Source Code Archiving to the Rescue of Reproducible Deployment

Ludovic Courtès, Timothy Sample, Simon Tournier, Stefano Zacchiroli

Source Code Archiving to the Rescue of Reproducible Deployment

Ludovic Courtès
ludovic.courtes@inria.fr
Inria
Bordeaux, France

Timothy Sample
samplet@ngyro.com
Saskatoon, Canada

Simon Tournier
simon.tournier@inserm.fr
Université Paris Cité
Paris, France

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

The ability to *verify* research results and to *experiment* with methodologies are core tenets of science. As research results are increasingly the outcome of computational processes, software plays a central role. GNU Guix is a software deployment tool that supports *reproducible* software deployment, making it a foundation for computational research workflows. To achieve reproducibility, we must first ensure the source code of software packages Guix deploys remains available.

We describe our work connecting Guix with Software Heritage, the universal source code archive, making Guix the first free software distribution and tool backed by a stable archive. Our contribution is twofold: we explain the rationale and present the design and implementation we came up with; second, we report on the archival coverage for package source code with data collected over five years and discuss remaining challenges.

KEYWORDS

reproducible research, replicability, digital preservation, functional package management, Guix, Software Heritage

ACM Reference Format:
Ludovic Courtès, Timothy Sample, Simon Tournier, and Stefano Zacchiroli. 2024. Source Code Archiving to the Rescue of Reproducible Deployment. In *ACM Conference on Reproducibility and Replicability (ACM REP '24), June 18–20, 2024, Rennes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3641525.3663622

1 INTRODUCTION

Now that software is an integral part of scientific experimental workflows, it must be held to the same standards of other scientific workflows: computational workflows must be transparent and reproducible. While such a statement is becoming consensual among scholars, its implications are often less understood: source code must be publicly available, with a license that grants the right to use it, to study it, to modify it, and to share those modifications. UNESCO's Recommendation on Open Science [20] further states:

> In the context of open science, when open source code is a component of a research process, enabling reuse and replication generally requires that it be accompanied with open data and open specifications of the environment required to compile and run it.

That last part—the *specifications of the environment*—is often overlooked or dismissed: researchers often resort to either imprecise natural-language build instructions or large binary images ("containers" or virtual machines) that let others run the software, but typically prevent them from experimenting with the code [21].

Guix is a software deployment tool developed by a large community since 2012 and which is seeing growing adoption [6, 7]. It can be used as a "package manager" such as those found on GNU/Linux distributions—`apt`, `dnf`, `pip`, etc.—or as a standalone system—just like Debian, Fedora, etc. It builds upon the *functional software deployment* model pioneered by Nix [8] and *reproducible builds* [12], making it a tool of choice when deploying software for reproducible research workflows.

Instead of capturing a list of package names and versions, Guix lets users capture the entire graph of package definitions, which includes, beyond version numbers, all the information required to build the packages, including how to fetch the source code and what its cryptographic hash must be, configuration options, and dependencies—*recursively*. Once they have captured this information, users can run the `guix time-machine` command to re-deploy *the exact same software environment*, bit for bit, on a different machine or at a different point in time [21].

This, of course, can only be true if one necessary condition holds: that package source code is available. Indeed, when the source code of a package disappears, Guix (unsurprisingly) cannot build that package anymore and thus becomes unable to redeploy the software environment. The entire foundation for reproducible deployment collapses if the source of one or more of the packages in our environment disappears. This scenario is far from unrealistic: source code hosting sites come and go, even those backed by large companies deemed "too big to fail" [9].

With the mission to save and archive all publicly-available source code, Software Heritage [5] (SWH for short) has the potential to let us fill this gap.

ACM REP '24, June 18–20, 2024, Rennes, France

ACM ISBN 979-8-4007-0530-4/24/06.
https://doi.org/10.1145/3641525.3663622

Contributions and paper structure. In the vast field of reproducible research, we focus exclusively on software deployment and under the assumption that we are only dealing with free and open source software, in line with the open science recommendations mentioned above. The following sections describe what it meant to "connect" Software Heritage and Guix, and the roadblocks we had to overcome. Section 3 describes our first contribution: the design and implementation of our bridge between Guix and SWH, including novel tools developed to address sub-problems. Section 4 provides an evaluation of the effectiveness of our solution, looking at 5 years worth of package source code referred to by Guix—our second contribution. Lastly, we describe related work and conclude on challenges that remain to be addressed.

Reproducibility package. A full reproducibility package for this work is available. See Section 7 for details.

2 BACKGROUND

To understand what we are trying to achieve, let us first describe the two components at play: Guix package definitions on one hand, and the Software Heritage archive on the other hand.

2.1 Guix Package Definitions

Guix is a software deployment tool that stands alone: it can only deploy software packages that have been *defined* in its own package collection. To date, Guix itself provides almost 30 000 packages, making it one of the five largest free software distributions according to Repology.[1] Each package definition specifies metadata, instructions on how to build the package, and references to dependencies (which are themselves other Guix packages). Like Nix and unlike Debian or Fedora, Guix at its core is a "source-based" deployment tool that builds software from source; it can optionally download pre-built binaries as a substitute for local compilation [7, 8].

Package definitions are declarative and embedded in the Scheme programming language [6]. Figure 1 shows the definition of the `python` package, simplified—we omitted fields that specify dependencies and build options. The `source` field declares an *origin* indicating that the source of `python` is the file `Python-3.10.7.tar.xz`, a so-called *tarball* to be downloaded over HTTPS. Crucially, the `sha256` field specifies the cryptographic hash of that file. When Guix fails to download the file, or if it obtains a different hash, it immediately aborts—an obvious prerequisite for reproducible software deployment.

The second package definition in Figure 1 is slightly different: source code is to be checked out using the Git version control system (VCS), from the `1.3.2` tag. The SHA256 hash, in this case, is that of the checked out directory once serialized as a so-called *normalized archive* or *nar*. The nar format was initially designed for Nix; unlike the venerable tar format (for "tape archive"), it omits Unix metadata unimportant in this context: file timestamps, access rights, and ownership information.

