HAL
open science

Referencing Source Code Artifacts: a Separate Concern in Software Citation

Roberto Di Cosmo, Morane Gruenpeter, Stefano Zacchiroli

To cite this version:

Roberto Di Cosmo, Morane Gruenpeter, Stefano Zacchiroli. Referencing Source Code Artifacts: a Separate Concern in Software Citation. Computing in Science and Engineering, inPress, pp.1-9. 10.1109/MCSE.2019.2963148 . hal-02446202

HAL Id: hal-02446202

https://hal.science/hal-02446202v1

Submitted on 22 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Referencing Source Code Artifacts: a Separate Concern in Software Citation

Roberto Di Cosmo, Inria and University Paris Diderot, France
roberto@dicosmo.org
Morane Gruenpeter, University of L'Aquila and Inria, France
morane@softwareheritage.org
Stefano Zacchiroli, University Paris Diderot and Inria, France
zack@irif.fr

***Abstract*—Among the entities involved in software citation, software source code requires special attention, due to the role it plays in ensuring scientific reproducibility. To reference source code we need identifiers that are not only unique and persistent, but also support *integrity* checking *intrinsically*. Suitable identifiers must guarantee that denoted objects will always stay the same, without relying on external third parties and administrative processes.**

We analyze the role of *identifiers for digital objects* (IDOs), whose properties are different from, and complementary to, those of the various *digital identifiers of objects* (DIOs) that are today popular building blocks of software and data citation toolchains.

We argue that both kinds of identifiers are needed and detail the syntax, semantics, and practical implementation of the persistent identifiers (PIDs) adopted by the Software Heritage project to reference billions of software source code artifacts such as source code files, directories, and commits.

***Index Terms*—software citation, digital preservation, reproducibility, open science, digital object identifier**

I. Introduction

This article builds upon key findings on identifiers for *software source code artifacts* from our previous work on digital archives [1]. We focus on its relevance for scientific reproducibility of experiments that rely on software, where one needs to identify source code at various granularities, retrieve byte-identical copies of source code artifacts described in a paper, and verify their integrity as a prerequisite for attempting to reproduce a given result.

We recall the schema of software source code identifiers that has been developed and deployed in the context of Software Heritage [2], a long term initiative aiming to collect, preserve, and share the entire body of software source code and its development history. We show that these identifiers are well-suited for addressing the need of scientific reproducibility when source code is involved.

The requirements that emerge from this particular setting, where one needs to handle tens of billions of different digital objects, cannot be fully satisfied by state-of-the-art identifier schemas broadly adopted for digital publications. Fortunately, we can leverage recursive data structures based on cryptographic hashes, such as Merkle trees [3], to build identifiers of digital objects that satisfy the overlapping requirements of long-term software source code preservation and scientific reproducibility.

II. Referencing Source Code for Reproducibility

Software and software-based methods are now widely used in research fields other than just computer science and engineering [4]. Recognition of the role that software plays in reproducing scientific results, and of the insufficient availability of its source code is growing [5], [6].

In the journey towards open science, and reproducible scientific results, open and unfettered access is needed to three main kinds of research outputs [7]:

1) **the scientific articles**;
2) **the data** used or produced in the research;
3) **the source code of the software** embodying the experiment logic.

Preserving software source code is as essential as preserving articles and datasets to promote both open science and reproducibility. Unfettered access to an executable version of the software is very valuable, but *source code* must be available too: it embodies the scientific *knowledge* underlying the computational calculation.

A. Software reference versus citation

A nice presentation of the many reasons why the research community needs to take software into account can be found in the *Software Citation Principles* [8].

When mentioning software in publications, though, we need to distinguish the software *project*, which refers to an endeavor to develop and maintain software artifacts, from the *software artifacts* (source code, executable binaries, etc.) that are the byproducts of that endeavor. The project is not a digital object; the resulting artifacts are.

Following [9], we distinguish software artifact *reference* from software project *citation*, that serve different purposes.

The main intention of a *citation* is to *give credit* to the authors of a given software (as a project), whereas the function of a *reference* is to *precisely identify* software artifacts, usually for reuse purposes. The two activities are intertwined, and sometimes citing the project without also referencing artifacts is not satisfactory. But for the specific needs of reproducibility, referencing software artifacts is often sufficient and may be easily done without, e.g., having to track down credit attribution, a vastly more difficult task that is worth a research article on its own [9].

In this article we focus on *references* and do not address *citations* further.

B. References for reuse and reproducibility

For reproducibility, software artifact references need to be very fine-grained, down to specific versions. And they should point to a persistent location, available over the long term. As noticed in [8], the usual ways of referencing software artifacts by just pointing to the project website or the current development repository are largely unsatisfactory, as these locations are ephemeral.

