Software Heritage: Why and How to Preserve Software Source Code*

Roberto Di Cosmo
Inria and University Paris Diderot
France
roberto@dicosmo.org

Stefano Zacchiroli
University Paris Diderot and Inria
France
zack@irif.fr

ABSTRACT

Software is now a key component present in all aspects of our society. Its preservation has attracted growing attention over the past years within the digital preservation community. We claim that source code—the only representation of software that contains human readable knowledge—is a precious digital object that needs special handling: it must be a first class citizen in the preservation landscape and we need to take action immediately, given the increasingly more frequent incidents that result in permanent losses of source code collections.

In this paper we present Software Heritage, an ambitious initiative to collect, preserve, and share the entire corpus of publicly accessible software source code. We discuss the archival goals of the project, its use cases and role as a participant in the broader digital preservation ecosystem, and detail its key design decisions. We also report on the project road map and the current status of the Software Heritage archive that, as of early 2017, has collected more than 3 billion unique source code files and 700 million commits coming from more than 50 million software development projects.

ACM Reference format:
Roberto Di Cosmo and Stefano Zacchiroli. 2016. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of 14th International Conference on Digital Preservation, Kyoto, Japan, September 2017 (iPRES2017),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software is everywhere: it powers industry and fuels innovation, it lies at the heart of the technology we use to communicate, entertain, trade, and exchange, and is becoming a key player in the formation of opinions and political powers. Software is also an essential mediator to access all digital information [1, 5] and is a fundamental pillar of modern scientific research, across all fields and disciplines [38]. In a nutshell, software embodies a rapidly growing part of our cultural, scientific, and technical knowledge.

Looking more closely, though, it is easy to see that the actual *knowledge* embedded in software is not contained into executable binaries, which are designed to run on specific hardware and software platforms and that often become, once optimized, incomprehensible for human beings. Rather, knowledge is contained in software *source code* which is, as eloquently stated in the very well crafted definition found in the GPL license [13], "the preferred form [of a program] for making modifications to it [as a developer]".

Yes, software *source code* is a unique form of knowledge which is designed to be *understood by a human being*, the developer, and at the same time is easily converted into *machine executable* form. As a digital object, software source code is also subject to a peculiar workflow: it is routinely evolved to cater for new needs and changed contexts. To really understand software source code, access to its entire development history is an essential condition.

Software source code has also established a new relevant part of our information commons [20]: the software commons—i.e., the body of software that is widely available and can be reused and modified with minimal restrictions. The raise of Free/Open Source Software (FOSS) over the past decades has contributed enormously to nurture this commons [32] and its funding principles postulate source code accessibility.

Authoritative voices have spoken eloquently of the importance of source code: Donald Knuth, a founding father of computer science, wrote at length on the importance of writing and sharing source code as a mean to understand what we want computers to do for us [19]; Len Shustek, board director of the Computer History Museum, argued that "source code provides a view into the mind of the designer" [33]; more recently the importance of source code preservation has been argued for by digital archivists [4, 22].

And yet, little *action* seem to have been put into long-term source code preservation. Comprehensive archives are available for variety of digital objects, pictures, videos, music, texts, web pages, even binary executables [18]. But source code in its own merits, despite its significance, has not yet been given the status of first class citizen in the digital archive landscape.

In this article, we claim that it is now important and urgent to focus our attention and actions to source code preservation and build a comprehensive archive of all publicly available software source code. We detail the basic principles, current status, and design choices underlying *Software Heritage*,[1] launched last year to fill what we consider a gap in the digital archiving landscape.

This article is structured as follows: Sections 2 through 4 discuss the state of the art of source code preservation and the mission of Software Heritage in context. Sections 5 and 6 detail the design and intended use cases of Software Heritage. Before concluding, Sections 7 through 9 present the data model, architecture, current status, and future roadmap for the project.

*Unless noted otherwise, all URLs in the text have been retrieved on March 30, 2017.

[1] https://www.softwareheritage.org

2 RELATED WORK

In the broad spectrum of digital preservation, many initiatives have taken notice of the importance of software in general, and it would be difficult to provide an exhaustive list, so we mention here a few that show different aspects of what has been addressed before.

A number of these initiatives are concerned with the execution of legacy software in binary form, leveraging various forms of virtualisation technologies: the Internet Archive [18] uses web-based emulators to let visitors run in a browser old legacy games drawn from one of their software collection; the E-ARK [8] and KEEP [9] projects brought together several actors to work on making emulators portables and viable on the long term, while UNESCO Persist [36] tries to provide a host organization for all activities related to preserving the executability of binaries on the long term.

NIST maintains a special collection of binaries for forensic analysis use [26]: while the content of the archive is not accessible to the public, it has produced interesting studies on the properties of different cryptographic hashes they use on their large collection of software binaries use [23].

The raising concern for the sore state of reproducibility of scientific results has spawn interest in preserving software for science and research, and several initiatives now offer storage space for depositing software artefacts: CERN's Zenodo [39] also provides integration with GitHub allowing to take snapshots of the source code of a software project (without its history).

Finally, the raise of software engineering studies on software repositories have led several researchers to build large collections of source code [17, 25, 35] and more recently databases with event metadata from GitHub [10, 14]; these initiatives have as main goal to provide a *research platform*, not an archive for *preservation*.

To the best of our knowledge, in the broader digital preservation landscape the niche of software archival *in source code form* has not been addressed in its own right before.

3 SOFTWARE SOURCE CODE IS AT RISK

Despite the importance of source code in the development of science, industry, and society at large, it is easy to see that we are collectively not taking care of it properly. In this section we outline the three most evident reasons why this is the case.

The source code diaspora. With the meteoric rise of Free/Open Source Software (FOSS), millions of projects are developed on publicly accessible code hosting platforms [34], such as GitHub, GitLab, SourceForge, Bitbucket, etc., not to mention the myriad of institutional "forges" scattered across the globe, or developers simply offering source code downloads from their web pages. Software also tend to move among code hosting places during its lifetime, following current trends or the changing needs and habits of its developer community.