Package source can also be fetched through other methods: by referring to a Git commit rather than a tag, or by using a different version control system such as Subversion. Currently, 44% of the packages get their source from a VCS when it was only 22% back in 2019; Figure 3 shows how the distribution has changed over time. In all these cases, a cryptographic hash of the content is always specified, as in the examples above. Source code is thus essentially *content-addressed*, with the URL and download method serving more as a hint. An implication is that if those hints become stale—e.g., the file is no longer available at the given URL—users can work around the problem: if a copy of the file or checkout is available elsewhere, one can run the `guix download` command with that new URL to feed it to Guix. Guix then finds the source with the expected hash and proceeds building it.

```
(define-public python
  (package
    (name "python")
    (version "3.10.7")
    (source
     (origin
       (method url-fetch)
       (uri (string-append "https://www.python.org/ftp/python/"
                           version "/Python-" version ".tar.xz"))
       (sha256
        (base32
         "0j6wvh2ad5jjq5n7sjmj1k66mh6lipabavchc3rb4vsinwaq9vbf"))))
    ;; various fields omitted
    (license license:psfl)))

(define-public python-scikit-learn
  (package
    (name "python-scikit-learn")
    (version "1.3.2")
    (source
     (origin
       (method git-fetch)
       (uri (git-reference
             (url "https://github.com/scikit-learn/scikit-learn")
             (commit version)))
       (sha256
        (base32
         "1hr024vcilbjwlwn32ppadri0ypnzjmkfxhkkw8gih0qjvcvjbs7"))))
    ;; various fields omitted
    (license license:bsd-3)))
```

Figure 1: Package definitions of Python and Scikit-learn.

Our goal, as designers of a reproducible deployment tool, is to ensure package source code can always be retrieved automatically and reliably, even once the original source code hosting site has vanished or has been compromised. This is where Software Heritage comes in.

2.2 Source Code Archiving with Software Heritage

Software Heritage [5] (SWH) is a digital preservation initiative with the aim of collecting, preserving for the long-term, and providing access to the entire body of software, in the preferred form for making modifications to it (referred to as simply "source code" in this article). The SWH archive[2] is the largest publicly available archive of software source code. At the time of writing it contains more than 17 billion unique source code files and 3.5 billion commits, coming from more than 250 million projects.

The SWH data model is a deduplicated *Merkle graph* [14], with nodes of different types, corresponding to the software artifacts

[1] https://repology.org, accessed 2024-01-18.

[2] https://archive.softwareheritage.org, accessed 2024-01-18.

commonly stored in modern version control systems (VCSs): individual source code files and directories, commits, releases, etc. SWH hence actually stores the full development history of software projects, rather than only the most recent version of archived software products. This enables restoring a hash-compatible version a Git repository that has disappeared (or been tampered with) from its usual hosting place—provided it was archived in time.

Each node in the SWH graph data model can be referenced *via* persistent, intrinsic identifiers called *SWHIDs* [4], which are computed as Merkle-style SHA1 hashes. For example, `swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d` is a SWHID referencing an archived commit of the Darktable image processing software, where `rev` is the node type: here a "revision" (akin to a "commit" in Git parlance). The current version of SWHID identifiers (version 1) is compatible with Git SHA1 hash, which allows revision SWHIDs to be constructed from Git commit identifiers and vice-versa. (Note that merely constructing a syntactically correct SWHID from Git does not mean the corresponding Git object has actually been *archived* in SWH.)

The SWH archive is populated primarily by *crawlers* ("pull" style) that track major forges (e.g., GitHub, GitLab, …), GNU/Linux distributions (e.g., Debian), and package manager repositories (e.g., PyPI, NPM, …). "Push" style archiving is also available; particularly relevant for this work is the *Save Code Now* feature. It allows users to trigger on-demand archiving of a specific Git repository that has either never been archived before or ought to be *re*-archived promptly, before the pull crawlers have a chance to notice its latest archived copy is out-of-date.

Various mechanisms to access the SWH archive are available, depending on the use case. Users can browse it *via* a forge-like Web interface at https://archive.softwareheritage.org. Developers can integrate with SWH via various *application programming interfaces* (APIs): Web, gRPC, file system, and GraphQL-based. We rely on the Web API[3] for the Guix/SWH integration described in this paper, using it for verifying that sources have been archived and triggering push archiving of missing sources. Researchers can also analyze the SWH archive data in bulk, *via* public large-scale datasets [15].

For the specific need of reconstructing repositories from the archive (e.g., in case of disappearance from their previous hosting place), a dedicated asynchronous service called the *Vault* is provided. Its need comes from the fact that, due to deduplication, an individual repository is stored by SWH as a (sub-)graph made of many nodes—e.g., tens of millions files, commits, etc., for a project like the Linux kernel. A user interested in retrieving a specific version of the `linux.git` repository will then file a Vault "cooking" request, *via* either the Web user interface or the API, which will then be processed by a dedicated pool of workers. When the bundle is ready to be downloaded, the user is notified, *via* mail or a Webhook trigger.

3 IMPLEMENTATION

The connection between Guix and Software Heritage goes both ways: first we must ensure that the SWH archive ingests source code of packages Guix refers to, and second Guix must fall back to retrieving source code from SWH. The sections below look at these two cases.

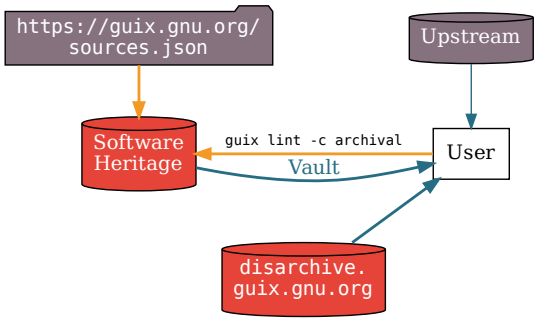

Figure 2: Populating the Software Heritage archive (orange arrows) and retrieving source code (blue arrows).