For reproducibility we also need a system of identifiers with specific properties that current systems do not provide. Let's consider ACM's Artifact Review and Badging policy, described at https://www.acm.org/publications/policies/artifact-review-badging. It defines the following properties, in order of increasing desirability:

- **Repeatability:** the ability to re-run an experiment by the *same* team using the same experimental setup—including all involved software artifacts. Results that are not repeatable are rarely suitable for publication.
- **Replicability:** the ability to re-run an experiment by a *different* team, reusing the described experimental setup, software artifacts included.
- **Reproducibility:** the ability to re-run an experiment by a different team, *without relying* on the experimental setup and software artifacts developed by the original team.

Furthermore, the following "badges" can be associated to software artifacts to capture their overall quality w.r.t. reproducibility and reuse:

- **Available:** software artifacts that have been made available via publicly-accessible long-term archives.
- **Functional:** software artifacts that meets the specific needs of an experiment and are documented, consistent, complete, and exercisable [by third parties].
- **Reusable:** functional (as above) software artifacts that are more generally useful than addressing the needs of a specific experiment.

The focus of this system is enabling researchers to reproduce and verify results, and not giving credit to authors. Even for the most bare requirement of *repeatability*, it is necessary to have at hand a precise reference to the software source code used for the experiment, as well as long-term archival of the referenced artifact.

As it is now customary in modern software development, we expect that the reference itself allows *integrity checking* upon artifact retrieval, enabling researchers that are attempting replication to rule out corruption or tampering with the digital objects as a potential cause for non-reproducibility. To this end, we need a system of identifiers that depends on *no middleman*, like central registries, *that could silently change the association between identifiers and referenced objects*.

In this paper we describe an already operational system of identifiers that satisfies all these extra properties: *high granularity, integrity, and no middleman*. We argue that, when used in conjunction with the long-term archival provided by Software Heritage [2], such a system provides a suitable mechanism for referencing source code artifacts in research articles.

TABLE I: Mechanism implementation in common systems of identifiers

Mech. / System	Handle	DOI	Ark	PURL	VDOI
Generation	Yes	Yes	Yes	Yes	Yes
Assignment	Yes	Yes	Yes	Yes	Yes
Verification	N.A.	N.A.	N.A.	N.A.	Yes
Retrieval	Yes	Yes	Yes	Yes	Yes
Reverse Lookup	N.A.	N.A.	N.A.	N.A.	N.A.
Description	Yes	Yes	Yes	N.A.	Yes

III. Identifier systems and their properties

This whole section recalls the key findings on identifier systems that the authors published to the attention of the digital preservation community in [1].

A. Identifier systems

A *system of identifier* is composed of a set of *labels* that can be used as references for objects and a set of *mechanisms* performing some or all of the following operations:

- **Generation:** create a new label
- **Assignment:** associate a label to an object
- **Verification:** given a label and an object, verify that they correspond
- **Retrieval:** given a label, provide a means of getting a copy of the corresponding object
- **Reverse lookup:** given a object, find the label that has been assigned to it, if any
- **Description:** given a label, provide a means of getting metadata describing the corresponding object

While these mechanisms can in principle be implemented by totally independent entities, the most common systems of identifiers conflate all these conceptually distinct mechanisms into a single logical component usually called *resolver*.

Despite the fact that the *verification* mechanism is of paramount importance in all the identification systems used in the digital landscape, we could not find any widely used system of identifiers that provides a reliable technical way of supporting verification, even if proposals in this sense have been around for quite a while [10], see Table I. For the specific needs of scientific reproducibility this is a relevant limitation of existing schemas.

B. General properties

The main properties of identifier systems that are relevant for the research software reference use case are:

- **Uniqueness:** one object should have only one canonical identifier.
- **Non ambiguity:** one identifier must denote only one object.
- **Integrity:** in most cases, and in particular for scientific reproducibility, one expects the object denoted by an identifier not to be silently changed later on. An identifier ensures integrity if a user can verify that the object

retrieved at any point in time is exactly the one that was associated to it at the beginning.

- **Persistence:** an identifier should keep its relevant properties on the long term, potentially even after the object it refers to has gone away. This term is used in the literature to capture different ideas, sometimes it just covers the requirement that an identifier should not disappear, while in other places the concept covers even integrity and non ambiguity.
- **No middleman:** to get the highest grade of resilience to external threats such as data loss or corruption, one should not rely on intermediaries for assigning identifiers in the beginning or using them later on (e.g., for retrieval). The name of this property is borrowed from security and cryptography (see, e.g., [11, Chapter 3].
- **Abstraction:** (opacity) early adopters of the Web started using URLs as persistent identifiers only to face dire consequences when it became evident that they are not persistent. As a consequence, recent identifier schemas, like DOI, Ark, or Handle, pushed the idea of identifiers that do not expose details that are subject to change, like the exact location of a resource. Similar ideas can be found in Cool URIs or PURLs. The intent is similar to that of Abstract Data Types in computer science, hence our preference for the term "abstract" over "opaque".
- **Gratis:** many traditional identifier systems, like ISBN, charge a fee for each identifier; several digital systems of identifiers have similar provisions (e.g., DOI fees [12], but also DNS [13]). In the case of digital resources that need to be created or modified frequently, and especially when their amount is very large, charging a per identifier fee is problematic, because it creates a significant barrier to adoption and engenders costs that can become greater than the fixed cost of the infrastructure needed to maintain them.

