Universal Source Code Provenance Tracking

1st Anonymous Author
University of Foo
Some, Where
foo@example.com

2nd Anonymous Author
University of Bar
Some, Where
bar@example.com

33rd Anonymous Author
University of Baz
Some, Where
baz@example.com

***Abstract*—Tracking the provenance of software, and in particular of source code artifacts, has become a required building block for software processes such as: code clone detection, the production of software bills of materials (BOMs) to ensure license compliance, and vulnerability tracking.**

In this paper we attack the challenging problem of designing a scalable software provenance solution that can track the provenance of the entire body of publicly available software source code. To that end we first characterize the current size and overall growth of public software development. We do so by analyzing the Software Heritage archive, which contains an unprecedented source code corpus built from major development forges and free/open source software distribution platforms.

Then we review different data models for capturing provenance information at this scale at the granularity of both revisions and individual source code files. Finally we benchmark these data models and show that the most compact among them is a viable solution to capture provenance at the scale of Software Heritage, on commodity hardware, and for the foreseeable future.

***Index Terms*—software provenance, software evolution, open source, clone detection, license compliance, source code tracking**

I. Introduction

Over the last three decades, the world of software development has been revolutionized under the combined effect of the massive adoption of free and open source software (FOSS), and the availability of a wealth of online platforms like GitHub, Bitbucket, and SourceForge that have sensibly reduced the cost of collaborative software development. One important consequence of this revolution is the fact that the source code and the development history of tens of millions of software projects is nowadays public, making an unprecedented corpus available to study software evolution.

Many interesting research articles have been published, reporting on a number of attempts to mine subsets of this massive dataset looking for patterns of interest for software engineering, ranging from the study of code clones [22]–[24] to automated vulnerability detection and repair [11], [16], [18], from code recommenders [28], [29] to software licence analysis and license compliance studies [26], [27].

An important building block for several of these studies is the ability to *identify the occurrences* of a given file in the reference corpus, also known as *provenance tracking* [9]. For example, when a vulnerability is identified in a source code file, it is important to find *other occurrences of this file*, be it in other versions of the same project, or in other projects. Similarly, when analyzing code clones or software licenses, it is important to find *the first occurrence* of a given source file.

Scaling up these studies to the whole corpus of publicly available source code, and making them reproducible, is a significant challenge. Up to now, we did not have a common infrastructure providing a *reference archive* of software development at a global scale. Hence, many authors resorted to using popular development platforms like GitHub as surrogates. 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, there are millions of projects developed elsewhere, including very high profile ones like GNOME.[2]

A new non profit initiative, Software Heritage [6], is bound to change radically this state of affairs: its mission is to collect, preserve, and make accessible the source code of all available software. The project has already amassed the largest corpus of source code ever built, with tens millions software projects archived from GitHub, Gitorious, Google Code, and Debian, growing by the day.

This corpus is stored in a Merkle Direct Acyclic Graph (DAG) [19], which offers several key advantages: it reduces storage requirements by natively deduplicating file clones, that are quite numerous [17], [20]; 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.

The Software Heritage archive is an unprecedented observatory of the software development landscape, and may become the missing reference infrastructure on top of which future software studies can be scaled up, expanded in scope, and made reproducible.

In order to fulfill this potential not just now, but for the long term, it is necessary to know better how this source code corpus being amassed *evolves over time*, which is the first research question addressed in this paper:

RQ1 how does public software development evolve over time? what is its growth rate?

To answer this question, we performed an extensive study of the Software Heritage corpus, continuing a long tradition of

[1]For example, tens of thousands of projects were 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/

software evolution studies [3], [4], [12], [13], that we expand here to a novel scale. We show evidence of exponential growth not only of the raw corpus, but also of the *original* (i.e., never published before) source code artifacts that are added to it.

Considering this exponential growth trend, it becomes important to understand whether it is possible to conceive efficient data models that allow to track software provenance across the whole corpus, not just today, but in the long term, at various granularities. These are then our next research questionsis:

RQ2 can we build an efficient and scalable representation of software source code provenance at the granularity of revisions?

RQ3 can we build an efficient and scalable representation of software source code provenance at the granularity of invididual files?

In order to answer these questions we have explored three data models that offer different space/time trade-offs and evaluated them on 35 years of development history extracted from Software Heritage. We have identified one of them (the *compact model*) as the best choice for a future-proof implementation of universal source code provenance tracking.

Paper structure: we review related work in Section II and recall the Software Heritage data model in Section III; then we address **RQ1** in Section IV; we turn to **RQ2** and **RQ3** by presenting the data models for provenance tracking in Section V and their experimental validation in Section VI; we address threats to validity in Section VII and we conclude discussing future work in Section VIII.

Reproducibility note: the sheer size of the Software Heritage archive (≈200 TB and a ≈100 B edges graph) makes impractical to share the entire dataset as a single bundle. Instead, we make available the full list of identifiers of the revisions we have analyzed at https://annex.softwareheritage.org/public/dataset/revisions-until-2018-02-13.txt.gz (19 GB); this allows to rebuild the dataset used in this paper using any Software Heritage mirror.

II. Related Work

The study of software evolution has been at the heart of software engineering since the seminal "Mythical Man Month" [2] and Lehman's laws [15], and the tidal wave of open source software, by making available a growing corpus of software projects, has spawned an impressive literature of evolution studies. Some 10 years ago a comprehensive survey [4] showed predominance of studies focusing on the evolution of individual projects. Since then large scale studies have become more frequent and the question of how software evolution laws need to be adapted to account for modern software development has attracted renewed attention, as shown in a recent survey [13] that advocates for more empirical studies to corroborate findings in the literature.

While research activity focused on mining software repositories is thriving, realizing massive scale empirical studies on the growth of the software corpus remains a challenging undertaking that requires completing very complex tasks, ranging from collecting massive amounts of source code [20], to building suitable platforms for analyzing them [8], [25]. Hence, up to now, most studies have resorted to selecting relatively small subsets[3] 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 [3], 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. Over the past years this has led to a significant slowdown in the growth of packages, that does not reflect the number of interesting software projects that could be packaged.

