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

Software Provenance Tracking at the Scale of Public Source Code

Guillaume Rousseau · Roberto Di Cosmo · Stefano Zacchiroli

the date of receipt and acceptance should be inserted later

Abstract We study the possibilities to track provenance of software source code artifacts within the largest publicly accessible corpus of publicly available source code, the Software Heritage archive, with over 4 billions unique source code files and 1 billion commits capturing their development histories across 50 million software projects.

We perform a systematic and generic estimate of the replication factor across the different layers of this corpus, analysing how much the same artifacts (e.g., SLOC, files or commits) appear in different contexts (e.g., files, commits or source code repositories). We observe a combinatorial explosion in the number of identical source code files across different commits.

To discuss the implication of these findings, we benchmark different data models for capturing software provenance information at this scale, and we identify a viable solution, based on the properties of isochrone subgraphs, that is deployable on commodity hardware, is incremental and appears to be maintainable for the foreseeable future. Using these properties, we quantify, at a scale never achieved previously, the growth rate of original, i.e. never-seen-before, source code files and commits, and find it to be exponential over a period of more than 40 years.

Keywords software evolution, open source, clone detection, source code tracking, mining software repositories, provenance tracking

G. Rousseau
Université de Paris, France, E-mail: guillaume.rousseau@univ-paris-diderot.fr

R. Di Cosmo
Inria and Université de Paris, France, E-mail: roberto@dicosmo.org

S. Zacchiroli
Université de Paris and Inria, France, E-mail: zack@irif.fr

1 Introduction

Over the last three decades, software development has been revolutionized under the combined effect of the massive adoption of free and open source software (FOSS), and the popularization of collaborative development platforms like GitHub, Bitbucket, and SourceForge [48], which have sensibly reduced the cost of collaborative software development and offered a place where historical software can be stored [47]. One important consequence of this revolution is that the source code and development history of tens of millions of software projects are nowadays public, making an unprecedented corpus available to software evolution scholars. We will refer to the full extent of this corpus as *public source code* in this paper.

Many research studies have been conducted *on subsets* of public source code, looking for patterns of interest for software engineering, ranging from the study of code clones [46, 51, 52] to automated vulnerability detection and repair [23, 32, 36], from code recommenders and reuse [27, 60, 61] to software licence analysis and compliance [17, 56, 58].

An important building block for several of these studies is the ability to *identify the occurrences* of a given file content, or a subpart of it, in the reference corpus, also known as *provenance tracking* [20]. For example, when a vulnerability is identified in a source code file [18] it is important to find *other occurrences of the exact same file content*, both in other versions of the same project and in different projects; similarly, when analyzing code clones or software licenses, it is important to find *the first occurrence* of a given exact source file, or a subpart of it.

Scaling up similar studies to the entire body of public source code, and making them reproducible, is a significant challenge. In the absence until recently of common infrastructures like Software Heritage [1, 14] and World of Code [34] that provide *reference archives* of public source code development, scholars have used popular development platforms like GitHub as a reference corpus. But development platforms are not archives: projects on GitHub come and go,[1] making reproducibility a moving target. And while GitHub is the most popular development platform today, millions of projects are developed elsewhere, including very high-profile ones like GNOME.[2] Software Heritage [1, 14] —with its mission to collect, preserve, and make all public source code accessible together with its development history—offers an opportunity to change this state of affairs. The project has amassed the largest public source code corpus to date, with more than 80 million software projects archived from GitHub, GitLab, PyPI, and Debian, growing by the day.

In this paper we leverage Software Heritage to perform a study addressing source code provenance tracking at this unprecedented scale. This work is part of a larger program whose objective is to build in the medium term a generic

[1] For example, hundreds of thousands of projects migrated from GitHub to GitLab.com in the days following the acquisition of GitHub by Microsoft in Summer 2018, see `https://about.gitlab.com/2018/06/03/movingtogitlab/`.

[2] See `https://www.gnome.org/news/2018/05/gnome-moves-to-gitlab-2/`

and easily deployable infrastructure to allow research teams from different communities to conduct a broad spectrum of analysis on top of the Software Heritage archive.

The Software Heritage corpus is stored in a Merkle Direct Acyclic Graph (DAG) [37], which offers several key advantages for large-scale provenance tracking: it reduces storage requirements by natively deduplicating *exact file clones*, that are quite numerous [33, 38]; it provides a means of *checking integrity* of the archive contents; and offers a *uniform representation* of both source code and its development history, independently of the development platform and version control system from which they were collected.

Research questions. In the first part of the article we focus on the various *layers* contained in the Software Heritage corpus, and study the number of *different contexts* in which original code artifacts re-appear over and over again, e.g., the same unmodified source code file found in different commits, or the same commit present in different repositories, leading to our first research question:

RQ1 To what extent are the same source code artifacts, and in particular lines of code, files, and commits, *replicated* in different contexts (files, commits, and repositories, respectively) in public source code?

By quantifying the *replication factor* of public source code artifacts, we find evidence of a combinatorial explosion in the number of contexts in which original source code artifacts appear, which is particularly significant in the replication of identical source code files across different commits.

In order to better characterize this phenomenon and its practical implications, we also address the following two sub-questions of **RQ1**:

RQ1.1 What is the impact of file size on the file replication factor?
RQ1.2 How does the replication factor of commits relate to forks and repository size?

We show that excluding small files reduces the amplitude of the observed combinatorial explosion, but does not remove it, showing that this phenomenon is not only related to the large dispersion of small generic files across repositories.

In the second part of this work we explore the implications of such a huge replication factor on the problem of *software provenance tracking* [19,20] at the massive scale of public source code, addressing our second research question:

RQ2 Is there an efficient and scalable data model to track software source code provenance at the level of individual files at the scale of public source code?

To address this practical question we evaluate three different data models for storing provenance information, which offer different space/time trade-offs. We evaluate these three data models on more than 40 years of public source code development history and find that one of them—which we call the *compact model*—allows to concisely track provenance across the entire Software

Heritage archive, both today and in the foreseeable future. The *compact* model achieves its performance by exploiting the creation time of software artifacts in the Software Heritage graph, and more precisely the properties of the *boundary* of subgraphs called *isochrone subgraphs*, which we introduce later in this paper.

In the last part of the paper, leveraging the fast scanning of the isochrone subgraph boundaries, we perform a large scale analysis of the evolution of public source code. We look into the production of *original* source code artifacts over time, that is, the amount of source code files and commits that have never been observed before (e.g., in other version control system (VCS) repositories or published tarballs and distribution packages) across the entire Software Heritage corpus, addressing our third research question:

RQ3 How does the public production of *original*—i.e., never seen before at any given point in time in Software Heritage—source code artifacts, and in particular files and commits, evolve over time? What are their respective growth rates?

To answer this question we perform an extensive study of the Software Heritage archive, continuing a long tradition of software evolution studies [10, 12,25,26,35], which we extend here by several orders of magnitude and perform over a period of more than 40 years. We show evidence of a remarkably stable *exponential growth rate* of original commits and files over time.

Paper structure. We review related work in Section 2 and make a short presentation of the Software Heritage dataset in Section 3. Section 4 explores **RQ1**, studying the replication factor of original source code artifacts across different contexts; Section 5 is dedicated to **RQ2**, studying data models for tracking provenance of artefacts and leading to the compact model, which is experimentally validated in Section 6. We address **RQ3** in Section 7 by computing the growth factor of public source code. Threats to validity are discussed in Section 8.2 before concluding with Section 9.

Replication package. Given the sheer size of the Software Heritage archive (≈200 TB and a ≈100 B edges graph), the most practical way to reproduce the findings of this paper is to first obtain a copy of the official Software Heritage Graph Dataset [41] and then focus on the source code revisions that we have analyzed for this paper. The full list of their identifiers is available on Zenodo (DOI 10.5281/zenodo.2543373) (20 GB); the selection criteria are described in Section 7.

2 Related Work

In this section we compare the present work with relevant literature in the field, starting with a discussion on the approaches to provenance tracking and considering then each of the stated research questions.

2.1 Software Provenance Tracking

The term *provenance* is generally used to denote the lineage of an artefact, which includes identifying its origin, what happens to it, and where it moves to over time.

Depending on the intended application and field of use, provenance can be looked at various granularities [4, 13]. On the finest granularity end of the spectrum, tracking the origin of programming building blocks like functions, methods or classes, code snippets, or even individual lines of code (SLOC) and abstract syntax trees (AST) [4], is useful when studying coding patterns across repositories [5, 17]. On the opposite end, at the coarsest granularity, tracking the origin of whole repositories is useful when looking at the evolution of forks [7, 28, 42, 53] or project popularity [8].

In between, tracking *file*-level provenance has been for more than a decade (and still is) the state-of-the-art in industrial tools and services to monitor compliance with open source license terms and detect presence of security vulnerabilities known in public vulnerability databases that need to be addressed. Companies like BlackDuck, Palamida, Antelink, nexB, TripleCheck, or FossID are actors and have developed patent portfolios in that domain [31, 44, 57]. Tools like FOSSology [55] and ScanCode are their open source counterparts. With few notable exceptions [3, 20, 33], file-level provenance has received little attention in the research community.

2.2 Public Source Code Replication (RQ1)

A few studies consider the amount of code duplication induced by the now popular pull-request development model [22] and more generally by the ease with which one can create copies of software components, even without forking them explicitly on collaborative development platforms.

The amount of exogenous code in a project can be extremely important, as shown in [33], which analyzed over 4 million non-fork projects from GitHub, selected by filtering on a handful of programming languages (Java, C++, Python, and JavaScript), and showed that almost 70% of the code consists of exact file-level clones. This remarkable work provides a very interesting picture of code cloning in a subset of GitHub at the precise moment in time it was performed, but did not study how cloning evolves over time, nor how it impacts the growth of the public source code corpus.

The approach proposed in this paper, which covers both the full granularity spectrum and does so at the scale of Software Heritage is, to the best of our knowledge, novel. It provides a clear overview of the combinatorial challenge that arises in tracking software source code provenance in the real world of public source code.

2.3 Large-scale Provenance Tracking (RQ2)

Until very recently most of the efforts to build very large source code corpuses—i.e., comparable in size to the full extent of public source code—and associated analysis infrastructures were maintained by private companies providing licence compliance solutions.[3] With the exception of the seminal work by Mockus [39], very limited information on data models and infrastructure to maintain such corpuses is available. Only recently academic work [1, 14, 34, 41] has highlighted the importance of building such large corpuses, making them more accessible, and aiming at being as exhaustive as possible both in terms of software distribution technologies (e.g., different VCS) and code hosting platforms (e.g., different forges).