3.1 Populating the Archive

As we have seen, the SWH archive has already been ingesting source code from a variety of repositories, covering a large subset of the code referenced by Guix packages. Our goal is to achieve *full coverage*—ensuring that the archive contains all the source code referenced by Guix packages at any time. To achieve that, Guix explicitly triggers source code archiving *via* two mechanisms represented by the orange arrows in Figure 2.

First, Guix packagers routinely run the `guix lint` command on packages they submit for inclusion. This command checks almost 30 properties on packages, such as making sure that they respect certain conventions and that the URLs they refer to are reachable. We extended `guix lint` with an `archival` check that (1) checks whether the code is already archived, and (2) submits a Save Code Now request (see Section 2.2) to SWH, *via* the Web API, if the source is not already archived *and* if it is a VCS checkout—the Save Code Now interface rejects requests to save arbitrary tarballs.

The second mechanism complements this: Guix publishes https://guix.gnu.org/sources.json, which is a machine-readable list of all its origins—URLs and hashes, along with commits or tags when referring to VCS checkouts. SWH periodically ingests all the tarballs and VCS repositories referenced by this file using a dedicated *crawler*, as described in Section 2.2. When it ingests a VCS repository, SWH preserves all its history; when it ingests a tarball, SWH only preserves the *contents* of the tarball and not the tarball itself.

3.2 Retrieving VCS Checkouts

Ensuring that source code is archived is only half of the job. How can we retrieve VCS checkouts from the SWH archive once the original hosting site has disappeared?

The `sha256` field of origins in package definitions make them content-addressed. However, as we have seen before, Guix and SWH each use a different method to compute the hash of directories: Guix computes the hash of a "normalized archive" (nar) whereas SWH

[3] https://archive.softwareheritage.org/api/, accessed 2024-01-19.

computes the hash of a Git tree. For a long time this mismatch made it impossible to query content by nar hash—a problem that has now been fixed, as we will see below.What the SWH archive does permit, though, is to query the checkout associated with the SHA1 identifier of a Git commit, and to browse the VCS snapshots of a given URL. We can distinguish several cases.

The easiest case is that of origins that refer to a Git commit by its SHA1 identifier. Using the Web API of the SWH archive, we can query the *directory* object associated with that identifier—this is effectively *content-addressed* access, made possible by the fact that revision identifiers in the SWH data model are equal to Git commit identifiers. To obtain the files comprised in that directory, Guix then uses the SWH Vault; if data has not been "cooked" yet, the download process has to wait until the Vault has made it available—see the blue arrow on Figure 2. Our implementation in Guix is transparent: package definitions do not need to be changed. Instead, Guix's download process automatically falls back to SWH when the URL specified in the origin is unreachable.

What about references to VCS *tags*? Git tags and commit identifiers illustrate the trilemma known as "Zooko's triangle" [22]. Compared to commit identifiers, tags have the advantage of being human-readable: the second example of Figure 1 makes it clear that the intent is to fetch a checkout for the tag corresponding to version 1.3.2 of Scikit-learn; for this reason, packagers often use them. But tags have two major drawbacks: they are not content-addressed, which make them context-dependent, and they are mutable—Git allows users to replace a tag pointing to a given commit with a tag pointing to a different commit.

In this case, Guix uses the SWH Web API to (1) look up the repository by URL, (2) look up the tag by name to obtain the corresponding commit identifier, and (3) download the corresponding directory from the Vault. We have anecdotal evidence that this process is "good enough" in most cases, but it is inherently brittle and could fail or return the wrong data: tags might have been modified, URLs might have been reused to host different code, etc. The worst that can happen is that Guix is unable to download the source, although SWH might contain it; hash mismatches are detected and cause Guix to abort, as we have seen in Section 2.

The fundamental mismatch in how Guix and SWH identify directories was addressed by a recent SWH feature deployed in January 2024: SWH now computes and exposes nar hashes for directories. These hashes are an extension of the SWH data model called *external identifiers* or *ExtIDs*; the Web API[4] lets us obtain the SWHID corresponding to a `nar-sha256` ExtID, which is exactly what is necessary to ensure content-addressed access in all cases. Consequently, the fallback code in Guix was changed to use that method. Since those ExtIDs have not yet been computed for previously-ingested origins, Guix still uses the method described earlier when lookup by `nar-sha256` fails.

The beauty of this content-addressed download method is that it works regardless of the origin type. In particular, it will still work when Git repositories start using SHA256 identifiers instead of SHA1—a feature that is slowly being deployed at this time—and it works for other version control systems too: Mercurial, Subversion, and CVS. Of these, Mercurial and CVS amount for less than 0.2% of the sources in the entire package collection, but Subversion amounts for 15.7% of all sources as it is used to retrieve the source of the more than 4000 individual TEX Live packages—see Figure 4.

[4]https://archive.softwareheritage.org/api/1/extid/doc/, accessed 2024-01-11.

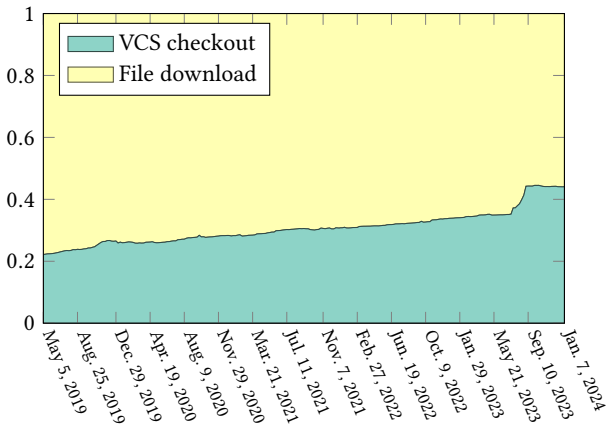

Figure 3: Relative high-level source types by sampled Guix commit.