A remarkable recent empirical study [12] has calculated the compound annual growth rate of over 4000 software projects, including popular open source products (Apache, Eclipse, GNOME, GNU, Linux, KDE, top GitHub repositories, ...) 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.

An interesting recent work has shown clearly how important exogenous code can be [17]. It analyzed over 4 million non-fork projects from GitHub, selected by filtering on a handful of programming languages (Java, C++, Python, and JavaScript), showing that almost 70% of the code consists file-level identical clones. This gives a picture of the cloning status in a subset of GitHub at the time it was performed, but no insights on how cloning evolves, or how it impacts the growth of the global source code corpus.

Software *provenance* is an essential building block of several research studies, in particular for studies addressing vulnerability tracking, license analysis, and code reuse [14]. Provenance tracking has been for over a decade a key element of industrial tools and services for license compliance developed by companies like BlackDuck, Palamida, Antelink, nexB, and more recently TripleCheck and FossID. Provenance is also addressed in detail by some patents [21], but with few exceptions [9] it has received little attention by the research community. We believe this is due to the lack of a reference archive on which this basic building block can be implemented once and then reused by other researchers.

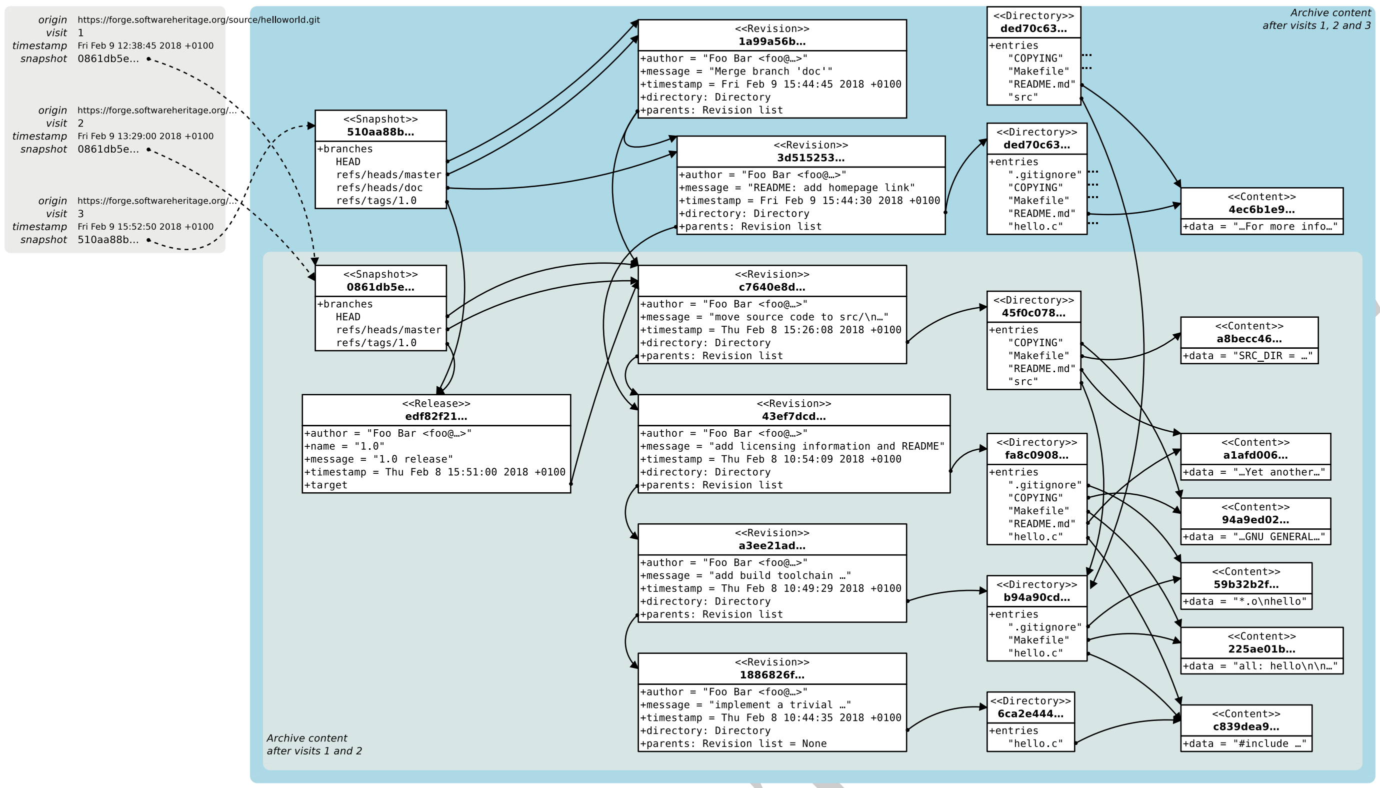
III. Background: Software Heritage

We give in this section a brief overview of the Software Heritage data model and archive, as they are the key ingredients leveraged in the following to track provenance at scale.

A. Data model

Public software *development* duplicates information a lot. It is so in modern software development, where entire repository

[3]Some studies have analyzed up to a few million projects, but this is still a tiny fraction of the total corpus of publicly available source code.