Different approaches are possible to track provenance at such a scale. Due to the sheer size of the dataset, it is natural to look for a preprocessing phase that precomputes provenance information for the whole dataset. Ideally, preprocessing should be *incremental*, precomputed information should be *complete*, space occupation should be *linear* w.r.t. the original corpus, and access to this information should happen in *constant time*. We are not aware of any approach that satisfies all these properties. The main design challenge for such systems is to determine, for a given set of use cases, the best compromise between preprocessing time and storage cost, and the gain obtained over different design choices.

The simplest approach, which enjoys the smallest storage fingerprint, represented by the *recursive* model in this article, is to not do any preprocessing at all, and just traverse the original dataset for every provenance query. As we will see, this may lead to *prohibitive execution time*.

Another approach, corresponding to the *flat* model in this article, consists in building a complete index associating each artefact to (all) its origins: this index can be maintained incrementally, provides constant access time, and works well on smaller corpuses [4], but we will see that it leads to *prohibitive space occupation* for the index, unless one decides to *give up on completeness*, and drop contents that appear too frequently, as is done in [34].

The new approach that we describe in this article, that we call the *compact* model, offers an interesting trade-off: it can be built and maintained *incrementally* and allows to keep *complete* provenance information. It provides an access time that is not constant, but is still way faster than the recursive model. Its storage fingerprint is not linear, but is way smaller than the flat model. It can be compared to what has been developed in other communities, such as those studying the properties of complex networks [30, 40].

The Software Heritage dataset [41], as the first publicly accessible dataset containing several billions of fingerprints of source code artifacts, allows to benchmark these different approaches on a reference real-world dataset, which

[3] Each claiming to have the largest knowledge base of software artifacts, see for example `https://en.wikipedia.org/wiki/Open_Hub`, `https://www.theserverside.com/discussions/thread/62521.html`

can be considered to be a good proxy of public source code in the context of the present work.

2.4 Growth of Public Source Code (RQ3)

The study of software evolution has been at the heart of software engineering since the seminal "Mythical Man Month" [9] and Lehman's laws [29]. The tidal wave of FOSS, making a growing corpus of publicly available software available, has spawned an impressive literature of evolution studies. Some 10 years ago a comprehensive survey [12] showed predominance of studies on the evolution of individual projects. Since then large-scale studies have become frequent and the question of how Lehman's laws need to be adapted to account for modern software development has attracted renewed attention, as shown in a recent survey [26] that advocates for more empirical studies to corroborate findings in the literature.

While Mining Software Repositories (MSR) research [24] is thriving, realizing large-scale empirical studies on software growth remains a challenging undertaking depending on complex tasks such as collecting massive amounts of source code [38] and building suitable platforms for analyzing them [16, 54]. Hence, up to now, most studies have resorted to selecting relatively small subsets[4] of the full corpus, using different criteria, and introducing biases that are difficult to estimate. For instance, an analysis of the growth of the Debian distribution spanning two decades has been performed in [10], observing initial superlinear growth of both the number of packages and their size. But Debian is a collection maintained by humans, so the number of packages in it depends on the effort that the Debian community can consent.

A recent empirical study [25] has calculated the compound annual growth rate of over 4000 software projects, including popular FOSS products as well as closed source ones. This rate is sensibly in the range of 1.20–1.22, corresponding to a *doubling in size every 42 months*. In this study, though, the size of software projects was measured using lines of code, without discriminating between original contents and refactored or exogenous code reused as-is from other projects.

The results we present here are to the best of our knowledge the first to explore the evolution of growth of publicly available software at very large scale over four decades.

3 Data Model and Dataset

In this section we present the data model and reference dataset used for the experiments discussed in the rest of the paper.

[4] Some studies have analyzed up to a few million projects, but this is still a tiny fraction of all public source code.

3.1 Data Model

Our *reference dataset* is extracted from Software Heritage. A toy yet detailed example of the corresponding data model is depicted in Fig. 1. The key principle is to deal with source code artifacts—collected from public version control systems (VCS), package manager repositories, and as many source code distribution places as possible—by storing them in a single, global (and huge) Merkle direct acyclic graph (DAG) [37], where all nodes are thoroughly deduplicated. The graph is typed in the sense that it contains different types of nodes. We recall in the following the main Software Heritage node types and their properties, referring the reader to [1, 14] for more details.

Terminological note. Software Heritage adopts a technology-neutral terminology which is now uncommon in the literature; in particular they use "revisions" for what is more commonly referred to as "commits" and use "contents" for *file* contents (with no attached metadata, including filenames) that are more commonly referred to as "blobs". In this article we adopt the more common terminology, i.e., blobs and commits, but below we *also* mention the Software Heritage terminology for ease of cross-referencing with existing bibliography on the archive data model.

The following node types can be found in the Software Heritage Merkle DAG:

Blobs (or *contents*, in Software Heritage terminology) raw file contents as byte sequences. Blobs are anonymous; "file names" are given to them by directories and are hence context dependent.

Directories lists of named directory entries. Each entry can point to blobs ("file entries"), recursively to other directories ("directory entries"), or even commits ("commit entries"), capturing links to external components like those supported by Git submodules and Subversion externals. Each entry is associated to a name (i.e., a relative path) as well as permission metadata and timestamps.

Commits (or *revisions*, in Software Heritage terminology) point-in-time states in the development history of a software project. Each commit points to the root directory of the software source code at commit time, and includes additional metadata such as timestamp, author, and a human-readable description of the change.

Releases (also known as *tags* in some VCS) particular commits marked as noteworthy by developers and associated to specific, usually mnemonic, names (e.g., version numbers or release codenames). Releases point to commits and might include additional descriptive metadata, e.g., release message, cryptographic signature by the release manager, etc.

Snapshots lists of pairs mapping development branch names (e.g., "master", "bug1234", "feature/foo") to commits or release nodes. Intuitively each snapshot captures the full state of a development repository, allowing to recursively reconstruct it if the original repository gets lost or tampered with.

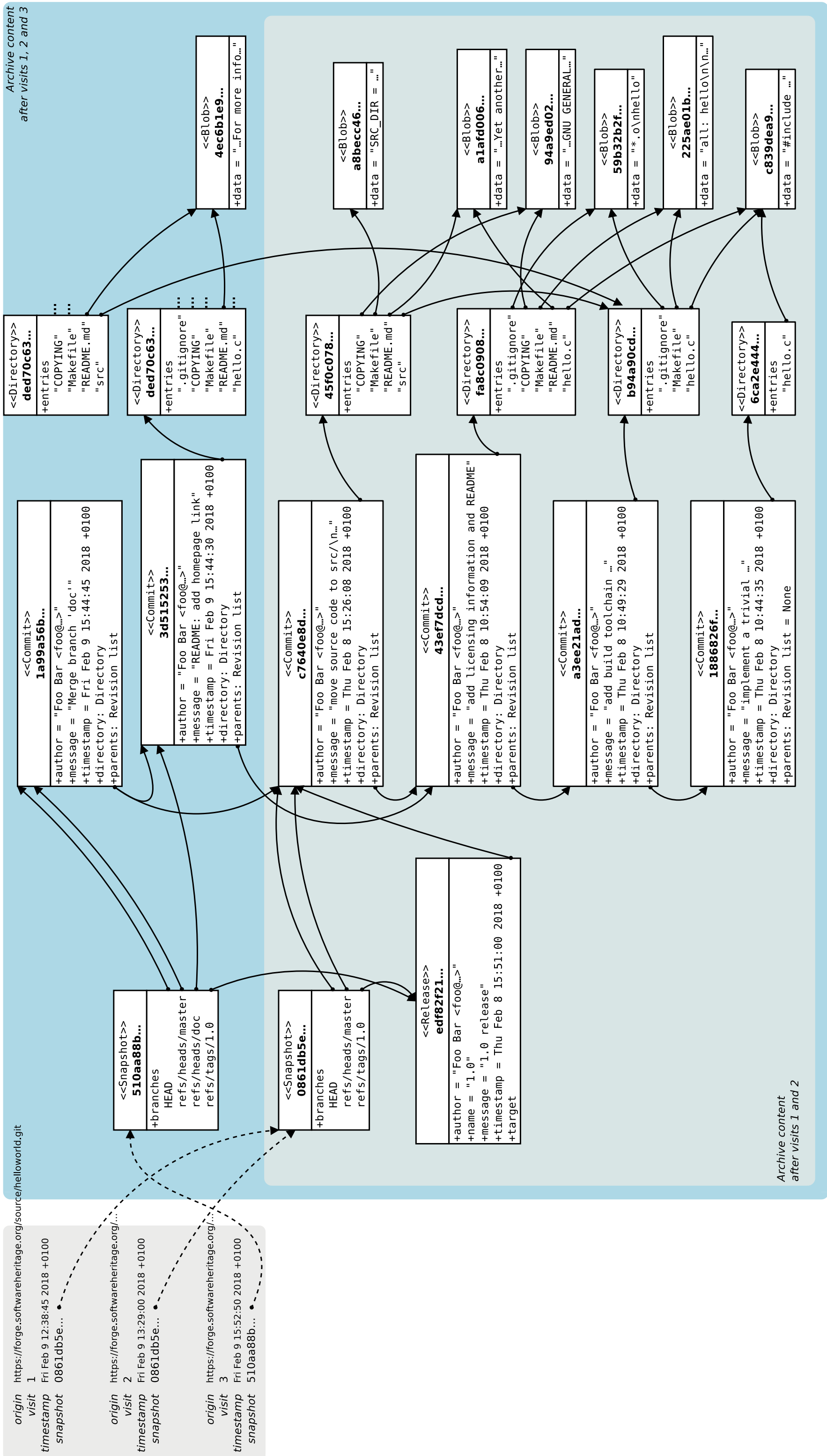


Fig. 1 Software Heritage data model: a Merkle direct acyclic graph (DAG) of public source code artifacts—file blobs, directories, commits, releases, project snapshots—equipped with crawling information of where (repositories, distribution packages, etc.) they have been observed.

Deduplication. In this data model, identical artefacts are coalesced in a single node, for all supported source code artifacts. Each blob is stored exactly once, no matter how many directories point to it, and referred to via cryptographic checksum key from multiple directories; each commit is stored once, no matter how many repositories include it; up to each snapshot, which is stored once no matter how many identical copies of repositories in exactly the same state (e.g., pristine forks on GitHub) exist.

This arrangement allows to store in a uniform data model both specific versions of archived software (pointed by release nodes), their full development histories (following the chain of commit nodes), and development states at specific points in time (pointed by snapshot nodes).

In addition to the Merkle DAG, Software Heritage stores *crawling information*, as shown in the top left of Fig. 1. Each time a source code origin is visited, its full state is captured by a snapshot node (possibly reusing a previous snapshot node, if an identical repository state has been observed in the past) plus a 3-way mapping between the origin (as an URL), the visit timestamp, and the snapshot object, which is then added to an append-only journal of crawling activities.