Once a particular version of a software is released, the question arises of how to distribute it to users. Here too the landscape is quite varied: some developers use the code hosting also for distribution, as most forges allow it. Other communities have their own archives organized by software ecosystems (e.g., CPAN, CRAN, ...), and then there are different software distributions (Debian, Fedora, ...) and package management systems (npm, pip, OPAM, ...), which also retain copies of source code released elsewhere.

It is very difficult to appreciate the extent of the software commons as a whole: we direly need a single entry point—a modern "great library" of source code, if you wish—where one can find and monitor the evolution of all publicly available source code, independently of its development and distribution platforms.

The fragility of source code. We have known for a long time that digital information is fragile: human error, material failure, fire, hacking, can easily destroy valuable digital data, including source code. This is why carrying out regular backups is important.

For users of code hosting platforms this problem may seem a distant one: the burden of "backups" is not theirs, but the platforms' one. As previously observer [37], though, most of these platforms are tools to enable collaboration and record changes, but do not offer any long term preservation guarantees: digital contents stored there can be altered or deleted over time.

Worse, the entire platform can go away, as we learned the hard way in 2015, when two very popular FOSS development platforms, Gitorious [11] and Google Code [16] announced shutdown. Over 1.5 million projects had to find a new accommodation since, in an extremely short time frame as regards Gitorious. This shows that the task of long term preservation cannot be assumed by entities that do not make it a stated priority: for a while, preservation may be a side effect of other missions, but in the long term it won't be.

We lack a comprehensive archive which undertakes this task, ensuring that if source code disappears from a given code hosting platform, or if the platform itself disappears altogether, the code will not be lost forever.

A big scientific instrument for software, or lack thereof. With the growing importance of software, it is increasingly more important to provide the means to improve its quality, safety, and security properties. Sadly we lack a research instrument to analyze the whole body of publicly available source code.

To build such a "very large telescope" of source code—in the spirit of mutualized research infrastructures for physicists such as the Very Large Telescope in the Atacama Desert or the Large Hadron Collider in Geneva—we need a place where all information about software projects, their public source code, and their development history is made available in a uniform data model. This will allow to apply a large variety of "big code" techniques to analyze the entire corpus, independently of the origin of each source code artifact, and of the many different technologies currently used for hosting and distributing source code.

4 MISSION AND CHALLENGES

In order to address these three challenges, in June 2016 the Software Heritage project was unveiled, with initial support by Inria, with the stated goal to *collect, organize, preserve, and make easily accessible* all publicly available source code, independently of where and how it is being developed or distributed. The aim is to build a *common archival infrastructure*, supporting multiple use cases and applications (see Section 6), but all exhibiting synergies with long-term safeguard against the risk of permanent loss of source code.

To give an idea of the complexity of the task, let's just review some of the challenges faced by the initial source code harvesting phase, ignoring for the moment the many others that arise in subsequent stages. First, we need to identify the code hosting places where source code can be found, ranging from a variety of well known development platforms to raw archives linked from obscure web pages. There is no universal catalog: we need to build one!

Then we need then to discover and support the many different protocols used by code hosting platforms to list their contents, and maintain the archive up to date with the modifications made to projects hosted there. There is no standard, and while we hope to promote a set of best practices for preservation "hygiene", we must now cope with the current lack of uniformity.

We must then be able to crawl development histories as captured by a wide variety of version control systems [28]: Git, Mercurial, Subversion, Darcs, Bazaar, CVS, are just some examples of the tools that need to be supported. Also, there is no grand unifying data model for version control systems: one needs to be built.

To face such challenges, it is important that computer scientist get directly involved: source code is the DNA of their discipline and they must be at the forefront when it comes to designing the infrastructure to preserve it.

5 CORE PRINCIPLES

Building the Software Heritage archive is a complex task, which requires long term commitment. To maximize the chances of success, we based this work on solid foundations, presented in this section as a set of core principles for the project.

Transparency and Free Software. As stated by Rosenthal [31], in order to ensure long term preservation of any kind of information it is necessary to know the inner workings of all tools used to implement and run the archive. That is why Software Heritage will develop and release exclusively Free/Open Source Software (FOSS) components to build its archive—from user-facing services down to the recipes of software configuration management tools used for the operations of project machines. According to FOSS development best practices development is conducted collaboratively on the project forge[2] and communication happens via publicly accessible media.

Replication all the way down. There is a plethora of threats, ranging from technical failures, to mere legal or even economic decisions that might endanger long-term source code preservation. We know that we cannot entirely avoid them. Therefore, instead of attempting to create a system without errors, we design a system which tolerates them.

To this end, we will build replication and diversification in the system at all levels: a geographic network of mirrors, implemented using a variety of storage technologies, in various administrative domains, controlled by different institutions, and located in different jurisdictions. Releasing our own code as FOSS is expected to further ease the deployment of mirrors by a variety of actors.

Multi-stakeholder and non-profit. Experience shows that a single for profit entity, however powerful, does not provide sufficient durability guarantees in the long term. We believe that for Software Heritage it is essential to build a non profit foundation that has as its explicit objective the collection, preservation, and sharing of our software commons.

In order to minimise the risk of having a single points of failure at the institutional level, this foundation needs to be supported by various partners from civil society, industry, and governments, and must provide value to all areas which may take advantage of the existence of the archive, ranging from the preservation of cultural heritage to research, from industry to education, (see Section 6).

The foundation should be run transparently according to a well-documented governance, and should be accountable to the public by reporting periodicly about its activities.

No a priori selection. A natural question that arises when building a long term archive is what should be archived in it among the many candidates available. In building Software Heritage we have decided to avoid any *a priori* selection of software projects, and rather archive them all.

The first reason behind this choice is pragmatic: we have the technical ability to archive every software project available. Source code is usually small in comparison to other digital objects, information dense and expensive to produce, unlike the millions of (cat) pictures and videos exchanged on social media. Additionally, source code is heavily redundant/duplicated, allowing for efficient storage approaches (see Section 7).

Second, software is nowadays massively developed in the open, so we get access to the history of software projects since their very early phase. This is a precious information for understanding how software is born and evolves and we want to preserve it for any "important" project. Unfortunately, when a project is in its infancy it is extremely hard to know whether it will grow into a king or a peasant. Consider PHP: when it was released in 1995 by Rasmus Lerdorf as PHP/FI (Personal Home Page tools, Forms Interpreter), who would have thought that it would have grown into the most popular Web programming language 20 years later?