origin https://forge.softwareheritage.org/source/helloworld.git
visit 1
timestamp Fri Feb 9 12:38:45 2018 +0100
snapshot 0861db5e...
origin https://forge.softwareheritage.org/...
visit 2
timestamp Fri Feb 9 13:29:00 2018 +0100
snapshot 0861db5e...
origin https://forge.softwareheritage.org/...
visit 3
timestamp Fri Feb 9 15:52:50 2018 +0100
snapshot 510aa88b...
Archive content after visits 1, 2 and 3
Archive content after visits 1 and 2
<<Snapshot>> 510aa88b… +branches HEAD refs/heads/master refs/heads/doc refs/tags/1.0
<<Snapshot>> 0861db5e… +branches HEAD refs/heads/master refs/tags/1.0
<<Release>> edf82f21… +author = "Foo Bar <foo@…>" +name = "1.0" +message = "1.0 release" +timestamp = Thu Feb 8 15:51:00 2018 +0100 +target
<<Revision>> 1a99a56b… +author = "Foo Bar <foo@…>" +message = "Merge branch 'doc'" +timestamp = Fri Feb 9 15:44:45 2018 +0100 +directory: Directory +parents: Revision list
<<Revision>> 3d515253… +author = "Foo Bar <foo@…>" +message = "README: add homepage link" +timestamp = Fri Feb 9 15:44:30 2018 +0100 +directory: Directory +parents: Revision list
<<Revision>> c7640e8d… +author = "Foo Bar <foo@…>" +message = "move source code to src/\n…" +timestamp = Thu Feb 8 15:26:08 2018 +0100 +directory: Directory +parents: Revision list
<<Revision>> 43ef7dcd… +author = "Foo Bar <foo@…>" +message = "add licensing information and README" +timestamp = Thu Feb 8 10:54:09 2018 +0100 +directory: Directory +parents: Revision list
<<Revision>> a3ee21ad… +author = "Foo Bar <foo@…>" +message = "add build toolchain …" +timestamp = Thu Feb 8 10:49:29 2018 +0100 +directory: Directory +parents: Revision list
<<Revision>> 1886826f… +author = "Foo Bar <foo@…>" +message = "implement a trivial …" +timestamp = Thu Feb 8 10:44:35 2018 +0100 +directory: Directory +parents: Revision list = None
<<Directory>> ded70c63… +entries "COPYING" "Makefile" "README.md" "src"
<<Directory>> ded70c63… +entries ".gitignore" "COPYING" "Makefile" "README.md" "hello.c"
<<Directory>> 45f0c078… +entries "COPYING" "Makefile" "README.md" "src"
<<Directory>> fa8c0908… +entries ".gitignore" "COPYING" "Makefile" "README.md" "hello.c"
<<Directory>> b94a90cd… +entries ".gitignore" "Makefile" "hello.c"
<<Directory>> 6ca2e444… +entries "hello.c"
<<Content>> 4ec6b1e9… +data = "…For more info…"
<<Content>> a8becc46… +data = "SRC_DIR = …"
<<Content>> a1afd006… +data = "…Yet another…"
<<Content>> 94a9ed02… +data = "…GNU GENERAL…"
<<Content>> 59b32b2f… +data = "*.o\nhello"
<<Content>> 225ae01b… +data = "all: hello\n\n…"
<<Content>> c839dea9… +data = "#include …"

Fig. 1. Software Heritage Merkle DAG with crawling information.

clones are produced to exchange even minor improvements via pull requests [10]; but it was the case also in the past when repositories migrated across platforms (e.g., from SourceForge to Google Code to GitHub) and technologies (e.g., from CVS to Subversion to Git) leaving stale copies behind. Further duplication is induced by software *distribution* platforms (e.g., package repositories) that massively rely on large mirror networks to distribute open source software packages. In order to be long-term viable, archival platforms need to tame this duplication; as we will see in the following doing so will also be beneficial for capturing provenance of software artifacts that exist, as identical copies, in a myriad of places.

Software Heritage [6] deals with this massive duplication by storing all archived source code artifacts in a single, huge Merkle direct acyclic graph (DAG) [19]. A toy yet detailed example of that structure is given in Fig. 1. Each node in the diagram corresponds to a source code artifact produced as part of software development:

a) Contents: (i.e., *blobs*) raw file contents as byte sequences, no matter the context where they have been found. Note that contents are anonymous; "file names" are given to them by directories.

b) Directories: lists of named directory entries, where each entry can point to content objects ("file entries"), to other directories ("directory entries"), or even to other revisions ("revision entries", capturing links to external components like those enabled by Git submodules or Subversion externals). Each entry is associated to a name (i.e., a relative path) as well as permission metadata and timestamps.

c) Revisions: (i.e., *commits*) point-in-time states in the development history of a software project. Each revision points to the root directory of the software source code at the time, and includes additional metadata such as revision timestamp, author, and a human-readable description of the change.

d) Releases: (i.e., *tags*) revisions marked as noteworthy and associated to specific, usually mnemonic, names (e.g., version numbers or release codenames). Releases point to revisions and might include additional descriptive metadata.

e) Snapshots: lists of pairs mapping development branch names (e.g., "master", "bug-1234", "feature-foo") to revisions or releases. 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.

Deduplication happens at the node level for all artifact types: each file content will be stored exactly once and referred to via cryptographic checksum key from multiple directories; each commit will be stored once, no matter how many branches include it; up to each snapshot, which will be stored once no matter how many identical copies of it in exactly the same state (e.g., pristine forks on GitHub) exist.

Note how 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 revision nodes), and development states at specific

TABLE I
Graph characteristics of the reference dataset: a Software Heritage archive copy as of February 13th, 2018.

(a) archive coverage

46.4 M software origins

(b) nodes

node type	quantity
content	3.98 B
revision	943 M
release	6.98 M
directory	3.63 B
snapshot	49.9 M
total	8.61 B

(c) edges

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

points in time (pointed by snapshot nodes).

In addition to the content of the Merkle DAG, Software Heritage stores *crawling information*, as depicted on 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 the same state has been observed in the past) and a 3-way mapping between the origin (as an URL), the time of the visit, and the snapshot object, which is added to an append-only journal of crawling activities.

B. Archive coverage

At the time we used it for the work presented in this paper, the Software Heritage archive was already the largest available corpus of software development artifacts [5], [6], encompassing:

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

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

Note that in the Software Heritage archive one can find all the essential building blocks of provenance information: the journal of crawling activities precisely records when and where a particular snapshot has been found. The challenge is to design a data model that can be used to efficiently answer provenance queries at this scale. For example, consider the problem of finding the first occurrence (by revision date) in which a given content (e.g., a specific `.java` file) appears, as well as reporting where it has been observed (the software origin URL) and when (the visit timestamp). Answering this query using only the Merkle DAG would require exploring

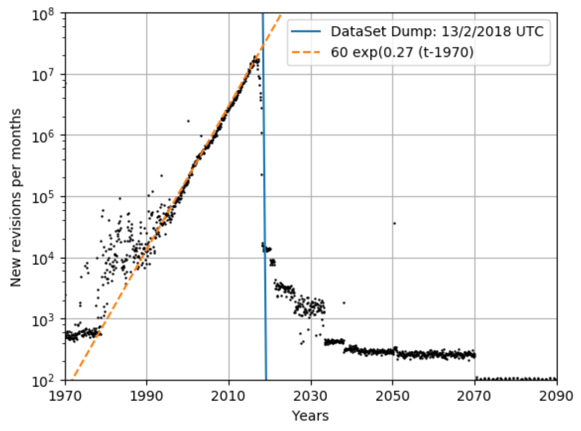


Fig. 2. Global production of original software artifacts over time, in terms of never-seen-before revisions.

all the paths leading to the specific content in a graph of a hundred billion edges.