3.2 Dataset

For this paper we used the state (called *reference dataset* in the following) of the full Software Heritage archive as it was on February 13th, 2018. In terms of raw storage size, the dataset amounts to about 200 TB, dominated by the size of blobs. As a graph, the DAG consists of ≈9 B nodes and ≈100 B edges, distributed as shown in Table 1.

Table 1 Descriptive graph size statistics about the reference dataset used in this paper: a Software Heritage archive copy as of February 13th, 2018.

(a) archive coverage

46.4 M software origins

(b) nodes

node type	**quantity**
blobs	3.98 B
commits	943 M
releases	6.98 M
directories	3.63 B
snapshots	49.9 M
total	8.61 B

(c) edges

edge type	**quantity**
commit → directory	943 M
release → commit	6.98 M
snapshot → release	200 M
snapshot → commit	635 M
snapshot → directory	4.54 K
directory → directory	37.3 B
directory → commit	259 M
directory → blob	64.1 B
total	103 B

At the time we used it for this paper, the Software Heritage archive was the largest available corpus of public source code [1,14], encompassing:

- a full mirror of public repositories on GitHub, constantly updated
- a full mirror of Debian packages, constantly updated
- a full import of the Git and Subversion repositories from Google Code at shutdown time
- a full import of Gitorious at shutdown time
- a one-shot import of all GNU packages (*circa* 2016)

This corpus is orders of magnitudes larger than previous work on provenance tracking [10,33,35].

4 Public Source Code Replication

We now look into *public source code replication*, i.e., how often the same artifacts (re-)occur in different contexts. Fig. 2 depicts the three layers of this phenomenon: a given line of code (SLOC) may be found in different source code file blobs; a given file blob may appear in different commits (in the same or different repositories); and a given commit may be found at multiple origins (e.g., the same commit distributed by multiple repositories and source packages).

To study this phenomenon and answer **RQ1** we perform in the following a focused analysis on the Software Heritage Merkle DAG. This will lead to quantitatively evaluate the *replication factor* of source code artifacts at each replication layer of Fig. 2.

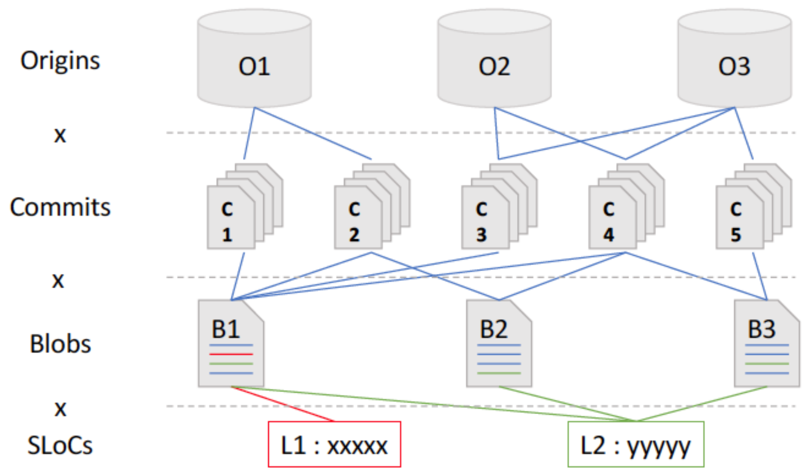


Fig. 2 The three layers of replication in public source code: SLOC occurring in source code files (blobs), blobs occurring in commits, commits found at different distribution places (origins).

4.1 Blob Replication Factor

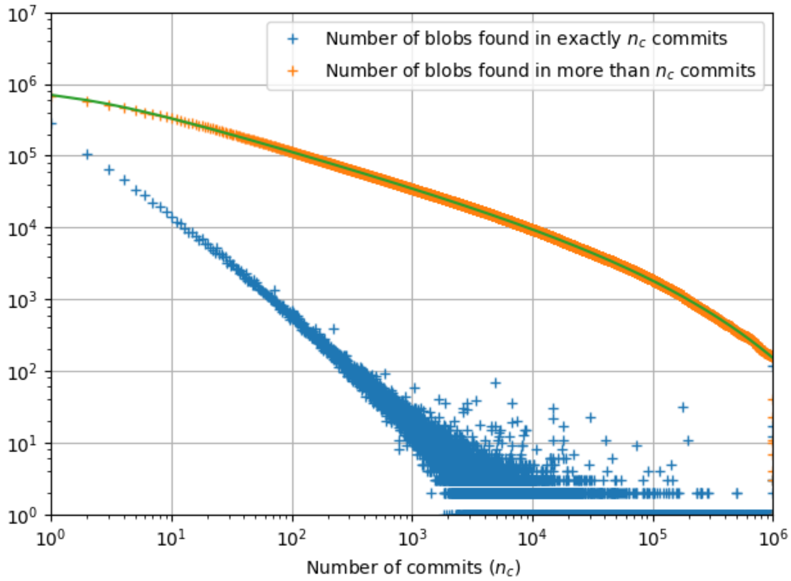
In order to assess the *blob* replication factor, i.e., how often the same file content appears in different commits, we took a random sample of about 1 million unique blobs. Since the cryptographic hash used in the Software Heritage Merkle DAG is uniformly distributed, this can be done easily by selecting all blobs whose hash has a given initial prefix: our random sample is made up of all blobs whose hash identifiers start with `aaa`. For each blob in that sample we counted how many commits contain it in the reference dataset. The resulting distribution of the replication factor is shown in the upper part of Fig. 3, together with the corresponding cumulative distribution.

Looking at the cumulative distribution it jumps out that the average replication factor is very high. It exhibits a characteristic decreasing power law ($\alpha \simeq -1.5$), only limited by an exponential cut-off. There are hence over a hundred of thousand blobs that are duplicated more than one hundred times; tens of thousand blobs duplicated more than a thousand times; and there are still thousands of blobs duplicated *more than a hundred thousands times*! Space-wise, keeping track of all the occurrences of the blob→commit layer of Fig. 2 is a highly nontrivial task.

We address **RQ1.1** by investigating the impact of file size on blob replication factor. We took two random samples of about 1 million blobs each, one with blob sizes up to 100 bytes and one with sizes between 10^5 and 10^6 bytes, and performed again the previous analysis.

The resulting normalized cumulative replication factors are shown on the bottom of Fig. 3. We can see that *the replication factor of small blobs is much higher* than that of average-sized and large blobs.

Hence, keeping track of the blob→commit occurrences only for files larger than, say, 100 bytes, would significantly simplify the problem with respect to the general case, as the total number of blob occurrences across different commits would be significantly lower. Omitting small files is indeed a technique often used by state-of-the-art industry solutions for software provenance tracking: Fig. 3 provides evidence on why it is effective (at the expense of completeness).



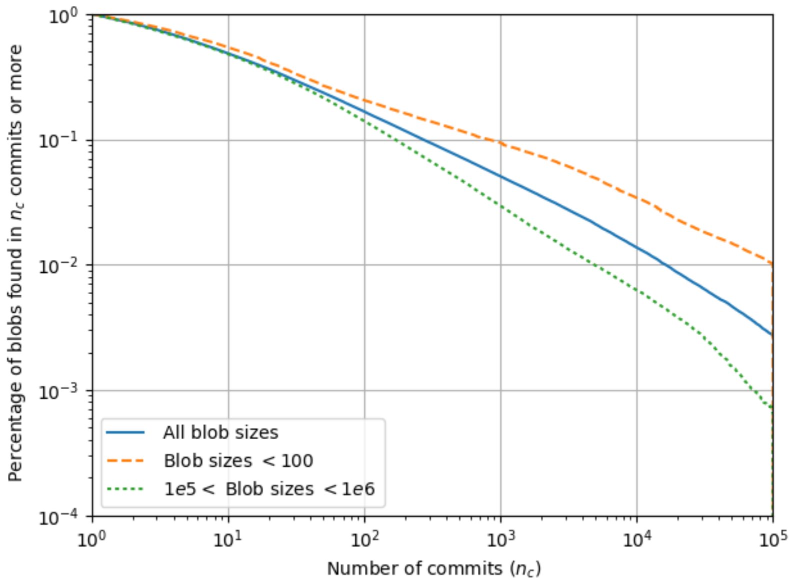


Fig. 3 Top: cumulative (upper curve) and simple (lower curve) replication factor of unique file blobs across unique commits. Bottom: normalized cumulative blob replication factor for the same sample (solid line) and two random samples of about 1 M blobs each, with blob sizes up to 100 bytes (dashed line) and between 10^5 and 10^6 bytes (dotted line).

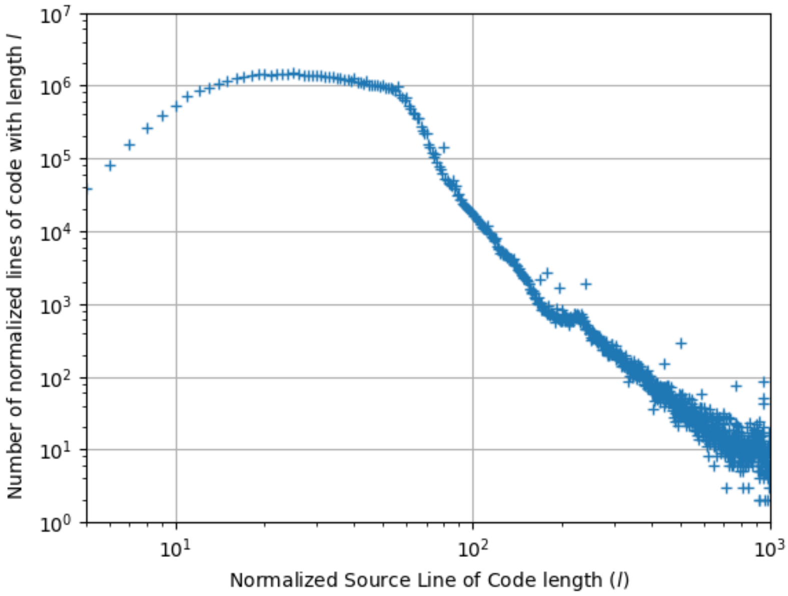


Fig. 4 Distribution of the length of normalized SLOC in a sample of 2.5 M blobs that appear at least once with `.c` extension.

4.2 SLOC Replication Factor

We now turn our attention to the bottom layer of Fig. 2: SLOC→blob. Since lines of code are hardly comparable across languages, we focused on the C language, which is well-represented in the corpus.

We took a random sample of ≈11.4 M unique blobs occurring in commits between 1980 and 2001, and selected from it blobs that appear at least once with `.c` extension and with sizes between 10^2 and 10^6 bytes, obtaining ≈2.5 M blobs. These thresholds have been determined experimentally with the goal of obtaining a temporally consistent subset of the reference dataset, containing more than 1 million file contents with the desired extension.

