Identifiers for Digital Objects: the Case of Software Source Code Preservation

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

In the very broad scope addressed by digital preservation initiatives, a special place belongs to the scientific and technical artifacts that we need to properly archive to enable scientific reproducibility. For these artifacts we need identifiers that are not only unique and persistent, but also support *integrity* in an *intrinsic* way. They must provide strong guarantees that the object denoted by a given identifier will always be the same, without relying on third parties and external administrative processes.

In this article, we report on our quest for these *identifiers for digital objects* (IDOs), whose properties are different from, and complementary to, those of the various *digital identifiers of objects* (DIOs) that are in widespread use today. We argue that both kinds of identifiers are needed and present the framework for intrinsic persistent identifiers that we have adopted in Software Heritage for preserving billions of software artifacts.

ACM Reference Format:
Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 2018. Identifiers for Digital Objects:, the Case of Software Source Code Preservation. In *Proceedings of 16th International Conference on Digital Preservation (iPRES2018).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In order to manipulate an entity it is essential to be able to refer to it. This is why *nouns* and *proper names* are such important expressions in natural language, and the subtleties of their nature have attracted the attention of brilliant philosophers, from Stuart Mill to Frege, Russel, and Kripke [13].

When building an information system, not necessarily computer based, one faces the same need to name the entities that the system will refer to, and we have become accustomed to call these names *identifiers*. Just like in the case of nouns and proper names, the apparently simple concept of identifier turns out to be rather complex and subtle. Its scope and meaning depend on the properties that one expects of it for each intended use case, and the complexity increases when computers get involved.

This article looks at the requirements and use cases that we have to consider when looking for a class of identifiers that are adapted for Software Heritage [2, 15], a long term initiative whose goal is to collect, preserve, and share a specific form of digital objects: software source code and its development history, which are essential for preserving the scientific and technical knowledge embedded in software [26].

Indeed, source code is a unique form of knowledge, designed to be understood by humans and readily convertible into machine executable instructions. While software in executable form is highly valuable as a tool when the specific environment for which it was compiled is available, source code can be read, studied and modified by humans directly, even if the machine for which it was designed has long disappeared. This is why Software Heritage has taken over the long overdue mission preserving it.

The requirements that emerge for this particular setting, where we need to handle billions of different digital objects, as shown in Figure 1, cannot be fully satisfied by well-known identifier schemas that are in use today in the framework of digital preservation. Fortunately, modern software development has adopted tools and techniques built on top of elegant concepts from computer science dating back to the 1980's [22], and it turns out that we can leverage these very same concepts for building identifiers of digital objects that satisfy all our requirements.

We believe that the analysis and results we report here will be relevant for many forms of digital objects, beyond source code.

The article is structured as follows: we detail the Software Heritage requirements and main use cases in Section 2, we provide a brief survey of digital identifiers in Section 3 and background on the Software Heritage data model in Section 4, then we present Software Heritage identifiers in Section 5, and validate them in Section 6. Section 7 concludes discussing future works and open perspectives.

2 REQUIREMENTS AND USE CASES

Software Heritage is an initiative with the set goal of building a universal archive of software source code, together with its development history as captured by state-of-the-art version control systems [15]. The archive can be accessed through a Web portal where the software source code it contains can be browsed and downloaded. To ensure the long-term preservation of the archive's content a network of mirrors is being created, promoting the use of diverse storage technologies and striving to achieve a broad variety of geographical locations and jurisdictions. The project serves the needs of a variety of stakeholders, ranging from cultural heritage to education, from scientific research to industry.

This implies that the identifiers provided for software artifacts must be able to satisfy a broad range of use cases coming from all these areas. In this section, we highlight a few significant ones and

iPRES2018, September 2018, Boston, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