C. Discussion

Many systems of digital identifiers strive to provide *uniqueness*, like URNs, ARK and DOI [14], [15], but they all rely on administrative structures to ensure it [16] and none of them provides technical guarantees. This fact leads to confusing issues like conflicting DOIs, an official list of which is maintained at https://www.crossref.org/06members/59conflict.html.

For *non ambiguity*, most common systems of identifiers rely on administrative care, leading to the risk that the same identifier end up denoting different objects over time; this issue is similar to what happens for URLs:

> *there is no general guarantee that a URL which at one time points to a given object continues to do so*
>
> — RFC 1738, Uniform Resource Locators (URL)

and is quite real, which was already pointed out, for example, by Arnab and Hutchison [10].

Despite the fact that the term "persistent identifier" is now used almost everywhere, for most resolver-based systems *persistence* is a property that is not technically guaranteed, as one can see clearly stated for example in [17]:

> *The only operational connection between a handle and the entity it names is maintained within the Handle System. This of course does not guarantee persistence, which is a function of administrative care.*

Two of the three remaining properties, *integrity* and *no middleman*, are largely ignored (and not satisfied) by the most common systems of identifiers:

> *the DOI (or any other similar system) does not have any mechanism to prove that a downloaded version of the document is the same as the document located through the resolution process [10]*

Abstraction seems now a generally appraised property, while the requirement for *gratuity* seems much stronger in the librarian community than in the scientific publishing one.

Finally, let us mention here the issues of versions and granularity. An object may be used to create a new object that is a modification of it, and one may want to keep track of the fact that the second one is derived from the first one. Some systems of identifiers allow to encode this versioning information in the object label. Similarly, an object may be composed of several other objects, and some systems of identifiers may want to encode in the object label the relation of containment.

D. DIOs versus IDOs

The reason why the stated requirements are so difficult to satisfy was already contained in the following key remark by Paskin [18]:

> *The term "Digital Object Identifier" is construed as "digital identifier of an object," rather than "identifier of a digital object": the objects identified by DOI names may be of any form—digital, physical, or abstract—as all these forms may be necessary parts of a content management system. The DOI system is an abstract framework which does not specify a particular context of its application, but is designed with the aim of working over the Internet.*

Indeed, all the systems of identifiers that are commonplace in the scholarly world are designed to provide digital identifiers for *any kind of object*, including people, or organizations, that have no canonical digital representation. These Digital Identifiers of Objects (or *DIOs*) make no assumptions on the nature of the object they represent, and hence they inherit all the epistemic issues of the traditional naming systems: the need of a central authority, the complexity related to handling different manifestations of the same conceptual object (like the PDF and the Postscript version of the same book), and more. This fact also explains why none of the systems of Table I supports reverse lookup.

On the other hand, for both scientific reproducibility and software source code archival at large, it is possible to use a system of identifiers for digital objects (or *IDO*). Such a system can be built assuming that it will only manipulate digital objects, which means giving up the ability to attach identifier to any kind of objects (like persons, ideas or institutions), but