Hence our approach to archive everything available: important projects will be pointed at by external authorities, emerging from the mass, less relevant ones will drift into oblivion.

Source code first. Ideally, one might want to archive software source code "in context", with as much information about its broader ecosystem: project websites, issues filed in bug tracking systems, mailing lists, wikis, design notes, as well as executables built for various platforms and the physical machines and network environment on which the software was run, allowing virtualization in the future. In practice, the resources needed for doing all this would be enormous, especially considering the *no a priori selection* principle, and we need to draw the scope line somewhere.

Software Heritage will archive *the entire source code* of software projects, together with their *full development history* as it is captured by state-of-the-art *version control systems* (or "VCS").

On one side, this choices allow to capture relevant context for future generations of developers—e.g., VCS history includes commit messages, precious information that detail *why* specific changes have been made to a given software at a given moment—and is precisely what currently nobody else comprehensively archives.

On the other side, a number of other digital preservation initiatives are already addressing some of the other contextual aspects we have mentioned: the Internet Archive [18] is archiving project

[2]https://forge.softwareheritage.org/

websites, wikis, and web-accessible issue trackers; Gmane [12] is archiving mailing lists; several initiatives aim at preserving software executables, like Olive [21], the Internet Archive, KEEP [9], E-ARK [8], and the PERSIST project [36], just to mention a few.

In a sense, Software Heritage embraces the Unix philosophy [30], focusing on the source code only, where it contribution is most relevant, and will go to great lengths to make sure that the source code artifacts it archives are easy to reference from other digital archives, using state-of-the-art linked data [3] technologies, paving the way to a future "semantic wikipedia" of software.

Intrinsic identifiers. The quest for the "right" identifier for digital objects has been raging for quite a while [2, 29, 37], and it has mainly focused on designing *digital identifiers* for objects that are not necessarily natively digital, like books or articles. Recent software development practices has brought to the limelight the need for *intrinsic identifiers* of natively digital objects, computed only on the basis of their (byte) *content*.

Modern version control systems like Git [6] no longer rely on artificial opaque identifiers that need third party information to be related to the software artifacts they designate. They use identifiers that can be *computed* from the object itself and are tightly connected to it; we call these identifiers *intrinsic*. The SHA1 cryptographic hash [7] is the most used approach for computing them today. The clear advantage of crypto-hard intrinsic identifiers is that they allow to check that an obtained object is exactly the one that was requested, without having to involve third party authorities.

Intrinsic identifiers also natively support integrity checks—e.g., you can detect alteration of a digital object for which an intrinsic identifiers was previously computed as a mismatch between its (current) content and its (previous) identifier—which is a very good property for any archival system.

Software Heritage will use intrinsic identifier for all archived source code. Pieces of information that are not natively digital, such as author or project names, metadata, or ontologies, non-intrinsic identifier will also be used. But for the long term preservation of the interconnected network of knowledge that is built natively by source code, intrinsic identifiers are best suited.

Facts and provenance. Following best archival practices, Software Heritage will store full *provenance* information, in order to be able to always state *what* was found *where* and *when*.

In addition, in order to become a shared and trusted knowledge base, we push this principle further, and we will store only *qualified facts* about software. For example, we will not store bare metadata stating that the programming language of a given file is, say, `C++`, or that its license is `GPL3`. Instead we will store qualified metadata stating that version `3.1` of the program `pygments` invoked with a given command line on this particular file reported it as written in `C++`; or that version `2.6` of the `FOSSology` license detection tool, ran with a given configuration (also stored), reported the file as being released under the terms of version 3 of the GPL license.

Minimalism. We recognize that the task that Software Heritage is undertaking is daunting and has wide ramifications. Hence we focus on building a core infrastructure whose objective is *only* collecting, organizing, preserving, and sharing source code, while establishing collaborations with any initiative that may add value on top or on the side of this infrastructure.

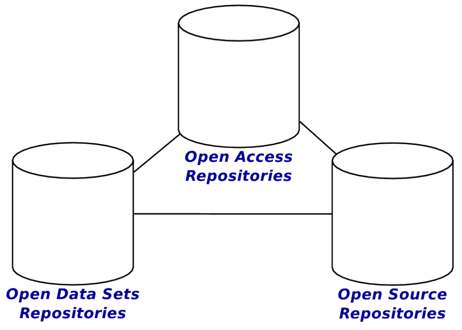

Figure 1: The scientific knowledge preservation trifecta

6 APPLICATIONS AND USE CASES

A universal archive of software source code enables a wealth of applications in a variety of areas, broader than preservation for its own sake. Such applications are relevant to the success of the archive *itself* though, because long term preservation carries significant costs: chances to meet them will be much higher if there are more use cases than just preservation, as the cost may then be shared among a broader public of potential archive users.

Cultural heritage. Source code is clearly starting to be recognized as a first class citizen in the area of cultural heritage, as it is a noble form of human production that needs to be preserved, studied, curated, and shared. Source code preservation is also an essential component of a strategy to defend against digital dark age scenarii [1, 5] in which one might lose track of how to make sense of digital data created by software currently in production.

For these reasons Inria has established an agreement with UNESCO on source code preservation,[3] whose main actions will be carried on in the context of Software Heritage.

Science. In the long quest for making modern scientific results reproducible, and pass the scientific knowledge over to future generations of researchers, the three main pillars are: scientific articles, that describe the results, the data sets used or produced, and the software that embodies the logic of the data transformation, as shown in Figure 1.

Many initiatives have been taking care of two of these pillars, like OpenAire [27] for articles and Zenodo [39] for data,[4] but for software source code, researchers keep pointing from their articles to disparate locations, if any, where their source code can be found: web pages, development forges, publication annexes, etc. By providing a central archive for all publicly available source code, Software Heritage contributes a significant building block to the edifice of reproducibility in all fields of science.

And there is more: in the specific field of Computer Science, there is a significant added value in providing a central repository where all the history of public software development is made available in a

[3]http://fr.unesco.org/events/ceremonie-signature-du-partenariat-unescoinria-preservation-partage-du-patrimoine-logiciel
[4]and more recently for self-selected software releases distributed via GitHub

uniform data model. It enables unprecedented big data analysis both on the code itself and the software development process, unleashing a new potential for Mining Software Repository research [15].