We will show that it is possible to solve this problem using commodity hardware and in a way that will stand the test of time. Before that, though, we will analyze the Software Heritage corpus to estimate how the graph of software development grows over time, which will give us precious insights on the complexity of the problem.

IV. Public Software Development Growth

In order to track provenance at this scale in the long term, we study here the factors that contribute to the sheer size of a provenance index. i.e., roughly speaking, the number of different *contexts* in which the source code artifacts we want to track *occur* and how this quantity evolves over time. We develop this understanding in this section studying the production of original, never-seen-before source code artifacts and how they spread across publicly developed software.

The fact that current development practices rely heavily on duplicating and/or reusing code [10], [17] makes simple metrics—such as counting the number of source code files, revisions (also known as *commits*), or even entire projects—ineffective at estimating how much *original* software is being produced.

Using Software Heritage as a representative sample of publicly developed software, we focus here on how *original* software artifacts grow and the relationship between the key components needed for provenance tracking: *revisions*, *contents* and *origins*.

A. *Evolution of* original *revisions*

We start by characterizing *the evolution of original revisions*. We have analyzed the entire reference dataset (see Table I), processing revisions in increasing timestamps order. A revision is *original* if the combination of its properties (or, equivalently, its identifier in the Merkle DAG) has never been encountered before. Results are shown in Fig. 2, which shows an exponential growth over time that can be accurately approximated by the fit line $60e^{0.27(t-1970)}$. At that rate, **the number of original revisions world-wide doubles every ≈30 months**.

This information is precious to estimate the resources needed for *archiving* publicly developed software: taking into account the long term evolution of storage costs[4] this growth looks managable, provided deduplication is used, as described in Section III. The sustainability of provenance tracking is more challenging, because artifact *occurrences* cannot be deduplicated as, by definition, they occur in different contexts. Viable provenance data models need to handle this growth; we will discuss them in Section V.

The growth rate of *original revisions* hints at the fact that both the production of novel source code artifacts and their provenance are interesting evolving complex networks [1], [7]. Provenance networks that link together occurrences of the same file constitute such networks, in which reuse of source code artifacts in novel contexts can be associated to *preferential attachment*. For scale-free networks it has been shown that growth and preferential attachment play an important role in inducing power law patterns. Determining if these networks are scale-free or not, and studying the growth dynamics between edges and nodes, potentially leading to *accelerating growth* [1], is outside the scope of this paper, but the revision growth trend hints at that and will play an important role in determining it.

Outliers in Fig. 2 deserve further explanation. Data points at the *Unix epoch* (1/1/1970) are over-represented, likely due to forged revision timestamps introduced when converting across version control systems—these revisions amount to 0.75% of the dataset. Other timestamps (0.1% of the dataset) are also clearly forged as they are in the future w.r.t. the timestamp at which the reference dataset has been taken.

B. Content popularity

Now we turn to another important factor in our analysis: the *popularity* of file contents among revisions, i.e., we study in how many unique revisions each unique content appears.

We took a random sample of about 1 M unique contents (all contents whose identifiers (hashes) start with `aaa`) and counted their occurrences in all revisions of the reference dataset. Simple and cumulative popularity are shown in the upper part of Fig. 3.

Content popularity in revisions is well approximated by a power law with exponential cutoff $f(x) = a(x+b)^{-\alpha}e^{-(\frac{x+b}{c})^{\gamma}}$ with parameters $a = 1.13392\cdot 10^6$, $b = 1.80697$, $c = 161128$, $\alpha = 0.489471$ and $\gamma = 0.449744$, obtained using an implementation of the nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm. This means that the average popularity is very high and is limited only by the existence of a slow exponential tail for popularity greater than several tens of thousands revisions. The detailed study of content popularity as a function of time and revisions—which we have already observed to grow exponentially itself—is outside the scope of

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

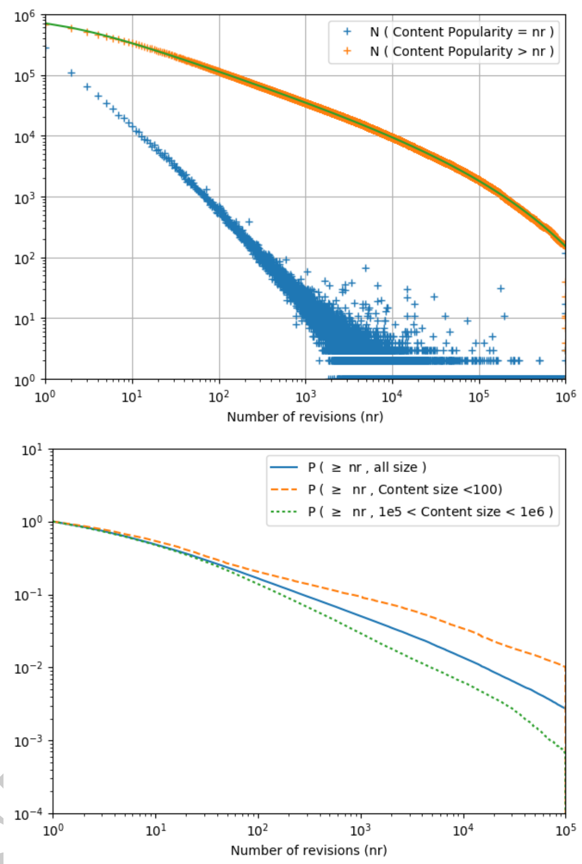


Fig. 3. Popularity of unique file contents as the number of unique revisions they appear in. Results for a random sample of the reference dataset are shown above; results for selected size-based samples below.

