	4	0	0	1.00	1.00
ffmpeg	8	0	7	0.53	1.00
vim	3	0	2	0.60	1.00
qgit	15	9	13	0.54	0.62
yasm	1	0	2	0.33	1.00
speexenc	4	0	1	0.80	1.00
pdftops	3	0	1	0.75	1.00
gd2topng	5	0	3	0.62	1.00
w3m	3	10	4	0.43	0.23
openssh	4	0	1	0.80	1.00
Average				0.64	0.89

Table 6: Precision and recall of the binary delta method on the known binaries

- The false negatives in qgit and other graphical packages are mostly small libraries that contain very few or no string literals, such as libjpeg and libXcursor. Obviously, such packages cannot be detected by the string scanner.

Nevertheless, the precision and recall of this method are promising.

To answer RQ2 and RQ3, we run the compression and binary delta scanner on the 10 statically linked binaries. To do so, we compared those binaries against a set of 134 libraries extracted from the Debian 6.0 "Squeeze" prerelease (Jan. 2011). Squeeze is based on GCC 4.4, while the static binaries were built with GCC 4.5. Therefore, this analysis should give some insight into how quickly repositories "decay" over time — that is, whether binaries produced several years apart (i.e., with different versions of compilers, upgraded source trees, and so on) are still comparable.

We did not use all libraries or binaries from these Debian releases because that would be prohibitively expensive for this evaluation. The selected binaries were chosen to include at least the dependencies of our static binaries. This obviously risks introducing a selection bias; however, since the number of dependencies for each static binary is small compared to the total number of libraries in the repository, we believe that this effect is at least somewhat limited.

Table 5 shows the results of running the compression-based scanner on the static binaries against Debian Squeeze. Table 6 shows the same for the binary delta scanner. Note that because the string method produces matches against packages (e.g., glibc), while the binary methods produce matches against files (e.g., libc.a and libm.a, which both come from Glibc), the number of matches is different from Table 4. Also, there may be duplicate matches: for instance, most programs will match against both libc.a and libc.so. This can skew the number of true positive.

There are a number of interesting points here:

- The xz-based compression method yields better results than the bsdiff-based delta method. Thus, while questions RQ2 and RQ3 can be answered in the affirmative, there is no good reason to use the delta method. The hypothesis that bsdiff's exploitation of the nature of differences between versions and variants of binaries would produce a better result than "direct" compression appears refuted.

- The high number of false positives in qgit and w3m is due to the internationalisation modules in /usr/lib/gconv (such as libJIS.so and libGB.so), which have high similarity and apparently match parts of the code in the static binaries.

Binary	Type	Size (MiB)	tp	fp	Precision
Vodafone Webby	Firmware	29	42	46	0.48
Asus WL500G	Firmware	2	26	12	0.68
Spotify	Core dump	344	27	61	0.31

Table 7: Precision of the string method on some third-party binaries

We have also compared the static binaries against a set of 135 libraries from Debian 5.0.8 "Lenny" (Feb. 2009), based on GCC 4.3. This leads to a significant reduction in the number of reported clones. For instance, for qgit the compression scanner produces 9 true positives and 9 false positives. The binary delta method yields a similar result. Thus, for large-scale use, it is important to have a binary repository that has sufficient coverage over time, especially since firmwares are often built with old toolchains.

Clone Detection in Opaque Binaries.

We applied the string-based clone detector to a set of third-party binaries, such as firmwares. For these, we can determine the precision but not the recall.

Table 7 shows the results of the string scanner on a small number of these binaries. They include firmwares and a core dump of an obfuscated executable (see Section 8). On large binaries, we find that the threshold of 100 for code clones to be displayed is too low: since the binary contains so many strings, even non-clones get a fairly high score due to non-unique matching strings. The top-ranked packages have much higher precision: for instance, the first 18 results for the Vodafone Webby firmware are all true positives.

We have not yet applied the compression and delta methods to large opaque binaries. In the future, we intend to load all binaries from all architectures of Debian into a repository and use it to identify cloning.

8. DISCUSSION

In this section we discuss various aspects and limitations of our work.

Usability. For an analysis tool such as BAT, it is important that it allows the user to investigate *why* it draws certain conclusions. Here the string method has a tremendous advantage over the binary methods: for each detected code clone, BAT presents the relevant evidence, i.e., the matching strings, the number of matches that are unique to the package, and so on. From a given string, the database allows the user to determine what other packages contain that string. All of this allows an experienced engineer to quickly determine whether matches are correct or false. (For instance, the overview of matching strings tends to immediately reveal packages that were matched due to symbol names rather than string literals.)

By contrast, the binary methods provide no such exploratory mechanisms. The reuse metrics are correlated with cloning, but users have no direct way to ascertain whether (say) a package with a score near the threshold indicates actual cloning.

Encrypted Binaries. Scanning of binaries is useless if a firmware is encrypted, unless decryption keys are provided. Encrypted firmwares are very common in for example Linux-based TVs. Nevertheless, the situation is not always hopeless. Programs that are encrypted or obfuscated can often be analysed by starting the program and then dumping its memory contents. For instance, we successfully applied BAT to a core dump of a running *Spotify* instance (a client for a music service whose binary is obfuscated). This works because BAT's methods are agnostic with respect to the format of the input file. However, the core dump may contain dynamically linked or loaded libraries that were not part of the binary's distribution (and therefore not necessarily a license issue). Thus the user must manually verify that any detected code cloning does not originate from those libraries.

Since in the string comparison technique we are comparing strings, BAT is easily defeated by string obfuscation. However, this does not appear to occur in practice.

Unpacking Firmware. There are file systems and compression methods that don't have standard "magic" markers, such as YAFFS2, or some instances of LZMA. It is hard for BAT to find these file systems and files and extra information has to be provided to the system before BAT can unpack them.

