Noname manuscript No.
(will be inserted by the editor)

The Debsources Dataset: Two Decades of Free and Open Source Software

Matthieu Caneill · Daniel M. Germán · Stefano Zacchiroli

Received: date / Accepted: date

Abstract We present the Debsources Dataset: source code and related metadata spanning two decades of Free and Open Source Software (FOSS) history, seen through the lens of the Debian distribution.

The dataset spans more than 3 billion lines of source code as well as metadata about them such as: size metrics (lines of code, disk usage), developer-defined symbols (ctags), file-level checksums (SHA1, SHA256, TLSH), file media types (MIME), release information (which version of which package containing which source code files has been released when), and license information (GPL, BSD, etc).

The Debsources Dataset comes as a set of tarballs containing deduplicated unique source code files organized by their SHA1 checksums (the source code), plus a portable PostgreSQL database dump (the metadata).

A case study is run to show how the Debsources Dataset can be used to easily and efficiently instrument very long-term analyses of the evolution of Debian from various angles (size, granularity, licensing, etc.), getting a grasp of major FOSS trends of the past two decades.

The Debsources Dataset is Open Data, released under the terms of the CC BY-SA 4.0 license, and available for download from Zenodo with DOI reference 10.5281/zenodo.61089.

This work has been partially performed at IRILL, center for Free Software Research and Innovation in Paris, France http://www.irill.org. Unless noted otherwise, all URLs in the text have been retrieved on September 1st, 2016. Authors are listed alphabetically.

M. Caneill
Université Grenoble Alpes, Grenoble, France, E-mail: caneill@imag.fr

D. Germán
University of Victoria, Victoria, Canada, E-mail: dmg@uvic.ca

S. Zacchiroli
Univ Paris Diderot, Sorbonne Paris Cité, IRIF, UMR 8243, CNRS, F-75205 Paris, France, and Inria, France, E-mail: zack@pps.univ-paris-diderot.fr

1 Introduction

Software is increasingly being distributed to final users by the means of software collections and deployed using package management tools. Some software collections are very tightly curated and integrated, like Free and Open Source Software (FOSS) *distributions*, others much more loosely so, like so called "app stores". The study of software evolution [19, 16] can no longer ignore software collections as relevant subjects of macro-level studies [11, 3], i.e., evolution studies conducted at the granularity of component releases rather than individual commits.

The study of software collections however poses specific challenges to scholars due to a common tendency at creating *ad hoc* software ecosystems made of homegrown tools, technical conventions, and social norms that might be hard to take into account when conducting empirical studies.

The Debsources platform [3] has been developed to counter those challenges in the specific case of Debian[1]—a general purpose FOSS operating system for desktop and server computers, which is one of the most reputed and oldest (est. 1993) distributions, often credited as the largest curated collection of FOSS components. Thanks to the free availability of both its source code and associated metadata, its conspicuous size, and its standardized package layout [13], Debian has become a popular subject of empirical software engineering studies [4] (see, among many others, [25, 11, 1]).

Debsources allows to gather, index, search, and publish on the Web the entire source code of Debian and metadata extracted from it. The most notable instance of Debsources is publicly available at `http://sources.debian.net` and indexes all currently active Debian releases, with several updates per day, as well as historical Debian releases going back almost 20 years.

Contributions In this paper we present the Debsources Dataset, a polished version of the data underpinning `http://sources.debian.net` suitable for a wide range of large-scale analyses on FOSS components released by Debian. The dataset is composed of two parts that can be used together or independently:

1. **Source code**. The dataset includes the source code of 10 Debian stable releases published over the past 2 decades, corresponding to 82 thousand packages for more than 30 million source code files. To reduce storage size, source code files have been deduplicated and organized in a manner that facilitates and speeds up empirical studies. The result of deduplication is 15 million unique files, requiring ≈ 320GB of disk space. After compression with `xz` the source code part of the dataset shrinks down to ≈ 90GB.
2. **Metadata**. Rich metadata regarding all shipped source code are also part of the dataset. Release metadata link together the 10 Debian releases, the packages that compose each of them, and the source code files that form

[1] `https://www.debian.org`

each package. In addition to release information, the dataset also contains the following content-oriented metadata:

- per-file size metrics including file size in bytes, number of lines (using `wc`), source lines of code (SLOC) divided by language, computed using both `sloccount`,[2] and `cloc`;[3]
- checksums: cryptographic hashes SHA1 and SHA256, as well as locality-sensitive TLSH [18] hashes of all files;
- MIME media type of each file, as detected by `file`[4];
- location, name, and type of developer-defined symbols (functions, data types, classes, methods, etc.) obtained indexing all source code with Exuberant Ctags;[5]
- applicable FOSS license for individual files, as detected by both `ninka` [8] and `fossology` [9].

Source code is shipped as a set of tarballs, metadata as a PostgreSQL[6] database dump.

Exploitation ideas The Debsources Dataset is a valuable resource for scholars interested in studying either the composition or the long-term evolution of FOSS. Here are a few ideas—some already realized, some still up for grabs—on how to exploit the dataset:

- Conduct or replicate long-term, macro-level *evolution studies* of FOSS. We show an example of this in Section 5.
  Several followup research questions remains unanswered, e.g.: does the use of different programming languages evolve in similar ways along the history of development? 20 years are enough to observe the raise and fall of programming languages and try to spot interesting adoption patterns. Using the Debsources Dataset those studies can be done both in aggregate ways (e.g., how many software projects are written in a given language over time?) and at per-project level (e.g., do all software projects written in a given language follow similar evolution patterns?).
- Study the structure and evolution of *license use* in FOSS, at different granularities: file, package, distribution. We address some of these in Section 5, but a lot remains to be done, most notably in the area of licensing of software components as aggregate wholes.
- Investigate *code reuse and cloning* along the whole history of all software packages contained in the dataset. Reuse without modification is trivial to track thanks to SHA1 and SHA256 checksums. Reuse *with* modification can be supported using ctags and/or TLSH hashes as fingerprinting techniques to track (modified) code copies, or by directly parsing the actual source code available in the dataset.

[2] http://www.dwheeler.com/sloccount/
[3] https://github.com/AlDanial/cloc
[4] http://www.darwinsys.com/file/
[5] http://ctags.sourceforge.net/
[6] http://www.postgresql.org

– The availability of source code can be further leveraged to support several kinds of static analysis studies. By focusing on source code files written in a specific programming language (e.g., C, C++), researchers can study the evolution over time of bugs that are detectable with a given static analysis tool (e.g., Coccinelle, Coverity).
  On the more practical side, the source code in the dataset also forms an interesting benchmark for *code search* at a scale. Multi-language, license-aware, automatic code completion backed by the Debsources Dataset would make for a very fun and useful toy for many developers.
– In comparison with other sub-fields, *release engineering* [2] is still relatively unexplored in empirical software engineering. The Debsources Dataset allows to follow the evolution of package-level structures along 20 years of Debian, and to mix-and-match with release metadata, metrics, and actual source code. Some open research questions in this area are: when and how software projects get split into multiple packages? does package organization change over time? does that affect release schedules? how do packages migrate from one development release to another? etc.

Paper structure Section 2 explains how the Debsources Dataset has been assembled. Section 3 describes the data schema and gives some statistics about its content. Section 4 shows how to get started using the dataset. Sections 5 presents a case study on how the dataset can be used to conduct a long-term, macro-level evolution study of FOSS from several angles using Debian as a proxy. Section 6 discusses the limitations of the dataset. Before concluding, Section 7 points to related work.

Dataset availability The Debsources Dataset is Open Data. The metadata part of the dataset is available under the terms of the Creative Commons Attribution-ShareAlike (CC BY-SA) license, version 4.0; the source code part of the dataset is available under the terms of the applicable FOSS licenses. The dataset is available for download from Zenodo[7] at `https://zenodo.org/record/61089`, with DOI reference `10.5281/zenodo.61089`.

2 Data gathering

The Debsources Dataset has been assembled by mirroring and extracting Debian source releases, organizing extracted source code to remove duplicates, running analysis tools over the obtained files, and injecting their results in a PostgreSQL database.

Given that both all Debian releases and the analysis tools we have used are freely available as FOSS, the dataset can be recreated from scratch following the blueprint given below. Be warned though that (re-)creating the dataset takes a significant amount of resources, both in terms of processing time and

[7] `https://zenodo.org/`

required disk space; details are given below. Also note that to simply *use* the Debsources Dataset you do *not* need to go through the process below, which is documented here for information and reproducibility purposes only. The dataset is ready to use as-is; see Section 4 for a quick start guide.

2.1 Blueprint

Database structure

0. As a preliminary step we have created the database structure. This can be achieved by replaying in a freshly created database the schema creation SQL statements that can be found in the Debsources Dataset database dump. The database content has then been filled as we went during the process described below.

Mirror current and historical Debian releases

1. *Current releases.* We have used debmirror[8] to retrieve all current Debian releases from a nearby mirror.[9] Binary packages can be ignored, and one can easily tune debmirror to only download source packages.
2. *Historical releases.* We have used `rsync` [23] to mirror `http://archive.debian.org`. This step is required to retrieve historical Debian releases that are no longer available from the regular mirror network.
3. *Releases metadata.* To load into the database release information we have used various sources of information. Each Debian release comes with a set of `Sources` files describing which packages/versions compose the release as well as other package metadata. We have parsed those files using the `python-debian` library[10] and stored the extracted information in the database.
   For the static information about each release (release name, date, etc.) we started from the list of Debian releases on Wikipedia,[11] double-checked with official Debian release announcements, and stored the obtained information in the database.

Source code extraction and deduplication

4. *Extract packages.* Debian source packages are formed by one or more tarballs and/or patch sets, along with a `.dsc` manifest file. We have looked for all such manifest files and extracted their content using `dpkg-source -x package_version.dsc` (dpkg-source is a Debian tool that can be found in the `dpkg-dev` package). Doing so merges upstream tarballs together

[8] `https://packages.debian.org/sid/debmirror`
[9] A list of Debian mirrors organized by geographical location is available at `https://www.debian.org/mirror/list`.
[10] `https://packages.debian.org/sid/python-debian`
[11] `https://en.wikipedia.org/wiki/List_of_Debian_releases`

and apply Debian-specific patches in the process. Thanks to the unicity of package ⟨name, version⟩ pairs, all packages can be extracted in the same directory; each one will be extracted in a separate sub-directory without conflicts.

5. *Deduplicate files.* Due to the presence of multiple versions of the same software, many source code files are exact copies of others. By deduplicating identical files we have cut down both disk usage and the number of files to process for any kind of batch analysis by a factor 2. To this end:
   - We have listed all regular files (e.g., with `find -type f`). This step excludes symbolic links, which can also result in processing multiple times the same files.
   - For each regular file, we have computed its SHA1 checksum and stored it in the database.
   - For each unique SHA1, we have created a file `XX/YY/SHA1`, whose content is identical to the original file and where `XXYY` are the first 4 characters of the checksum.
   - *Mirror deduplicated files to preserve extensions.* Some file analysis tools require file extensions to properly identify the file language, type, or format. For example, without a `.c` extension many C files will not be recognized as such by default by neither `cloc` nor `sloccount`. To facilitate the analysis with such tools we have created a directory tree with the same structure of the deduplicated tree above, but having extensionful symlinks as its leaves. Each symlink is named `XX/YY/SHA1.ext` and points to `XX/YY/SHA` in the deduplicated tree.