Industry. Industry is growing more and more dependant on FOSS components, which are nowadays integrated in all kinds of products, for both technical and economic reasons. This tidal wave of change in IT brought new needs and challenges: ensuring technical compatibility among software components is no longer enough, one also needs to ensure compliance with several software licenses, as well as closely track software supply chain, and bills of materials to identify which specific variants of FOSS components were used in a given product.

Software Heritage makes two key contributions to the IT industry that can be leveraged in software processes. First, Software Heritage intrinsic identifiers can precisely pinpoint specific software versions, independently of the original vendor or intermediate distributor. This *de facto* provides the equivalent of "part numbers" for FOSS components that can be referenced in quality processes and verified for correctness independently from Software Heritage (they are intrinsic, remember?).

Second, Software Heritage will provide an open provenance knowledge base, keeping track of which software component—at various granularities: from project releases down to individual source files—has been found where on the Internet and when. Such a base can be referenced and augmented with other software-related facts, such as license information, and used by software build tools and processes to cope with current development challenges.

The growing support and sponsoring for Software Heritage coming from industry players like Microsoft, Huawei, Nokia, and Intel provides initial evidence that this potential is being understood.

7 DATA MODEL

In any archival project the choice of the underlying data model—at the *logical* level, independently from how data is actually stored on physical media—is paramount. The data model adopted by Software Heritage to represent the information that it collects is centered around the notion of *software artifact*, that is at the key component of the Software Heritage archive, and we describe it in what follows. It is important to notice that according to our principles, we must store with every software artifact full information on where it has been found (provenance), that is also captured in our data model, so we start by providing some basic information on the nature of this provenance information.

7.1 Source code hosting places

Currently, Software Heritage uses of a curated list of *source code hosting places* to crawl. The most common entries we expect to place in such a list are popular collaborative development forges (e.g., GitHub, Bitbucket), package manager repositories that host source package (e.g., CPAN, npm), and FOSS distributions (e.g., Fedora, FreeBSD). But we may of course allow also more niche entries, such as URLs of personal or institutional project collections not hosted on major forges.

While currently entirely manual, the curation of such a list might easily be semi-automatic, with entries suggested by fellow archivists and/or concerned users that want to notify Software Heritage of the need of archiving specific pieces of endangered source code. This approach is entirely compatible with Web-wide crawling approaches: crawlers capable of detecting the presence of source code might enrich the list. In both cases the list will remain curated, with (semi-automated) review processes that will need to pass before a hosting place starts to be used.

7.2 Software artifacts

Once the hosting places are known, they will need to be periodically looked at in order to add to the archive missing software artifacts. Which software artifacts will be found there?

In general, each software distribution mechanism will host multiple releases of a given software at any given time. For VCS, this is the natural behaviour; for software packages, while a single version of a package is just *a* snapshot of the corresponding software product, one can often retrieve both current and past versions of the package from its distribution site.

By reviewing and generalizing existing VCS and source package formats, we have identified the following recurrent artifacts as commonly found at source code hosting places. They form the basic ingredients of the Software Heritage archive:[5]

file contents (AKA "blobs") the raw content of (source code) files as a sequence of bytes, *without* file names or any other metadata. File contents are often recurrent, e.g., across different versions of the same software, different directories of the same project, or different projects all together.

directories a list of *named* directory entries, each of which pointing to other artifacts, usually file contents or sub-directories. Directory entries are also associated to arbitrary metadata, which vary with technologies, but usually includes permission bits, modification timestamps, etc.

revisions (AKA "commits") software development within a specific project is essentially a time-indexed series of copies of a single "root" directory that contains the entire project source code. Software evolves when a developer modifies the content of one or more files in that directory and record their changes.

Each recorded copy of the root directory is known as a "revision". It points to a fully-determined directory and is equipped with arbitrary metadata. Some of those are added manually by the developer (e.g., commit message), others are automatically synthesized (timestamps, preceding commit(s), etc).

releases (AKA "tags") some revisions are more equals than others and get selected by developers as denoting important project milestones known as "releases". Each release points to the last commit in project history corresponding to the release and might carry arbitrary metadata—e.g., release name and version, release message, cryptographic signatures, etc.

Additionally, the following crawling-related information are stored as provenance information in the Software Heritage archive:

origins code "hosting places" as previously described are usually large platforms that host several unrelated software

[5]as the terminology varies quite a bit from technology to technology, we provide both the canonical name used in Software Heritage and popular synonyms

projects. For software provenance purposes it is important to be more specific than that. Software origins are fine grained references to where source code artifacts archived by Software Heritage have been retrieved from. They take the form of ⟨*type*, *url*⟩ pairs, where *url* is a canonical URL (e.g., the address at which one can `git clone` a repository or `wget` a source tarball) and *type* the kind of software origin (e.g., `git`, `svn`, or `dsc` for Debian source packages).

projects as commonly intended are more abstract entities that precise software origins. Projects relate together several development resources, including websites, issue trackers, mailing lists, as well as software origins as intended by Software Heritage.

The debate around the most apt ontologies to capture project-related information for software hasn't settled yet, but the place projects will take in the Software Heritage archive is fairly clear. Projects are abstract entities, which will be arbitrarily nestable in a versioned project/sub-project hierarchy, and that can be associated to arbitrary metadata as well as origins where their source code can be found.

snapshots any kind of software origin offers multiple pointers to the "current" state of a development project. In the case of VCS this is reflected by *branches* (e.g., master, development, but also so called feature branches dedicated to extending the software in a specific direction); in the case of package distributions by notions such as *suites* that correspond to different maturity levels of individual packages (e.g., stable, development, etc.).

A "snapshot" of a given software origin records all entry points found there and where each of them was pointing at the time. For example, a snapshot object might track the commit where the master branch was pointing to at any given time, as well as the most recent release of a given package in the stable suite of a FOSS distribution.

visits links together software origins with snapshots. Every time an origin is consulted a new visit object is created, recording when (according to Software Heritage clock) the visit happened and the full snapshot of the state of the software origin at the time.