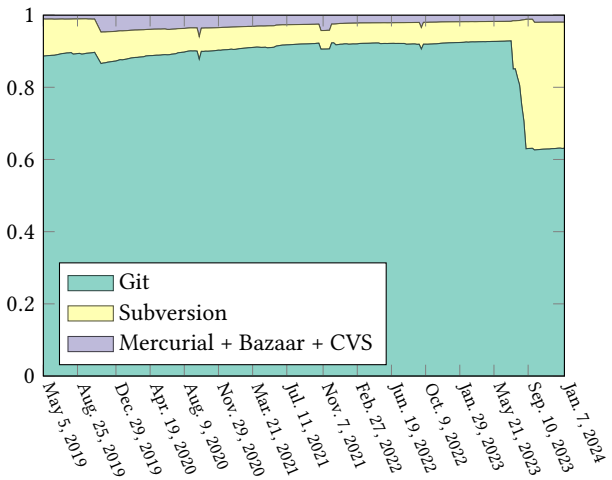

Figure 4: Relative VCS source types by sampled Guix commit.

3.3 Retrieving Source Code Tarballs

When SWH ingests source code from a tarball (or any other archival file format), it unpacks the tarball and stores only its contents, rather than keeping the entire file. This is the natural approach given its graph-oriented representation of software projects. On the other hand, Guix considers the tarball an atomic input and expects to be able to retrieve it intact and verify that it is unmodified. The process of creating a tarball is, in general, not reproducible: even when using the same inputs and tools, timestamps and nondeterminism can result in small differences in the resulting files [12]. Therefore, Guix cannot directly retrieve a tarball from SWH, nor can it retrieve the same contents and synthesize one that would pass its own verification.

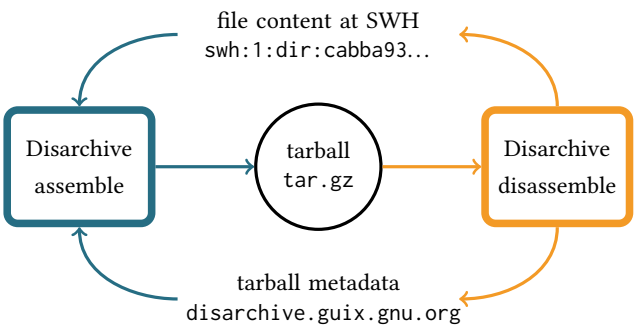

Figure 5: Disarchive tarball disassembly (orange arrows) takes a "tarball" as input and produces metadata along with a SWHID pointing to the tarball contents. Assembly (blue arrows) reconstructs the tarball by combining its metadata and its contents.

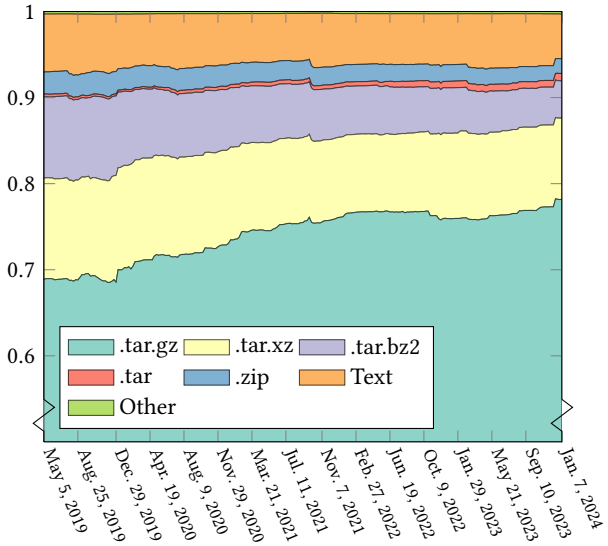

Figure 6: Relative download source types by sampled Guix commit (truncated at 50%).

To solve this mismatch, we have developed a way to decompose tarballs into two parts: the file system content and a description of the process that transformed it into the tarball. If the description is sufficiently detailed, we can run this process in reverse and arrive again at the original tarball given these two parts. Since SWH already stores the contents, if we could provide the corresponding description, Guix could revive the original tarball.

We created a tool called Disarchive[5] that implements this strategy: it can *disassemble* a tarball to obtain a description and a link to its contents in SWH, and can *assemble* that tarball given its contents and the description, as shown in Figure 5. Currently, Disarchive supports decomposing plain tar files as well as compressed files in gzip, bzip2, and XZ formats. These formats represent the majority of non-VCS Guix sources (see Figure 6). For tar files, it stores the metadata for each file in the order the files appeared in the original tarball: this includes file metadata (modification time, Unix owner and group, permissions) but also low-level details about the tar headers (details in the representation of tar members that vary between tar implementations and variants).

To describe a compressed file, Disarchive guesses the compression system used and then verifies its guess—e.g., it tries both GNU gzip and zlib, with various flags, until it matches the given gzip file. This brute-force approach is admittedly not very elegant, but it works in practice. Testing shows that 97.3% of the 41,521 compressed tarballs referred to by Guix (see Section 4.2) can be disassembled using this approach.

The tarball description resulting from Disarchive's disassemble step looks like that shown in Figure 7. It is a tree (a Scheme s-expression) showing, in this case, the components of the gzipped tar file: there is first a *gzip member* with a header and footer and whose body was compressed with the `gnu-best-rsync` method; the gzip member has a *tarball* as its input, which, in turn, has a *directory reference* as its input. That last element points to the actual contents of the tarball, referred to by a SWHID.

```
(disarchive
  (version 0)
  (gzip-member
    (name "sed-4.8.tar.gz")
    (digest (sha256 "53cf3e1..."))
    (header (mtime 0) (extra-flags 2) (os 3))
    (footer (crc 1582442600) (isize 10516480))
    (compressor gnu-best-rsync)
    (input
      (tarball
        (name "sed-4.8.tar")
        (digest (sha256 "626c2e3..."))
        (default-header
          (chksum (trailer " "))
          (typeflag 0)
          (magic "\x00\x00\x00\x00\x00\x00")
          (version "\x00\x00")
          (data-padding ""))
        (headers
          ("sed-4.8/"
           (mode 493)
           (mtime 1579061438)
           (chksum 3662)
           (typeflag 53))
          ;; many headers omitted
          ("sed-4.8/bootstrap.conf"
           (size 3129)
           (mtime 1578639009)
           (chksum 5071)))
        (padding 1024)
        (input
          (directory-ref
            (version 0)
            (name "sed-4.8")
            (addresses (swhid "swh:1:dir:f36d96f..."))
            (digest (sha256 "994ba02..."))))))))
```

Figure 7: Disarchive disassemble output for `sed-4.8.tar.gz`.