Compute content metadata

6. *Extension-agnostic metadata.* For each unique SHA1, we have retrieved the corresponding unique file and computed its size, media type (using the Unix command `file`), number of lines (using the Unix command `wc`), SHA256, and TLSH checksums.
7. *Extension-sensitive metadata.* For each unique ⟨SHA1, extension⟩ pair, we have ran `cloc`,[12] `sloccount`,[13] and exuberant `ctags`[14] on the corresponding extensionful symlink. Note that all these operations can also be performed in batch on several files at once. For instance, one can execute `ctags --recurse` on the outermost `00` SHA1 directory, to process all files whose SHA1 starts with `00` at once. The trade-off here is between the output size of each tool invocation and the number of invocations.

   All obtained metadata have been stored in the database.

Compute license information

8. *License detection.* We ran `fossology` and `ninka` on all unique files and stored the output of both license detection tools in the database.

[12] https://github.com/AlDanial/cloc
[13] http://www.dwheeler.com/sloccount/
[14] http://ctags.sourceforge.net/

2.2 Required resources

The dataset (re-)creation process is I/O-bound and might require up to 1.5TB of working disk space during processing.

About 200GB are needed to store the (compressed) source mirror of both current and historical Debian releases. After extraction, but before deduplication, 800GB of additional disk space are required to store the bulk of the extracted source code. Deduplication and release metadata extraction can then be run, resulting in extra 400GB between the new files and the working database. At this point the 1TB of disk space occupied by compressed and uncompressed source code can be freed.

Processing the deduplicated source code to compute checksums, SLOCs using `sloccount`, `ctags`, and disk usage took us about 10 days on a single server-grade machine with rather slow (by today standards) 7.2kRPM spinning disks. Computing SHA256, TLSH, media types, and SLOCs using `cloc` required approximately 1 week using an Apple Mac Pro and a Promise RAID. The process of running `fossology` and `ninka` on all source files took approximately 25 days on a virtual machine using the Western Canada Research Grid.[15]

In total, a realistic estimate for recreating from scratch the Debsources Dataset on a single machine equipped with fast SSD drives is in between 4 and 5 weeks; a couple of weeks more with spinning disks.

3 Bird's eye view

The Debsources Dataset comes in two major parts: a set of tar files containing the source code and a PostgreSQL database dump with metadata about source code files and their relationship to Debian packages and releases.

3.1 Source code

The first part of the Debsources Dataset is a set of tarballs containing the deduplicated source code files. They were divided into 16 tarball (each one weighting 5–6GB) to facilitate distribution. Each tarball, named `debsources.`*X*`.tar.xz` (where *X* is an hexadecimal digit: 0–9, a–f), includes all source code files whose SHA1 start with *X*. The files contained in these tarballs are further divided (or "sharded") in sub-directories based on the first 4 characters of their SHA1. For example, a file whose SHA1 is `deadbeef[...]` will have path `de/ad/deadbeef[...]` and can be found in tarball number `d`.

As discussed in Section 2, the additional tarball `debsources-ext.tar.xz` contains file extension information as a set of symlinks to the actual source code files.

[15] https://www.westgrid.ca/

Table 1 Various versions of the `xournal` package in Debian, all containing a file named `src/xo-interface.h` with the same SHA1 checksum.

Release	Package	Version	SHA1 of `src/xo-interface.h`
lenny	xournal	0.4.2.1-0.1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
squeeze	xournal	0.4.5-2	e09a07941a3c92140c994fcdda7f74bce1af4ca3
wheezy	xournal	0.4.6~pre20110721-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3
jessie	xournal	1:0.4.8-1	e09a07941a3c92140c994fcdda7f74bce1af4ca3

Consider file `src/xo-interface.h`, which can be found in four different versions of the `xournal` package in Debian. Its metadata are depicted in Table 1. After extraction (see Section 4) source code can be found in two top-level directories: `debsources` and `debsources.ext`. The first one contains the actual source code, the second extensionful symlinks to them. In our example we will have the following on-disk layout (names ending in '`/`' represent directories, and '`->`' a symlink and its destination):

```
debsources/
...
debsources/e0/
debsources/e0/9a/
debsources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
debsources.ext/
...
debsources.ext/e0/
debsources.ext/e0/9a/
debsources.ext/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3.h ->
   ../../../sources/e0/9a/e09a07941a3c92140c994fcdda7f74bce1af4ca3
...
```

Even though files are renamed to match their SHA1 checksums, the directory structure of individual packages is not lost. Full original paths are available as part of the metadata database described next. As a preview, the following SQL query can be used to reconstruct the paths at which a file with the SHA1 of our example can be found in the dataset, together with the corresponding package names and versions:

```
SELECT release_id as release,
       package_name as package, package_version as version,
       encode(path, 'escape') as path
FROM releases
NATURAL JOIN package_info
NATURAL JOIN paths
NATURAL JOIN path_info
NATURAL JOIN files
WHERE sha1='e09a07941a3c92140c994fcdda7f74bce1af4ca3'
```

When run the query will return the following tuples:

release	package	version	path
lenny	xournal	0.4.2.1-0.1	src/xo-interface.h
squeeze	xournal	0.4.5-2	src/xo-interface.h
wheezy	xournal	0.4.6~pre20110721-1	src/xo-interface.h
jessie	xournal	1:0.4.8-1	src/xo-interface.h

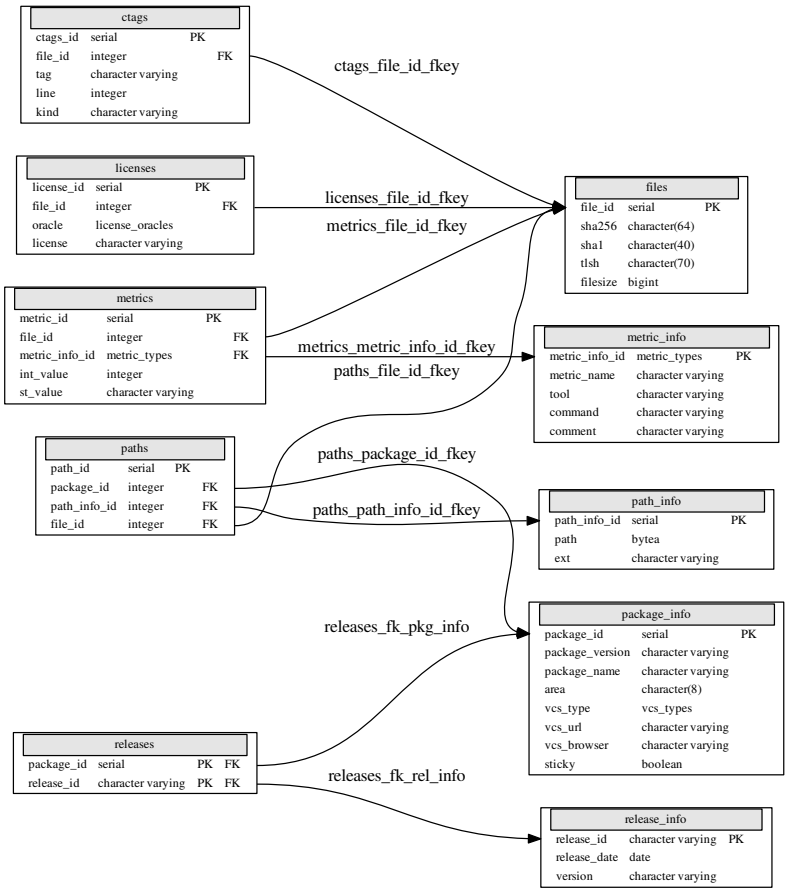

Fig. 1 Database schema. Primary key fields are denoted with "PK", foreign keys "FK"; arrows indicate referential integrity constraints.

3.2 Metadata

The second part of the dataset is a Postgres database, containing all source code metadata. The database schema is shown in Fig. 1. A brief description of each table is given below.

3.2.1 Intrinsic information

The following tables describe releases, packages, files and the relationships among them:

Table 2 Tools used to extract file-level metadata.

Tool	Version	Command
file	5.14	`file --mime-type`
cloc	1.66	`cloc --by-file --follow-links --skip-uniqueness \ --sql-append`
sloccount	2.26	`sloccount --duplicates --follow --details`
wc (coreutils)	8.13	`wc -l`

- *package_info:* information about Debian source packages contained in the dataset, such as package names, versions, and associated attributes (e.g., project homepage). Package names are generally lowercase variants of the original (or "upstream") FOSS project names, e.g., bash, linux (the kernel), libreoffice, etc. Package versions include both the upstream project version and the Debian package revision separated by a dash, e.g.: "1.2.3-4".
- *release_info:* information about the 10 Debian releases in the dataset; name, version, and release date of each one are included.
- *releases:* mappings between source packages and Debian releases.
- *path_info:* the full path name of every file in the dataset. The table also contains, as a separate field, the file extension. Note that, consistently with the POSIX standard, paths are stored as raw byte sequences; there is no guarantee that they can be interpreted as valid Unicode characters without further knowledge of the applicable character encoding. The vast majority of paths that are encoded in UTF-8 can be parsed at query time using suitable Postgres functions.
- *files:* deduplicated files together with their checksums (SHA1, SHA256, and TLSH) and size (in bytes). This table lists all unique files present in the dataset.
- *paths:* ternary mappings between *packages*, *path_info*, and *files*. This table indicates, for a given package and path, the corresponding unique file. For symbolic links, the *file_id* column will be NULL.

3.2.2 Derived information

The following tables describe information that have been extracted from Debian source code following the process described in Section 2.

- *licenses:* license information. This table maps unique files to the corresponding FOSS licenses as identified by the license detection tools (or "oracles") `ninka` and `fossology`.
- *metric_info:* information about the tools used to compute file-level information. For reproducibility reasons, this table includes tool name, version, command line used to run it, and a *comment* field with additional human-readable information. Due to how representative of the available file-level information this table is, its content is given in Table 2.
- *metrics:* links files to the information extracted from them. Some derived information are integer-valued (e.g., the output of `wc -l`), some string-valued (e.g., `file` output), some both (`cloc` and `sloccount` output both

Table 3 Size of Debsources Dataset metadata as a Postgres database. The entire database requires ≈ 40GB of disk space (including indexes, which are not listed below).