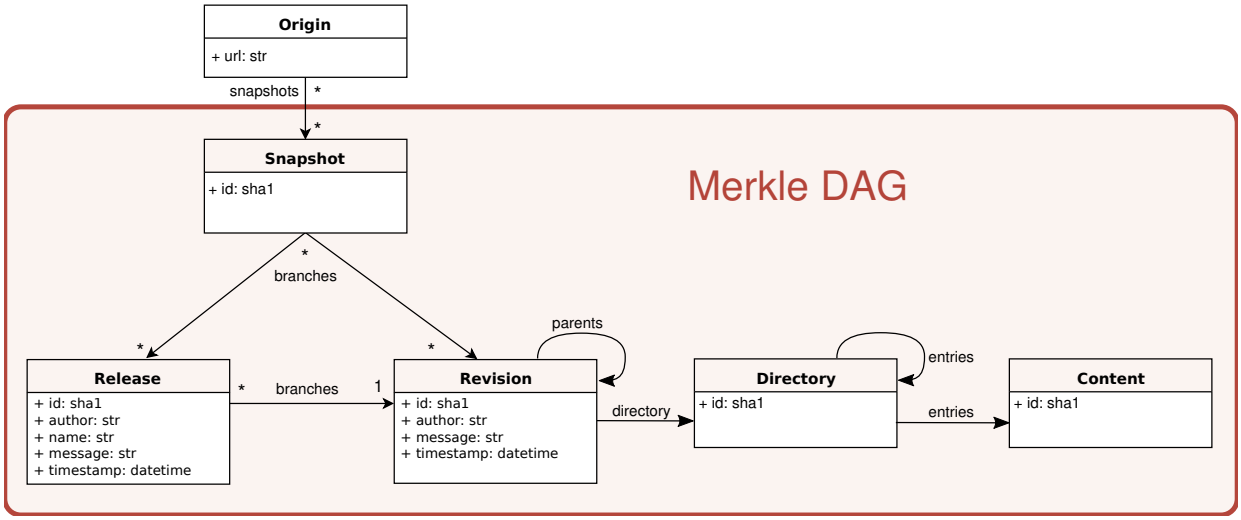


Fig. 1: Topology of the Software Heritage Merkle DAG, which captures software source code together with its full development history.

in exchange all the properties that are difficult or impossible to satisfy in traditional systems of identifiers become feasible.

The key insight is that to get all the nice properties above one must build identifiers *from the object itself*, using for example a hashing function, and this is why we call this kind of identifiers *intrinsic*.

This approach only works well if the digital object has a *canonical representation*, on which the hashing function can be applied: it will not be very useful for identifying documents like research articles, that can be represented in various formats, but is perfectly suited for software artifacts.

IV. Data Model

In order to develop an identifier system for billions of source code artifacts archived for the long-term in Software Heritage [2] we use a data model based on Merkle DAG (Direct Acyclic Graph) [3]. Nodes and edges are connected using hashing functions and represent the history of software development as captured by modern version control systems. We recall the key concepts, and refer the reader to [19] for full details.

The topology of the Merkle DAG is shown in Figure 1. Nodes represent: individual source code files ("content" in the figure), source code directories, commits ("revisions"), software releases, and entire snapshots of the state of a given software development project. Origins are URLs and, strictly speaking, not part of the Merkle DAG itself; but allows to identify where source code artifacts have been encountered.

Edges between the various nodes serve different purposes depending on the type of source/destination nodes. Edges between directories and contents simply form on-disk source code structures. Edges between revisions (and from each revision to the source code root directory at the time) represent the evolution of software development over time and support both code "forks" and "merges". Releases just annotate interesting commits at a given point in time.

Fig. 2: Intrinsic identifiers in the Merkle DAG: example of how identifiers for directory nodes are computed.

In addition, various attributes (not shown in picture) are attached to nodes and edges, depending to their types, e.g., revisions have metadata about who did the commit and when it happened, directories attach local path names to successors, and releases carry human-targeted labels such as "1.0".

As a consequence of the graph being a *Merkle* DAG, the identifier of each node is uniquely identified by a cryptographic checksum (SHA1 in this case). For instance, identifiers for directory nodes are computed as shown in Figure 2. First, a textual serialization of the node (known as "manifest") is produced; in the manifest outbound edges are represented by the identifiers of the target nodes and metadata such as path names are included. Then, a SHA1 checksum of the manifest is computed, returning the desired identifier.

As a result, deduplication is built-in: if the same file appears in several software projects, it will be represented (and hence stored) only once; its hash will be used as unique identifier to link to its content from multiple directories. This

TABLE II: EBNF grammar of Software Heritage persistent identifiers

```
<identifier> ::= "swh" ":" <schema_version> ":" <object_type> ":" <object_id> ;
<schema_version> ::= "1" ;
<object_type> ::=
    "snp"  (* snapshot *)
  | "rel"  (* release *)
  | "rev"  (* revision *)
  | "dir"  (* directory *)
  | "cnt"  (* content *)
  ;
<object_id> ::= 40 * <hex_digit> ;  (* intrinsic object id, as hex-encoded SHA1 *)
<hex_digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
              | "a" | "b" | "c" | "d" | "e" | "f" ;

<identifier_with_context> ::= <identifier> [<lines_ctxt>] [<origin_ctxt>] ;
<lines_ctxt> ::= ";" "lines" "=" <line_number> ["-" <line_number>] ;
<origin_ctxt> ::= ";" "origin" "=" <url> ;
<line_number> ::= <dec_digit> + ;
<url> ::= (* RFC 3986 compliant URLs *)  ;
```

process happens for directories appearing in multiple commits, commits appearing in multiple projects, up to graph roots (i.e., snapshot nodes).

