

Software Heritage – Why and How to Preserve Software Source Code

April 5, 2019

Contents

1	Software source code at risk	2
1.1	The source code diaspora	2
1.2	The fragility of source code	2
2	Mission and challenges	2
2.1	Mission	2
2.2	Source code harvesting challenges	3
3	Data model	3
3.1	Source code hosting places	3
3.2	Software artifacts	4
3.3	Provenance information	5
3.4	Data structure	5
4	Architecture and data flow	6
4.1	Data Flow Ingestion	6
4.2	Listing	7
4.3	Loading	7
4.4	Scheduling	8
4.5	Archive	9
5	Current status & road-map	10
5.1	Listers	10
5.2	Loaders	10
5.3	Archive coverage	10
5.4	Features	11

1 Software source code at risk

1.1 The source code diaspora

1. **development of software:** with the rise of FOSS:
 - millions of projects are developed on **publicly accessible code hosting platforms** (e.g. *GitHub, GitLab, SourceForge, Bitbucket, etc.*)
 - myriad of “**institutional forges**” scattered across the globe
 - **source code downloads** offered by **developers**
2. **distribution of software:**
 - **principle:**
 1. software tend to *move* among **code hosting places** during its lifetime
 2. the movement is controlled by *current trends* or the *changing needs and habits* of its **developer community**
 - **means of distribution:**
 1. using **code hosting platforms** for **distribution** as well (*most forges allow it*)
 2. using **archives organized by software ecosystems** (e.g. *CPAN, CRAN, etc.*)
 3. keeping **copies of source code released elsewhere:**
 - **software distributions** (e.g. *Debian, Fedora, etc.*)
 - **package management systems** (e.g. *npm, pip, OPAM, etc.*)

1.2 The fragility of source code

1. **problem:** **digital information** (*including source code*) is **fragile** and can be **easily destroyed**: (*human error, material failure, fire, hacking, etc.*)
2. **solution:** **regular backups** in dedicated locations
3. **code hosting platforms as a location for backups:**
 - **pro:** users of code hosting platform don't have to worry about backups since it's **the platform's problem not theirs**
 - **cons:**
 1. most of these platforms are used for **collaboration and record changing** primarily and not for **long-term code preservation**
 2. **digital contents** stored on them can be **altered and/or deleted over time**
 3. the **entire platform can go away** (e.g. *Glitorious and Google Code, with 1.5 million software projects forced to find a new accommodation since*)

2 Mission and challenges

2.1 Mission

1. **Software Heritage:**
 - project unveiled in June 2016
 - initial support by **INRIA**
2. **goal:** *collect, organize, preserve, and make easily accessible all publicly available source code, independently of where and how it is being developed or distributed*

2.2 Source code harvesting challenges

1. challenge 1:

- **identify the code hosting places where source code can be found** (e.g. variety of well-known development platforms to raw archives linked from obscure web pages)
- **solution:** building a **universal catalog** for these **code hosting places**

2. challenge 2:

- **discover and support the many different protocols** used by code hosting platforms to list their contents
- **maintain the modifications made to projects** hosted there
- **solution:** *best practices for preservation “hygiene”* as there’s currently **no uniformity**

3. challenge 3:

- **development histories captured by a wide variety of version control systems** (e.g. *Git, Subversion, Darcs, Bazaar, CVS, etc.*)
- no grand **unifying data model** for **version control systems**
- **solution:** build a grand unifying data model for version control systems and **crawl the development histories captured by them**

3 Data model

1. Data model centered around storing “**software artifacts**” and their corresponding “**provenance information**”, regardless of **data collection**:

- **software artifact:** a **key component** in the **Software Heritage archive**
- **provenance information:** **full information** about where the software was found

3.1 Source code hosting places

1. **code hosting platforms:** stored in a **curative list**, to be **crawled**

2. **types of code hosting platforms:**

- **collaborative development forges:** e.g. *GitHub, Bitbucket, etc.*
- **package manager repositories:** e.g. *CPAN, npm, etc.*
- **FOSS software distributions:** e.g. *Debian, Fedora, FreeBSDn etc.*
- **other types:** e.g. *URLs of personal or institutional project collections not hosted on forges*