[5]https://ngyro.com/software/disarchive.html, accessed 2024-01-19.

As can be guessed from this example, the difficult part in designing Disarchive was to find out which details about tar and compression formats needed to be kept so that Disarchive could faithfully represent the variety of tarballs actually used without loss of information while keeping the output reasonably compact. The format shown here has proved to meet these requirements for many thousands of tarballs, as we will see in Section 4. The storage costs for those descriptions are modest. As an example, for `sed-4.8.tar.gz`, which contains 987 files, the description takes 140 KiB uncompressed and only 14 KiB when compressed with gzip at its maximum level.

Using Disarchive, we have built and are continuously updating a database[6] of these tarball descriptions. If a source tarball is no longer available from its original location, Guix can automatically recreate it with Disarchive by downloading its contents from SWH and its description from the database—see the second blue arrow on Figure 2.

Maintaining a separate database like this is at odds with our stated goal of full coverage by the SWH archive. As such, work is ongoing to integrate Disarchive and its database into SWH properly, so that these descriptions can be archived there for the long term. Doing so will not be particularly costly in terms of archival storage: Disarchive descriptions are really small in comparison to other source code artifacts already archived by SWH. In conjunction with ExtIDs, this will enable the SWH archive to produce tarballs that are byte-identical to tarballs observed in the wild—e.g., as a new Vault download format.

3.4 Limitations and Mitigation

Our implementation suffers from several limitations, as we have seen before. Source code tarballs are the most challenging part. As of this writing, some archive formats are not supported by Disarchive, including lzip compression, Zip files, and some unusual forms of gzip compression. As we will see in Section 4, these amount to less than 2.6% of the package source code though.

The Subversion version control system poses unique challenges: unlike other VCSes, it allows users to retrieve individual sub-directories within a repository. Luckily, SWH now computes `nar-sha256` ExtIDs for them, allowing Guix to recover them. However, TEX Live packages in Guix work by *combining* several directories checked out from Subversion—typically a source and a documentation directory—and the nar hash in package definitions is computed over that combination. This defeats content-addressed lookups because those combinations do not exist as such in the SWH archive. As of this writing, the Guix and SWH team are discussing a solution whereby SWH would also store the nar hash for these combinations.

Another limitation has to do with the way Guix uses the SWH Vault: the Vault needs to "cook" source code archives before Guix can download them, and that process can take from minutes to days depending on the size of the artifact to be built and the load of the service. That we cannot guarantee timely downloads is a significant usability problem. One solution we are considering is to have SWH "pre-cook" any source referenced by Guix packages. The storage cost of such a policy change is currently being assessed before implementation and deployment can proceed.

[6] https://disarchive.guix.gnu.org, accessed 2024-01-19.

Table 1: "Link rot" empirical evaluation of all package sources over five years.

	May 2019 v1.0.0	Apr. 2020 v1.1.0	Nov. 2020 v1.2.0	May 2021 v1.3.0	Dec. 2022 v1.4.0
#sources	8794	11659	13609	15520	20184
avail.	91.5%	92.4%	95.0%	95.7%	96.4%
missing	8.5%	7.6%	5.0%	4.3%	3.6%
hash mis.	87	63	69	66	52

4 EVALUATION

In this section we evaluate "link rot"—source code disappearance over time—and whether the upstream source code of Guix packages exists in the SWH archive.

4.1 Source Code "Link Rot"

The first question we asked ourselves is: what is the extent of the problem? How much of the source code referenced by Guix packages has disappeared or has been tampered with? To answer this question, we attempted to download the source code of all the packages found in Guix at the time of past releases, covering 5 years of history from version 1.0.0 (released in May 2019) to today (January 2024). Even though Guix was already 6 years old when 1.0.0 was released, we choose this as the starting point because it represents the oldest point in history supported as a target for `guix time-machine`.

To estimate link rot, we re-downloaded all the source code of all the packages defined in Guix since version 1.0.0 from their upstream location, turning off the fallback mechanisms described in Section 3. Table 1 shows the fraction of package source code that could still be downloaded from its initial location. After one year, 96.4% of the package source code is still available unaltered from its upstream location; it decreases to 91.5% after five years. Among the sources reported as *missing*, a fraction was actually still available for download but had been tampered with (*hash mismatch*): that represents 1% of the sources after five years, and 0.3% after one year. These findings are consistent with those reported in an earlier study [10].

To put it in perspective, 3.6% corresponds to 726 packages of version 1.4.0 that are already "lost" a year later. How serious is this? Obviously, this largely depends on what packages one is trying to deploy, directly but also *indirectly*. Looking at the number of dependents of the packages whose source is missing (their *rank* in the graph) gives a more accurate picture. As an example, `openjdk-9.181.tar.bz2` is unavailable from its original upstream URL as it appears in Guix 1.4.0; the `openjdk 9.181` package had 184 dependents, so we would effectively be losing *185 packages*, not just one, if this tarball were unrecoverable. Merely looking at the fraction of missing package sources underestimates the real impact.

4.2 Preservation of Guix

The second question we wanted to answer is this: how much of the source code of Guix packages is actually archived in SWH? Note that we are concerned here with *preservation*. Code available in SWH is not necessarily *recoverable* today by Guix due to the limitations

Table 2: SWH archive coverage of collected sources, including coverage of select commits.