Since identifiers are *cryptographic* checksums, node identifiers are tamper-proof and can be used to verify the integrity of referenced software artifacts. If, say, a file is changed during transfer or due to media bit rot, the receiver can independently re-compute its identifier upon reception and verify it does not match the identifier used to fetch it, immediately realizing that the object got corrupted or has been wilfully altered.

V. Software Heritage Identifiers

Different experiments might need to reference and archive software source code at different granularities: a single source code file, a tarball, a commit in a version control system (VCS), etc. We now address the goal of how to identify source code artifacts at all those granularities, also satisfying the requirements of Section II. To that end we recall in this section the full details of Software Heritage identifiers from [1].

To each source code artifact in the Software Heritage archive we associate a *persistent identifier* (PID) computed through cryptographic hashes. A PID can point to any node in the graph described in the previous section: contents, directories, revisions, releases, snapshots. Each PID embeds a strong cryptographic checksum computed on the entire set of node properties and successors, forming a Merkle structure where each node is labeled with the identifier and provides a secure and efficient integraty mechanism.

A. Syntax

Syntactically, PIDs are generated by the EBNF grammar given in Table II.

B. Semantics

The `swh` prefix makes explicit that these identifiers are related to Software Heritage, and the colon (`:`) is used as separator between the logical parts of identifiers. The schema version (currently `1`) is the current version of this identifier schema; future editions will use higher version numbers, possibly breaking backward compatibility (but without breaking the resolvability of old identifiers).

A persistent identifier points to a single object, whose type is given by `<object_type>`:

- **snp** identifiers points to snapshots,
- **rel** to releases,
- **rev** to revisions,
- **dir** to directories,
- **cnt** to contents.

The actual referenced object is identified by `<object_id>`, which is a hex-encoded SHA1 cryptographic checksum computed on the content and metadata of the object itself (see https://docs.softwareheritage.org/devel/apidoc/swh.model.html for details).

C. Examples

swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2

points to the content of a file containing the full text of the GPL3 license.

swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505

points to a directory containing the source code of the Darktable photography application as it was at some point on 4 May 2017.

swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d

points to a commit in the development history of Darktable, dated 16 January 2017, that added undo/redo supports for masks.

swh:1:rel:22ece559cc7cc2364edc5e5593d63ae8bd229f9f

points to Darktable release 2.3.0, dated 24 December 2016.

swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453

points to a snapshot of the entire Darktable Git repository taken on 4 May 2017 from GitHub.

D. Contextual information

It is often useful to complement persistent identifiers with contextual information about the object's setting.

One can do so with Software Heritage identifiers, using the semicolon (`;`) in the grammar as separator between PID and contextual information. Each piece of contextual information is specified as a key/value pair, using the equal sign (=) as a separator. The following pieces are supported:

- **Software origin:** a URL where a given object has been observed in the wild.
- **Line numbers:** a line number or range, pointing *within* the given object.

For example, the following identifier

```
swh:1:dir:c6f07c2173a458d098de45d4c459a8f1916d900f;
origin=https://github.com/id-Software/Quake-III-Arena
```

points to the source code root directory of the computer game Quake III Arena with the origin URL where it was found; while

```
swh:1:cnt:41ddb23118f92d7218099a5e7a990cf58f1d07fa;
lines=64-72
```

points to a comment segment with the warning "NOLI SE TANGERE" in a file in the Apollo-11 source code.

E. Resolution

Persistent identifiers are not directly browsable URLs, but they can be resolved in various ways. Any identifier can be given to the Software Heritage Web user interface via the URL pattern https://archive.softwareheritage.org/⟨identifier⟩ to reach the referenced object. Both in-browser and programmatic use via a dedicated REST API endpoint is available.

The following third-party resolvers also support resolution of Software Heritage persistent identifiers:

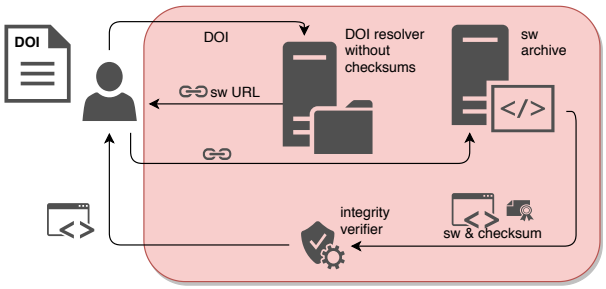
- **Identifiers.org**
- **Name-to-Thing (N2T)**

F. Verification

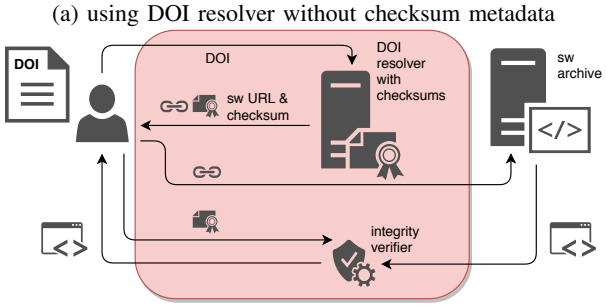
Software Heritage identifiers can be generated and verified independently by anyone using the open source `swh-identify` tool, developed by Software Heritage and distributed via PyPI as `swh.model` (Software Heritage identifier swh:1:rev:6cab1cc81118877e2105c32b08653509475f3eaa; origin=https://pypi.org/project/swh.model/).