Table	Disk size	Tuples
ctags	23 GB	186.5M
files	5944 MB	15.5M
metrics	3549 MB	46.7M
paths	3259 MB	30.5M
licenses	2976 MB	31.0M
path_info	1895 MB	11.7M
package_info	14 MB	82113
releases	7248 KB	97471
metric_info	32 KB	4
release_info	32 KB	10

Table 4 Size of Debsources Dataset source code.

Tarball	Disk usage (compressed)	Disk usage (expanded)
`debsources.*.tar.xz`	89GB (total)	317GB
`debsources-ext.tar.xz`	422MB	61GB
`debsources.dump.xz`	3.1GB	see Table 3

detected language and number of SLOCs). For this reason this table allows to store an integer (field `int_value`) and/or a string (`st_value`). The attribute `comment` in table *metrics_info* documents which field is relevant for which metric.

– *ctags:* `ctags` results for each file. The table contains one entry for each developer-defined symbols in a given source file, together with the precise file location at which the symbol was found and the symbol type (function, data type, method, etc).

3.3 Dataset size

To give an idea of the size of the dataset, Table 3 lists the sizes of all tables in the database, as both number of tuples and required disk space. If space is at a premium, some large tables (e.g., *ctags*) can be deleted without compromising the referential integrity of the database. Similarly, Table 4 details the required disk space to locally host the source code part of the Debsources Dataset.

4 Getting started

This section describes the steps necessary to use the Debsources Dataset. The two parts—source code and metadata—can be used independently, but the metadata are needed if you want to be able to relate individual source code files to their context.

4.1 Metadata

The metadata part of the Debsources Dataset comes as a plain-text SQL dump of a PostgreSQL database, compressed in `xz` format. The dump has been obtained from Postgres 9.4 using `pg_dump`, but it should be compatible with any version of Postgres ≥ 9.1.

To import the metadata you should first install Postgres, then create a dedicated database (e.g., *debsources*), and finally import the dump into it. For the last two steps you can proceed as follows, acting as a user with suitable Postgres permissions:

```
1. createdb debsources
2. xzcat debsources.dump.xz | psql debsources
```

On a modern high-end laptop equipped with a fast SSD disk, the import takes about 1.5 hours. The freshly imported database will require about 40 GB of disk space (see Table 3 for details).

4.2 Source code

To decompress the source code you should first create a directory that will contain all of it and move into that directory. Then:

- (Optional) Expand the tarball with the extension symlinks:
  `tar xaf /path/to/compressed/dataset/debsources-ext.tar.xz`
- Create a sub-directory called `debsources` and move into it:
  ```
  1. mkdir debsources
  2. cd debsources
  ```
- Extract the actual source code. For each of the `debsources.`*`X`*`.tar.xz` tarballs, execute:
  `tar xaf /path/to/compressed/dataset/debsources.X.tar.xz`

The result will be a Debsources Dataset directory containing two sub-directories, `debsources` for extensionless deduplicated source code files sharded by SHA1, and `debsources.ext` with extensionful symbolic links pointing into it, as described in Section 3.1.

One advantage of the deduplicated files is that they are file system agnostic and can be expanded onto any file system. This is not the case for the original source code files. For example, these two paths `net/netfilter/xt_tcpmss.c` and `net/netfilter/xt_TCPMSS.c` exist in the Linux kernel. The two paths point to files with different content, but have file names that differ only in capitalization. This is supported by file systems such as ext4, but will cause a file name clash on JFS (case preserving, Mac OS) or NTFS (while the file system is case sensitive *per se*, case sensitivity depends on the application creating the files). The impact of these low-level issue can be significant: there are more than twenty such cases in the Linux kernel alone. By renaming the files to their SHA1 we avoid similar issues.

5 Case study: long-term macro-level evolution

In the following we show how the Debsources Dataset can be used to conduct a long-term, macro-level evolution analysis of FOSS projects, as they can be observed through the lens of the Debian distribution. We focus on aspects such as source code size (under various metrics), programming language popularity, package size, package maintenance, and software licensing.

The analyses we conduct are both qualitative and quantitative, and in part replicate and extend previous findings [11, 3]. The research questions we will address are:

RQ i. *How does the size of Debian evolve over time?* Looking at various metrics we will study how and at which rate Debian grows across releases.

RQ ii. *How much Debian changes between releases?* By studying package versions and their content, we can measure the amount of packages that are updated across Debian releases and to what extent they are.

RQ iii. *How has the popularity of programming languages changed over the last 20 years?* By looking at the evolution of SLOCs per language, we identify which languages are gaining (or losing) traction among FOSS projects represented in Debian.

RQ iv. *Which licenses apply to Debian source code files?* We identify which software licenses are used in Debian at a file-by-file granularity, irrespectively of the containing package.

RQ v. *Which licenses can be found in Debian source packages?* By aggregating file licenses by package we can study the expected license variability when reusing entire software packages.

RQ vi. *How has license use evolved in Debian over time?* We explore the evolution of license use over time by comparing the licensing of files and packages that belong to different Debian releases.

5.1 Growth over time

The evolution of Debian size over time (RQ i) can be studied under various metrics. We take into account the following ones: number of packages, number of source code files, disk usage of (uncompressed) source code, lines of code (SLOCs), and developer-defined symbols (or "*ctags*").

In the Debsources Dataset packages can be found in the *package_info* table, that has one row per package. Source code files can be found in table *paths*, which in turn points to unique files listed in table *files*. All file-level metrics, except ctags, are in table *metrics*, column *int_value*, distinguished by metric type (column *metric_info_id*).[16] File-level metrics can then be grouped by package following the *metrics* → *files* ↔ *paths* → *package_info* chain of

[16] Note that two different SLOC metrics are available in the dataset: as computed by `sloccount` and `cloc`. Each tool has its strength and weaknesses. For this case study we use `sloccount` numbers.

relationships. Ctags are stored in the separate *ctags* table because, whereas they can be used as a size/complexity metric for individual source code files, they primarily act as an index which doesn't fit the general model of the *metrics* table. Per-packages metrics can be further aggregated by release using the *releases* table. Per-release metrics can finally be sorted by time using the *release_date* field of the *release_info* table.

Table 5 Debian release sizes by various metrics—number of packages, files (and files explicitly recognized as source code by `sloccount`), disk usage of uncompressed source packages, lines of code, developer-defined symbols (ctags). See also Table 6 for additional statistics parameters about these measures.

Release	Version	Packages	Files (k)	Source files (k)	Disk usage (GB)	ctags (M)	SLOCs (M)
hamm	2.0	1373	348.4	152.5	4.1	4.1	34.9
slink	2.1	1880	484.6	224.4	6.0	6.2	51.9
potato	2.2	2962	686.0	292.6	8.6	7.4	68.8
woody	3.0	5583	1394.5	563.3	18.2	17.2	140.7
sarge	3.1	9050	2394.0	870.6	34.1	24.2	210.1
etch	4.0	10 550	2879.7	1092.7	45.0	30.3	272.1
lenny	5.0	12 517	3713.9	1437.2	61.8	38.3	332.7
squeeze	6.0	14 951	4908.1	1952.2	89.1	52.3	444.4
wheezy	7	17 564	7310.5	2751.4	131.7	69.5	636.8
jessie	8	21 041	8375.0	3404.2	167.0	95.6	784.3

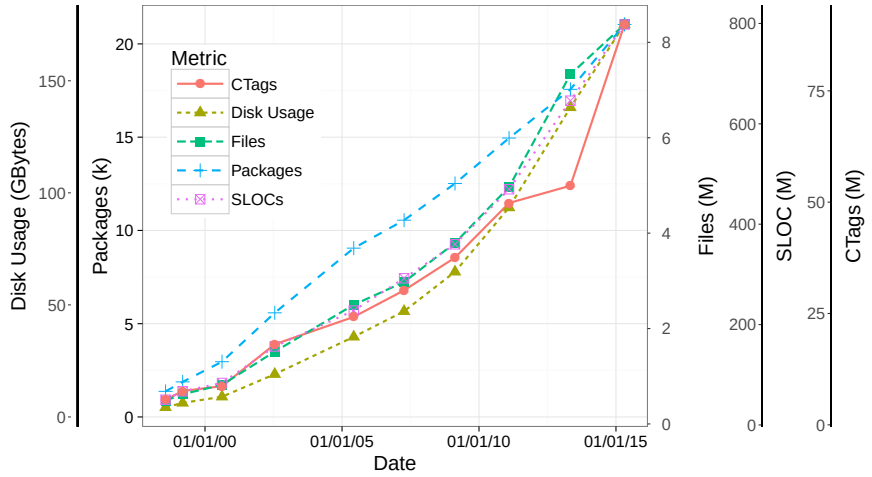

Fig. 2 Debian release size over time, under various metrics.

Release size The above query plan can easily be translated to SQL queries and run on the Debsources Dataset. Query results are shown in Table 5, and plotted in Fig. 2 over time.

In absolute terms, Debian has scaled to a point where the last stable release (Jessie) contains more than 21 thousand packages, and almost 800 millions lines of code. If we look at the metrics evolution over time we notice that the five considered metrics exhibit similar growth rates. Four of them (ctags, disk usage, files, and SLOCs) are very highly correlated and grow super-linearly, with an apparent slow down in the most recent stable release. The other metric (package count) is more regular and almost perfectly linear.

This discrepancy gives some insights about Debian technical management. Packages are the units at which software is maintained in Debian: each package is under the responsibility of a (group of) maintainer(s). A super-linear growth in the number of packages would need a super-linear growth in the number of maintainers to be sustainable in the long-term or, alternatively, an increase in the amount of packages maintained by the same people. While there is some evidence of the latter [20] on shorter time-frames (about a decade) than the one considered here (two decades), it also seems that Debian is focusing on sustainable size increases rather than trying to package every available FOSS product bearing the risk of stretching its forces too thin.

Table 6 Averages, median, and maximum of various size metrics, over packages and per release. See Table 5 for totals.

	Files		Disk usage (KB)		SLOCs		
Release	**median**	**max**	**median**	**max**	**median (K)**	**avg. (K)**	**max (M)**
hamm	65	17.8	780	0.2	4.63	25.4	1.2
slink	64	17.8	782	0.1	4.37	27.6	1.3
potato	58	27.3	732	0.2	3.46	23.2	2.0
woody	60	29.8	784	0.4	3.61	25.2	2.9
sarge	62	68.6	904	0.9	3.74	23.2	4.0
etch	65	27.2	1012	0.4	4.54	25.8	5.6
lenny	66	59.6	1000	0.9	4.41	26.5	5.9
squeeze	69	57.2	960	2.3	4.17	29.7	7.9
wheezy	69	182.4	924	2.8	3.97	36.2	13.9
jessie	67	182.4	808	2.8	3.40	37.3	14.9

Package size Thanks to the mapping between metrics and packages, we can also study the distribution of package sizes in different Debian releases: it is plotted in Fig. 3 for selected releases. Averages, medians, and maximums of selected metrics over packages are given in Table 6.