7.3 Data structure

With all the bits of what we want to archive in place, the next question is how to organize them, i.e., which logical data structure to adopt for their storage. A key observation for this decision is that source code artifacts are massively duplicated. This is so for several reasons:

- code hosting diaspora discussed in Section 3;
- copy/paste (AKA "vendoring") of parts or entire external FOSS software components into other software products;
- large overlap between revisions of the same project: usually only a very small amount of files/directories are modified by a single commit;
- emergence of DVCS (*distributed* version control systems), which natively work by replicating entire repository copies around. GitHub-style pull requests are the pinnacle of this, as they result in creating an additional repository copy at each change done by a new developer;
- migration from one VCS to another—e.g., migrations from Subversion to Git, which are really popular these days—resulting in additional copies, but in a different distribution *format*, of the very same development histories.

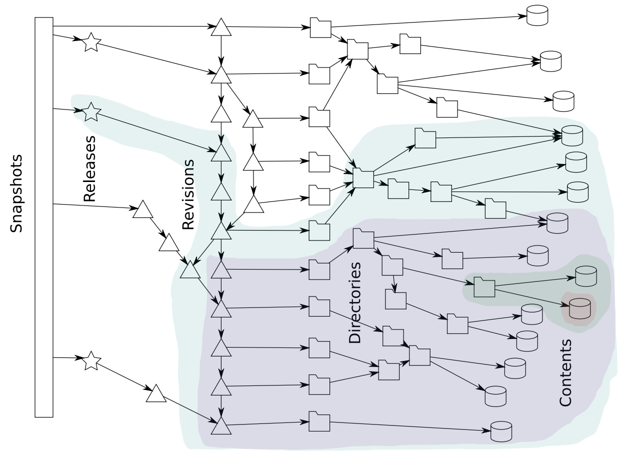

Figure 2: Software Heritage direct acyclic graph data model

These trends seem to be neither stopping nor slowing down, and it is reasonable to expect that they will be even *more* prominent in the future, due to the decreasing costs of storage and bandwidth.

For this reason we argue that any sustainable storage layout for archiving source code in the very long term should support *deduplication*, allowing to pay for the cost of storing source code artifacts that are encountered more than once…only once. For storage efficiency, deduplication should be supported for all the software artifacts we have discussed, namely: file contents, directories, revisions, releases, snapshots.

Realizing that principle, the Software Heritage archive is conceptually a single (big) Merkle Direct Acyclic Graph [24] (DAG), as depicted in Figure 2. In such a graph each of the artifacts we have described—from file contents up to entire snapshots—correspond to a node. Edges between nodes emerge naturally: directory entries point to other directories or file contents; revisions point to directories and previous revisions, releases point to revisions, snapshots point to revisions and releases. Additionally, each node contains all metadata that are specific to the node itself rather than to pointed nodes; e.g., commit messages, timestamps, or file names. Note that the structure is really a DAG, and not a tree, due to the fact that the line of revisions nodes might be forked and merged back.

In a Merkle structure each node is identified by an intrinsic identifier (as per our principles detailed in Section 5) computed as a cryptographic hash of the node content. In the case of Software Heritage identifiers are computed taking into the account both node-specific metadata and the identifiers of child nodes.

Consider the revision node shown in Figure 3. The node points to a directory, whose identifier starts with `fff3cc22...`, which has also been archived. That directory contains a full copy, at a specific point in time, of a software component—in the example

```
directory: fff3cc22cb40f71d26f736c082326e77de0b7692
parent: e4feb05112588741b4764739d6da756c357e1f37
author: Stefano Zacchiroli <zack@upsilon.cc>
date: 1443617461 +0200
committer: Stefano Zacchiroli <zack@upsilon.cc>
commiter_date: 1443617461 +0200
message:
  objstorage: fix tempfile race when adding objects

  Before this change, two workers adding the same
  object will end up racing to write <SHA1>.tmp.
  [...]
```

revision_id: 64a783216c1ec69dcb267449c0bbf5e54f7c4d6d

Figure 3: A revision node in the Software Heritage DAG

a component that we have developed ourselves for the needs of Software Heritage. The revision node also points to the preceding revision node (`e4feb051...`) in the project development history. Finally, the node contains revision-specific metadata, such as the author and committer of the given change, its timestamps, and the message entered by the author at commit time.

The identifier of the revision node itself (`64a78321...`) is computed as a cryptographic hash of a (canonical representation of) all the information shown in Figure 3. A change in any of them—metadata and/or pointed nodes—would result in an entirely different node identifier. All other types of nodes in the Software Heritage archive behave similarly.

The Software Heritage archive inherits useful properties from the underlying Merkle structure. In particular, deduplication is built-in. Any software artifacts encountered in the wild gets added to the archive only if a corresponding node with a matching intrinsic identifier is not *already* available in the graph—file content, commits, entire directories or project snapshots are all deduplicated incurring storage costs only once.

Furthermore, as a side effect of this data model choice, the entire development history of all the source code archived in Software Heritage—which ambitions to match all published source code in the world—is available as a unified whole, making emergent structures such as code reuse across different projects or software origins, readily available. Further reinforcing the use cases described in Section 6, this object could become a veritable "map of the stars" of our entire software commons.

8 ARCHITECTURE & DATA FLOW

Both the data model described in the previous section and a software architecture suitable for ingesting source code artifacts into it have been implemented as part of Software Heritage.

8.1 Listing

The ingestion data flow of Software Heritage is shown in Figure 4. Ingestion acts like most search engines, periodically crawling a set of "leads" (in our case the curated list of code hosting places discussed in Section 7) for content to archive and further leads. To facilitate software extensibility and collaboration, ingestion is split in two conceptual phases though: listing and loading.

Listing takes as input a single hosting place (e.g., GitHub, PyPi, or Debian) and is in charge of enumerating all software origins (individual Git or Subversion repositories, individual package names, etc.) found there at listing time.

The details of how to implement listing vary across hosting platforms, and dedicated lister software components need to be implemented for each different *type* of platform. This means that dedicated listers exist for GitHub or Bitbucket, but that the GitLab lister—GitLab being a platform that can be installed on premises by multiple code hosting providers—can be reused to list the content of any GitLab instance out there.