VI. Validation

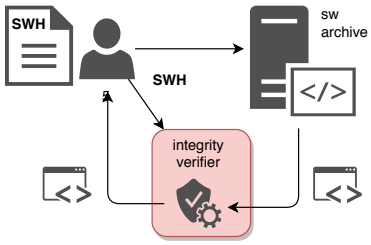
We recall in this section the findings of the self-assessment exercise against the properties discussed in Section III that was performed in [1, Section 6]: we refer the interested reader to it for more details on hash collisions.



(a) using DOI resolver without checksum metadata



(b) using DOI resolver with checksum metadata



(c) using Software Heritage identifiers

Fig. 3: Trusted third parties (shown as rounded red boxes) for software artifact retrieval and verification in three different scenarios.

Uniqueness: identifiers are computed using a cryptographic hash. By construction and due to the Birthday paradox the chances of giving the same identifier to different objects are negligible.

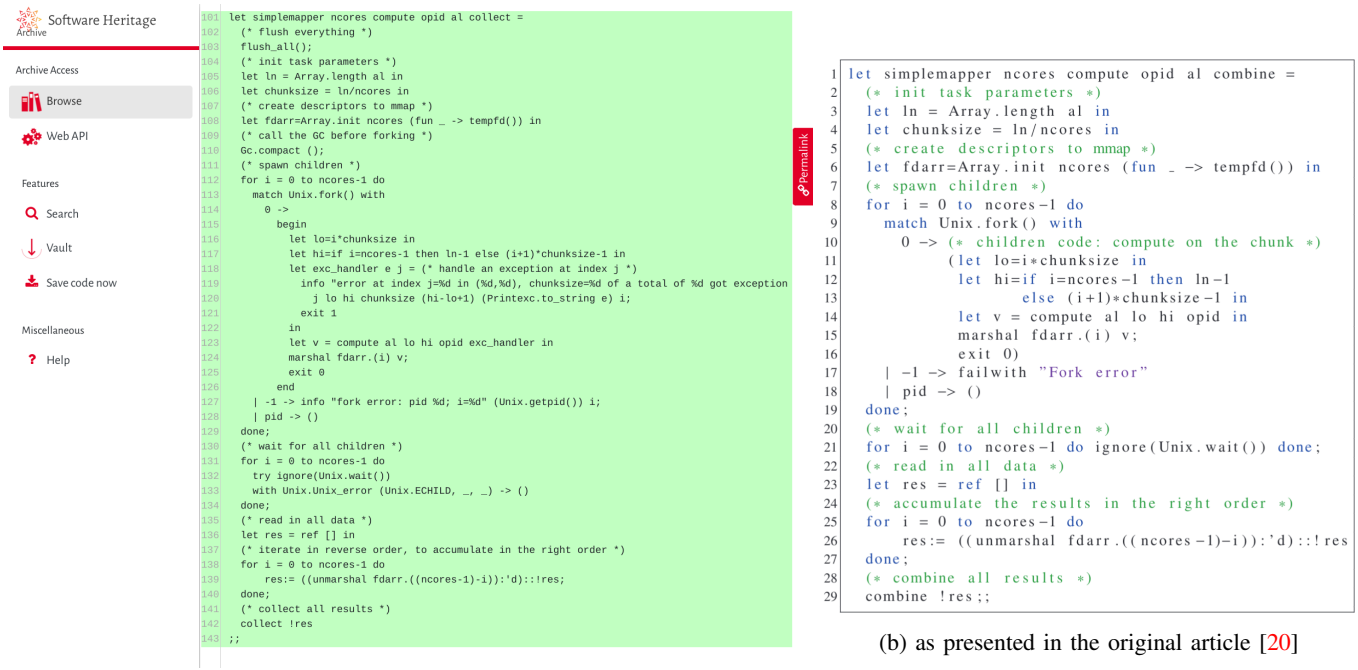
Non ambiguity: at each granularity level, each identifier is designating only one object, without ambiguity.

Persistence: the Software Heritage archive guarantees that nothing will be deleted intentionally and will undertake the task of perpetually maintain old version of the identifier schema, even when new versions of it will be released.