Commit	Date	Stored		Missing		Total
ee3ce0d	mag 5, 2019	6313	71.7%	1879	21.3%	8810
3d76112	ago. 25, 2019	6870	73.4%	1812	19.4%	9362
34085ea	dic. 29, 2019	8146	79.1%	1524	14.8%	10 300
fafe234	apr. 19, 2020	9470	79.8%	1453	12.2%	11 863
cb97d07	ago. 9, 2020	11 430	87.5%	613	04.7%	13 070
60a10a1	nov. 29, 2020	12 388	88.7%	507	03.6%	13 963
ba0dc1d	mar. 21, 2021	13 894	89.4%	443	02.9%	15 541
5c3489a	lug. 11, 2021	15 100	90.6%	353	02.1%	16 666
b11badf	nov. 7, 2021	16 247	90.9%	306	01.7%	17 879
31ecd80	feb. 27, 2022	17 470	91.9%	274	01.4%	19 016
77db24f	giu. 19, 2022	17 985	92.1%	275	01.4%	19 536
79358a9	ott. 9, 2022	18 805	92.3%	271	01.3%	20 373
bea2240	gen. 29, 2023	19 367	92.9%	278	01.3%	20 852
7b3f571	mag 21, 2023	20 323	92.8%	326	01.5%	21 903
2eb6df5	set. 10, 2023	24 775	94.0%	378	01.4%	26 370
25bcf4e	gen. 7, 2024	25 473	93.8%	349	01.3%	27 157
All commits		58 530	85.5%	5950	08.7%	68 473

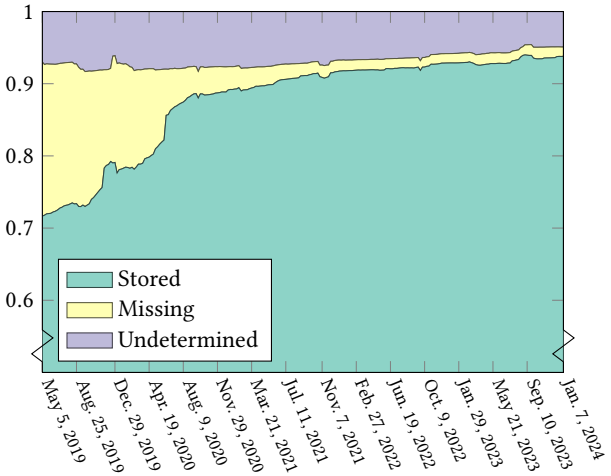

Figure 8: Relative SWH archive coverage by sampled Guix commit (truncated at 50%).

outlined in Section 3.4. Availability is a necessary condition for code to be recoverable, though.

To answer this question, we implemented a set of tools on top of Guix to conduct the analyses described below [16]. We have extracted a list of nearly all of the sources referenced by Guix since version 1.0.0. We sampled the history of the Guix repository, analyzing one commit per week from May 2019 [18] until January 2024 [19].[7] This gives us 243 snapshots of the history of the package collection. At each snapshot the package graph was built and crawled, and each external reference was collected as both a cryptographic hash and a machine-readable description of how to obtain it. There are 68 473 sources in total.

As described in Section 3, the cryptographic hashes from Guix cannot generally be used to locate content in the SWH archive. Therefore, we attempted to download and verify these sources in order to compute SWHIDs for each of them. Tarball sources are processed with Disarchive, and the underlying directory SWHID is extracted from the Disarchive specification. For bare files such as patches, we compute a content SWHID. Git sources are cloned and the directory SWHID of the checkout is taken from Git itself. Subversion sources in Guix can be composed of selected sub-directories of a checkout. To account for this, we take the SWHID of each sub-directory. Other, smaller categories of sources, like Mercurial and CVS, are ignored.

This approach can fail in a number of ways: the sources may be already unavailable; they may be available but fail to verify because they have changed since they were first referenced by Guix; or Disarchive may be unable to process a tarball. Nevertheless, we found SWHIDs for 64 480 (94.2%) of the sources.

The SWH Web API provides the `known`[8] endpoint to query the existence of SWHIDs in bulk. Using this, we found that the SWH archive covered 58 530 (85.5%) of the total sources, or 90.8% of the sources for which we found SWHIDs—see Table 2.

Breaking down the results by commit shows that recent commits have far better coverage than older commits. The earliest commit is missing 21.3% of total sources while the latest commit is missing only 1.3%. This difference is clear in Figure 8, which shows relative SWH coverage by the publish date of individual commits. The main reason for this improvement is that SWH started loading sources as listed by Guix in September 2020 (see Section 3.1). The sources collected for this report will be loaded by SWH using the same system, which will improve the coverage of earlier commits.

The true coverage is likely even better than these numbers show, as many of the sources we were unable to process may be in the SWH archive. This suggests that *complete preservation* is an achievable goal.

4.3 Automatic Source Code Recovery

It is one thing to know that source code is *preserved* by SWH; it is a different thing to be able to automatically *recover* it. This is in part due to the limitations discussed in Section 3.4 but also, more importantly, due to the fact that the recovery mechanism *is itself* improving over time. Consider this command:

```
guix time-machine -q --commit=v1.0.0 -- install emacs
```

It installs (and potentially rebuilds) Emacs and all its dependencies as they were defined in Guix 1.0.0 from 2019. SWH support back then was in its infancy: it was able to retrieve Git checkouts and little more. Thus, the command above ends up using this less-capable code, or even buggy code, which may fail to recover source code, even though today's SWH support in Guix can do so. Two bugs illustrate the problem: the fallback mechanism currently does not fire upon hash mismatches[9], meaning that source that has been tampered with upstream is *not* automatically recovered; the Vault Web API recently started responding with HTTP redirects

[7] Sampling reduces the number of commits to analyze by tens of thousands, easing the computational burden of analysis. However, any external references that appeared and disappeared between samples are missed. The number of missed sources can be estimated by searching the text of the Guix repository (over the given period) for cryptographic hashes. Such a search suggests that only about 1500 (2.1%) sources were missed due to sampling.

[8] https://archive.softwareheritage.org/api/1/known/doc/, accessed 2024-02-08.

[9] https://issues.guix.gnu.org/28659, accessed 2024-02-11.

that Guix code did not follow[10], preventing automatic recovery altogether. These are "normal" bugs that happen in any software development effort and eventually get fixed, but because Guix lets users travel back in its history, those bugs make *automatic* recovery a real challenge.