3. **listing of code hosting platforms:**

- currently **entirely manual**
- can be **semi-automatic**:
 1. **manual aspect:** entries suggested by **archivists and/or concerned users**
 2. **automatic aspect:** **Web crawlers:** e.g. *detecting the presence of source code to enrich the list*
 3. **review process:** **semi-automated process** that needs to pass **before a hosting place can be listed**

3.2 Software artifacts

1. process:

- **hypothesis:** once **code hosting platforms** are known, **periodic checks** to **archive missing software artifacts**
- **basis:** in general, any **software distribution mechanism** will host **multiple releases** of a **given product** at **any given time**:
 1. **VCS:** *natural behavior*
 2. **software packages:** **current and previous versions of packages** (*i.e. current and previous snapshots of corresponding software*)
- **consequence:** *generalizing VCS and source package formats* -> **recurrent software artifact types** that:
 1. are commonly on **code hosting platforms**
 2. constitute the **basic ingredients** of the **Software Heritage archive**

2. types :

- **file contents (Blobs):**
 1. **definition:** **raw content of source code** (*i.e. sequence of bytes*), **without file names or any other metadata**
 2. **recurrent:**
 - across **different versions of the same software**
 - **different directories of the same project**
 - **different projects**
- **directories:**
 1. **definition:** a **list of named directory entries** pointing to **other artifacts** (*file contents or sub-directories*)
 2. **metadata:** **entries are associated to arbitrary metadata** :
 - **varying with technologies**
 - *e.g. permission bits, modification timestamps, etc.*
- **revisions (Commits):**
 1. **software development:** *a time-indexed series of copies of a single “root” directory that contains the entire project source code*
 2. **software evolution:** *modifying the content of one or more files in that directory and recording their change*
 3. **definition:** each **recorded copy** of a the **root directory** is called a **revision**:
 - **directory with arbitrary metadata**
 - **manually added metadata:** *commit message*
 - **automatically added metadata:** *timestamps, preceding versions, etc.*
- **releases (Tags):** a release is a revision **achieving a project milestone**:
 1. each release points to the **last commit in project history** corresponding to it along with **arbitrary metadata**
 2. **metadata examples:** *release name, release version, release message, cryptographic signatures, etc.*

3.3 Provenance information

1. **process:** the **crawling-related information** are stored as **provenance information** in the **Software Heritage archive**
2. **types:**
 - **origins:**
 1. **definition:** software origins are **fine grained references** to where **source code artifacts** archived by **Software Heritage** were retrieved from
 2. **representation:** **<type, url> pairs:**
 - **type:** the kind of software origin (*e.g. Git, svn, dsc (Debian source packages)*)
 - **url:** **canonical URL** (*e.g. the address at which one can git clone a repository or wget a source tarball*)
 - **projects:** **abstract entities** to precise software origins that:
 1. are **arbitrarily nestable in a versioned project/sub-project hierarchy**
 2. **release several development resources** (*e.g. websites, issue trackers, mailing lists, etc.*)
 3. can be associated to **arbitrary metadata** and software origins (*i.e. where their source code can be found*)
 - **snapshots:**
 1. software origins and the **state of a development project:** any kind of software origin offers **multiple pointers** to the **current state of a development project:**
 - **VCS: branches** (*e.g. master, development or feature branches*)
 - **package distribution: suites** (*i.e. different maturity levels of individual packages (e.g. stable, development, etc.)*)
 2. **definition:** a snapshot of a given software origin at a **given time*** **records all entry points found there and what they were pointing at**
 3. **examples:**
 - **VCS:** a snapshot object that *tracks the master branch commits at any given time*
 - **FOSS distribution:** a snapshot object that **tracks the most recent release of a given package in the stable suite**
 - **visits:**
 1. **role:** **linking** software origins and snapshots together
 2. **definition:** **every time** an origin is **consulted**, a new visit **object is created**, **recording:**
 - **when the visit happened** (*according to Software Heritage clock*)
 - the **full snapshot** of the **state** of the software origin