Integrity: using a cryptographic hash as identifier ensures that modifications to the denoted object, however minimal, would yield a different identifier with an extremely high probability. Users can recompute identifiers on retrieved objects and verify they match.

No middleman: the link between an object to its identifier does not depend on the resolution of an online service. These identifiers can be used and verified outside the system that creates and maintains them.

Figure 3 compares the proposed approach with the state-of-the-art in terms of third parties and communication channels that should be trusted to verify the integrity of source code



(a) as archived in Software Heritage

(b) as presented in the original article [20]

Fig. 4: Code fragment from the published article compared to the content in the Software Heritage archive

artifacts. In the common case of DOI resolvers that do not include artifact checksums as part of metadata (Figure 3a), one has to trust the entire toolchain. Storing artifact checksums as part of DOI metadata (Figure 3b) is a significant improvement, in which the artifact archive is no longer trusted: tampering there (or in the communication with it) can be detected; DOI intermediaries still have to be trusted though. The proposed approach (Figure 3c) minimizes the trusted parties and channels: only a reliable checksum verifiers is needed—and several exist already.

Abstraction: the proposed identifier schema does not expose any piece of information that is subject to change over time.

Gratis: the proposed identifiers are intrinsic, meaning they can be independently computed by anyone, using freely available software, incurring no costs for identifier creation or attribution. By construction the obtained identifiers will be the same everywhere, allowing cross-referencing.

Hence, we argue that the Software Heritage identifier schema provides a systems of identifiers for digital objects (IDO) that satisfies the stated requirements for scientific reproducibility and long-term source code preservation.

Note that the optional contextual information of Section V-D are not strictly needed for reproducibility, but it is convenient to store extra information, like the location from where the archived source code has been obtained, to allow tracking future evolution of referenced software artifacts.

VII. Showcase

Software Heritage supports exact referencing of source code artifacts in two unique ways: on the one hand, it provides a *universal archive* that stores the source code, and its full development history; on the other hand, it uses *the same intrinsic identifiers* for all its 10 billion contents, no matter where the source code comes from, or the version control system used to develop it.

We now look at a real world example of how this can significantly improve the workflow of referencing source code in research articles for the purpose of scientific reproducibility.

In 2011, Marco Danelutto and the first author started work on Parmap, a minimalist OCaml library that implements a map-reduce framework for multicore architectures in a concise and elegant way. The software project was developed using git on the Gitorious forge, and described in the paper *A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml* [20]. In order to make the code available to all and facilitate reuse the article, published in June 2012, pointed to the open source release of Parmap linking to https://gitorious.org/parmap.

Alas, Gitorious was closed down in June 2015, and that URL is now broken. Luckily, Software Heritage has archived `parmap` along with all repositories from Gitorious: that repository, with all its development history, can now be recovered. The same can be done for all other legacy articles referencing code on that lost platform. The *universal archive* functionality is essential, as the code can be salvaged without requiring proactive actions by researchers.

The unique identifiers provided by Software Heritage allow to go much further, and enable *precise traceability of code versions and fragments therein*. In Figure 1 of the Parmap article, the authors show the core part of the code implementing the parallel functionality, consisting of 29 lines. In 2012, we had no way to reference these exact lines in the version of the code associated to the published article.

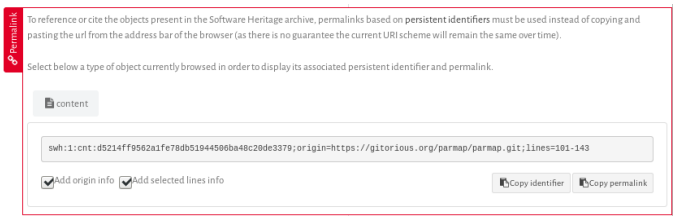


Fig. 5: Obtaining a Software Heritage identifier using the permalink box on the archive Web user interface

Today, using the proposed identifiers, the same code fragment can be precisely identified as:

swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;
origin=https://gitorious.org/parmap/parmap.git;
lines=101-143

Figure 4a shows side-by-side the code as archived by Software Heritage and as shown in the paper, allowing to notice that the code in the article was slightly simplified w.r.t. the actual implementation. Today, a corresponding clickable link could be easily added in the caption of the figure (see page 5 of an updated version of the original article).

Software Heritage identifiers are easy to obtain using the permalink box on the archive's interface, as shown in Figure 5.

The authors should also reference in their article the exact *version* of the software project containing the code, which is also possible using a Software Heritage revision identifier:

swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;
origin=https://gitorious.org/parmap/parmap.git

VIII. Conclusion

Software is an important product of research, and needs to be properly mentioned in research articles, both to give academic credit to the persons involved and to support reproducibility of research. We consider that these two concerns are both important, but separate: while *citations* are essential for giving credit, *references* are sufficient for reproducibility.