Increasingly, more and more very large packages are present in Debian: at the time of Jessie the *chromium-browser* and *linux* packages have, respectively, more than 15M and 12M SLOC. When Hamm was released its biggest package was *xfree86*, with "only" 1.2M SLOC. At the same time the per-release averages of package size are going up, whereas medians are going down. Overall it appears that: i) smaller and smaller packages are getting added to Debian, ii)

larger and larger packages are getting added too; with (ii) dominating more and more the total size of releases. A possible explanation for (i) comes from the packaging of relatively new software ecosystems that are increasingly releasing very small packages, e.g., Python's PyPi, R's CRAN, Node.js' NPM, etc. (ii) on the other hand seems due to behemoth software packages such as Web browsers, that are becoming self-contained work environments that need to (re)implement more, and more complex, functionalities that were historically available from separate packages.

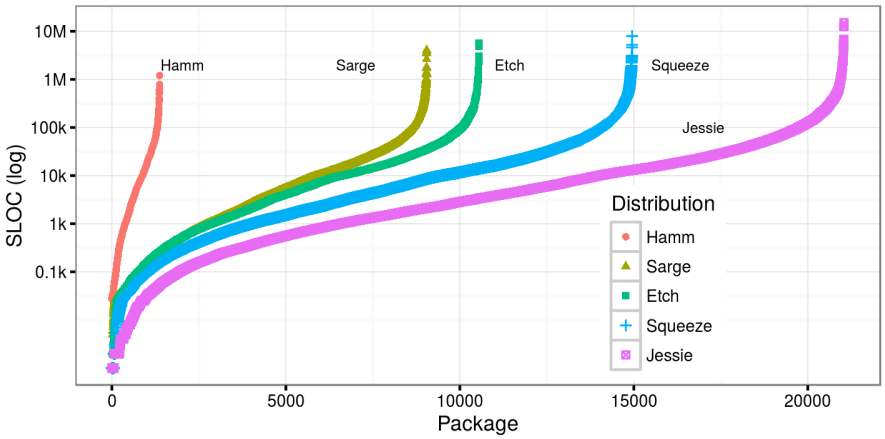

Fig. 3 Size of packages per distribution (measured in SLOC, y-axis). Each integer in the x-axis represents one package. E.g., in Jessie ≈ 7500 packages have sizes less or equal to 1k SLOC, while ≈ 20 000 packages have sizes less or equal to 100k SLOC.

5.2 Package maintenance

RQ ii is about the amount of changes that Debian users can expect when upgrading from one stable release to another. As the pairs ⟨name, version⟩ uniquely identify packages throughout Debian history, and as those pairs are available in the *package_info* table, we can leverage the Debsources Dataset to compare the sets of packages shipped by different Debian releases. Furthermore we can dissect package versions into their upstream and Debian-specific parts (see Section 3) to related changes in the Debian archive to upstream ones.

The top-half of Table 7 summarizes the amount of changes between pairs of Debian releases. *Common* packages are those that appear in both releases, in the same or different versions. *Unchanged* packages appear in both releases with the same "upstream" version, ignoring Debian-specific version changes (hence: *unchanged* ⊆ *common*).

It is interesting to note that 73 packages have remained at the same upstream version between Hamm and Jessie, for more than 17 years, whereas

Table 7 Changes between Debian releases: 'c' for common, 'u' for unchanged, and 'm' for modified packages.

from \ *to*	**slink**	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**	**jessie**
hamm	1324c 842u	1198c 463u	1079c 270u	958c 175u	864c 148u	782c 124u	719c 100u	670c 81u	649c 73u
slink		1657c 742u	1455c 384u	1281c 252u	1155c 210u	1037c 172u	941c 136u	881c 113u	852c 101u
potato			2456c 935u	2118c 551u	1881c 436u	1683c 352u	1497c 271u	1399c 220u	1348c 201u
woody				4588c 1688u	3953c 1156u	3497c 908u	3018c 633u	2786c 520u	2648c 458u
sarge					7671c 3832u	6828c 2597u	5896c 1717u	5349c 1367u	5042c 1164u
etch						9230c 4578u	8033c 2906u	7212c 2203u	6778c 1813u
lenny							10823c 5271u	9624c 3673u	8999c 2928u
squeeze								13098c 6802u	12201c 4890u
wheezy									16160c 8427u

	from previous suite to **slink**	**potato**	**woody**	**sarge**	**etch**	**lenny**	**squeeze**	**wheezy**	**jessie**
modified pkgs	556m	1305m	3127m	4462m	2879m	3287m	4128m	4466m	4881m
changed files per pkg	54.6%	64.4%	65.3%	67.5%	58.9%	59.8%	60.4%	57.3%	54.7%

their Debian revisions have evolved. Among these packages we can find for instance *netcat*, a network tool that hasn't changed upstream for that long, but seems to be still working just fine in Debian (otherwise it would have been removed from recent releases). This hints at the fact that long lasting unchanged packages might have been abandoned upstream, but are still maintained in Debian via patches applied by distribution maintainers.

The bottom-half of Table 7 focuses on upgrades from a release n to the immediately subsequent release $n+1$, which is the most common (and the only officially supported) upgrade path in Debian. The table shows the number of *modified* packages between consecutive releases (packages which exist in both releases, but in different upstream versions), as well as the proportion of source code files updated in these packages. The latter can be computed using the already discussed mapping between files and packages, together with either SHA1 or SHA256 checksums, both available in the *files* table.

The percentage of common and unchanged packages w.r.t. the previous release oscillates around 87% (common) and 43% (unchanged) with low variance. This suggests that Debian users experience high stability in terms of which packages are available across releases (almost 90%), as well as a steady flow (around 60%) of new upstream releases that are incorporated by Debian maintainers. The number of changed files per package on the other hand gives insights into how much new upstream releases touch the actual source code that form packages. This measure is also pretty stable across all Debian releases, ranging between 54% and 67%. Note however that this does not tell us how much individual files have been changed, only how many of them have: bumping copyright year in a file header or rewriting the file from scratch will still account for one source code file change. More precise evaluations of "how much" source code has changed can be performed leveraging TLSH hashes, that are readily available in the Debsources Dataset as well.

5.3 Programming language popularity

To address RQ iii (programming language popularity) we can simply aggregate per-package first, and per-release then, the SLOC counts available in the *metrics* table as computed by both `sloccount` and `cloc`. In either case the field *st_value* is used to detail the detected programming language. For consistency with RQ i, in the following we present `sloccount` results.

The evolution of programming languages in Debian is presented in Table 8 and plotted in Fig. 4 and 5. In both cases we restrict presented results to the most popular languages, using the Jessie release as a reference. Fig. 4 shows the evolution of language popularity in absolute SLOCs, while Fig. 5 shows the proportion over release size measured in SLOC.

Results show that C has always and still is the dominant language in Debian, since a big part of the core operating system (the Linux kernel, the GNU suite, etc) is written in C. However, while the absolute amount of C code has been steadily increasing, its proportion over the total is decreasing since

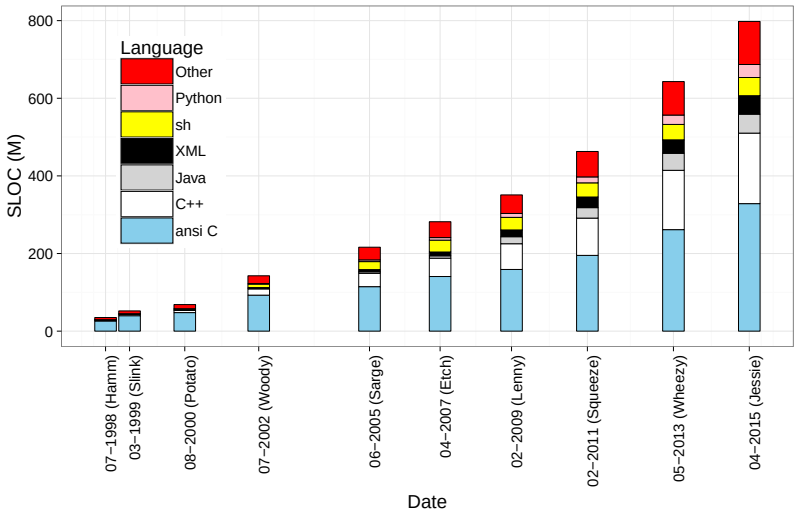

Fig. 4 Evolution of the most popular (top-6 plus other) programming languages in Debian by total number of SLOC per release.

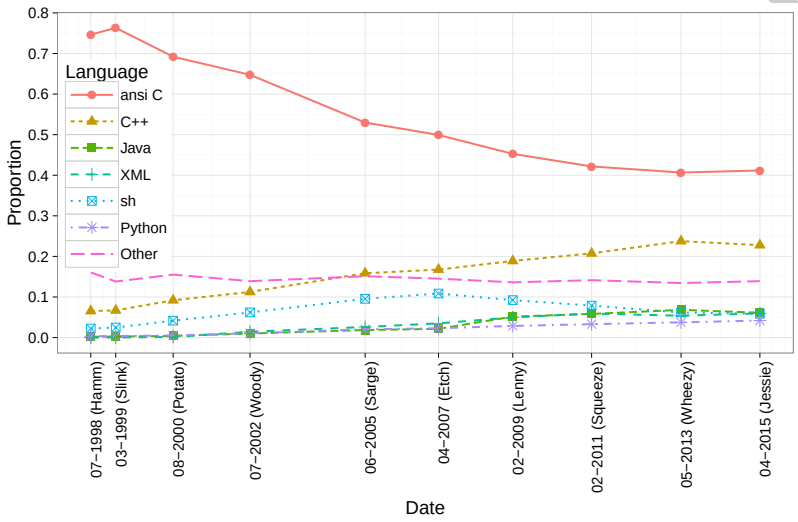

Fig. 5 Evolution of the most popular (top-6 plus other) programming languages in Debian as a proportion of release size in SLOC.

the Slink release (1999). Other languages, and most notably C++, are getting more and more relevant. The proportion of C code seems to have been stable for the past 3 releases though, at about 41% of the total.

Without a comprehensive reference base for FOSS source code it is impossible to determine how representative these numbers are of out-of-Debian trends. But the comparison with programming languages trends on other plat-

Table 8 Most popular programming languages in Debian releases, in MSloc. Numbers between parentheses represent percentage of total. (Quantities < 0.1 have been omitted and replaced by ≈ 0.)