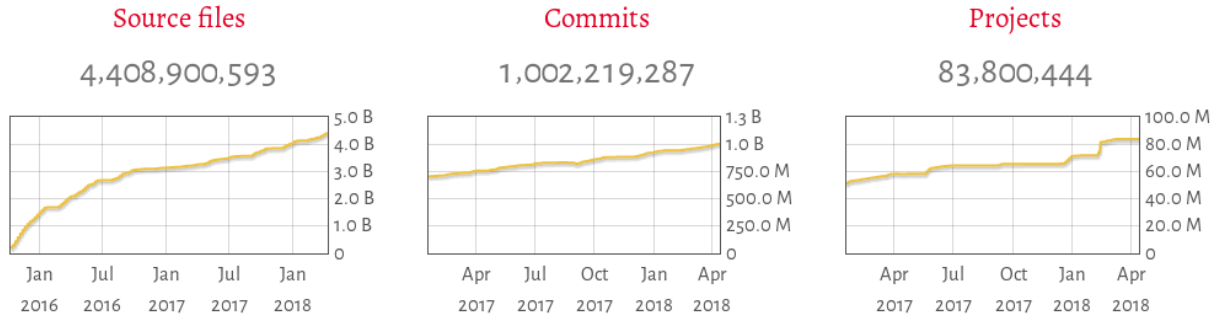

Figure 1: Growth of the Software Heritage archive, as of April 2018

elicit from them a set of requirements for the Software Heritage identifiers.

2.1 Software development

Building a universal, vendor-neutral, persistent reference software archive is an answer to the need for systemic solution to software source code preservation, which is not addressed by existing development and hosting platforms [30]. Software Heritage needs to cater for the needs of *software developers*, that can obtain, for each archived software artifact, an unambiguous identifier that enables collaborative development and facilitate spotting and fixing errors.

Reference, versioning, granularity. Hence, identifiers in Software Heritage must be able to reference a specific version of the source code of a project, at different levels of granularity: a repository, a release, a directory, down to a single file.

Note that software is rarely fully self-contained. In addition to internal resources shipped as part of the software source code, it might also depend on external resources—such as software libraries, runtime environments, data, multimedia, etc.—which are needed to use the software in practice (at compilation and/or runtime). The identifiers we are looking for here cover the state of source code as seen by the software developers of a given project, in the sense that they will change when internal resources are changed, but not when external ones are.

Integrity. Furthermore, in current practice, software developers expect the identifiers they use to provide a means of checking integrity: having an identifier and the (allegedly) corresponding object must be enough to independently verify that the object has not been tampered with. Hence, identifiers in Software Heritage must provide the same ability, in order to be adopted by software developers.

2.2 Software citation

While software is becoming a legitimate product of research [3], the practice of citing software is still inconsistent: some cite the manual of the software, some the software paper, others just the URL of the software repository or homepage. Despite the efforts of various working groups, an agreed upon standard for software citation has not emerged yet.

When citing software, it is important to distinguish between a software *project*, which refers to the software as a concept and its creation environment (and which is not a digital object) and *software artifacts* themselves (source code, binaries, etc.), which usually are digital objects.

In the scholarly ecosystem, we recognize three goals for referencing software:

credit and attribution to be able to identify a *software project* and ensures that the authors have their contributions recognized and rewarded.

reuse and reproducibility to be able to reproduce an experiment, identifying a specific *artifact* of the used software is essential.

integrity checking mechanism to be able to guarantee the accuracy and consistency of the link between a specific *software artifact* and its identifier.

In this paper, we do not address the complex issue of identifying a software project, but we fully address the issue of identifying software artifacts produced by the practice of collaborative development of software source code, which is the main focus of Software Heritage.

The usual ways of designating software artifacts by mentioning the software project or the current development repository are unsatisfactory for several reasons:

- The software itself will evolve and without mentioning the actual version, the reference won't be accurate.
- The authors can delete the current repository, and content can disappear, making reuse impossible.
- The hosting service of a repository can shut down (e.g., Gitorious, Google Code) and the content might move elsewhere.

Therefore, no matter which solution is retained for software citation, an archive of source code that provides a *unique, persistent and intrinsic* identifier for a specific software *artifact* will ensure direct access to the software source code without ambiguity.

2.3 Software evolution tracking

Software traceability is a key component in the Software Heritage archive where we can view and analyze the transformation and evolution of the software source code through its development

history. For this use case it is necessary to identify the *origin* of software source code, i.e., where it has been retrieved from and, hopefully, where newer versions can be found in the future.

In the context of Software Heritage, an origin is a URL representing the location Software Heritage has crawled. A given software project can correspond to multiple origins (e.g., the collaborative repository where it has been developed, its mirrors on other forges, its forks by contributors, etc.); at this point mapping all such origins to a single software project entity cannot be fully automated yet.