One measure we took recently to mitigate that is to further *decouple* the download mechanism from the rest of the packaging machinery. For example, downloads of files as well as Git checkouts can now be delegated to the Guix build daemon (*via* the special `builtin:download` and `builtin:git-download` "builders"). The build daemon evolves and incorporates improvements in its fallback code; those improvements will be beneficial even when downloading today's source code 5 years from now. A second mitigation on our road map is a new `guix` command to recover source code referenced by past revisions using present-day techniques.

5 RELATED WORK

More and more frequently, scientists willing to share their computational workflows and to make them reproducible resort to *workflow systems* such as Snakemake and Nextflow. These tools, in turn, often delegate deployment of the computational environment to *container engines* such as Docker, podman, or Singularity/Apptainer [11].

Container engines have been advocated as a tool for reproducible research workflows for almost a decade now [3]. Container engines run images typically built from a large base image on top of which additional software is installed by various means—using the distribution's package manager or additional tools such as Conda and pip. By shipping the container image, one enables others to re-run the exact same code. Unfortunately, the build process of those images is rarely reproducible, and provenance tracking and the source/binary correspondence are lost [21]. As a result, recipients cannot really tell what software they are running, nor experiment with variants of that software.

Maneage [1] is a framework that aims to support reproducible computational workflows and reproducible paper authoring. To do so, it provides a set of Makefiles to download and build software from source. However, its download method has no fallback: one can no longer deploy the workflow, potentially irreversibly, once one of the source code tarballs has become unavailable.

Many free operating system distributions and deployment tools predate Guix of course, and they all have had to ensure to some level that source code of their packages is available. However, few have a stated goal of allowing "time travel"—being able to redeploy and possibly rebuild software at a later point in time [13]. Most major distributions such as Gentoo, Debian, and FreeBSD Ports maintain copies of the source code of their packages, at least for a certain amount of time. Most notably Debian operates the https://snapshot.debian.org service, which contains all version of all Debian packages in both source and binary form (including development versions never shipped in a stable release), but "only" going as far back as 2005 (Debian was created in 1993). Older releases of Debian are available *via* the regular distribution network, but at the coarser granularity of stable releases.

Conversely, many of the package managers that fill a particular niche, such as Brew, Conda, pip, or Spack, do not have source code mirrors in place. Package definitions in Brew and Spack, for example, refer directly to the upstream source code location, with limited mitigation against disappearance or tampering of source code. The pip package manager fetches packages from the Python Package Index (PyPI), where some packages are available as binaries only, in the "wheels" format; remaining packages are available as source and retained for an indefinite amount of time though the project does not commit to any retention policy. Spack implements a source code mirror mechanism[11] but the project does not appear to maintain such a cache. Brew and Conda do not have any source code mirroring or download fallback in place, to our knowledge.

The deployment model pioneered by Nix and that Guix builds upon gives a natural solution to source code caching [7, 8]. The mechanism that allows users to download pre-built binaries as a *substitute* for a local build also applies to source code. Consequently, substitute servers automatically act as a cache for source code too. This solution, however, typically does not offer durability guarantees: those cached sources may end up being deleted from substitute servers after an unspecified amount of time.

The mechanism described in Section 3.1 that allows SWH to pull a list of source code URLs from a JSON file published by the Guix project was initially implemented by a Nix developer. The Nix project publishes a JSON file similar to that of Guix, which thus allows Nix to ensure that its own package source code is also being archived. However, to date, Nix and its package collection do not implement a SWH-based download fallback as described in Section 3.

SwhFS [2] is a file system for Linux that allows users to "mount" subsets of the Software Heritage archive and navigate them as it they were part of the local file system. It is meant to bridge the gap between common software development activities such as searching through development history and source code archiving. As such, it is possible to use SwhFS to retrieve code disappeared from its previously known hosting place, *via* the relevant SWHID identifiers. It is not possible to do so *via* tarball hash though, as the SwhFS interface is based on SWHIDs. The persistence characteristics of SwhFS are inherited from Software Heritage, and hence analogous to those of the approach proposed in this paper.

Disarchive, described in Section 3.3, is, to our knowledge, a new approach to the tarball archiving issue. In 2007, the Debian Project developed the pristine-tar tool[12] to address this problem. To preserve a compressed file, pristine-tar uses a similar guessing approach to Disarchive, and was its inspiration. To preserve a tarball, it generates a fresh tarball in a controlled way using the system implementation of `tar`, and then stores a *binary delta* between the fresh tarball and the original [17]. Later it can generate another fresh tarball and apply that delta to recreate an exact copy of the original upstream tarball. Disarchive instead opts for a more transparent approach with an explicit representation of tarball metadata. This avoids potential stability issues with the output of the `tar` program, and on tests of a few thousand tarballs, results in smaller descriptions on average.

[10] https://issues.guix.gnu.org/69058, accessed 2024-02-12.

[11] https://spack.readthedocs.io/en/latest/mirrors.html, accessed 2024-02-06.

[12] https://manpages.debian.org/unstable/pristine-tar/pristine-tar.1.en.html, accessed 2024-02-06.

6 CONCLUSION

The ability to *preserve* source code in a long-term archive and to automatically *recover it* from the archive when deploying software might sometimes be dismissed as a technicality. Our view is that addressing this technical issue is in fact the very first step required to achieve reproducible software deployment, itself a prerequisite of reproducible research workflows.

By connecting Guix to the Software Heritage archive and by designing Disarchive to bridge the gap between them, we have been able to ensure that 94% of the packages provided by Guix today have their source code archived—85% if we look at all the packages that have been provided by Guix over the past 5 years. Our experience is that automatic source code recovery is even more challenging than preservation; future work includes allowing users to recover old package source code using present-day techniques. To our knowledge, this is the first time a software deployment tool is backed by a stable archive. The relevance of this work goes beyond Guix itself and extends to the wealth of deployment tools and scientific workflow managers that similarly need to be able to reliably download package source code.