Release	total	ada	ansic	asm	cpp	erlang
hamm	35	0.24 (0.68)	27 (77)	0.39 (1.1)	1.9 (5.5)	NA (NA)
slink	52	0.26 (0.51)	4.5 (78)	0.64 (1.2)	3.0 (5.7)	NA (NA)
potato	69	0.42 (0.61)	49 (7.6)	0.57 (0.83)	5.8 (8.5)	0.21 (0.30)
woody	15	0.58 (0.41)	94 (67)	2.6 (1.9)	15 (1.4)	≈ 0 (≈ 0)
sarge	21	1.1 (0.53)	120 (56)	2.8 (1.3)	33 (16)	≈ 0 (≈ 0)
etch	270	0.76 (0.28)	140 (53)	4.5 (1.6)	46 (17)	0.69 (0.25)
lenny	330	0.85 (0.26)	160 (49)	4.1 (1.2)	64 (19)	0.82 (0.25)
squeeze	440	1.3 (0.29)	210 (46)	4.8 (1.1)	96 (22)	1.3 (0.28)
wheezy	640	1.6 (0.25)	290 (46)	8.2 (1.3)	150 (23)	1.6 (0.25)
jessie	780	1.8 (0.23)	360 (46)	1.5 (1.3)	180 (23)	1.8 (0.23)

Release	f90	fortran	haskell	java	lisp
hamm	≈ 0 (≈ 0)	0.70 (2.0)	NA (NA)	≈ 0 (0.17)	0.11 (0.32)
slink	≈ 0 (≈ 0)	1.0 (2.0)	≈ 0 (≈ 0)	0.13 (0.25)	2.5 (4.8)
potato	≈ 0 (≈ 0)	1.4 (2.1)	≈ 0 (≈ 0)	0.27 (0.40)	3.4 (4.9)
woody	≈ 0 (≈ 0)	2.3 (1.6)	0.28 (0.20)	1.4 (1.0)	5.1 (3.7)
sarge	≈ 0 (≈ 0)	2.9 (1.4)	0.98 (0.47)	4.0 (1.9)	6.9 (3.3)
etch	≈ 0 (≈ 0)	2.1 (0.76)	0.58 (0.21)	6.1 (2.2)	7.2 (2.6)
lenny	0.29 (≈ 0)	2.3 (0.68)	0.67 (0.20)	18 (5.4)	8.1 (2.4)
squeeze	0.76 (0.17)	2.5 (0.56)	0.93 (0.21)	27 (6.1)	9.7 (2.2)
wheezy	1.1 (0.17)	8.2 (1.3)	1.6 (0.25)	44 (7.0)	8.8 (1.4)
jessie	7.5 (0.95)	9.7 (1.2)	2.0 (0.25)	50 (6.3)	11 (1.4)

Release	makefile	ml	objc	pascal	perl	python
hamm	2.3 (6.7)	≈ 0 (0.26)	≈ 0 (0.16)	0.17 (0.49)	≈ 0 (≈ 0)	0.49 (1.4)
slink	0.15 (0.28)	≈ 0 (0.11)	0.22 (0.43)	≈ 0 (0.10)	0.79 (1.5)	0.20 (0.39)
potato	0.21 (0.31)	0.15 (0.22)	0.41 (0.60)	0.31 (0.45)	1.4 (2.0)	0.36 (0.52)
woody	0.37 (0.26)	0.38 (0.27)	0.55 (0.39)	0.43 (0.31)	3.0 (2.1)	1.5 (1.1)
sarge	0.55 (0.26)	0.76 (0.36)	0.76 (0.36)	1.4 (0.65)	6.3 (3.0)	4.4 (2.1)
etch	0.68 (0.25)	1.3 (0.47)	1.0 (0.37)	1.1 (0.41)	8.1 (3.0)	6.5 (2.4)
lenny	0.75 (0.23)	1.8 (0.55)	1.1 (0.32)	0.87 (0.26)	9.4 (2.8)	1.1 (3.0)
squeeze	0.69 (0.16)	2.6 (0.59)	1.2 (0.27)	3.8 (0.84)	13 (2.9)	16 (3.5)
wheezy	0.66 (0.10)	3.8 (0.59)	1.7 (0.26)	4.4 (0.69)	18 (2.8)	25 (3.9)
jessie	0.72 (≈ 0)	4.1 (0.52)	1.9 (0.25)	5.5 (0.70)	20 (2.5)	35 (4.5)

Release	php	ruby	sh	sql	tcl	yacc
hamm	≈ 0 (≈ 0)	≈ 0 (≈ 0)	0.92 (2.6)	≈ 0 (≈ 0)	0.35 (1.0)	0.19 (0.54)
slink	≈ 0 (≈ 0)	≈ 0 (≈ 0)	1.5 (2.9)	≈ 0 (≈ 0)	0.50 (0.97)	0.25 (0.48)
potato	≈ 0 (≈ 0)	≈ 0 (≈ 0)	3.3 (4.8)	≈ 0 (≈ 0)	0.67 (0.97)	0.32 (0.47)
woody	0.58 (0.41)	≈ 0 (≈ 0)	9.5 (6.8)	≈ 0 (≈ 0)	1.2 (0.86)	0.45 (0.32)
sarge	1.8 (0.87)	0.46 (0.22)	21 (1.2)	≈ 0 (≈ 0)	2.1 (1.0)	0.56 (0.26)
etch	3.0 (1.1)	1.2 (0.45)	31 (12)	0.51 (0.19)	1.7 (0.64)	0.65 (0.24)
lenny	4.0 (1.2)	2.0 (0.61)	33 (9.9)	0.66 (0.20)	1.9 (0.58)	0.67 (0.20)
squeeze	4.7 (1.1)	4.3 (0.96)	38 (8.6)	1.5 (0.34)	2.5 (0.55)	0.81 (0.18)
wheezy	5.8 (0.92)	4.2 (0.66)	42 (6.6)	2.4 (0.38)	2.6 (0.41)	1.0 (0.16)
jessie	8.1 (1.0)	5.2 (0.66)	49 (6.2)	3.9 (0.49)	3.1 (0.40)	1.2 (0.15)

forms (e.g., GitHub [15]) is striking. Whereas GitHub developers seem to be flocking to JavaScript, Java, Ruby, and PHP, a foundational operating system

like Debian is still prominently composed of system-level languages like C and C++.

Table 9 Median file size (in SLOC) per language for the most popular languages.

Release	ada	ansic	asm	cpp	erlang	f90	fortran	haskell	java	lex	lisp
hamm	40	69	26	62	-	47	67	-	44	180	130
slink	38	68	43	60	-	11	61	12	36	190	120
potato	40	71	34	57	170	11	73	44	34	170	120
woody	47	86	75	64	42	16	81	36	37	210	140
sarge	47	79	40	66	43	38	89	37	43	180	120
etch	43	80	32	64	210	61	79	57	46	170	120
lenny	42	79	25	61	200	89	78	44	44	180	130
squeeze	45	76	20	62	160	96	76	58	45	190	120
wheezy	47	80	23	66	130	98	110	32	47	190	110
jessie	38	75	21	65	110	79	96	39	47	210	110

Release	make	ml	objc	pascal	perl	php	python	ruby	sh	sql	tcl	yacc
hamm	43	26	150	140	62	≈ 0	59	≈ 0	20	19	87	520
slink	42	22	120	12	66	≈ 0	61	≈ 0	23	16	92	510
potato	42	32	140	63	63	19	68	31	23	17	97	600
woody	46	35	170	72	61	45	65	31	27	16	80	320
sarge	47	44	160	92	63	39	60	38	35	11	94	320
etch	47	50	150	300	70	46	59	45	38	11	92	320
lenny	44	49	150	230	64	46	58	42	39	11	84	320
squeeze	32	50	140	84	56	44	59	38	37	26	69	320
wheezy	15	48	130	79	58	39	60	36	33	20	72	340
jessie	11	43	110	88	53	37	62	32	34	20	65	350

File size We can drill down to investigate median file sizes (in SLOC) per language and their evolution over time. Detecting the programming language of each source file can be done in a number of ways, each one with their own strengths and weaknesses. Two ways of doing that is delegating language detection to `cloc` and `sloccount`, using the *st_value* column of the *metrics* table; alternatively one can look at file extensions (field *ext* in table *path_info*) and map popular extensions to programming languages. We have adopted the `sloccount` approach in the following.

Table 9 presents per release and per language median file sizes for the most popular languages. For the top-6 of them, their evolution over time is plotted in Fig. 6.

Most of the studied languages are shown to be relatively stable in their median file size over time. This is the case for mainstream languages such as C, C++, and Java, as well as several others such as Perl and Lisp (not plotted). Median file size also appears to be a rather intrinsic characteristic of a programming language, that is not really affected by how popular the language is in a large FOSS ecosystem.

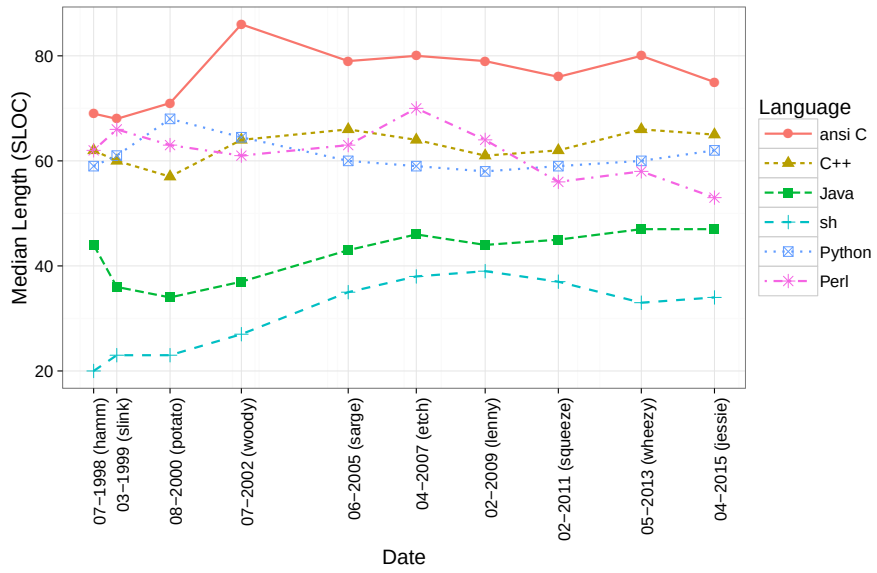

Fig. 6 Evolution over time of the median file size (in SLOC) per language, based on file extension.

5.4 Debian licensing over time

One of the most important characteristics that define a FOSS component is its license. It is very important to know the license of a package, as it determines the rights and obligations of anybody wanting to reuse and further distribute the software (either as a component or as a stand-alone product). Since the conception of Debian in 1993 the FOSS license landscape has evolved significantly. Many licenses released new versions, others have been created, and some ceased to be used. The long history of Debian creates a perfect subject to evaluate how FOSS licenses use has evolved over time, and the popularity of licenses currently in use.

Creating a census of licenses used in a large software distribution is not an easy task though. The first challenge is how to identify the licenses of individual files. Then, one needs to consider the overall licenses of aggregate/composite software bundles, such as packages in the case of FOSS distributions. The license of a package as a whole is indeed not necessarily the same of the files that compose it: due to how license compatibility works, a package might have most of its files under license A, but its aggregate license might end up being license $B \neq A$. Unfortunately there is no well-established convention for documenting either one, and tools for license identification have thus far focused on discovering the license of individual files. In order to answer RQ iv, in the following we focus on the license of individual files. We do not study the license of packages as wholes. We do, however, aggregate file licenses by package in order to study license variability within packages, answering RQ v.