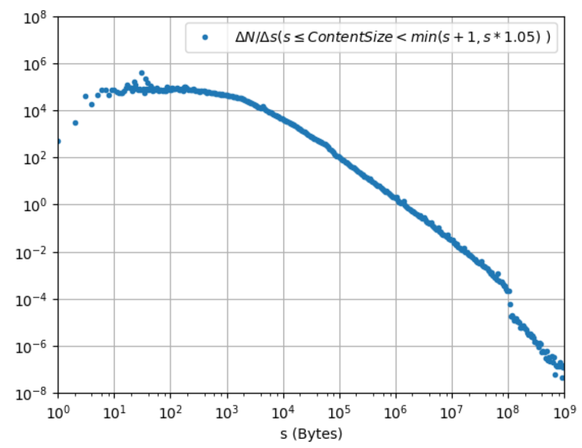


Fig. 4. Distribution of content sizes for a subset of the reference dataset (1/16th, or ≈250 M contents).

this article and left as future work. But clearly such growth plays a major role in making the fully general version of provenance tracking challenging to deal with.

Can we narrow down provenance tracking to selected subsets of file contents, in order to reduce its magnitude? To explore this we sliced contents by size, a technique often

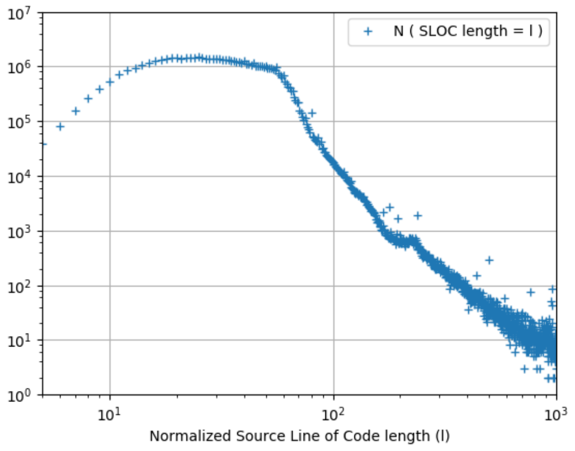


Fig. 5. Distribution of normalized SLOC lengths in a sample of 2.5 M contents that appear at least once with `.c` extension.

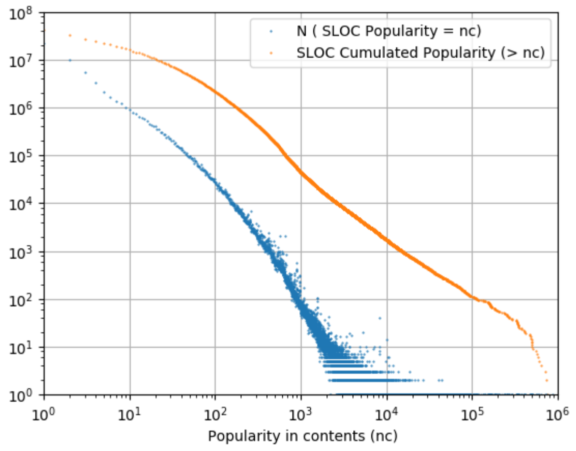


Fig. 6. Popularity of normalized SLOCs as the number of unique contents they appear in. Dataset: same of Fig. 5.

used by current industry solutions which, e.g., ignore files smaller than 100 bytes. To determine slices we looked at the distribution of content sizes, shown in Fig. 4 for 1/16th of the reference dataset ($\approx$250 M contents). To limit the amplitude of fluctuations and avoid empty bins, bin sizes vary exponentially for large value, i.e., (n+1)-th bin size = 1.05 $\cdot$ n-th bin size, and bin amplitudes are normalized by bin size.

The log-log scale clearly shows a decrease in power law for size $> 10^4$ and up to 10^8, with an exponent that can be manually estimated at $\approx$1.8.[5] Hence we looked into two size-based subsets: up to 100 bytes (middle of the plateau before 10^4) and between 10^5 and 10^6 (middle of the descent before 10^8). We took 1 M contents randomly selected within each interval and studied their popularity as before. Normalized cumulative popularity results are shown in the bottom part of Fig. 3, together with the results for the full sample for comparison.

Small contents are much more popular than both average-sized and large contents. The comparison shows, for small contents, a slower decrease in popularity followed by an exponential cut-off; while for large contents, the decline in popularity is faster with an earlier cut-off.

It is indeed the case that by ignoring small contents the amplitude of provenance tracking can be significantly reduced. But doing so would be justifiable only for specific use cases (e.g., small contents might be considered not "original enough" from an intellectual property standpoint and hence not worth tracking for specific use cases) while our goal is to support all use cases, so we keep our focus on provenance tracking for the entire corpus.

C. *Lines of code length and popularity*

For some use cases it might be interesting to implement provenance tracking at finer granularities, so we also performed a few analysis at the level of single lines of code (SLOC). Since lines of code are hardly comparable across languages, we focused on the C language, which is well-represented in our corpus. To that end we took a random sample of $\approx$11.4 M unique contents occurring in revisions between 1980 and 2001, and selected from it contents that appear at least once with `.c` extension and with size between 10^2 and 10^6 bytes, obtaining $\approx$2.5 M contents. We then split contents by line and, to remove equivalent formulations of the same SLOC, *normalized* lines by removing blanks and trailing `";"` (semicolon). We obtained $\approx$64 M normalized SLOCs.

The distribution of normalized SLOC lengths between 4 and 1000 characters is shown in Fig. 5. **Lines with length 15 to 60 normalized characters are the most represented**, with fairly stable presence within that range. Afterwards the amounts of lines of code with a given length drop steeply, reaching very long line outliers, probably due to phenomena like inlining and obfuscation. For SLOC provenance tracking there does not seem to exist any obvious length-based threshold that would reduce the problem magnitude.

We also looked into the popularity of SLOCs across unique contents, as we did for content popularity in revisions; results are shown in Fig. 6. As for contents, popularity shows a decrease in power law over several decades, followed by an exponential cutoff close to the finite size of the dataset.