Individual SLOC have been extracted by first splitting blobs into physical lines at usual line separators (linefeed and/or carriage return) and then normalizing lines by removing blanks (spaces and tabs) and trailing semicolons (after removing blanks). The obtained *normalized lines* have been compared using byte equality. At the end of this process we obtained ≈64 M normalized SLOC.

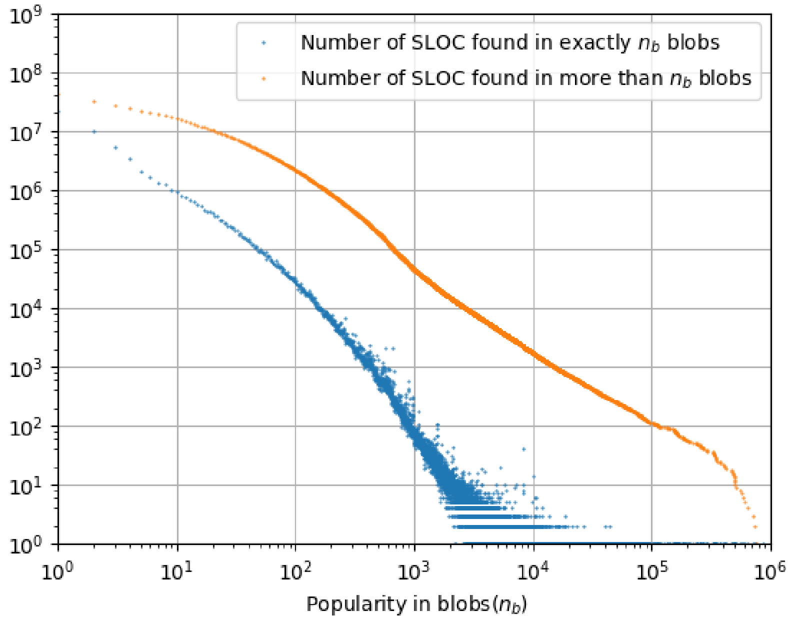


Fig. 5 Replication factor of normalized SLOC as the number of unique blobs they appear in. Dataset: same of Fig. 4.

The replication factor of SLOC across unique blobs is shown in Fig. 5. We observe a much faster decrease w.r.t. the replication factor of blobs in commits ($\alpha \simeq -2.2$), providing evidence that keeping track of SLOC→blob occurrences would be less problematic than blob→commit.

We also computed the distribution of normalized SLOC lengths between 4 and 1000, which is shown in Fig. 4. We observe that lines with length 15 to 60 normalized characters are the most represented, with a fairly stable presence within that range, and a steep decrease for longer lines, confirming previous observations in the field. Hence, for SLOC→blob occurrences there is no obvious length-based threshold that would offer a significant gain.

4.3 Commit Replication Factor

Finally we look into the commit→origin layer of Fig. 2. To that end we took a random sample of ≈6% of all commits) and performed again the previous study of blob replication across commits, this time considering commit replication across origins. Results are shown in Fig. 6.

Commits replication across origins shows larger fluctuations near the end of the range, but decreases steadily before that, and way more steeply ($\alpha \simeq -1.5$)

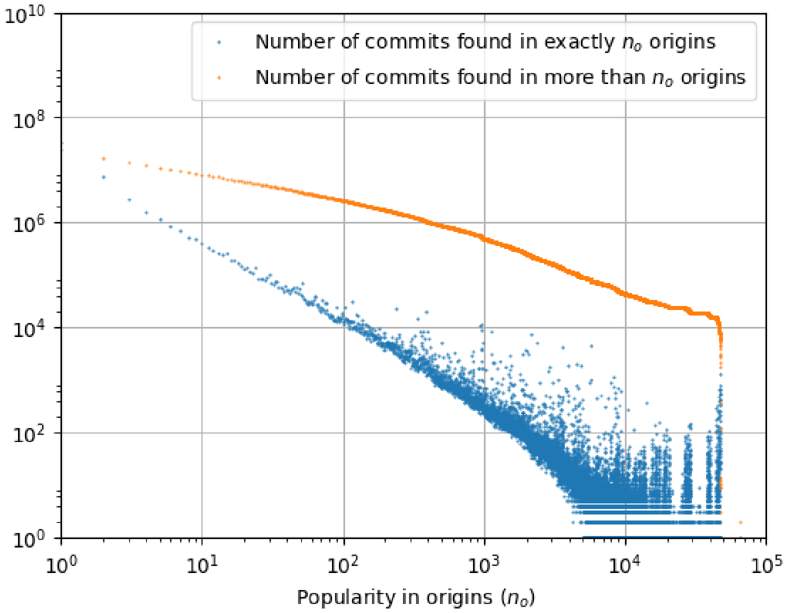


Fig. 6 Replication of commits across origins.

than it was the case for blob→commit replication (see Fig. 3 for comparison): the replication factor of commits across origins is way smaller than that of blobs in commits.

While this result is sufficient to assess the respective impact on public source code replication of the considered layers, we dug further into origin sizes to better understand which origins participate into commit→origin replication, and address **RQ1.2**.

We have considered two different measures of origin size. One that simply counts the number of commits found at each origin. Another that associates commits found at multiple origins *only to the origin that contains the largest number of commits*, and then counted them as before. When a project is forked, the second measure would always report a commit as belonging to the fork with the most active development, which is an approximation of the "most fit fork", while stale forks would decay.

This measure has many good properties: it will follow forks that resurrect projects abandoned at their original development places, it does not rely on platform metadata for recognizing forks, and is hence able to recognize *exogenous forks* across unrelated development platforms (e.g., GitHub-hosted forks of the Linux kernel which is not natively developed on GitHub).

Fig. 7 shows the impact that the "most fit fork" measure has on the number of commit→origin occurrences over the whole dataset. Starting with relatively small repositories, (≈100 commits) the number of occurrences to track is lower than for the simpler measure, with a difference growing up to a full order of magnitude for repositories hosting 10 K commits.

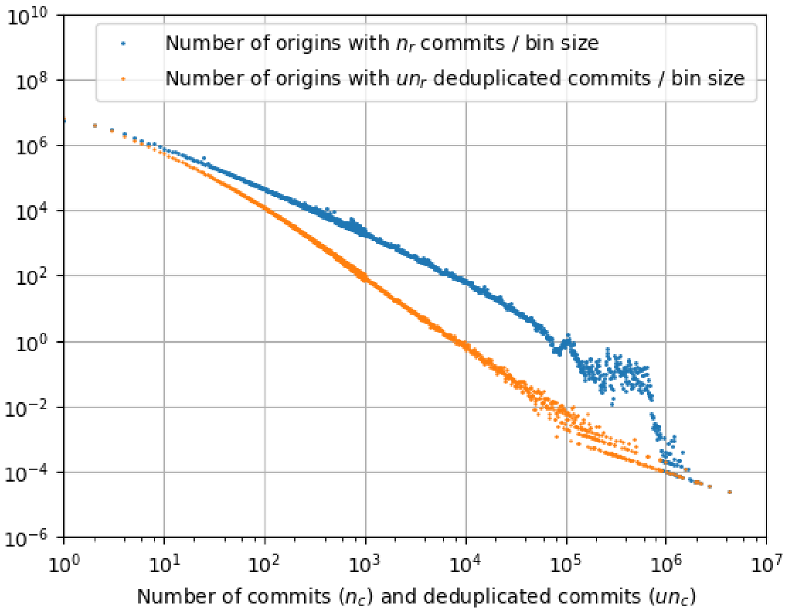


Fig. 7 Distribution of origin size as the number of commits they host.

5 Compact Provenance Modeling

We now consider the problem of tracking software provenance across a corpus as large and as fast growing as all public source code, addressing **RQ2** in this section. In short, the goal is to keep track of all the different places (blobs, commits, origins) in which any given source code artifact (SLOC, blob, commit) can be found—detailed requirements are given below, in Section 5.1.

What are the implications of our findings on public source code growth and replication, on the *feasibility* of maintaining such a complete provenance index? An important fact that emerges from the analyses is that, size-wise, the most challenging part is the layer associating file blobs to all the commits they appear in, because blobs are duplicated across commits much more than commits across origins or SLOC across blobs.

Hence in the following we will focus on concisely representing blob→commit provenance mappings. If we can effectively deal with that replication layer, dealing *also* with the commit→origin and SLOC→blob ones will be compatible and fully modular extensions of the proposed approach.

5.1 Requirements

Supported queries. At least two queries should be supported: first occurrence and all occurrences. The *first occurrence* query shall return the earliest occurrence of a given source code artifact in any context, according to the commit timestamp. The *all occurrences* query will return all occurrences. The two queries answer different use cases: first occurrence is useful for prior art assessment and similar intellectual property needs; all occurrences is useful for impact/popularity analysis and might be used to verify first occurrence results in case of dubious timestamps.

Granularity. It should be possible to track the provenance of source code artifacts at different granularities including at minimum file blobs and commits.

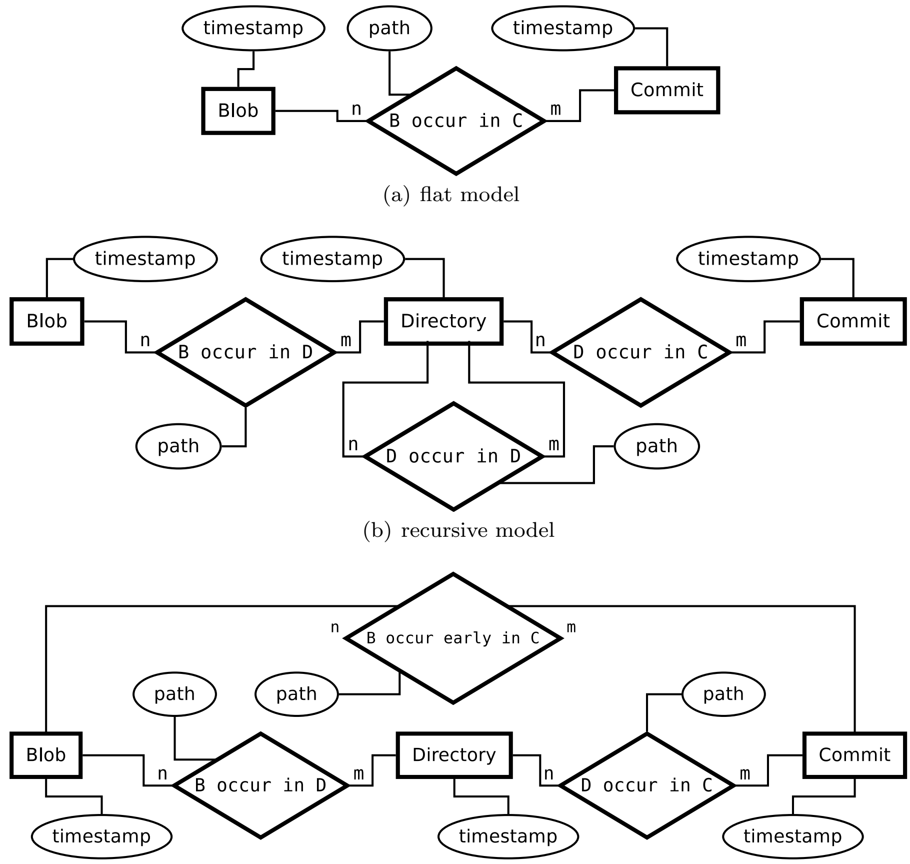
Scalability. It should be possible to track provenance at the scale of at least Software Heritage and keep up with the growth rate of public source code. Given that the initial process of populating provenance mappings might be onerous, and that some use cases require fresh data (e.g., impact/popularity), we also require *incrementality* as part of scalability: the provenance index must support efficient updates of provenance mappings as soon as source code artifacts (old or new) are observed in new contexts.