In this article, we have focused on the key properties that *references* need to satisfy in the context of scientific reproducibility, some of which traditional digital identifiers of an object (DIOs) do not enjoy, in particular the ability to independently verify object integrity.

Cryptographic hashes widely used in software development can be used as identifiers of digital objects (IDOs) that satisfy all the key requisites, and lie at the core of the Software Heritage identifier schema that is used in production to identify over 10 billion different objects in the project archive.

We look forward to wider adoption of these IDOs in the research community for software artefacts, and all digital objects that have a canonical representation.

References

[1] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, "Identifiers for digital objects: the case of software source code preservation," in *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA*, Sep. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01865790

[2] J.-F. Abramatic, R. Di Cosmo, and S. Zacchiroli, "Building the universal archive of source code," *Commun. ACM*, vol. 61, no. 10, pp. 29–31, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3183558

[3] R. C. Merkle, "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*, ser. Lecture Notes in Computer Science, C. Pomerance, Ed., vol. 293. Springer, 1987, pp. 369–378. [Online]. Available: https://doi.org/10.1007/3-540-48184-2_32

[4] R. Van Noorden, B. Maher, and R. Nuzzo, "The top 100 papers," *Nature*, pp. 550–553, Oct.4 2014. [Online]. Available: http://doi.org/10.1038/514550a

[5] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, "Measuring reproducibility in computer systems research," *Department of Computer Science, University of Arizona, Tech. Rep*, vol. 37, 2014. [Online]. Available: http://reproducibility.cs.arizona.edu/tr.pdf

[6] S. Krishnamurthi and J. Vitek, "The real software crisis: Repeatability as a core value," *Commun. ACM*, vol. 58, no. 3, pp. 34–36, Feb. 2015. [Online]. Available: http://doi.org/10.1145/2658987

[7] R. Vicente-Sáez and C. Martínez-Fuentes, "Open Science now: A systematic literature review for an integrated definition," *Journal of business research*, vol. 88, pp. 428–436, 2018. [Online]. Available: https://doi.org/10.1016/j.jbusres.2017.12.043

[8] A. M. Smith, D. S. Katz, and K. E. Niemeyer, "Software citation principles," *PeerJ Computer Science*, vol. 2:e86, 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.86

[9] P. Alliez, R. Di Cosmo, B. Guedj, A. Girault, M.-S. Hacid, A. Legrand, and N. P. Rougier, "Attributing and referencing (research) software: Best practices and outlook from inria," *Computing in Science Engineering*, pp. 1–14, 2019, preprint available at https://hal.archives-ouvertes.fr/hal-02135891.

[10] A. Arnab and A. Hutchison, "Verifiable digital object identity system," in *Proceedings of the ACM Workshop on Digital Rights Management*, ser. DRM '06. New York, NY, USA: ACM, 2006, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1179509.1179514

[11] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C 2nd edition*. John Wiley & Sons, 2007.

[12] Crossref, "Doi fees," 2017, online; retrieved 09 April 2018. [Online]. Available: https://web.archive.org/web/20180129114723/https://www.crossref.org/fees/

[13] J. Charles, "Web interests tangle over dns proposal," *IEEE Software*, vol. 14, no. 4, pp. 100–105, July 1997. [Online]. Available: https://doi.org/10.1109/MS.1997.595968

[14] I. D. Foundation, "Factsheet: Doi system and internet identifier specifications," 2015, online; retrieved 09 April 2018. [Online]. Available: https://www.doi.org/factsheets/DOIIdentifierSpecs.html

[15] T. C. D. Library, "Archival resource key," 2001. [Online]. Available: http://n2t.net/e/ark_ids.html

[16] W. Y. Arms, "Uniform resource names: Handles, purls, and digital object identifiers," *Commun. ACM*, vol. 44, no. 5, pp. 68–, May 2001. [Online]. Available: http://doi.acm.org/10.1145/374308.375358

[17] S. Sun, L. Lannom, and B. Boesch, "Handle system overview," Internet Requests for Comments, RFC Editor, RFC 3650, November 2003.

[18] N. Paskin, "Digital object identifier (doi) system," *Encyclopedia of library and information sciences*, vol. 3, pp. 1586–1592, 2010.

[19] R. Di Cosmo and S. Zacchiroli, "Software heritage: Why and how to preserve software source code," in *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*, Sep. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01590958/

[20] M. Danelutto and R. Di Cosmo, "A "Minimal Disruption" skeleton experiment: Seamless map & reduce embedding in OCaml," *Procedia CS*, vol. 9, pp. 1837–1846, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2012.04.202