D. *Origin size and popularity*

Another factor that impacts provenance tracking is the amount of repository forks, because each fork induces extra data points to capture the additional location of *all* source code artifacts contained in the fork. To assess this impact we looked at the distribution of origin sizes using two metrics: the number of revisions hosted at a software origin and the number of *origin-deduplicated revisions*. The latter metric makes revisions that appear in multiple origins count for only one of them and, specifically, for the biggest one among them in terms of non-deduplicated revisions. This measure is very robust in the sense that it will naturally follow the

[5]The cut-off at 10^8 is an artifact likely due to the size limitations imposed by GitHub. See: https://help.github.com/articles/what-is-my-disk-quota/

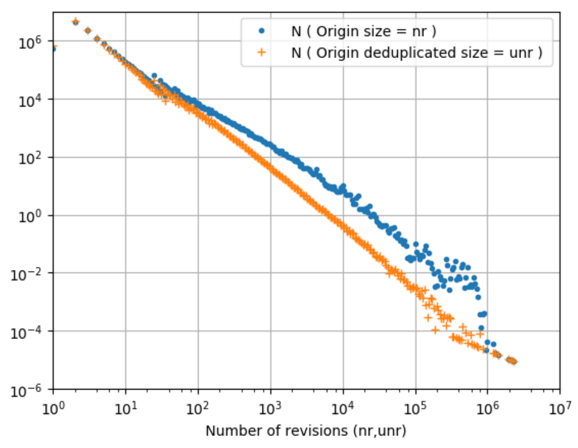


Fig. 7. Distribution of origin size as the number of revisions they host. Deduplicated size is obtained by counting revisions found at multiple origins only for the largest (non-deduplicated) among them.

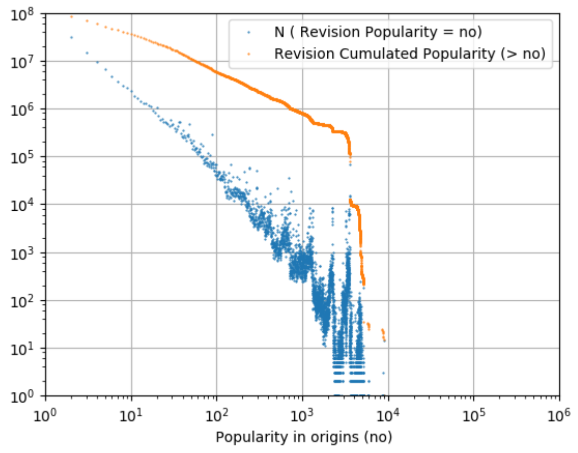


Fig. 8. Popularity of revisions across origins for the same dataset of Fig. 7.

main development line (or *most fit fork*) of a project, assigning deduplicated revisions to it, while stale forks decay. Also, the metric will follow forks that resurrect projects abandoned at their original development places. Note that this approach does not rely on platform metadata for recognizing forks; as such it will recognize *exogenous forks* across unrelated development platforms (e.g., GitHub-hosted forks of the Linux kernel that is not natively developed on GitHub).

Results are shown in Fig. 7 for ≈12% of the origins, which contain ≈29% of the revisions (or about 5.4 M origins and 272.5 M revisions) in the reference dataset. Again, bin sizes vary exponentially for large values and their amplitude is normalized by bin size.

Note how, starting from relatively small repositories (≈100 revisions) deduplication reduces significantly provenance amplitude, with a gap growing up to a full order of magnitude in difference for repositories hosting 10 K revisions.

As the last element of provenance characterization, we would like to know how popular revisions are across origins, assessing the impact of their duplication on provenance amplitude. To that end we replicated the previous study of content popularity onto revisions. Results are shown in Fig. 8.

Revision popularity shows an erratic behavior near the end of the range, but decreases steadily before, and way more steeply than it was the case for content popularity across revisions (see Fig. 3 for comparison).

An important fact that emerges from these analyses is that the content→revision mapping is the most challenging to deal with, because contents are duplicated across revisions much more than revisions are duplicated across origins. Hence in the following we will focus on efficiently implementing the content→revision mapping, as the revision→origin mapping will be a straightforward and modular addition.

V. Compact Provenance Modeling

A. Requirements

We can now attack the problem of how to practically capture and query provenance information. Specifically, we set forward to define a data model that satisfies the following requirements:

a) Supported queries: The *first occurrence* query shall return the most ancient occurrence of a given source code artifacts in any context, according to the revision timestamp. The *all occurrences* query generalizes first occurrence to return all occurrences instead. 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.

b) Granularity: Support tracking the provenance of source code artifacts at different granularities including at least file contents and revisions.

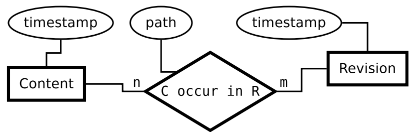
c) Scalability: Support tracking provenance at the scale of Software Heritage and keep up with the growth rate of public software development (see Section IV). Given that the initial process of populating provenance mappings might be very onerous, and that some use cases require fresh data (e.g., impact/popularity), we consider *incrementality* as part of scalability: the data model must support efficient updates of the provenance mappings as soon as source code artifacts (old or new) are observed in new contexts.

d) Compactness: Enable storing and querying provenance mappings on state-of-the-art consumer hardware, without requiring dedicated hardware or expensive cloud resources.

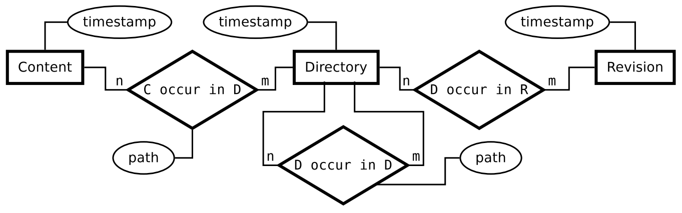
e) Streaming: For the *all occurrences* query a non-compressible performance bottleneck is the transfer time required to return a sheer amount of results. Viable provenance data models should perform *no worse* than that, immediately returning some of the results, piping up the rest for later.