Compactness. While we put no hard boundaries on total required storage resources, it should be possible to store and query provenance information using state-of-the-art consumer hardware, without requiring dedicated hardware or expensive cloud resources.

Streaming. For the *all occurrences* query a significant and incompressible performance bottleneck is the transfer time required to return the potentially very large result. A viable provenance solution should hence allow to return results incrementally, piping up the rest for later.

5.2 Provenance Data Models

We study three different data models for provenance tracking, that we call respectively *flat*, *recursive*, and *compact*. Their Entity-Relationship (E-R) representations are shown in Fig. 8.



(a) flat model

(b) recursive model

(c) compact model

Fig. 8 Provenance tracking models, entity-relationship (E-R) views

Flat model. This is our baseline for tracking provenance, shown in Fig. 8(a). In this model provenance mappings are “flattened” using a single B(lob) occur in C(commit) relation, that also keeps track of file paths relatively to the root directory of the associated commit. The cardinality of B occur in C is n-m (rather than 1-n), because the same blob can appear multiple times in a given commit at different paths. Each commit carries as attribute the commit timestamp, in order to answer the question of *when* the occurrence happened. Each blob carries as attribute the timestamp of its earliest occurrence, i.e., the minimum timestamps among all associated commits.

Given suitable indexing on blob identifiers (e.g., using a B-tree), the flat model adds no read overhead for the all occurrences query. The same is valid for first occurrence, given suitable indexing on timestamp attributes, which is required to retrieve path and commit.

Updating provenance mappings when a new commit comes in requires traversing the associated directory in full, no matter how many sub-directories or blobs in it have been encountered before, and adding a relationship entry for each of its nodes.

Recursive model. While the flat model shines in access time at the expenses of update time and compactness, the recursive model shown in Fig. 8(b) does the opposite. It is intuitively a "reverse" Merkle DAG representation of the Software Heritage data model (Fig. 1), which maps blobs to directories and directories to commits.

Each entity has a timestamp attribute equal to the timestamp of the earliest commit in which the entity has been observed thus far. When processing an incoming commit r_{t_2} (with timestamp t_2) it is no longer necessary to traverse in full the associated directory: if a node n is encountered that is already present in the model with a timestamp $t_1 < t_2$, recursion can stop because the subtree rooted at n, which is known to be already present due to the Merkle DAG properties, has already been labeled with timestamps earlier than t_2 and needs not to be updated; we just need to add an entry in the corresponding occurrence table for n with timestamp t_2.

Thanks to the sharing offered by the directory level, the recursive model is as compact as the original Merkle structure, with no flattening involved. The all occurrences query is slow in this model though, as for each blob we need to walk up directory paths before finding the corresponding commits. Response time will hence depend on the average directory depth at which queried blobs will be found. First occurrence is faster, but still incurs some read overhead: given a blob we have to walk up all directories and then lookup the corresponding commits whose timestamps equate the timestamp of the blob being queried.

Compact model. Fig. 8(c) shows a compromise version between the flat and recursive models, which is both storage-compact and capable of quickly answering the required queries. The tables for blob, directory, and commit entities are progressively populated as the structure is built, with a timestamp attribute denoting the earliest known occurrence, as before.

To understand how the compact model is built and queried we introduce the following notion:

Definition 1 (Isochrone subgraph) *given a partial provenance mapping $\mathcal{P}$ associating a timestamp of first occurrence to each node in a Merkle DAG, the* isochrone subgraph *of a commit node R (with timestamp t_R) is a subgraph rooted at R's directory that only contains directory nodes whose timestamps in $\mathcal{P}$ are equal to t_R.*

Intuitively, when processing commits chronologically to update the entity tables and the provenance mappings, the isochrone subgraph of a commit starts with its root directory and extends through all directory nodes containing never-seen-before source code artifacts. Due to Merkle properties each directory containing at least one novel element is itself novel and has a fresh intrinsic identifier. Everything *outside* the isochrone subgraph on the other hand has been observed before, in at least one previously processed commit.

Given this notion, the upper part of the compact model (B(LOB) OCCUR EARLY IN C(OMMIT) in Fig. 8(c)) is filled with one entry for each blob attached to any directory in the isochrone subgraph. As a consequence of this, the first occurrence of any given blob will always be found in B OCCUR EARLY IN C although other occurrences—depending on the order in which commits are processed to update provenance mappings—may also be found there.

The relation D(IRECTORY) OCCUR IN C(OMMIT) is filled with one entry, pointing to the commit being processed, for each directory *outside* the isochrone subgraph that is referenced by directories *inside* it, i.e., D OCCUR IN C contains one entry for each directory→directory edge crossing the isochrone frontier. Finally, the relation B(LOB) OCCUR IN D(IRECTORY) is filled with one entry for each blob (recursively) referenced by any directory added to the D OCCUR IN C relation.

Filling the compact model is faster than the flat model: when we reach a directory d at the frontier of an isochrone subgraph, we only need to visit it in full the first time, to fill B OCCUR IN D, and we need not visit d again when we see it at the frontier of another isochrone subgraph in the future.

The compact model is slower than the recursive model though, as we still need to traverse the isochrone subgraph of each commit. Read overhead for first occurrence is similar to the flat model: provided suitable indexing on timestamps we can quickly find first occurrences in B OCCUR EARLY IN C. Read overhead for all occurrences is lower than the recursive model because all blob occurrences will be found via B OCCUR IN D without needing to recursively walk up directory trees, and from there directly linked to commits via D OCCUR IN C.

5.3 Design Rationale and Trade-off

Intuitively, the reason why the compact model is a good compromise is that we have many commits and a very high number of file blobs that occur over and over again in them, as shown in Section 4.1. Consider now two extreme cases: (1) a set of commits all pointing to the same root directory but with metadata differences (e.g., timestamp or author) that make all those commits unique (due to Merkle intrinsic properties); (2) a set of commits all pointing to different root directories that have no file blobs or (sub)directories in common.

In case (1) the flat model would explode in size due to maximal replication. The recursive model will need just one entry in D OCCUR IN C for each commit. The compact model remains small as the earliest commit will be flattened (via B OCCUR EARLY IN C) as in the flat model, while each additional commit will add only one entry to D OCCUR IN C (as in the recursive model).

In case (2) the flat model is optimal in size for provenance tracking purposes, as there is no sharing. The recursive model will have to store all deconstructed paths in D OCCUR IN D. The compact model will be practically as small as the flat model: all commits are entirely isochrones, so the B occur early in C relation will be the same as the B occur in C relation of the flat model, and the only extra item is the Directory table.

Reality will sit in between these two extreme cases, but as the compact model behaves well in both, we expect it to perform well on the real corpus too. The experimental evaluation reported in the next section validates this intuition.

6 Experimental Evaluation

To compare the size requirements of the provenance data models described in Section 5 and address *efficiency* and *scalability* criteria in **RQ2**, we have monitored the growth of each model while processing incoming commits to maintain provenance mappings up to date.

Specifically, we have processed in chronological order commits from the reference dataset with timestamps strictly greater than the Unix epoch (to avoid the initial peak of forged commits that will be shown later in Section 7) and up to January 1st, 2005, for a total of ≈38.2 M commits. For each commit we have measured the number of entities and relationship entries according to the model definitions, that is:

Flat model: one entity for each blob and commit; plus one B occur in C entry for each blob occurrence

Recursive model: as it is isomorphic to the Merkle DAG, we have counted: one entity for each blob, directory, and commit; plus one relationship entry for each commit→directory, directory→directory, and directory→blob edge

Compact model: after identifying the isochrone subgraph of each commit, we counted: one entity for each blob and commit, plus one entity for each directory outside the isochrone graph referenced from within; as well as one relationship entry for each blob attached to directories in the isochrone graph (B occur early in C), one D occur in C entry for each directory→directory edge crossing the isochrone frontier, and one B occur in D entry for each blob present in directories appearing in D occur in C.

These measures are abstract, in the sense that they do not depend on any specific technology used to store provenance information in an information system. They show how much information, measured in terms of E-R elements (entities and relationship entries, respectively) would need to be stored by *any* information system in order to capture the same provenance information, in the various models.

Processing has been done running a Python implementation of the above measurements on a commodity workstation (Intel Xeon 2.10GHz, 16 cores, 32 GB RAM), parallelizing the load on all cores. Merkle DAG information has been read from a local copy of the reference dataset, which had been previously mirrored from Software Heritage. In total, commit processing took about 4 months (or ≈0.3 seconds of per-commit processing time, on average), largely dominated by the time needed to identify isochrone subgraphs.

Table 2 Size comparison for provenance data models, in terms of entities (nodes), relationship entries (edges), and ratios between the amount of relationship entries in the various models. Same dataset of Fig. 9.

	Flat	**Recursive**	**Compact**
entities	80 118 995 commits: 38.2 M blobs: 41.9 M	148 967 553 commits: 38.2 M blobs: 41.9 M directories: 68.8 M	97 190 442 commits: 38.2 M blobs: 41.9 M directories: 17.1 M
relationship entries	654 390 826 907	2 607 846 338 blob–directory: 1.29 B directory–commit: 38.2 M directory–directory: 1.28 B	19 259 600 495 blob–directory: 13.8 B directory–commit: 2.35 B blob–commit: 3.12 B
relationship ratios	$\frac{\text{flat}}{\text{compact}} = 34.0$	$\frac{\text{flat}}{\text{recursive}} = 251$	$\frac{\text{compact}}{\text{recursive}} = 7.39$

Note that this approach is not *globally* optimal in terms of total processing time, as the isochrone graph frontier is re-computed over and over again for related commits. The total processing time would have been shorter if we only calculated the isochrone boundary over the entire graph and then applied updates. This chronological analysis of commit, while not globally fastest, facilitates the monitoring of abstract size requirements for each model as a function of the total size of the monitored corpus.