3.4 Data structure

1. **fact:** **source code is duplicated:**
 - **code hosting diaspora**
 - **vendor:** *copy/paste of entire external FOSS components into other software products*
 - usually a **very small number of files/directories** are **modified by a single commit** -> *large overlap between revisions of the same project*
 - **emergence of DVCS (Distributed VCS):**
 1. **definition:** *natively work by replicating entire repository copies around*

2. **example:** *GitHub pull requests: creation of an additional repository copy at each change done by a new developer*
 - **migration from one VCS to another:** *e.g. Subversion (SVN) -> Git resulting in additional copies, but in different distribution formats, of the very same development histories*
2. **data structure:**
- **principle:**
 1. **deduplication** should be used for **long term preservation and storage efficiency**
 2. **software artifacts** that need to be **deduplicated:** file contents, directories, revisions, releases, snapshots
 - **model:** the **Software Heritage archive** is a **Merkel DAG** (*Direct Acyclic Graph*) (cf. Figure 2 in article)
 - **nodes:**
 1. each **artifact** in the **archive hierarchy**, *from blobs to entire snapshots*, is a **node**
 2. each **node** contains all **metadata** that are **specific to the node itself** (e.g. *commit messages, timestamps, file names*)
 3. each **node** is **identified by an intrinsic identifier:**
 - **computed from the node itself** (i.e. *a cryptographic hash of the node content*)
 - **node content:** *node-specific metadata and the identifiers of child nodes represented in a canonical form*
 - **edges between nodes:**
 1. directory entries **point to other** directories or file contents
 2. revisions **point to** directories and **previous** revisions
 3. releases **point to** revisions
 4. snapshots **point to** revisions and releases
 - **example of a revision node in the Software Heritage Merkel DAG** (cf. Figure 3 in article)
 - **properties inherited from the Merkel data structure:**
 1. **built-in deduplication:**
 - **hash property:** *any software artifact encountered gets added to the archive, only if a corresponding node with a matching intrinsic identifier is not already available in the graph*
 - **consequence:** file contents, directories, project snapshots are **deduplicated** -> **storage costs only once**
 2. **side effect property:** the **entire development history of all source code archived in Software Heritage is available as a unified whole** -> *code reuse across different projects or software origins readily available*

4 Architecture and data flow

Requirement: *an architecture suitable for ingesting source code artifacts into the data model defined for Software Heritage*

4.1 Data Flow Ingestion

1. **definition:** **periodically crawling** a set of “**leads**” (i.e. *curated list of code hosting places*) for content to archive and further leads (like a search engine)

2. **architecture: split into two conceptual phases** (*listing and loading*) -> **facilitate extensibility and collaboration** (cf. Figure 4 in article)

4.2 Listing

1. **definition:**

- **input:** a single code hosting platform (e.g. GitHub, Bitbucket, PyPI, Debian)
- **role:** enumerating all software origins (e.g. individual Git/SVN repositories, individual package names, etc.) found at listing time

2. **implementation: dedicated lister software components for each different type of platform** (e.g. dedicated listers for GitHub, Bitbucket, etc.)