B. Provenance models

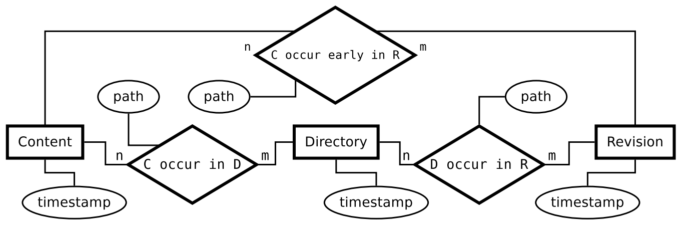
We study three different data models for provenance tracking, that we call respectively *flat*, *recursive*, and *compact*. Their Entity-Relationship (E-R) diagrams are shown in Fig. 9.



(a) flat model



(b) recursive model



(c) compact model

Fig. 9. Provenance tracking models, entity-relationship (E-R) views

a) Flat model: this is our baseline for tracking provenance, shown in Fig. 9(a). In this model provenance mappings are "flattened" using a single `C(ontent) occur in R(evision)` relation, that also keeps track of content paths relatively to the root directory of the associated revision. The cardinality of `C occur in R` is n-m (rather than 1-n), because the same content might appear multiple times in a given revision at different paths. Each revision carries as attribute the revision timestamp, in order to answer the question of *when* the occurrence happened. Each content carries as attribute the timestamp of its earliest occurrence, i.e., the minimum timestamps among all associated revisions.

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

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

b) Recursive model: while the flat model shines in access time at the expenses of update time and compactness, the recursive model shown in Fig. 9(b) does the opposite. It is intuitively a "reverse" Merkle DAG representation, which maps contents to directories and directories to revisions.

Each entity has a timestamp attribute equal to the timestamp of the earliest revision in which the entity has been observed thus far. When processing an incoming revision r_{t_2} (with timestamp t_2) it is no longer necessary to fully traverse the associated directory: if a node is encountered whose timestamp t_1 in the model is s.t. $t_1 < t_2$, recursion can stop as the subtree rooted there has already been added in the past; we just need to add a single occurrence for the node 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 as for each content we need to walk up directory paths before finding the corresponding revisions. Response time will hence depend on the average directory depth at which queried contents will be found. First occurrence is faster, but still incurs some read overhead: given a content we have to walk up all directories (and then revisions) whose timestamps equate the timestamp of the content being queried.

c) Compact model: Fig. 9(c) shows a compromise version between the flat and recursive models, which is both storage-compact and capable of quickly answering the target queries. A timestamp attribute denoting the earliest known occurrence is associated to content, directory, and revision entities, as before. To understand how the compact model is used 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 structure, the isochrone subgraph *of a revision node R (with timestamp t_R) in the Software Heritage graph is a subgraph rooted at R's directory, which only contains directory nodes whose timestamps in $\mathcal{P}$ are equal to t_R.*

Intuitively, when processing revisions chronologically to update provenance mappings, the isochrone subgraph of a revision 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. Everything outside the isochrone subgraph has been observed before, in at least one previously processed revision.

Given this notion, the upper part of the compact model (`C occur early in R`) is filled with one entry for each content attached to any directory in the isochrone subgraph. As a consequence of this, the first occurrence of any given content will always be found in `C occur early in R` although other occurrences—depending on the order in which revisions are processed to update the provenance mappings—will also be found there.

The relation `D occur in R` is filled with one entry, pointing to the revision being processed, for each directory *outside* the isochrone subgraph that is referenced by directories *inside* it, i.e., `D occur in R` contains one entry for each directory-to-directory edge crossing the isochrone frontier. Finally, the relation `C occur in D` is filled with one entry for each content (recursively) referenced by any directory added to the `D occur in R` relation.

Filling the compact model is faster than the flat model because we stop traversing the directory hierarchy when we reach the frontier of the isochrone subgraph; it is slower than the recursive model case, though, as we still need to traverse the isochrone subgraph of each revision. Read overhead for first occurrence is similar to the flat model: provided suitable indexing on timestamps we can quickly find first occurrences in `C occur early in R`. Read overhead for all occurrences is lower than the recursive model because all content occurrences will be found via `C occur in D` without needing to recursively walk up directory trees, and from there directly linked to revisions via `D occur in R`.

C. Discussion

Intuitively, the reason why the compact model is a good compromise is that we have many revisions and a very high number of file contents that occur over and over again in them, as we have seen in Fig. 3. Consider then two extreme cases: (1) a set of revisions all pointing to the same root directory but with metadata differences (e.g., timestamp or author) that make all revisions unique; (2) a set of revisions all pointing to different root directories that have no file contents or (sub)directories in common.

In case (1) the flat model would explode in size due to maximal duplication. The recursive model will need just one entry in `D occur in R` for each revision. The compact model remains small as the earliest revision will be flattened (via `C occur early in R`) as in the flat model, while each additional revision will add only one entry to `D occur in R` (as in the recursive model).

In case (2) the flat model will be 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 as small as the flat model, because all revisions are entirely isochrones.

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

VI. Evaluation

To compare the provenance data models described in the previous section we have measured the evolution of the sizes of the three models while processing incoming revisions to maintain provenance mappings up to date.

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

a) Flat model: one entity for each content and revision; plus one `C occur in R` entry for each content occurrence

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

TABLE II
Size comparison for provenance data models, in terms of entities and relationship entries. Same dataset of Fig. 10.

	Flat	**Recursive**	**Compact**
entities	80 118 995	148 967 553	97 190 442
	rev: 38.2 M	rev: 38.2 M	rev: 38.2 M
	cont: 41.9 M	cont: 31.9 M	cont: 31.9 M
		dir: 68.8 M	dir: 17.1 M
rel. entries	654 390 826 907	2 607 846 338	19 259 600 495
		cont–dir: 1.29 B	cont–dir: 13.8 B
		dir–rev: 38.2 M	dir–rev: 2.35 B
		dir–dir: 1.28 B	cont–rev: 3.12 B
rel. ratios	$\frac{flat}{compact} = 34.0$	$\frac{flat}{rec.} = 251$	$\frac{compact}{rec.} = 7.39$

c) Compact model: we have first determined the isochrone subgraph of each revision, then counted: one entity for each content and revision, plus one entity for each directory outside the isochrone graph referenced from within; as well as one relationship entry for each content attached to directories in the isochrone graph (`C occur early in R`), one `D occur in R` entry for each directory-to-directory edge crossing the isochrone frontier, and one `C occur in D` entry for each content present in directories appearing in `D occur in R`.