Measured sizes over time, measured in terms of entities and relationship entries for each model, are given in Table 2. They show, first, that the amount of relationship entries dominate that of entities in all models, from a factor 18 (recursive model) up to a factor 8000 (flat). Dealing with mappings between source code artifacts remains the main volumetric challenge in provenance tracking. As further evidence of this, and as a measure of the overall amplitude of provenance tracking for all public source code, we have also computed the number of relationship entries for the flat data model *on the full reference dataset*, obtaining a whooping $8.5 \cdot 10^{12}$ entries in B OCCUR IN C.

Second, sizes show that the Merkle DAG representation, isomorphic to the recursive model, is indeed the most compact representation of provenance information, although not the most efficient one to query. The compact model is the next best, 7.39 times larger than the recursive model in terms of relationship entries. The flat model comes last, respectively 251 and 34 times larger than recursive and compact.

Fig. 9 shows the evolution of model sizes over time, as a function of the number of unique blobs processed thus far. After an initial transition period, trends and ratios stabilize making the outlook of long-term viability of storage resources for the compact model look sustainable.

Furthermore, the comparison between the compact (orange line) and flat (blue line) model shows that, at the cost of a small increase in the number of entities, the compact model performs much better in terms of relationship

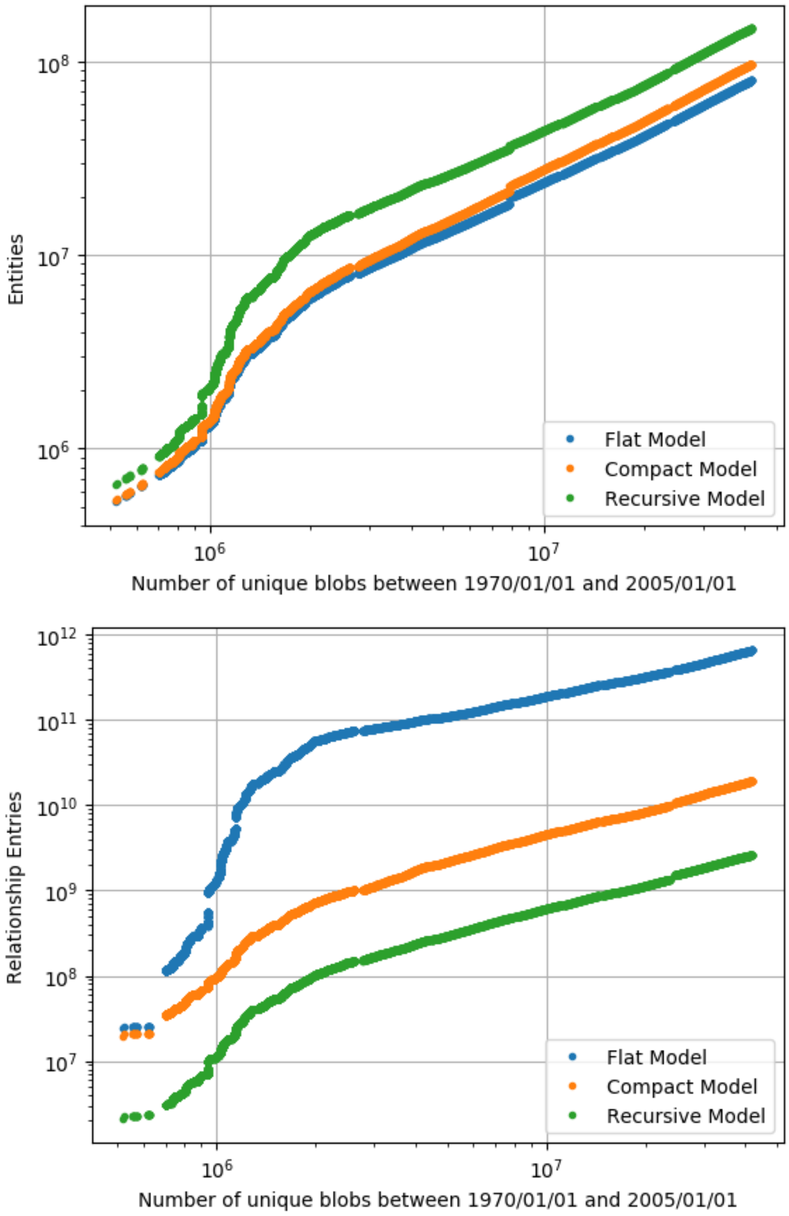


Fig. 9 Evolution over time of the sizes of different provenance data models, in terms of entities (top) and relationship entries (bottom). Data for Software Heritage commits up to 2005-01-01, excluding Unix epoch.

entities. And even if in terms of entities a small divergence can be observed over time ($\frac{1}{10}$ of an order of magnitude), the gain in terms of relationship entries makes it worthwhile (1.5 orders of magnitude).

In order to relate these figures to real-world storage requirements, we have also filled a MongoDB-based implementation of the compact model—including all attributes of Fig. 8(c) and needed indexes—while processing commits to perform the above measurements. Extrapolating the final MongoDB size to the full reference dataset we obtain an on-disk size of 13 TB. While large, such a database can be hosted on a consumer workstation equipped with ≈4000$ of SSD disks, without having to resort to dedicated hardware or substantial investments in cloud resources. Using the compact model, universal source code provenance tracking can lay at the fingertips of every researcher and industrial user.

7 Public Source Code Growth

Current development practices rely heavily on duplicating and reusing code [22, 33], which makes it non trivial to estimate how much *original* software is being produced: summing up the usual metrics—such as number of source code files or commits—across a wealth of software projects will inevitably end up in counting the same original source code artifacts multiple times.

In this section we report on the first large scale analysis of the growth of original software artifacts, in terms of revisions and contents, that we have performed leveraging the fully-deduplicated data model that underlies Software Heritage, and using fast scanning of the boundary of the isochrone subgraphs.

We have analyzed the entire reference dataset (see Table 1), processing commits in increasing chronological order, and keeping track for each blob of the timestamp of the *earliest* commit that contains it, according to commit timestamp. This analysis is done by running on the reference dataset a parallel, incremental version of isochrone subgraph detection (as per Section 5) optimized to keep only the first occurrence of each blob from a set of time-sorted commits.

A commit is considered to be *original* at time t if the combination of its properties (or, equivalently, its identifier in the Merkle DAG) has never been encountered before during processing—i.e, if no commits with a timestamp earlier than t had the same commit identifier. Similarly, a file blob is considered to be original if it has never been recursively referenced by the source code directory of commits with a timestamp earlier than t.

Results are shown in Fig. 10. They provide very rich information, answering **RQ3** for both commits and blobs.

We discuss first a few outliers that jump out. Data points at the *Unix epoch* (1/1/1970) account for 0.75% of the dataset and are clearly over-represented. They are likely due to forged commit timestamps introduced when converting across version control systems (VCS). This is probably also the main reason behind commits with timestamps in the "future", i.e., after the dataset times-

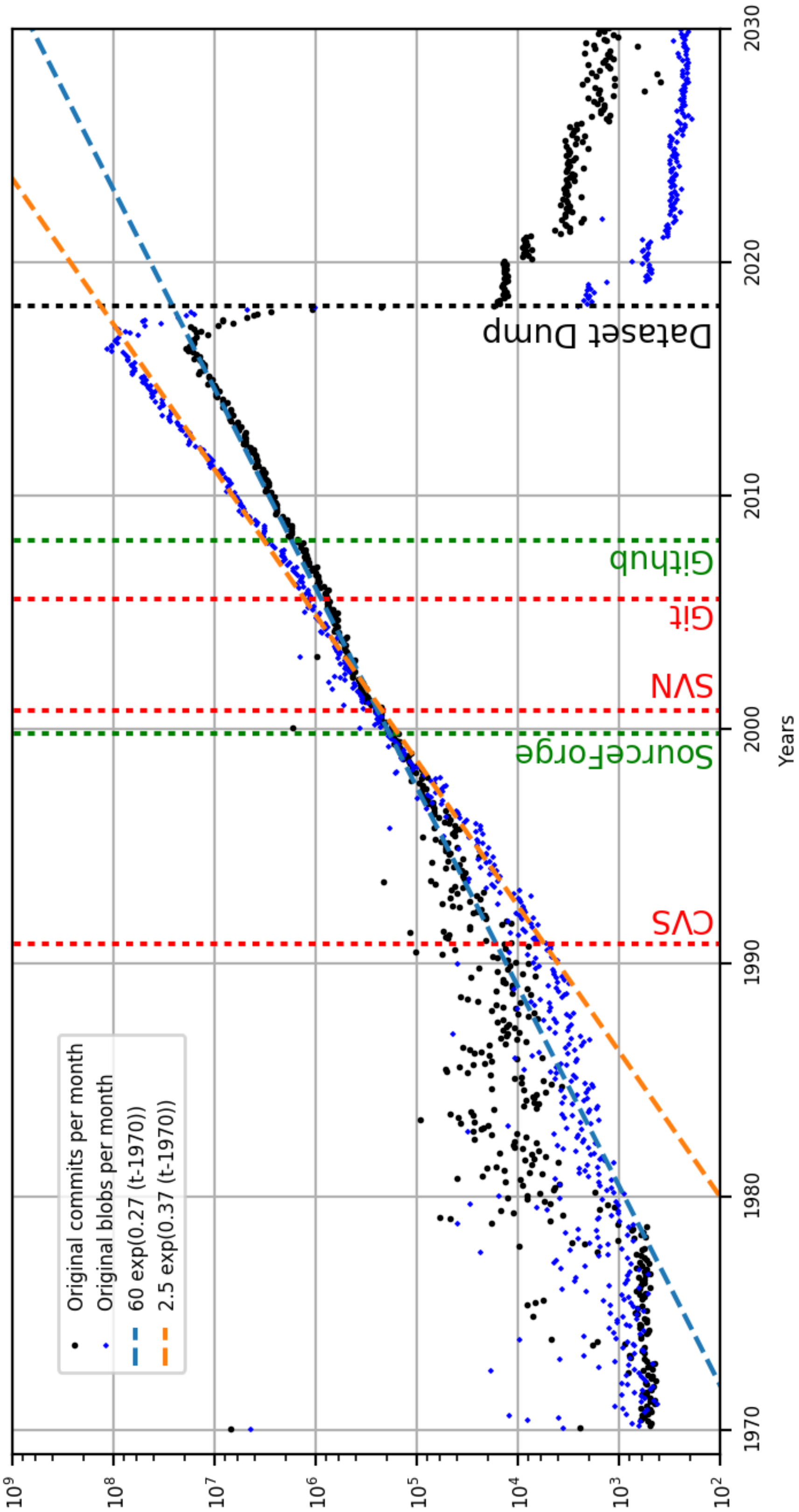


Fig. 10 Global production of original software artifacts over time, in terms of never-seen-before revisions and file contents (lin-log scale). Major events in the history of version control systems and development forges are materialised by vertical bars.

tamp; these commits in the future account for 0.1% of the dataset. The sharp drop before the dataset timestamp is a consequence of the lag of Software Heritage crawlers w.r.t. its data sources.