3. **listing disciplines:**

- **full listing:**
 1. collecting the entire list of origins available at a given code hosting platform at once
 2. **pro:** making sure that no origin is being overlooked
 3. **con:** costly if done too frequently on large platforms (e.g. as of 2017, GitHub has more than 55 million Git repositories)
- **incremental listing:**
 1. collecting only the new origins since the last listing
 2. **pro:** quickly update the list of origins available at a code hosting platform
 3. **remark:** to be used after a full listing has been executed

4. **listing styles:**

- **pull style:** the archive periodically checks the code hosting platforms to list origins
- **push style:**
 1. code hosting platforms, properly configured to work with Software Heritage, **contact back the archive at each change in the list of origins**
 2. **pro:** minimize the lag between the appearance of a new software origin and its ingestion in Software Heritage -> optimization on top of the pull style
 3. **con:** risk of losing notifications -> software origins not being considered for archival

4.3 Loading

1. **definition:** actual ingestion in the archive of source code found at known software origins: extraction of software artifacts in software origins and adding them to the archive

2. **implementation: specific to the technology used to distribute source code:**

- **one loader** for each type of VCS (e.g. Git, SVN, Mercurial, etc.)
- **one loader** for each source package format (e.g. Debian source packages, source RPMs, tarballs, etc.)

3. **native deduplication w.r.t. the entire archive:** any artifact (blob, 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

4. deduplication use case:

- **first encounter ever:** the **Git loader** will load all its **software artifacts** (file contents, revisions, etc.) into the **Software Heritage archive**
- **next encounter of an identical repository:** *nothing will be added at all to the archive*
- **next encounter with a slightly different repository** (e.g. a repository containing a dozen additional commits not yet integrated in the official release of Linux): *only the corresponding revision nodes, new file contents and directories pointed by them will be loaded into the archive*

4.4 Scheduling

1. definition:

- **listing and loading happen periodically on a schedule**
- **keeping track of when the next listing/loading actions need to happen:**
 1. for each **code hosting platform** (*for listers*)
 2. for each **software origin** (*for loaders*)
- **remark:**
 1. even when **push-style listing** is performed, we still want to **periodically list pull-style** to stay on the safe side
 2. *scheduling is not always needed for listing*

2. update lag vs. resource consumption:

- **problem:**
 1. number of **hosting platforms to list** is **not enormous**, but the amount of software origins to **load into the archive** can easily reach **hundreds of millions given the size of major code hosting platforms**
 2. **listing/loading too frequently** from that many **code hosting platforms** -> *unwise resource consumption and unwelcome to maintainers of those platforms*
- **solution: adaptive scheduling discipline** -> *balance between update lag and resource consumption*

3. adaptive scheduling:

- **fruitful actions:**
 1. each run of a **periodic action** (*listing/loading*) can be **fruitful**, if it resulted in **new information since the last visit**
 2. **fruitful listing:** *the discovery of new software origins*
 3. **fruitful loading:** *the overall state of the consulted origins differs from the last observed one*
- **process:**
 1. if a **scheduled action** has been **fruitful** -> the consulted site has seen **activity since the last visit** -> *increase the frequency of future visits*
 2. else (*no activity*) -> *decrease the frequency of future visits*
 3. **exponential backoff strategy:** if **activity** is noticed -> *visit frequency is doubled, else it is halved*

4.5 Archive

1. **logical representation: Merkel DAG data structure**
2. **physical storage: different technologies** due to the differences in **size requirements** for storing **different parts of the graph**
3. **blob nodes storage:**
 - **storage space:** occupy the **most space** as they contain the **full content of all archived source code files**
 - **storage technology:**
 1. **key-value object storage** (*key = intrinsic identifier of the Merkel DAG node*)
 2. **pros:**
 - **horizontal scaling: distribution of the object storage over multiple machines**
-> *performance and redundancy*
 - **key-value paradigm is very popular among current storage technologies** -> *easily host copies of the bulk of the archive on premise/public cloud offerings*
4. **rest of the DAG storage: RDBMS (Postgres):**
 - **roughly one table per node type**
 - **key: intrinsic identifier of Merkel DAG node**
 - **pros:**
 1. **horizontal scaling across multiple servers**
 2. **master/slave replication and point-in-time recovery** -> *performance and recovery*
5. **hash object storage:**
 - **problem: hash collisions** if two **different objects hash to the same intrinsic identifier**
-> *risk of storing only one of the nodes*
 - **solution: multiple cryptographic checksums with unicity constraints** on each of them to *detect collisions before adding new software artifact to the archive*
 - **types of checksums:**
 1. **used: SHA1, SHA256, "salted" SHA1 checksums** (*in the style of what Git does*)
 2. **in the process of adding: BLAKE2 checksums**
6. **node mirroring:**
 - **change feed:**
 1. each **type of node** is associated to a **change feed** that **takes note of all changes performed to the set of objects in the archive**
 2. **persistent**
 3. the **archive is append-only** -> under normal circumstances, each **feed will only list additions of new objects as soon as they are ingested into the archive**
 - **pro of using change feeds:** ideal for **mirror operators** -> *after a full mirror step, can cheaply remain up to date w.r.t. the main archive*
7. **retention policy:**
 - **retention policy example:** *each file content must exist in at least 3 copies*
 - **process: a software component of the archive:**
 1. **keeps track of the number of copies** of a given file content and **where each of them is**