There are also situations where file systems and compression methods have been slightly altered by the vendor so that files cannot be unpacked with the standard unpacking tools. However, unless compression is involved, the generic scanners should still be able to produce usable results on such files.

Granularity. BAT detects code cloning at a high level of granularity, e.g., that a firmware contains the Linux kernel. While this is useful to know, it is not necessarily sufficient. For instance, if the upstream supplier applied a patch to the Linux kernel, then the patched source code must be made available to customers. BAT however cannot determine whether the source code provided by the supplier matches the binary.

Pre-flashed Firmware. BAT will only be able to scan what is in a binary file, but this does not necessarily reflect what is actually shipped in the device. Often a firmware update only partially updates contents on a flash chip, but leaves other parts alone. A good example is the bootloader, which is commonly pre-flashed on a chip and not updated by newer firmwares. This could still present a legal risk: RedBoot and u-boot are two GPL licensed bootloaders that are used on many devices and for which sources are often missing from GPL source code archives. To avoid this it would be best to let BAT scan dumps of the contents of the flash chip instead of firmware update files.

Threats to Validity. The main *external* threat to the validity of our evaluation is that some binaries may not allow deconstruction to a level where the generic scanners can operate, i.e., where strings and machine-language structures become visible. Thus, as discussed above, our work cannot be applied to binaries that are encrypted or use unsupported compression layers. *Internal* validity is primarily at risk from the limited size of the evaluation, from our manual inspection to classify the results of the scans as false or true positives, and from selection bias in the evaluation of the compression and delta methods – against a larger repository of binaries, the false positive rate may well increase.

Binary Clone Detection. In order to evaluate the applicability of binary clone detection for BAT, we implemented the Sæbjørnsen binary code clone detector algorithm [20]. This revealed some limitations for our use-case. Ideally, we want to extract the feature vectors and use them as fingerprints stored in a database, like we do for strings. The algorithm as presented does not immediately allow this. Its final filtering steps require access to (a subset of) the original instructions and function boundaries. Thus, with the unmodified algorithm, we must keep around a corpus of binary libraries that new candidate executables are run against, as we do for e.g. binary deltas. The algorithm's strength – it finds all possible code clone pairs – presented a challenge for us: for the purposes of license violation detection, we are not interested in self-clones inside a given binary, so we only extract the clusters where regions from more than one binary are present. Another potential pitfall is the (expected) tendency of the algorithm to find more clones in bigger executables. We tried to compensate for this by focusing on a given percentile of the largest clone pairs, with the idea that large clone pairs across two different binaries are more likely to represent actual code clones. Despite these modifications, we did not obtain consistent results comparable to the other techniques presented in this paper. We still think the algorithm holds promise, but further study is needed to tune and extend it for license violation detection.

9. CONCLUSION

In this paper, we motivated the need for tools to detect code cloning in binaries such as firmwares. We showed three methods to detect cloning in binaries, and discussed our initial experiences with these methods as implemented on top of the Binary Analysis Tool. This experience suggests that our techniques are feasible. However, more work is needed to improve the quality of BAT's results, in particular to mitigate the effects of internal cloning in the source code repository. We also intend to evaluate the compression and delta methods on a much larger repository.

Availability. The Binary Analysis Tool is available under a free software license at http://www.binaryanalysis.org/.

10. ACKNOWLEDGMENTS

The Binary Analysis Tool was created by Loohuis Consulting and Opendawn. Initial development was sponsored by NLnet Foundation, and further development has been sponsored by Linux Foundation and NLnet Foundation. This research is partially supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*, as well as the NIRICT LaQuSo Build Farm project.

11. REFERENCES

[1] E. Andersen et al. BusyBox hall of shame. http://busybox.net/shame.html, 2008.

[2] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of Internet malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID'07)*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–95, Washington, DC, USA, 1995. IEEE Computer Society.

[4] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4), Apr. 2005.

[5] I. J. Davis and M. W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. pages 242–246, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[6] M. Di Penta, D. German, and G. Antoniol. Identifying licensing of jar archives using a code-search approach. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 151–160, May 2010.

[7] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 145–154, New York, NY, USA, 2010. ACM.

[8] 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)*, pages 583–592. IEEE Computer Society, May 2004.

[9] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:37–58, Jan. 2006.

[10] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 147–156, New York, NY, USA, 2010. ACM.

[11] D. M. German, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 81–90, Washington, DC, USA, 2009. IEEE Computer Society.

[12] 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, pages 437–446, New York, NY, USA, 2010. ACM.

[13] A. Hemel. The GPL compliance engineering guide. http://www.loohuis-consulting.nl/downloads/compliance-manual.pdf, 2008–2010.

[14] L. Hewlett-Packard Development Company. FOSSology. http://www.fossology.org/, 2009–2011.

[15] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, Dec. 2008.

[16] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF generic differencing and compression data format. RFC 3284, June 2002.

[17] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

[18] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society.

[19] C. Percival. Naive differences of executable code. http://www.daemonology.net/bsdiff/, 2003.

[20] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[21] Software Freedom Law Center. Best Buy, Samsung, Westinghouse, and eleven other brands named in SFLC lawsuit. http://www.softwarefreedom.org/news/2009/dec/14/busybox-gpl-lawsuit/, 2009.

[22] A. Vance. The defenders of free software. http://www.nytimes.com/2010/09/26/business/26ping.html, 2010.

[23] H. Welte. Free and open source software license compliance. http://www.openfoundry.org/en/workshop/details/115, 2010.

[24] H. Welte and A. Hemel. The gpl-violations.org project. http://gpl-violations.org/, 2005–2011.