Focusing on the core part of the figure we remark that in the early years, before the introduction of forges and advanced VCS, the number of commits is relatively small (tens to hundreds of thousands only), and their evolution is rather irregular.

After the creation of the first popular forge, SourceForge (1999), we observe on the other hand a remarkably regular exponential growth lasting twenty years. For original commits, growth can be accurately approximated by the fit line $60e^{0.27(t-1970)}$; at this rate **the amount of original comits in public source code doubles every ≈30 months**. For original blobs, growth is accurately approximated by the fit line $2.5e^{0.37(t-1970)}$; at this rate **the amount of original public source code blobs doubles every ≈22 months**.

Finally, we remark that the *difference* in the growth rates of original commits and original file blobs means that over the past twenty years **the average number of original file blobs *per commit* has been doubling every ≈7 years**. Over time, developers are squeezing more file changes in what is recognizable as commit in the use dataset.

8 Discussion

8.1 Impact

On the granularity and scale of source code provenance tracking. We have explored in this paper the feasibility of tracking software provenance—where a software source code artifact comes from and, more generally, occurs—at various granularities and at the scale that currently best approximates the full body of publicly available source code.

Doing so requires first and foremost assessing the amplitude of the problem (**RQ1**), i.e., how many occurrences one has to track to deal with the problem at the stated scale. To characterize that amplitude we have proposed a layered approach consisting of three different layers: SLOC occurring in individual blobs (i.e., unique source code file contents), blobs occurring in source code trees pointed by commits, commits occurring in software repositories. We have shown qualitatively and quantitatively using the Software Heritage dataset that, although each of this layers contributes to the *replication* of source code artifacts reoccurring in different places, the layer that contribute the most to that replication is blob→commit. We argue that if one can—from both a theoretical complexity and technological point of view—deal with that replication layer, then the problem of source code provenance tracking can be dealt with in its full generality and at all discussed granularities.

We do not argue for the superiority of any particular granularity in source code provenance tracking, as what is "best" will heavily depends on the intended application. The goal of the current work is to study the problem in its more general formulation and at a scale that is as close as possible to the current state of the world of public source code, which led us to focus on the corpus made available by Software Heritage.

By proposing and experimentally validating the compact model for storing provenance information (**RQ2**) we have shown that it is feasible to track provenance information at the scale of Software Heritage, down to the granularity of individual source files. The result is a remarkably efficient trade-off requiring storage resources similar to those of the recursive model, but ensuring faster answer to provenance queries. To the best of our knowledge this is a significant improvement over the current industry state of the art of provenance tracking, where file-level provenance tracking relies on filtering out specific types of blobs, commits or origins based on criteria like size, popularity or presence of releases, and has to give up completeness. The approach we present in this article is to the best of our knowledge the only one that is complete, and amenable to implementation using accessible hardware resources.

The approach we propose is not intended to be compared with the state of the art in *academic* works that require granularities finer than the file, like function or method-level software clone detection [43, 45]. In order to track the evolution and migration of very fine-grained source code parts, these approaches use language-specific techniques, e.g., parsing, to split individual source code blobs into meaningful and individually trackable snippets [46], and as a consequence are tested at much smaller scale on language specific selections of the global corpus.

When addressing public source code, or even "only" very large source code archives [1,34], we are not yet in a position to properly parse source code written in thousands of different programming languages [6]: language recognition alone is a very challenging undertaking with such diversity.

If one wants to perform studies that are independent of the programming language, a natural approach is to split files into SLOCs and track individual (or groups of) SLOCs. We have shown experimental evidence that the replication factor of SLOCs across individual blobs poses a less challenging problem than the one we have dealt with in this paper. Hence, we expect that implementing tracking of SLOCs to be a manageable modular addition to the provenance tracking approach proposed in this article, and we will try to do so in future work.

Sustainable archival and provenance tracking at the scale of public source code. Our study of the growth of original content (**RQ3**), both in terms of blobs and commits, over four decades, shows a remarkably stable *exponential growth*. This calls into question right away the *sustainability* of any effort to build and archive that stores all public source code, like Software Heritage and, more recently and with a narrower scope, World of Code [34]. Based on the findings in this paper, and taking into account the long term evolution of storage costs[5], we see that storage cost is decreasing more rapidly—and has been so for a long

[5] see, e.g., `https://hblok.net/blog/storage/`

time—than the growth rate of the original content that needs to be archived. This provides evidence that full scale source archival is sustainable in the foreseeable future, for archives that adopt a deduplication approach similar to the one used in Software Heritage.

This exponential growth also calls into question the ability to handle provenance tracking not only at the *scale*, but also at the *speed* of public source code growth. Indeed, the fact that one can build the needed data structures *once* at the scale of Software Heritage, does not guarantee *per se* that these can be maintained at the needed speed.

For the recursive model, the answer is easy: since it is just a copy, with arrows reversed, of the Merkle DAG used for the archive, maintaining it is no more difficult than maintaining the archive itself. The downside is that query costs might become even more prohibitive as time passes.

For the compact model, our experimental validation provides several elements to support its long-term sustainability. First, the model is *incremental*: the process we have followed to populate it in Section 6 is the same that will need to be followed to keep it up to date w.r.t. public source code growth. In particular, there is no need to sort all commits by timestamp beforehand, incoming ones can be processed on the fly. Second, the vast majority of incoming commits that need to be added will tend to be naturally chronologically sorted, which is the optimal scenario for the compact model. Indeed, since we deduplicate before processing, old commits that are already archived will not need to be considered, and our findings in the study of software growth over time show that the amount of commits with timestamps incorrectly set to the Unix epoch is marginal. In these condition, which are close to optimal, the long-term storage requirements for provenance tracking using the compact model will be marginal w.r.t. actual archival costs.

8.2 Threats to Validity

Analyzing the largest corpus of source code available today is a very challenging technical undertaking. We discuss here the main threats to the validity of the findings obtained by doing so in this paper, starting from potential internal validity threats and discussing external validity last.

Dataset selection. The main concern for internal validity is that, due to its sheer size, we could not perform all estimates and experiments on the reference dataset, i.e., the full Software Heritage graph dataset. While the results reported in Section 7 are obtained by exploring the full reference dataset, other findings are obtained by analysing smaller subsets.

For some specific experiments, like the data model benchmark, using the full dataset is simply not feasible: testing the flat data model in full would require an inordinate amount of space, due to the replication factor identified in Section 4, so we tested all three models on a subset.

For other experiments, like the estimation of the replication factor across the layers of the Software Heritage archive, our approach to minimize potential bias has been to use sizable subsets obtained by random sampling on the full starting corpus,

Finally, when comparing provenance data models, we have quantitatively estimated sizes, but only qualitatively estimated read overhead—rather than benchmarking it in production—in order to remain technology-neutral. We believe that making complete estimates on a subset and giving indications based on large-scale extrapolations provides sufficient evidence to compare the behaviour of the three different models.

Timestamps trustworthiness. In order to determine the first occurrence of software artifacts, we have have trusted the timestamps provided by the commits themselves, despite the fact that commit timestamps can be forged. On the one hand, this approach is consistent with previous software evolution studies that consider timestamp forging a marginal phenomenon. On the other hand, the correctness of the timestamps is irrelevant for evaluating the performance of the provenance model, as we only need to single out *a* first occurrence, no matter how it is determined.

Deduplication granularity. The data model used by Software Heritage performs deduplication *globally* across all repositories and stops at file level.

This has an impact on what can be easily traceable using the proposed provenance model: while it is possible to determine the occurrences of a given unmodified file content (blob) in the global corpus, it is not possible to identify natively slightly modified variants of a file, like trivial changes such as blanks, which are allowed by Type 1 software clones.

One can foresee various more refined data models and strategies to identify non exact clones, like normalizing blob contents before hashing or, alternatively, by increasing the granularity to SLOC level, but this will lead to an even higher replication factor than what we have observed.

Another limitation of the data model is that it does not contain explicit edges to link a particular file content to its predecessors or successors in a given development history, which would be very useful to answer interesting software phylogeny queries, certainly related to software provenance tracking, but outside the scope of the present work. We remark that it is *possible* to recover those ancestry relations: by following commit chains one can determine when two different blobs (A and B) are pointed by the same path (a common heuristic used in version control systems) in neighbor commits. When that is the case one can establish an ancestry relation “blob A originates from blob B”. In general this will be a many-to-many relationship, as the same blob pairs might be related by different commit chains, possibly occurring in different repositories.

Tracking content at the level of SLOC. The estimate of the replication factor for SLOCs we performed relies on the selection of a specific, albeit very popular, programming language (C), and on the application of a filter that is meant to remove irrelevant formatting for that specific programming language.

In order to build a full provenance tracking model at the granularity of the line of code, of snippers, or of programming language constructs like functions or methods, more work will be needed, possibly involving the use of language specific tools and heuristics.

External validity. The main question about external validity is to what extent the findings we report from an analysis of the Software Heritage archive can be generalised to the full corpus of public source code.

As a first remark, we notice that since Software Heritage does not cover the full extent of public source code, it is quite possible that some given file or commit has been previously accessible on a forge not crawled by Software Heritage.

Nevertheless, we stress the fact that Software Heritage is the largest publicly accessible source code archive in the world and spans the most popular code hosting and development platforms. We therefore consider that this is the best that can be done at present.

We also acknowledge the habit of using *software* development platforms for collaboration tasks other than software development (e.g., collaborative writing), particularly on GitHub, but we did not try to filter out non-software projects. On the one hand we expect software development to remain the dominant phenomenon on major forges, and on the other hand non-software projects might still contain interesting code snippets that are worth tracking. Also, as demonstrated in the paper, it is not *necessary* to filter out non-software project in order to build a practical provenance tracking solution.

Finally, there is a time lag in the archival of public source code artifacts that is clearly visible in Fig. 10. We consider it to be a delay induced by Software Heritage crawling activities, and this bias may be corrected by excluding data too close to the dataset dump date.