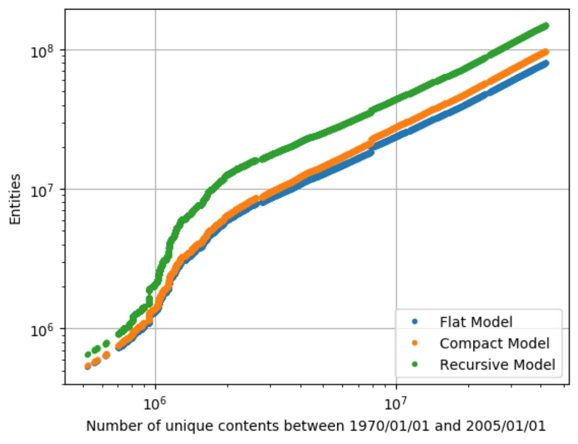
Processing has been done running a Python implementation of the above measurements on a commodity workstation (Intel Xeon 2.10GHz, 16 cores, 32 GB RAM), parallelizing the load on all available cores. Merkle DAG information have been read from a local copy of the reference dataset, which had been previously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

Final sizes measured in terms of entities and relationship entries are shown in Table II. 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 artefacts remains the main volumetric challenge in tracking provenance. As further evidence of this, and as a measure of the overall amplitude of universal provenance tracking, 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 `C occur in R`.

Second, sizes tell us that the Merkle DAG representation, isomorphic to the compact model, is indeed the most compact representation of provenance information, although not the most efficient 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. 10 shows the evolution of model sizes over time, as a function of the number of unique contents 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 good.

In order to relate these figures to actual storage requirements, we have also filled a MongoDB-based implementation



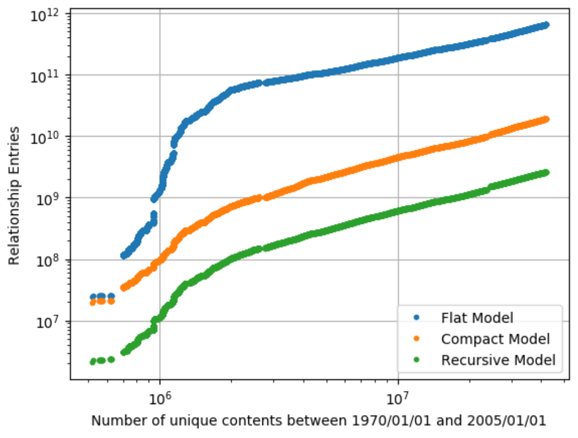


Fig. 10. Evolution over time of the sizes of different provenance data models, in terms of entities and relationship entries. Data for Software Heritage revisions up to 2005-01-01, excluding UNIX epoch.

of the compact model—including all attributes of Fig. 9(c) and needed indexes—while processing revisions to perform the above measurements. Extrapolating the final MongoDB size (including indexes) 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 ≈2000$ of SSD disks, hence without having to resort to dedicated data centers or substantial investments in cloud resources. Universal source code provenance can lay at the fingertips of every researcher and industrial user!

VII. Threats to Validity

a) Internal validity: The main concern for internal validity is that we did not have the resources available to perform all estimates and experiments on the full Software Heritage archive. Our main growth results is measured on the full reference dataset, while other results are projections extrapolated from smaller subsets of it. This may lead to bias, which we believe to have countered using random sampling.

Along the same line, 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.

Finally, we trusted commit timestamps to determine first occurrences, knowing well that commit timestamps might be forged. On the one hand, this is consistent with previous software evolution studies that generally seem to consider timestamp forging a marginal phenomenon. On the other hand, what matters for our purposes is just the ability to separate *a* "first" occurrence from the others, which we support no matter what discrimination criterion is used.

b) External validity: In terms of generality, while Software Heritage does not cover all existing free/open source software, it is the largest source code archive in the world and spans the most popular code hosting and development platforms. We think this is the best that can be done today.

Finally, we acknowledge the habit of using *software* development platforms for tasks other than software development (e.g., collaborative writing), particularly on GitHub. We did not make an effort to filter out non-software projects, but we expect software development to be still the largely dominant use of the platforms represented in our corpus.

VIII. Conclusion

The emergence of Software Heritage as a universal source code archive, with its compact Merkle-based representation of development artifacts, enables analysis of the evolution of software development at an unprecedented scale.

It is now well known that the amount of code clones across GitHub projects is huge [17]. The first main contribution of this paper is to show that, even factoring out exact code clones, *the production of unique original revisions* at the scale of Software Heritage, spanning tens of millions of software projects over several decades, *doubles every 30 months*.

This means that the perceived overall growth of the public software ecosystem is quite real, so that to enable sustainable large scale studies on this new commons we need to find scalable, efficient ways for tracking the *provenance* of source code artifacts.

To this end, we have studied three data models that allow to answer the essential questions "what are the first/all occurrence(s) of this file/commit?" in a given source code corpus. The models represent different trade-offs between space requirements and access overhead, which we have estimated and verified experimentally. We have shown that the *compact* data model has the best space/time trade-off and allows to track universal software provenance at the scale of Software Heritage on consumer hardware, paving the way to a variety of interesting big code studies.

In future work we intend to, on the one hand, further characterize the software evolution trends presented here, exploring how other types of original source code artifacts evolve over time, as well as studying the characteristics of provenance graphs as evolving complex networks. On the other hand we want to increase the granularity at which provenance is tracked, most notably breaking file boundaries and reaching down to individual lines of code, supporting queries such as what are the first/all occurrences of a given line of code across the entire Software Heritage corpus.

References

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[3] 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.

[4] 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.

[5] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for digital objects: the case of software source code preservation. In *iPRES 2018: 15th International Conference on Digital Preservation*, 2018. To appear.

[6] 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.

[7] Sergey N Dorogovtsev and Jose FF Mendes. Evolution of networks. *Advances in physics*, 51(4):1079–1187, 2002.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In 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*, pages 11–20. IEEE Computer Society, 2009.

[21] Guillaume Rousseau and Maxime Biais. Computer Tool for Managing Digital Documents, February 2010. CIB: G06F17/30; G06F21/10; G06F21/64.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.