Listing can be done fully, i.e., collecting the entire list of origins available at a given hosting place at once, or incrementally, listing only the new origins since the last listing. Both listing disciplines are necessary: full listing is needed to be sure that no origin is being overlooked, but it might be unwieldy if done too frequently on large platforms (e.g., GitHub, with more than 55 million Git repositories as of early 2017), hence the need of incremental listing to quickly update the list of origins available at those places.

Also, listing can be performed in either pull or push style. In the former case the archive periodically checks the hosting places to list origins. In the latter code hosting sites, properly configured to work with Software Heritage, contact back the archive at each change in the list of origins. Push looks appealing at first and might minimize the lag between the appearance of a new software origin and its ingestion in Software Heritage. On the other hand push-only listing is prone to the risk of losing notifications that will result in software origins not being considered for archival. For this reason we consider push an optimization to be added on top of pull, in order to reduce lag where applicable.

8.2 Loading

Loading is responsible of the actual ingestion in the archive of source code found at known software origins.

Loaders are the software components in charge of extracting source code artifacts from software origins and adding them to the archive. Loaders are specific to the technology used to distribute source code: there will be one loader for each type of version control system (Git, Subversion, Mercurial, etc.) as well as one for each source package format (Debian source packages, source RPMs, tarballs, etc).

Loaders natively deduplicate w.r.t. the entire archive, meaning that any artifact (file content, revision, etc.) encountered at any origin will be added to the archive only if a corresponding node cannot be found in the archive as a whole.

Consider the Git repository used for the development of the Linux kernel, which is fairly big, totaling 2 GB on disks for more than 600 000 revisions and also widely popular with thousands of (slightly different) copies available only on GitHub. At its first encounter ever, the Git loader will load essentially all its file contents, revisions, etc., into the Software Heritage archive. At the next encounter of an identical repository, nothing will be added at all. At the encounter of a slightly different copy, e.g., a repository containing a dozen additional commits not yet integrated in the

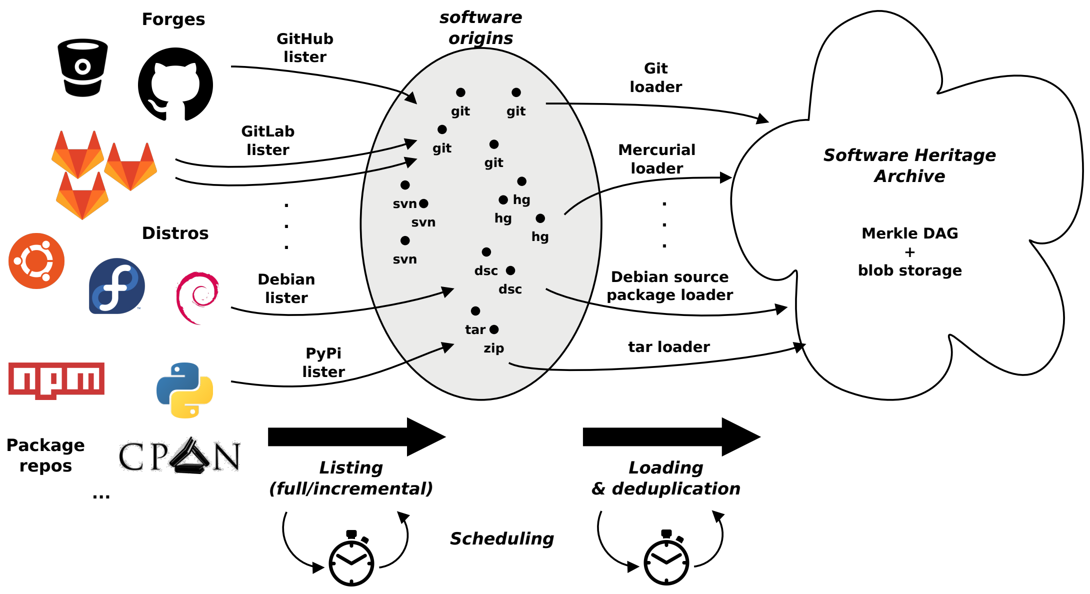

Figure 4: Ingestion data flow in Software Heritage

official release of Linux, only the corresponding revision nodes, as well as the new file contents and directories pointed by them, will be loaded into the archive.

8.3 Scheduling

Both listing and loading happen periodically on a schedule.[6] The *scheduler* component of Software Heritage is in charge of keeping track of when the next listing/loading actions need to happen, for each code hosting place (for listers) or software origins (loaders).

While the amount of hosting places to list is not enormous, the amount of individual software origins can easily reach the hundreds of millions given the current size of major code hosting places. Listing/loading from that many Internet sites too frequently would be unwise in terms of resource consumption, and also unwelcome by the maintainers of those sites. This is why we have adopted an adaptive scheduling discipline that strikes a good balance between update lag and resource consumption.

Each run of periodic action, such as listing or loading, can be "fruitful" or not. It is fruitful if and only if it resulted in new information since the last visit. For instance, listing is fruitful if it resulted in the discovery of new software origins; loading is if the overall state of the consulted origin differs from the last observed one. If a scheduled action has been fruitful, the consulted site has seen some activity since the last visit, and we will increase the frequency at which that site will be visited in the future; in the converse case (no activity), visit frequency will be decreased.

Specifically, Software Heritage adopts an *exponential backoff* strategy, in which the visit period is halved when activity is noticed, and doubled when no activity has been observed. Currently, the fastest a given site will be consulted is twice day (i.e., every 12 hours) and the slowest is every 64 days. Early experiences with large code hosting sites seem to tell that ≈90% of the repositories hosted there quickly fall to the slowest update frequency (i.e., they don't see any activity in 2-month time windows), with only the remaining ≈10% seeing more activity than that.

[6]as discussed, even when listing is performed in push style, we still want to periodically list pull-style to stay on the safe side, so scheduling is always needed for listing as well

8.4 Archive

At a logical level, the Software Heritage archive corresponds to the Merkle DAG data structure described in Section 7. On disk, the archive is stored using different technologies due to the differences in the size requirements for storing different parts of the graph.