8.3 Future Work

To the best of our knowledge this work is the first establishing the feasibility of source code provenance tracking at file granularity and at a scale comparable to the full body of public source code. We point out here some new research and practical questions that this result brings up.

Source code artifact multiplication. From the point of view of characterizing the replication factor, we have scratched the surface addressing the need of estimating the *total* number of occurrences that need to be tracked at the various replication layers—as that is the challenge that needs to be faced in order to solve provenance tracking in its full generality. It would be very interesting to refine the analysis we have done by adding control variables related to different kinds of projects (e.g., small v. large, active v. inactive, mobile apps

v. server software v. desktop applications) and development models (e.g., pull-request based v. direct push, truly distributed v. centralized), to determine whether they contribute differently, and how, to the multiplication of source code artifacts that need tracking. This may lead to insights that would help in addressing provenance tracking at smaller scales that what we have studied here, e.g., in large in-house collaborative development efforts as those being proposed in the context of inner source [11, 49].

Software provenance networks. To measure the replication factor in the various layers we have *de facto* established new derived graphs on top of the starting Merkle DAG data model; they map source code artifacts to where they can be found in the wild. The growth rate of original blobs and commits suggests that both the original Merkle DAG graph and these new derived graphs are interesting evolving complex networks, naturally occurring as byproducts of the human activity of developing and distributing software, like others that have been studied in the past [2, 15].

The topology and nature of these networks (scale-free or not, small-world or not, etc.) as well as the modeling of how they evolve over time, which we have started to observe in this work, are important and actionable research subjects. For instance, it would be interesting to know whether phenomena like preferential attachment [2] (which has been observed in other technology related network such as the Web), potentially leading to accelerating growth, play a role also in these cases or not.

SLOC provenance tracking. While we have established the theoretically feasibility of operating a global provenance database at file-level granularity, and provided early evidence that SLOC-level tracking is also doable due to its smaller replication footprint, we have not actually tried to operate a global provenance tracking database at SLOC granularity at the proposed scale. We intend to do so in future work and provide a return of experience on the practical challenges that need to be faced.

Long-term growth. We have shown that it is possible to study the growth of public source code over four decades at the scale of Software Heritage, staying at the level of commits and file contents (blobs).

This naturally leads to the question of performing a similar study at finer granularities, like that of lines of codes, and see how the growth rate at this finer level compartes to the growth rates we have found for commits and blobs.

The discrepancies in the growth rate observed at the different layers of the Software Heritage archive (commits v. blobs in our study, likely SLOC v. commits v. blobs in the future) naturally lead to questions about the long-term evolution of developer productivity and habits. Why do developers change steadily varying number of files in their commits? Is it due to the qualities of the tools (e.g., editors, VCS) that become available? Or are there other variables affecting this long-term observable change? Answering these research questions will allow to get a better understanding of the dynamics of the creation of new software artifacts.

9 Conclusion

We have considered the problem of tracking the provenance of source code, i.e., keeping track of *where* individual source code artifacts can be found within a reference corpus. We have attacked that challenge at file-level granularity and at global scale, using as reference corpus the best available approximation of the entire body of publicly available source code, spanning major forges and free/open source software distributions via the Software Heritage archive, for a total of more than 40 million projects.

Our first contribution is a qualitative and quantitative assessment of the amount of replication of original file contents (i.e., never observed before in the reference corpus at any given point in time) across different commits and of original commits across different software distribution places (e.g., VCS repositories or distribution packages). The findings presented in this article offer insights into the deep structure of public software development and distribution, and allow to establish that the blob→commit layer is the one presenting the most significant combinatorial explosion. Hence, in order to address the provenance tracking problem at the scale of public source code in its full generality, one can focus on finding and efficient data model for this layer.

The second contribution is the development and comparison of three data models designed to answer the software provenance questions of “what are the first/all occurrences of a given file blob/commit in the reference corpus?”. We introduce a novel *compact model* for provenance tracking, based on the notion of isochrone subgraphs, and establish experimentally that it offers a time/space trade-off that allows to track software provenance at the scale of Software Heritage on cheap, consumer-grade hardware.

The third and final contribution is a quantitative analysis of the growth of public software development, factoring out exact code clones. Since the advent of version control systems, the production of unique original commits doubles every 30 months, and the production of unique original files is even faster, doubling every 22 months. These findings provide solid evidence that the proposed provenance tracking approach is viable not only today, but also in the foreseeable future, based on the observed growth rates and long-term storage cost trends.

Acknowledgements

The authors would like to thank the anonymous reviewers for precious feedback that allowed us to significantly improve this article.

References

1. Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, October 2018.
2. Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.
3. Carol V. Alexandru, Sebastiano Panichella, and Harald C. Gall. Reducing redundancies in multi-revision code analysis. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 148–159. IEEE Computer Society, 2017.
4. Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019.
5. Miltiadis Allamanis and Charles A. Sutton. Mining source code repositories at massive scale using language modeling. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 207–216. IEEE Computer Society, 2013.
6. Thomas J. (Tim) Bergin. A history of the history of programming languages. *Commun. ACM*, 50(5):69–74, May 2007.
7. Marco Biazzini and Benoit Baudry. May the fork be with you: novel metrics to analyze collaboration on github. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 37–43. ACM, 2014.
8. H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, October 2016.
9. Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.
10. Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. The Debsources dataset: two decades of free and open source software. *Empirical Software Engineering*, 22(3):1405–1437, 2017.
11. Maximilian Capraro and Dirk Riehle. Inner source definition, benefits, and challenges. *ACM Computing Surveys (CSUR)*, 49(4):67, 2017.
12. Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1–7:35, March 2008.
13. Julius Davies, Daniel M. Germán, Michael W. Godfrey, and Abram Hindle. Software bertillonage - determining the provenance of software development artifacts. *Empirical Software Engineering*, 18(6):1195–1237, 2013.
14. Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan*, September 2017. Available from `https://hal.archives-ouvertes.fr/hal-01590958`.
15. Sergey N Dorogovtsev and Jose FF Mendes. Evolution of networks. *Advances in physics*, 51(4):1079–1187, 2002.
16. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.
17. Daniel M. Germán, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In Godfrey and Whitehead [21], pages 81–90.
18. Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In Zaidman et al. [59], pages 18–21.
19. Michael W. Godfrey. Understanding software artifact provenance. *Sci. Comput. Program.*, 97:86–90, 2015.

20. Michael W. Godfrey, Daniel M. German, Julius Davies, and Abram Hindle. Determining the provenance of software artifacts. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 65–66, New York, NY, USA, 2011. ACM.
21. Michael W. Godfrey and Jim Whitehead, editors. *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. IEEE Computer Society, 2009.
22. Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.
23. Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 85–96, New York, NY, USA, 2016. ACM.
24. Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.
25. Les Hatton, Diomidis Spinellis, and Michiel van Genuchten. The long-term growth rate of evolving software: Empirical results and implications. *Journal of Software: Evolution and Process*, 29(5), 2017.
26. Israel Herraiz, Daniel Rodríguez, Gregorio Robles, and Jesús M. González-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.*, 46(2):28:1–28:28, 2013.
27. T. Ishio, R. G. Kula, T. Kanda, D. M. German, and K. Inoue. Software Ingredients: Detection of Third-Party Component Reuse in Java Software Release. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 339–350, May 2016.
28. Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. Why and how developers fork what from whom in github. *Empirical Software Engineering*, 22(1):547–578, 2017.
29. Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.
30. Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.
31. Douglas Andrew Levin, Palle Martin Pedersen, and Ashesh C. Shah. Resolving license dependencies for aggregations of legally protectable content, June 2009. CIB: H04K1/00; G06Q10/00; G06Q50/00; H04L9/00.
32. Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2201–2215, New York, NY, USA, 2017. ACM.
33. Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *PACMPL*, 1(OOPSLA):84:1–84:28, 2017.
34. Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: an infrastructure for mining the universe of open source VCS data. In Storey et al. [50], pages 143–154.
35. Vadim Markovtsev and Waren Long. Public git archive: a big code dataset for all. In Zaidman et al. [59], pages 34–37.
36. Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.
37. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.
38. Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In Godfrey and Whitehead [21], pages 11–20.

39. Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 11–20, Washington, DC, USA, 2009. IEEE Computer Society.
40. Mark Newman, Albert-Laszlo Barabasi, and Duncan J. Watts. *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press, Princeton, NJ, USA, 2006.
41. Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: public software development under one roof. In Storey et al. [50], pages 138–142.
42. Ayushi Rastogi and Nachiappan Nagappan. Forking and the sustainability of the developer community participation–an empirical investigation on outcomes and reasons. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 102–111. IEEE, 2016.
43. Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.
44. Guillaume Rousseau and Maxime Biais. Computer Tool for Managing Digital Documents, February 2010. CIB: G06F17/30; G06F21/10; G06F21/64.
45. Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical Report 115, Queen's School of Computing, 2007.
46. Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Ccfindersw: Clone detection tool with flexible multilingual tokenization. In Jian Lv, He Jason Zhang, Mike Hinchey, and Xiao Liu, editors, *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 654–659. IEEE Computer Society, 2017.
47. Diomidis Spinellis. A repository of Unix history and evolution. *Empirical Software Engineering*, 22(3):1372–1404, 2017.
48. Megan Squire. The lives and deaths of open source code forges. In Lorraine Morgan, editor, *Proceedings of the 13th International Symposium on Open Collaboration, OpenSym 2017, Galway, Ireland, August 23-25, 2017*, pages 15:1–15:8. ACM, 2017.
49. Klaas-Jan Stol and Brian Fitzgerald. Inner source–adopting open source development practices in organizations: a tutorial. *IEEE Software*, 32(4):60–67, 2014.
50. Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors. *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE / ACM, 2019.
51. Jeffrey Svajlenko and Chanchal Kumar Roy. Fast and flexible large-scale clone detection with cloneworks. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 27–30. IEEE Computer Society, 2017.
52. Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.
53. Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. Network structure of social coding in github. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 323–326. IEEE, 2013.
54. Nitin M. Tiwari, Ganesha Upadhyaya, and Hridesh Rajan. Candoia: a platform and ecosystem for mining software repositories tools. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 759–764. ACM, 2016.
55. Timo Tuunanen, Jussi Koskinen, and Tommi Kärkkäinen. Automated software license analysis. *Automated Software Engineering*, 16(3-4):455–490, 2009.
56. C. Vendome. A large scale study of license usage on github. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 772–774, May 2015.
57. Ray Waldin and Jing Zhang. Determining a document similarity metric, July 2009. CIB: G06F17/30.
58. Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. Germán, and Katsuro Inoue. Analysis of license inconsistency in large collections of open source projects. *Empirical Software Engineering*, 22(3):1194–1222, 2017.

59. Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors. *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018.
60. T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9, May 2007.
61. Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 563–572. IEEE Computer Society, 2004.