Finally, in order to answer RQ vi (*how has license use evolved in Debian over time?*), we analyze the evolution of our answers to RQ iv (file licensing) and RQ v (source package licensing) across all Debian stable releases.

As automatic license identification of a file is still difficult and error prone, we avoid developing in house heuristics and rather resort to the two tools that are considered the state of the art in license identification: `ninka` [8] and `fossology` [9]. Both tools are capable of identifying commonly used licenses, but vary in the way that they deal with less common ones. For example, `ninka` is capable of identifying many variants of the BSD and MIT family of licenses, while `fossology` groups them into "MIT-style" and "BSD-style". On the other hand `fossology` is capable of identifying more licenses and, where applicable, uses the license identifiers standardized by SPDX [22] for its reports.

In terms of Debsources Dataset use, the license information we have extracted using both `ninka` and `fossology` are readily available from table *licenses*. The detected licenses of individual files as returned by the two tools can be discriminated using the *oracle* field of that table, whose value will be either "ninka" or "fossology". Note that the detected license, available in field *license*, is tool-dependent: we have favored preserving the full information returned by license identification tools over data uniformity. For the sake of brevity in the following we only discuss `fossology` results, but interested scholars can use the Debsources Dataset to explore `ninka` results.

While the Debsources Dataset contains the output of `ninka` and `fossology` for every unique file, in the following we only report about licensing of source code files (excluding, e.g., binary files such as raster images). To that end we will ignore all files not recognized as being source code by `sloccount`.

5.4.1 Individual file licensing

Table 10 Number of different licenses identified in each release.

Release	Licenses
hamm	281
slink	326
potato	437
woody	620
sarge	949
etch	1125
lenny	1352
wheezy	1879
jessie	2039

Table 10 shows the total amount of *different* licenses identified for each release in the dataset. As it can be observed, the number of identified licenses is very large and has grown an order of magnitude across Debian history. Part of the reason is that, when a file is licensed under two or more licenses, such combination of licenses is considered to be a different license by license identification tools. For example, in several Debian releases Firefox is licensed under

a combination of the MPL, GPL-2.0, and LGPL-2.1. In most releases, few licenses account for most of the identified licenses: the top 50 most frequently identified licenses (including "No_license_found") correspond to 94–97% of release source files.

Table 11 shows the top identified licenses in the oldest and newest Debian releases available in the dataset. As it can be observed, most frequently files do not have a license that `fossology` can directly identify. Fig. 7 shows the evolution over time of the most common identified licenses. As it can be seen, the most used licenses have been the GPL and BSD families, with recent increases for Apache-2.0 and the Mozilla Public License (MPL).

Table 11 Top identified licenses in two selected releases.

Release	License	Files	Prop.(%)	Accum. (%)
Hamm	No_license_found	72,533	47.5	47.5
	GPL-2.0+	22,983	15.1	62.6
	LGPL-2.0+	14,608	9.6	72.2
	BSD-style	3,667	2.4	74.6
	See-doc(OTHER)	2,490	1.6	76.2
	MIT-style	2,457	1.6	77.8
	UnclassifiedLicense	2,359	1.5	79.4
	GPL	2,329	1.5	80.9
	BSD-4-Clause-UC	2,112	1.4	82.3
	See-file	1,938	1.3	83.5
	X11	1,887	1.2	84.8
Jessie	No_license_found	1,011,088	29.7	29.7
	GPL-2.0+	432,482	12.7	42.4
	Apache-2.0	168,655	5.0	47.4
	GPL-3.0+	160,233	4.7	52.1
	GPL-2.0	148,364	4.4	56.4
	LGPL-2.1+	141,747	4.2	60.6
	BSD	115,135	3.4	64.0
	LGPL-2.0+	87,153	2.6	66.5
	BSD-3-Clause	72,634	2.1	68.7
	MIT	71,022	2.1	70.8
	EPL-1.0	66,755	2.0	72.7

5.4.2 Package licensing and variability

In order to address RQ v (source package licensing), it is not practical to simply report each and every license found in very package. Instead, we develop several metrics, each one highlighting different aspects of the licensing of source code files belonging to a given package:

– How detectable are the licenses of the package source files? For this purpose we compute the proportion of files for which a license was identified over the total number of files. `fossology` reports *No_license_found* when it does not find the license of a given file, and *UnclassifiedLicense* when it finds one it does not know. Hence we consider a file to have an *identifiable*

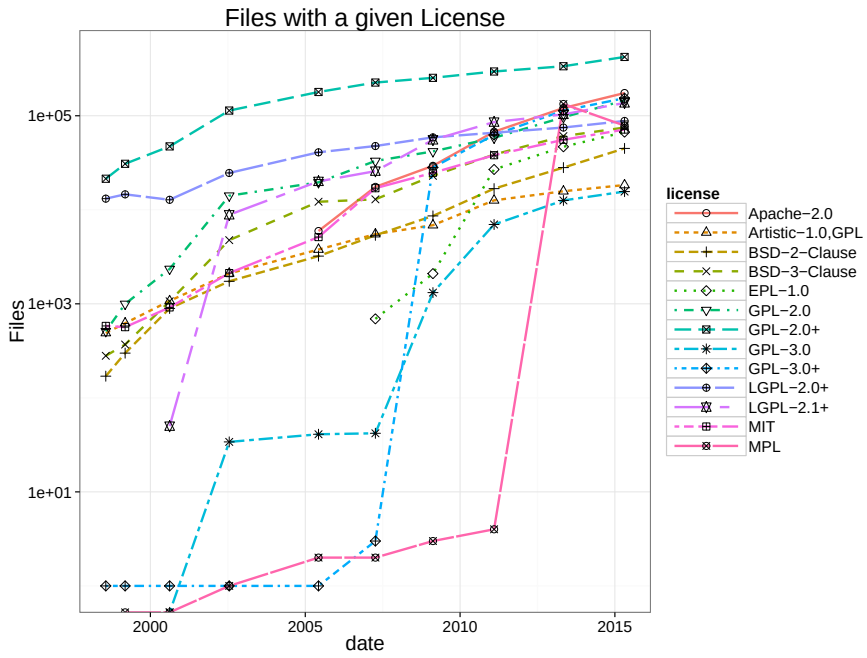

Fig. 7 Evolution of the number of files with a given license, as detected by `fossology`.

license if `fossology` reports a license other than *No_license_found* and *UnclassifiedLicense*. The *proportion of files without a license* is the number of source files without an identifiable license divided by the total number of source files.

- How many *different* licenses can be found in a given package? In this case we ignore files without an identifiable license.
- What is the *dominant license* identified in each package? We define such a license as the most commonly identifiable license, counted as the number of files it applies to. If two or more licenses are equally frequent, all of them are considered to be equally dominant.
- How much *diversity* is there in the licenses of the files of a package? The more licenses a package contains, the larger it will be the problem space of determining its license as an aggregate—due to how license compatibility works the problem will not necessarily be more *difficult*, but more options will have to be considered.
  To establish license diversity within a package we use Wilcox's Analog of the Mean Difference (MNDif). It represents the average of the absolute differences of all the possible pairs of license frequencies. Intuitively, it is the equivalent of a GINI coefficient, but applicable to categorical data. A value of 0 implies that all files have the same license, while a value of 1 that all licenses are equally represented, i.e., each license is used by the same number of files.

When aggregating by package, different licensing patterns appear. Fig. 8 shows box plots with the proportion of files that do not have a detectable license. The median number of source files without an identifiable license has fluctuated between 50% and 60%, showing the same pattern over time. It is important to mention that `sloccount` is relatively aggressive on what it considers source code. For example, `sloccount` considers Makefiles and configuration and installation scripts to be source code; these files do not normally include a license. For this reason we also include the box plots for C (Fig. 9) and Java files (Fig. 10). As it can be seen, their current median is below 5% in both cases, and over time, the proportion of files without a license keeps dropping. It seems that, at least from the point of view of `fossology` and for mainstream programming languages, FOSS development practices (and in particular writing down license annotations) are evolving in a way that makes automatic license detection easier. There is still plenty of room for improvement though.

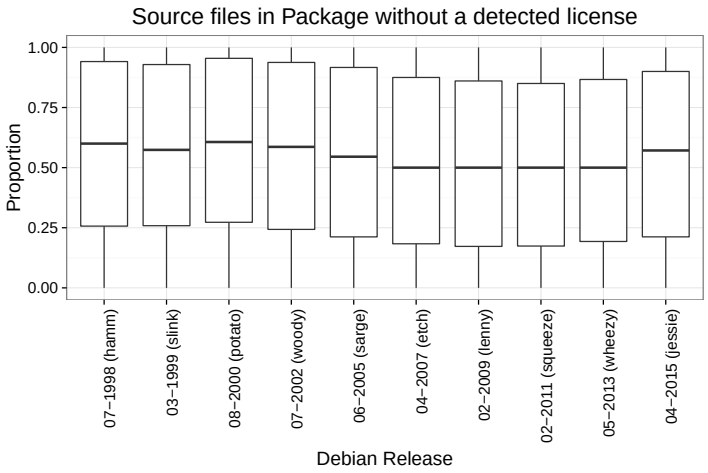

Fig. 8 Box plots of the proportion of files with no identifiable license.

The number of licenses used per package is generally very small, as shown in Fig. 11. The median is 2; the third quartile has decreased from 3 to 2 licenses in recent releases (taking into account identifiable licenses only).

With regard to the chosen licenses, we present in Fig. 12 the evolution of licenses that occur at least once in a package. As it can be seen variants of the GPL licenses are still, by far, the most commonly used, and in particular versions 2.0 and 2.0+ (i.e., "version 2 or any later version").

Fig. 13 shows the evolution of dominant licenses in Debian packages, according to our definition. The top license is, once again, GPL-2.0+, followed by: Artistic-1.0/GPL dual-licensing (the licensing choice of Perl and most Perl libraries), GPL-3.0+, and Apache-2.0.

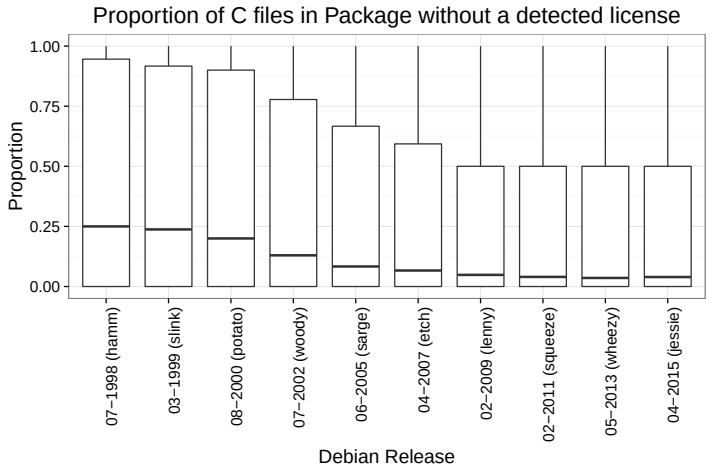

Fig. 9 Box plots of the proportion of C files with no identifiable license.