File content nodes require the most storage space as they contain the full content of all archived source code files. They are hence stored in a key-value object storage that uses as keys the intrinsic node identifiers of the Merkle DAG. This allows trivial distribution of the object storage over multiple machines (horizontal scaling) for both performance and redundancy purposes. Also, the key-value access paradigm is very popular among current storage technologies, allowing to easily host (copies of) the bulk of the archive either on premise or on public cloud offerings.

The rest of the graph is stored in a relational database (RDBMS), with roughly one table per type of node. Each table uses as primary key the intrinsic node identifier and can easily be sharded (horizontal scaling again) across multiple servers. Master/slave replication and point-in-time recovery can be used for increased performance and recovery guarantees. There is no profound reason for storing this part of the archive in a RDBMS, but for what is worth our early experiments seem to show that graph database technologies are not yet up to par with the size and kind of graph that Software Heritage already is with its current coverage (see Section 9).

A weakness of deduplication is that it is prone to hash collisions: if two *different* objects hash to the same identifier there is a risk of storing only one of them while believing to have stored them both. For this reason, where checksums algorithms are no longer considered strong enough for cryptographic purposes,[7] we use multiple checksums, with unicity constraints on *each* of them, to detect collisions before adding a new artifact to the Software Heritage archive. For instance, we do not trust SHA1 checksums alone when adding new file contents to the archive, but also compute SHA256, and "salted" SHA1 checksums (in the style of what Git does). Also, we are in the process of adding BLAKE2 checksums to the mix.

Regarding mirroring, each type of node is associated to a change feed that takes note of all changes performed to the set of those objects in the archive. Conceptually, the archive is append-only, so under normal circumstances each feed will only lists additions of new objects as soon as they get ingested into the archive. Feeds are persistent and the ideal branching point for mirror operators who, after an initial full mirror step, can cheaply remain up to date w.r.t. the main archive.

On top of the object storage, an archiver software component is in charge of both enforcing retention policies and automatically heal object corruption if it ever arises, e.g., due to storage media decay. The archiver keeps track of how many copies of a given file content exist and where each of them is—we currently operate two in-house mirror of the entire object storage, plus a third copy currently being populated on a public cloud. The archiver is aware of the desired retention policy, e.g., "each file content must exist in at least 3 copies", and periodically swipe all known objects for adherence to the policy. When fewer copies than desired are known to exist, the archiver asynchronously makes as many additional copy as needed to satisfy the retention policy.

The archiver also periodically checks each copy of all known objects—randomly selecting them at a suitable frequency—and verifies it for integrity by recomputing its intrinsic identifier and comparing it with the known one. In case of mismatch all known copies of the object are checked on-the-fly again; assuming at least one pristine copy is found, it will be used to overwrite corrupted copies, "healing" them automatically.

9 CURRENT STATUS & ROADMAP

The Software Heritage archive grows incrementally over time as long as new listers/loaders get implemented and periodically run to ingest new content.

Listers. In terms of listers, we initially focused on targeting GitHub as it is today by far the largest and most popular code hosting platform. We have hence implemented and put in production a GitHub lister, capable of both full and incremental listing. Additionally, we have recently put in production a similar lister for Bitbucket. Common code among the two has been factored out to an internal lister helper component that can be used to easily implement listers for other code hosting platforms.[8] Upcoming listers include FusionForge, Debian and Debian-based distributions, as well as a lister for bare bone FTP sites distributing tarballs.

Loaders. Regarding loaders, we initially focused on Git as, again, the most popular VCS today. We have additionally implemented loaders for Subversion, tarballs, and Debian source packages. A Mercurial loader is also in the working.

[7] note that this is already a higher bar than being strong enough for archival purposes
[8] see https://www.softwareheritage.org/?p=9594 for a detailed technical description

Archive coverage. Using the above software components we have already been able to assemble what, to the best of our knowledge, is the largest software source code archive in existence.

We have fully archived once, and routinely maintain up-to-date, GitHub into Software Heritage, for more than 50 million Git repositories. GitHub itself has acknowledged Software Heritage role as 3rd-party archive of source code hosted there.[9]

Additionally we have archived, as one shot but significant in size archival experiments, all releases of each Debian package in between 2005–2015, and all current and historical releases of GNU projects as of August 2015. We have also retrieved full copies of all repositories that were previously available from Gitorious and Google Code, now both gone. At the time of writing the process of ingesting those repositories into Software Heritage is ongoing.

In terms of storage, each copy of the Software Heritage object storage currently occupies ≈150 TB of individually compressed file contents. The average compression ration is 2x, corresponding to 300 TB of raw source code content. Each copy of the RDBMS used to store the rest of the graph (Postgres) takes ≈5 TB. We currently maintain 3 copies of the object storage and 2 copies of the database, the latter with point-in-time recovery over a 2-week time window.

As a logical graph, the Software Heritage Merkle DAG has ≈5 billion nodes and ≈50 billion edges. We note that more than half of the nodes are (unique) file contents (≈3 B) and that there are ≈750 M revision/commit nodes, collected from ≈55 M origins.

Features. The following functionalities are currently available for interacting with the Software Heritage archive:

content lookup allows to check whether specific file contents have been archived by Software Heritage or not. Lookup is possible by either uploading the relevant files or by entering their checksum, directly from the Software Heritage homepage.

browsing via API allows developers to navigate through the entire Software Heritage archive as a graph. The API offered to that end is Web-based and permits to lookup individual nodes (revisions, releases, directories, etc.), access all their metadata, follow links to other nodes, and download individual file contents. The API also gives access to visit information, reporting when a given software origin has been visited and what its status was at the time.

The API technical documentation[10] has many concrete examples of how to use it in practice.

The following features are part of the project technical road map and will be rolled out incrementally in the future:

Web browsing equivalent to API browsing, but more convenient for non-developer Web users. The intended user interface will resemble state-of-the art interfaces for browsing the content of individual version control systems, but will be tailored to navigate a much larger archive.

provenance information will offer "reverse lookups" of sort, answering questions such as "give me all the places and timestamps where you have found a given source code artifact".

[9] https://help.github.com/articles/about-archiving-content-and-data-on-github/
[10] https://archive.softwareheritage.org/api/

This is the key ingredient to address some of the industrial use cases discussed in Section 6.

metadata search will allow to perform searches based on project-level metadata, from simple information (e.g., project name or hosting place), to more substantial ones like the entity behind the project, its license, etc.

content search conversely, content search will allow to search based on the *content* of archived files. Full-text search is the classic example of this, but in the context of Software Heritage content search can be implemented at various level of "understanding" of the content of individual files, from raw character sequences to full-fledged abstract syntax trees for a given programming language.