2. periodically swipe all known objects for adherence to the policy
 3. when fewer copies than desired exists, additional copies as needed to satisfy the retention policy are asynchronously made by the archiver
8. object corruption automatic healing:
- **example of an object corruption scenario:** *storage media decay*
 - **process: a software component of the archive:**
 1. **periodically checks each copy of all known objects:** *random selection at a suitable frequency*
 2. **recomputes the intrinsic identifier of each copy and compares it with the known one to verify its integrity**
 3. **in case of a mismatch:**
 - **all known copies of the object are checked on-the-fly again**
 - **assuming one pristine copy is found, it will be used to overwrite corrupted copies** -> *automatic healing*

5 Current status & road-map

5.1 Listers

1. **implemented listers:**
 - **GitHub and Bitbucket listers: full and incremental listing** (*put in production*)
 - **consequence: refactoring of common code** -> **lister helper component** to *easily implement listers for other code hosting platforms*
2. **upcoming listers:**
 - *FusionForge, Debian and Debian-based distributions*
 - *bare bone FTP sites distributing tarballs*

5.2 Loaders

1. **implemented loaders:** *Git, SVN, tarballs and Debian source packages*
2. **upcoming loader:** *Mercurial*

5.3 Archive coverage

1. **GitHub archiving:**
 - **full archiving once**
 - **routinely maintain GitHub up-to-date:** *more than 50 million Git repositories*
2. **Debian archiving:**
 - **full archiving once**
 - **all releases of Debian packages** (*2005-2015*)
3. **other archives:**
 - **as of August 2015:** *all current and historical releases of GNU projects*
 - **full copies of all repositories** previously available on **Glitorious and Google Code:** *ongoing ingestion into Software Heritage*

5.4 Features

1. available features:

- **content lookup:**
 1. check whether **specific file** contents have been **archived by Software Heritage**
 2. **uploading** file contents or **directly entering** their **checksum** from **Software Heritage homepage**
- **browsing via API: Web-based API**, allowing **developers** to **navigate** through the **entire Software Archive archive as a graph** :
 1. **look up individual nodes** (revisions, releases, directories, *etc.*)
 2. **access their metadata**
 3. **follow links to other nodes**
 4. **download individual file contents**
 5. **visit information: reporting** when a given software origin has been **visited** and its **status at the time**
 6. *documentation and concrete examples for practical use online*

2. road-map features to be integrated incrementally:

- **web browsing:** equivalent to **API browsing** but for **non-developer Web** users; *i.e. GUI*:
 1. state-of-the-art **interfaces** for **browsing the contents of individual VCS**
 2. *tailored to navigate a much larger archive*
- **provenance information: reverse lookup:** *all the places and timestamps where a given source code artifact has been found*
- **metadata search: searches based on project-level metadata:**
 1. **simple information:** *project name, hosting place, etc.*
 2. **substantial information:** *entity behind the project, license, etc.*
- **content search: searches based on the content of archived files:**
 1. **full-text search**
 2. **raw character sequences**
 3. **syntax trees for a given programming language**