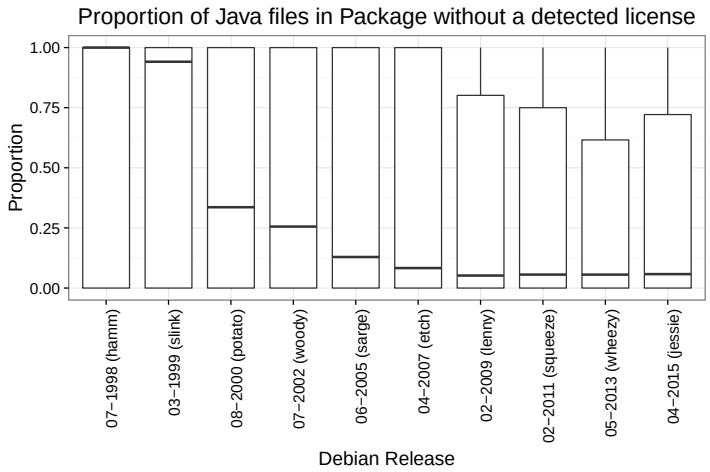

Fig. 10 Box plots of the proportion of Java files with no identifiable license.

With regard to the variability of licenses in packages, we present in Fig. 15 the box plot of the MNDif of the licenses per package in each release. As it can be seen, most packages have very small license variability, and license diversity seems to be decreasing over time. This might be due to new, popular programming language ecosystems that manage to impose, either with legal agreements or simply via "bandwagon" effects, a specific license to all the new modules and libraries that will be developed in the language.

Fig. 14 shows a scatter plot showing MNDif vs the number of source files in a package, for the most recent Debian release (Jessie). A pattern stands out: the more source files a package has, the less license diversity. This might

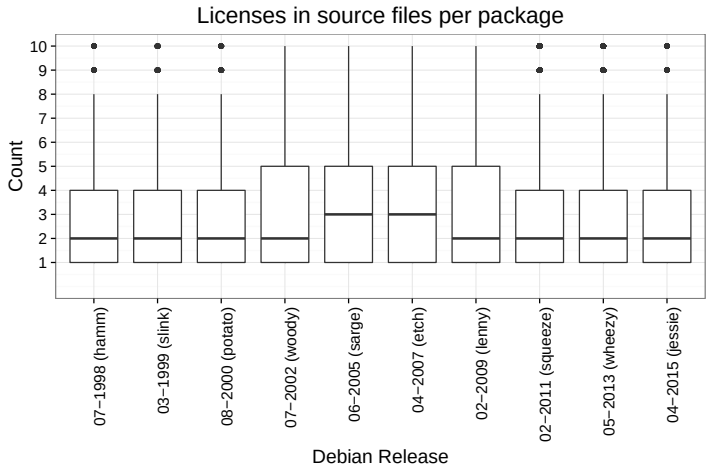

Fig. 11 Box plots with the number of different identified licenses per package.

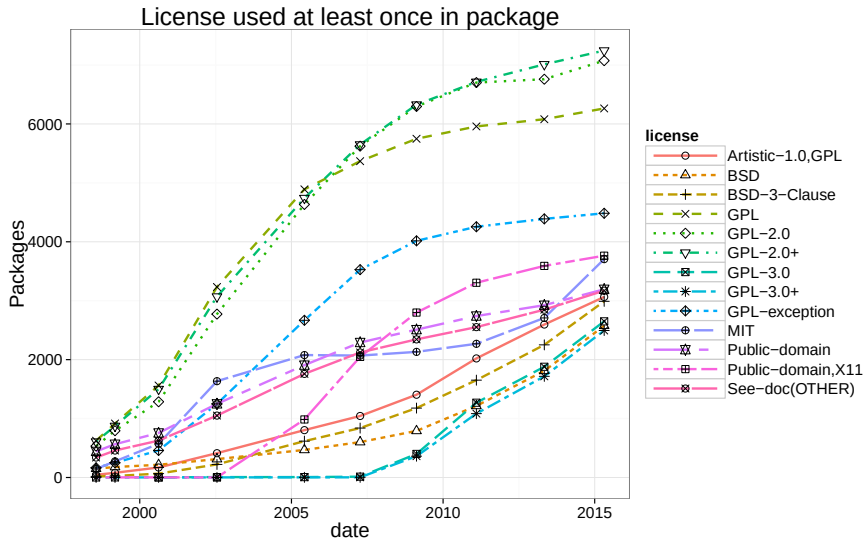

Fig. 12 Evolution of the number packages that use a license at least once.

seem counter intuitive at first, because more files would appear to give more opportunities for reusing code from other FOSS projects and hence adopt a new license, increasing diversity. Our intuition is that such aspect is countered by the fact that larger, well-established FOSS projects tend to be governance-heavy, cautious when importing external code in their own repositories (e.g., due to long-term maintainability concerns), if not simply used to impose a specific license as a condition to accept external code contributions.

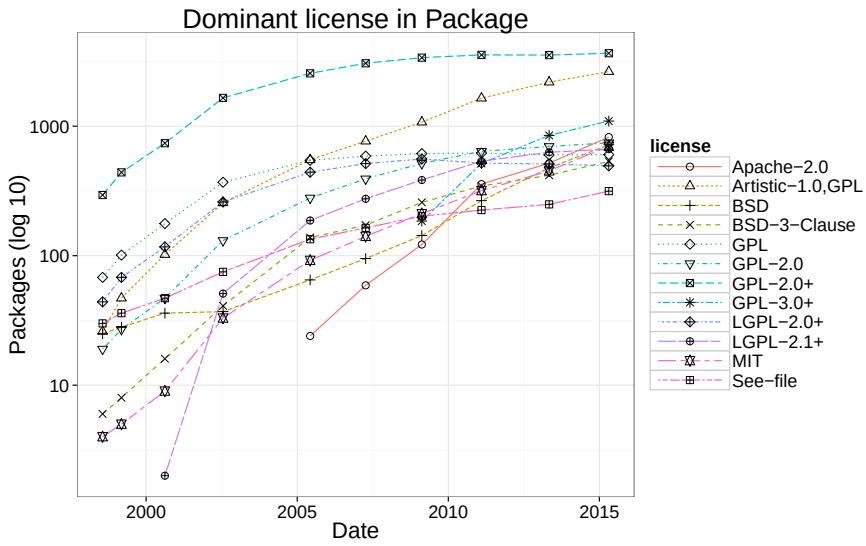

Fig. 13 Evolution of the number of packages that have a given dominant license.

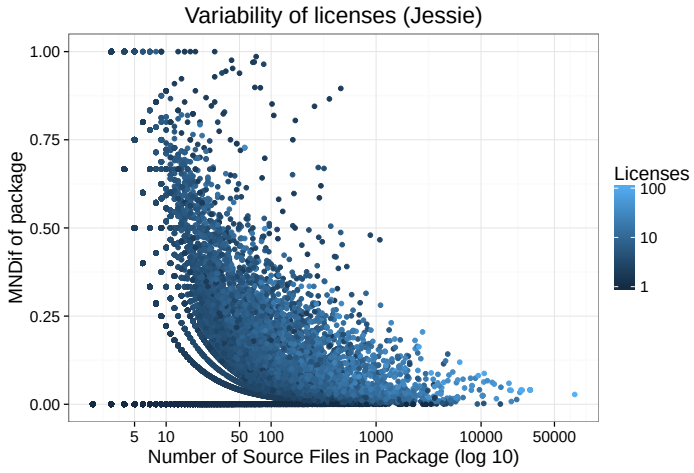

Fig. 14 Each point represents a package in Jessie (version 8): its MNDif vs number of source files. As it can be seen, smaller packages tend to have more variability in their licensing.

To the best of our knowledge this is the first study of FOSS license popularity at this scale, and in particular over such a long time frame. While there exist reports on the Web about FOSS license popularity, and most notably from Black Duck,[17] such reports do not disclose the adopted methodology nor are clear on the underlying sample of observed FOSS projects, making them non-reproducible. Furthermore they do not properly document how licenses

[17] https://www.blackducksoftware.com/top-open-source-licenses

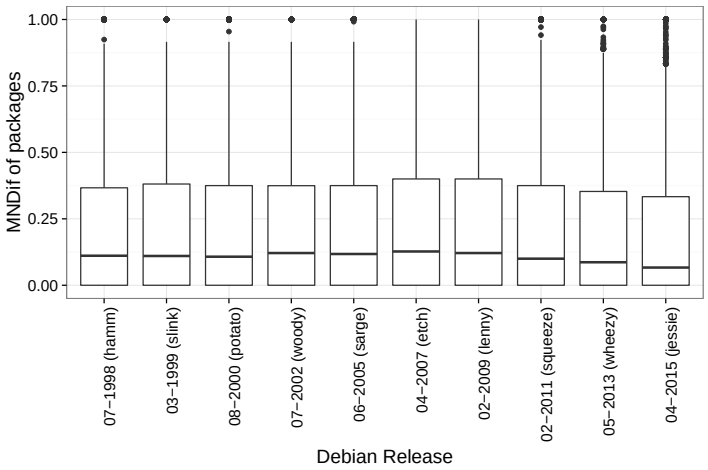

Fig. 15 Box plots showing the MNDif of packages per release. A MNDif of zero means no variability, while 1 means that every license in the package is equally represented.

are counted, which is an important and tricky aspect of surveying FOSS license use [14].

5.5 Looking back, Debsources Dataset advantages

Looking back at this case study we can attempt a self-assessment of the advantages induced by using the Debsources Dataset as a starting point. Limitations and threats to validity will be discussed in the next section.

Using Debsources Dataset metadata (the database) we have been able to study Debian growth over time as well as the correlation and distribution of the chosen metrics. We have also been able to get insights on software engineering practices such as package maintenance and study over time the popularity of programming languages and software licenses, aggregating at different granularities (file, package, release). In all cases data gathering boiled down to crafting and executing relatively straightforward (Postgre)SQL queries. Statistical analysis and plotting have then been implemented externally (using GNU R), processing Postgres query output.

Without the Debsources Dataset the starting point of the case study would have necessarily been retrieving and unpacking all Debian releases, followed by running all measurement/mining tools on the obtained source code. We have documented in Section 2.2 how, in terms of resources, doing so would have required about 1.5 months of processing time (on a single machine) and 1.5TB of disk space. None of this has been necessary to run our case study. More importantly than savings in computational resources though, the Debsources Dataset relieves scholars from the responsibility of figuring out which-tools-

to-use-when in order to mine Debian-specific data sources; the starting point becomes a relatively straightforward ER data model.