Our ambition with Guix is to provide a tool that can redeploy the same software environment years later, whether or not pre-built binaries are available—a tool scientists can rely on to share their work among peers and to inspect and modify it. The work presented here naturally stems from this goal—and so do reproducible builds and the functional deployment model embraced by Guix. It is at odds with the more widespread approach that consists in storing pre-built software images with little or no provenance tracking and little or no support for inspection and experimentation.

Ten years of experience with Guix have allowed us to identify further challenges towards that goal—addressing problems that do not appear when "time-traveling" at the scale of one year, such as the influence of system time and hardware details on software build processes. Our work, going forward, is to tackle those remaining stumbling blocks to achieve reproducible software deployment viable over the course of many years.

7 DATA AVAILABILITY

A full reproducibility package for this work is available for download from Zenodo at https://doi.org/10.5281/zenodo.11256698.

REFERENCES

[1] Mohammad Akhlaghi, Raúl Infante-Sainz, Boudewijn F. Roukema, Mohammadreza Khellat, David Valls-Gabaud, and Roberto Baena-Gallé. 2021. Toward Long-Term and Archivable Reproducibility. *Computing in Science & Engineering* 23, 3 (2021), 82–91. https://doi.org/10.1109/MCSE.2021.3072860

[2] Thibault Allançon, Antoine Pietri, and Stefano Zacchiroli. 2021. The Software Heritage Filesystem (SwhFS): Integrating Source Code Archival with Development. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 45–48. https://doi.org/10.1109/ICSE-COMPANION52605.2021.00032

[3] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *SIGOPS Oper. Syst. Rev.* 49, 1 (jan 2015), 71–79. https://doi.org/10.1145/2723872.2723882

[4] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 2018. Identifiers for Digital Objects: The case of software source code preservation. In *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, MA, USA, September 24-28, 2018*, Nance McGovern and Ann Whiteside (Eds.). https://hdl.handle.net/11353/10.923616

[5] Roberto Di Cosmo and Stefano Zacchiroli. 2017. 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 25-29, 2017*, Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto (Eds.). https://hdl.handle.net/11353/10.931064

[6] Ludovic Courtès. 2013. Functional Package Management with Guix. In *European Lisp Symposium*. Madrid, Spain. https://hal.inria.fr/hal-00824004/en

[7] Ludovic Courtès. 2022. Building a Secure Software Supply Chain with GNU Guix. *The Art, Science, and Engineering of Programming* 7 (06 2022). Issue 1. https://doi.org/10.22152/programming-journal.org/2023/7/1

[8] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*. USENIX, 79–92.

[9] Emily Escamilla, Martin Klein, Talya Cooper, Vicky Rampin, Michele C. Weigle, and Michael L. Nelson. 2022. The Rise of GitHub in Scholarly Publications. In *Linking Theory and Practice of Digital Libraries*, Gianmaria Silvello, Oscar Corcho, Paolo Manghi, Giorgio Maria Di Nunzio, Koraljka Golub, Nicola Ferro, and Antonella Poggi (Eds.). Springer International Publishing, Cham, 187–200.

[10] Emily Escamilla, Martin Klein, Talya Cooper, Vicky Rampin, Michele C. Weigle, and Michael L. Nelson. 2023. Cited But Not Archived: Analyzing the Status of Code References in Scholarly Articles. In *Leveraging Generative Intelligence in Digital Libraries: Towards Human-Machine Collaboration: 25th International Conference on Asia-Pacific Digital Libraries, ICADL 2023, Taipei, Taiwan, December 4–7, 2023, Proceedings, Part II* (Taipei, Taiwan). Springer-Verlag, Berlin, Heidelberg, 194–207. https://doi.org/10.1007/978-981-99-8088-8_17

[11] Samuel Grayson, Darko Marinov, Daniel S. Katz, and Reed Milewicz. 2023. Automatic Reproduction of Workflows in the Snakemake Workflow Catalog and nf-core Registries. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability* (Santa Cruz, CA, USA) *(ACM REP '23)*. Association for Computing Machinery, New York, NY, USA, 74–84. https://doi.org/10.1145/3589806.3600037

[12] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 2 (March 2022), 62–70.

[13] Arnaud Legrand and Pedro Velho. 2023. [Re] Velho and Legrand (2009) - Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. *ReScience C* 6, 1 (Dec. 2023), 20. https://doi.org/10.5281/zenodo.10275726

[14] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 293)*, Carl Pomerance (Ed.). Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[15] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2020. The Software Heritage Graph Dataset: Large-scale Analysis of Public Software Development History. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 1–5. https://doi.org/10.1145/3379597.3387510

[16] Timothy Sample. 2024. *Preservation of Guix*. swh:1:rev:1e719900a301d266044eef6cfa04c7b200a60f0e;origin=https://git.ngyro.com/preservation-of-guix/

[17] The Debian Project. [n. d.]. *pristine-tar*. https://salsa.debian.org/debian/pristine-tar/

[18] The Guix contributors. 2019. *GNU Guix 1.0.0*. swh:1:rev:6298c3ffd9654d3231a6f25390b056483e8f407c;origin=https://git.savannah.gnu.org/git/guix.git

[19] The Guix contributors. 2024. *GNU Guix*. swh:1:rev:25bcf4eda05b501758b11a53823867dc500ac7d1;origin=https://git.savannah.gnu.org/git/guix.git

[20] UNESCO. 2021. UNESCO Recommendation on Open Science. https://www.unesco.org/en/natural-sciences/open-science

[21] Nicolas Vallet, David Michonneau, and Simon Tournier. 2022. Toward practical transparent verifiable and long-term reproducible research using Guix. *Nature Scientific Data* 9 (October 2022). Issue 1. https://doi.org/10.1038/s41597-022-01720-9

[22] Zooko Wilcox-O'Hearn. 2001. Names: Distributed, Secure, Human-Readable: Choose Two. https://web.archive.org/web/20011020191610/http://zooko.com/distnames.html [Online, Archived; accessed 27. Jan. 2024].