10 CONCLUSIONS

Software Heritage is taking over the long overdue task of collecting *all publicly available source code*, with all its development history, organizing it into a canonical structure, and providing unique, intrinsic identifiers for all its parts, enabling its better fruition while ensuring its long term preservation.

This is a significant undertaking, that faces a broad range of challenges, from plain technical complexity to economic sustainability. To maximize the chances of success, the core design principles of Software Heritage include technical choices like archival minimalism and deduplication, up to organizational decisions like running the project as a multi-stakeholder, transparent, non-profit initiative, open to collaboration with other digital archival initiatives for all non source code aspects of software archiving.

Software Heritage is run as an open collaborative project, and we call for digital archivists and computer scientists to critically review our work and join the mission of archiving our entire software commons. Challenges abound, but we believe they are worth it.

ACKNOWLEDGMENTS

Software Heritage is a far reaching project that could not exist without the dedication and vision of many people and institutions that we would like to sincerely thank here.

First and foremost Inria as an institution, and its CEO Antoine Petit in particular, for providing the support needed to move from paper to reality. Jean-François Abramatic, who spent countless hours to make the project take off, as chair of the advisory board and "compagnon de route" of the first author. The Software Heritage development team, for their tireless commitment, with shout out for Guillaume Rousseau, Antoine R. Dumont, and Nicolas Dandrimont—team members since day one.

Last but not least the special people who worked internally at the many sponsor organizations convincing them to commit resources to this mission and the many others, too numerous to list here, who contributed along the way.

REFERENCES

[1] David Anderson. The digital dark age. *Communications of the ACM*, 58(12):20–23, 2015.
[2] William Y. Arms. Uniform resource names: handles, purls, and digital object identifiers. *Commun. ACM*, 44(5):68, 2001.
[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227, 2009.
[4] Michel Castagné. Consider the source: The value of source code to digital preservation strategies. *SLIS Student Research Journal*, 2(2):5, 2013.
[5] Vinton G Cerf. Avoiding" bit rot": Long-term preservation of digital information [point of view]. *Proceedings of the IEEE*, 99(6):915–916, 2011.
[6] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.
[7] Quynh Dang. Changes in federal information processing standard (fips) 180-4, secure hash standard. *Cryptologia*, 37(1):69–73, 2013.
[8] E-ark (european archival records and knowledge preservation) project. http://www.eark-project.com/, 2014.
[9] Keep: Eu cooperating. https://www.keep.eu/, 2000.
[10] Github archive. https://www.githubarchive.org/, 2017. Retrivede March 2017.
[11] GitLab. Gitlab acquires gitorious to bolster its on premises code collaboration platform. https://about.gitlab.com/2015/03/03/gitlab-acquires-gitorious/, 2015.
[12] Gmane. http://gmane.org, 2017.
[13] GNU. Gnu general public license, version 2, 1991. retrieved September 2015.
[14] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, *9th IEEE Working Conference of Mining Software Repositories, MSR*, pages 12–21. IEEE Computer Society, 2012.
[15] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.
[16] Google Project Hosting. Bidding farewell to google code. https://opensource.googleblog.com/2015/03/farewell-to-google-code.html, 2015.
[17] James Howison, Megan Conklin, and Kevin Crowston. Flossmole: A collaborative repository for FLOSS research data and analyses. *IJITWE*, 1(3):17–26, 2006.
[18] Internet archive: Digital library of free books, movies, music & wayback machine. https://archive.org, 2017.
[19] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.
[20] Nancy Kranich and Jorge Reina Schement. Information commons. *Annual Review of Information Science and Technology*, 42(1):546–591, 2008.
[21] Gloriana St Clair Mahadev Satyanarayanan, Benjamin Gilbert, Yoshihisa Abe, Jan Harkes, Dan Ryan, Erika Linke, and Keith Webster. One-click time travel. Technical report, Technical report, Computer Science, Carnegie Mellon University, 2015.
[22] Brian Matthews, Arif Shaon, Juan Bicarregui, and Catherine Jones. A framework for software preservation. *International Journal of Digital Curation*, 5(1):91–105, 2010.
[23] Steve Mead. Unique file identification in the National Software Reference Library. http://www.nsrl.nist.gov, 2014. smead@nist.gov.
[24] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.
[25] Audris Mockus. Large-scale code reuse in open source software. In *FLOOS'07: 1st International Workshop on Emerging Trends in FLOSS Research and Development.* IEEE, 2007.
[26] NIST. [US] national software reference library. http://www.nsrl.nist.gov, 2014.
[27] Openaire. https://www.openaire.eu/, 2014.
[28] Bryan O'Sullivan. Making sense of revision-control systems. *Communications of the ACM*, 52(9):56–62, 2009.
[29] Norman Paskin. Digital object identifier (doi) system. *Encyclopedia of library and information sciences*, 3:1586–1592, 2010.
[30] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.
[31] David Rosenthal, Rob Baxter, and Laurence Field. Towards a shared vision of sustainability for research and e-infrastructures. https://www.eudat.eu/news/towards-shared-vision-sustainability-research-and-e-infrastructures, 24-25 September 2014. EUDAT conference.
[32] Charles M Schweik and Robert C English. *Internet success: a study of open-source software commons*. MIT Press, 2012.
[33] Leonard J. Shustek. What should we collect to preserve the history of software? *IEEE Annals of the History of Computing*, 28(4):110–112, 2006.
[34] Megan Squire and David Williams. Describing the software forge ecosystem. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 3416–3425. IEEE, 2012.
[35] 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.
[36] Unesco persist programme. https://unescopersist.org/, 2015.
[37] Herbert Van de Sompel and Andrew Treolar. A perspective on archiving the scholarly web. *Proceedings of the International Conference on Digital Preservation (iPRES)*, pages 194–198, 2014.
[38] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. The top 100 papers. *Nature*, pages 550–553, October4 2014.
[39] Zenodo. https://zenodo.org/, 2013.