A counter argument here is that the metrics and information we were interested in were all already available in the dataset; unsurprisingly, given we initially included them in the dataset for our own needs. The first response to this is that a significant part of the metadata included in the dataset is intrinsic to either how Debian works (e.g., Debian release information) or the nature of the referenced objects (e.g., file size, SHA checksums). We expect that most studies interested in using Debian as a FOSS sample will need these information; the Debsources Dataset alleviates the need of having to mine them over and over again.

Second, when it comes to mining new facts that are not included in the Debsources Dataset, the source code part of the dataset and how it is organized offers many benefits. Most notably it saves space; thanks to deduplication the required disk space is cut by approximately 50%. As many kind batch source code analyses are I/O bound, a similar saving in processing time should generally be expected as well. Source code organization also simplifies analysis, since the files are sharded into a (relatively speaking) small number of directories; for example, one can run `cloc` recursively only 256 times, once per each top-level directory. Another benefit is that this code organization avoids the challenges of having to deal with the original path names. Some of these paths use extended character sets, or include characters that might not be handled properly by (buggy) research tools (e.g., apostrophes and white spaces), or might differ only in capitalization resulting in name clashes on some popular file systems.

It is also reasonable to expect that newly mined facts from Debian source code will need to be correlated, one way or another, with metadata that *are* available in this dataset. It is at the border of source code mining and related metadata that the Debsources Dataset offers the most time- and space-saving opportunities for empirical software engineering scholars.

6 Threats to validity

Due to the fact that at the time Debian neither had a source package format that can be extracted using today's `dpkg-source`, nor package indexes (`Sources` files), the Debsources Dataset does not include the first 3 Debian releases: Buzz (1996), Rex (1996) and Bo (1997). The first release included in the dataset is Hamm (1998). Additionally, due to a regression in `dpkg-source`,[18] 12 packages from historical releases cannot be extracted and are missing from the dataset. We do not expect such a tiny number of packages to significantly impact the usefulness of the dataset.

It is important to note that the Debsources Dataset does not fully round-trip with Debian mirrors: it is not possible to fully reconstruct source packages

[18] http://bugs.debian.org/740883

by only using the dataset. This is because the dataset is not supposed to be as precise as a backup system in capturing detailed file characteristics such as ownership, permissions, and extended attributes.

`sloccount` and Exuberant Ctags are starting to show their age and suffer from a lack of active maintenance. Most notably, they do not support languages like Scala and JavaScript, which might then be underrepresented in the dataset. The case of JavaScript is particularly worrisome, due to its increasing popularity for server-side Node.js applications. On the front of SLOC counting the issue is mitigated by the presence of counts as returned by `cloc`, which is a more modern tool with support for recent languages. At the same time we consider important to *also* have `sloccount` results in the dataset, as it is used as a reference tool in many works in the literature.

Regarding licensing data, the main threat to construct validity is the reliability of license identification as implemented by the tools that we used to detect licenses. `fossology` is a mature tool, widely used in the software industry; `ninka` is more experimental but shines in specific areas such as discriminating among license variants. With respect to external validity, we make no specific claims. While Debian is a good proxy for well-established FOSS products, Debian requires clear licensing on any software that it incorporates, hence what is observable in Debian might not reflect all of FOSS.

More generally, while we claim that the Debsources Dataset is representative, by construction, of Debian trends, any extrapolation of findings based on this dataset to more general FOSS trends should stand on its own ground. Debian is likely representative of enterprise use of FOSS as a base operating system, where stable, long-term and seldomly updated software products are desirable. Conversely Debian is unlikely representative of more dynamic FOSS environments (e.g., modern Web-development with micro libraries) where users, who are usually developers themselves, expect to receive library updates on a daily basis. Debian trends on size, language popularity, and licensing are likely not directly transferable to those contexts.

7 Related work

Debsources [3] is the software platform used to produce a previous version of the Debsources Dataset [27], which contained only metadata (no source code) and only a subset of the metadata described in this paper. Debsources can be used to recreate similar datasets for any other FOSS distribution that uses the Debian source package format, including Ubuntu[19] and hundred others [5].

Reproducing the findings of a former macro-level software evolution study [11] motivated in part the development of Debsources. That study also shows the results of running `sloccount` on Debian releases over the 1998–2007 period. The Debsources Dataset covers twice that period, offers more and more diverse metadata (ctags, disk size, checksums, license information),

[19] http://www.ubuntu.com/

and is publicly available from archival storage (Zenodo), whereas the dataset URL from [11] has been down for a few years now. Most notably the availability of the Debsources Dataset allows today to conduct studies similar to [11] without having to mirror Debian from different websites, run sloccount, manually classify by release, etc. All needed metadata are already available in an easy-to-query format. When it comes to missing metadata, they can be extracted from the source code shipped as part of the Debsources Dataset, minimizing the required computational effort thanks to source code deduplication.

The Ultimate Debian Database (UDD) [17] has assembled a large dataset about Debian and some of its derivatives, and is a popular target for mining studies and challenges [25]. UDD however lacks the time axis and is focused on distribution-level metadata. As such it lacks most of the content-oriented metadata (ctags, checksums, license information, etc.) that are available in the Debsources Dataset.

Other studies have targeted different aspects of the Debian ecosystem, such as discussion on its mailing lists (e.g., [21]). Those studies cannot be supported by the Debsources Dataset which focuses on source code. However, the presence of time-indexed metadata in the dataset allows to correlate mailing list discussions, and in particular development discussions that often touch specific versions of specific packages, with release and release dates.

Boa [6] is a Domain Specific Language (DSL) and an infrastructure to mine FOSS project collections like forges. Boa's dataset is larger in scope than the Debsources Dataset (e.g., it contains SourceForge) and also more fine grained, reaching down to the version control system level, but does not correspond to curated software collections like FOSS distributions. That has both advantages (it allows to peek into unsuccessful projects) and disadvantages: contained projects are less likely to be representative of what was popular at the time. The time horizon of BOA is also more limited than that of the Debsources Dataset.

FLOSSmole [12] is a collaborative collection of datasets obtained by mining FOSS projects. Many datasets in there are about Debian but no one is, by far, as extensive as the Debsources Dataset.

With respect to our case studies, analyses similar to the one of Section 5 on software evolution have been conducted in the past on various distributions, e.g., [24, 10], even though only punctually on individual releases. On the specific aspect of licensing, `ninka` has been used in the past to scan Debian Lenny [8]. Several empirical studies on licensing have used other FOSS distributions as data sources, such as Fedora Core 12 [7] and Debian Wheezy [26].

8 Conclusion

We have presented the Debsources Dataset: source code, release metadata about it, measurements, checksums, and license information spanning two decades of Free and Open Source Software (FOSS) history, as it can be observed through the lens of the popular Debian distribution. Using the dataset

we have conducted a long-term, macro-level evolution study of Debian looking from angles as diverse as size, package maintenance, programming language popularity, and software licensing.

The Debsources Dataset contains increasingly more fine-grained information (packages → releases → source code files → checksums → developer-defined symbols) about more than 3 billion lines of source code, from popular FOSS projects of their times. The dataset also contains the corresponding source code, deduplicated at file-level granularity, resulting in a factor 2 gain in the space required to store it in uncompressed form. Deduplication allows to efficiently process the available source code to mine further facts and correlate them with existing metadata.

The Debsources Dataset is publicly available as Open Data, documented, and reproducible using data and source code from the Debian project, as well as a variety of tools that are all available as FOSS. However, recreating it takes a non-negligible amount of storage and computational resources. Its availability as a ready to use dataset can therefore ease the work of scholars interested in macro-level software evolution, and in the history and composition of FOSS.

References

1. Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99, 2009.
2. Bram Adams, Christian Bird, Foutse Khomh, and Kim Moir. 1st international workshop on release engineering (RELENG 2013). In *ICSE'13*, pages 1545–1546, 2013.
3. Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014.
4. Serge Demeyer, Alessandro Murgia, Kevin Wyckmans, and Ahmed Lamkanfi. Happy birthday! a trend analysis on past msr papers. In *MSR 13: 10th Working Conference on Mining Software Repositories*, MSR'13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE.
5. DistroWatch distribution search — Debian-based distributions. http://distrowatch.com/search.php?ostype=Linux&basedon=Debian&status=Active.
6. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE / ACM, 2013.
7. Daniel M. German, Massimiliano Di Penta, and Julius Davis. Understanding and auditing the licensing of open source software distributions. In *18th International Conference on Program Comprehension (ICPC'2010)*, pages 84–93, May 2010.

8. Daniel M. German, Yuki Manabe, and Katsuro 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. ACM, 2010.
9. Robert Gobeille. The fossology project. In *MSR 2008: The 5th Working Conference on Mining Software Repositories*, pages 47–50. ACM, 2008.
10. Jesús M González-Barahona, MA Ortuno Perez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. Counting potatoes: the size of debian 2.2. *Upgrade Magazine*, 2(6):60–66, 2001.
11. Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.
12. James Howison, Megan Conklin, and Kevin Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *IJITWE*, 1(3):17–26, 2006.
13. Ian Jackson et al. Debian policy manual. Available at https://www.debian.org/doc/debian-policy/, 1996.
14. Michael Kerrisk. Surveying open source licenses. Available at https://lwn.net/Articles/547400/, 2013.
15. Alyson La. Language trends on github. Available at https://github.com/blog/2047-language-trends-on-github, 2015.
16. Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
17. Lucas Nussbaum and Stefano Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR*, pages 52–61. IEEE, 2010.
18. Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh - a locality sensitive hash. In *CTC, 4th Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.
19. Frederick P. Brooks, Jr. *The mythical man-month: essays on software engineering*. Addison-Wesley, 2 edition, 1995.
20. Gregorio Robles, Jesus M Gonzalez-Barahona, and Martin Michlmayr. Evolution of volunteer participation in libre software projects: evidence from debian. In *Proceedings of the 1st International Conference on Open Source Systems*, pages 100–107, 2005.
21. Sulayman Sowe, Ioannis Stamelos, and Lefteris Angelis. Identifying knowledge brokers that yield software engineering knowledge in oss projects. *Information and Software Technology*, 48(11):1025–1033, 2006.
22. Kate Stewart, Phil Odence, and Esteban Rockett. Software package data exchange (SPDX™) specification. *International Free and Open Source Software Law Review*, 2(2):191–196, 2011.
23. Andrew Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University Canberra, 1999.

24. David A Wheeler. More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.1.03.html, 2001.
25. Jim Whitehead and Thomas Zimmermann, editors. *Mining Software Repositories, MSR 2010*. IEEE, 2010.
26. Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. A method to detect license inconsistencies in large-scale open source projects. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 324–333, Piscataway, NJ, USA, 2015. IEEE Press.
27. Stefano Zacchiroli. The Debsources dataset: Two decades of Debian source code metadata. In *MSR 2015: The 12th Working Conference on Mining Software Repositories*, pages 466–469. IEEE, 2015.