Note that information about abstract software projects, e.g. descriptive and usage metadata, are generally not available as part of the software source code. As such, changes in such external metadata are not covered by the identifiers we are looking for.

2.4 Long term digital preservation

The curation of the entire software commons comes with the great responsibility of identifying each and every element while their number is steadily growing.

Therefore, we need identifiers that will guarantee *uniqueness* and *persistence* in the long term that must be:

gratis because there are billions of software artifacts to index, growing fast

intrinsic because one cannot expect third-party resolvers to be around forever

2.5 Summing up

Collecting the Software Heritage requirements from the above use cases, we see that we need identifiers that are *unique, persistent, intrinsic, gratis* and allow support for *versioning* and identifying objects at different levels of *granularity*.

In the next section, we will embark on a quest to find identifiers with these properties.

3 IDENTIFIER SYSTEMS AND THEIR PROPERTIES

When dealing with digital objects, the usual approaches that worked well for physical objects like books do not carry over easily: as it was already remarked almost 20 years ago, *"Identifying objects in digital libraries seems simple but proves to be surprisingly complex"* [5].

We believe that this complexity comes from the great variety of properties that different communities expect from a (digital) identifier [14] and the absence of a clear, shared definition of what identifiers actually are [6].

In this section, we offer a simple conceptual framework for breaking up identifiers into their basic constituent parts, and present a survey of the properties of identifiers that are found in the literature.

3.1 Identifier systems

As a first remark, it is important to notice that while one often speaks of "identifiers", what we are really dealing with is a *system* that is composed of a set of *labels* that can be used as references for objects and the system *mechanisms* performing some or all of the following operations:

generation create a new label

assignment associate a label with 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, when surveying the abundant existing literature, we have found that, with very rare exceptions, the most common systems of identifiers conflate all these conceptually distinct mechanisms into a single logical component[1] usually called *resolver*.

Table 1: 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

Despite the fact that the *verification* mechanism is of paramount importance to 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 [6, 29], see Table 1.

3.2 General properties

We now summarise the *properties* of identifier systems that we have come across during our survey, and that are most relevant for the use cases discussed in Section 2 (see also [20]).

uniqueness one object should have only one canonical identifier

non ambiguity one identifier must denote only one object

persistence an identifier should keep its relevant properties in 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 even covers integrity and non ambiguity

integrity in most cases, one expects the object denoted by an identifier to not 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 with it at the beginning

no middle man to get the highest grade of resilience to external threats, one should not rely on a central authority for assigning identifiers in the beginning or using them later on

[1] This single logical component may be built as a distributed system where all these operations are delegated from one resolver instance to another.

abstraction (opacity) early adopters of the Web started using URLs as persistent identifiers, only to face dire consequences when it became evident that over time they were not persistent. As a consequence, more recent identifier schemas, like DOI, Ark, or Handle, pushed the idea of using identifiers that do not expose details that are subject to change, like the exact location of a resource; similar ideas can be seen in the Cool URIs or in PURLs; the motivation and intent is really the same as that of Abstract Data Types in computer science, hence our preference for the term *abstract* w.r.t. the more commonly used term *opaque*

gratis (free of charge) many traditional systems of identifiers, like the ISBN [1], charge a fee for each identifier, and some digital systems of identifiers have similar provisions [9, 12]; in the case of digital resources that need to be created or modified frequently and smoothly, and in the case when the amount of such resources is very large, charging a per-identifier fee is often seen as non acceptable [20], because it creates a significant barrier to adoption and engenders costs that can become quickly much greater than the fixed cost of the infrastructure needed to maintain them.

3.3 Discussion

Many digital identifier systems strive to provide *uniqueness*, like URNs, ARK and DOI [17, 18], but they all rely on administrative structures to ensure it [5] and none of them provides technical guarantees. This fact leads to confusing issues like conflicting DOIs.[2]

For *non-ambiguity*, most common identifier systems rely on administrative care, leading to the risk that the same identifier ends up denoting different objects over time; this issue is similar to what happens for URLs,[3] and is quite real, as was already pointed out, for example, in [6].

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 [27]:

> *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 middle man*, are largely ignored (and not satisfied) by the most common systems of identifiers,[4] 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 identifier systems offer means to encode this versioning information in the object label. Similarly, an object may be composed of several other objects, and some identifier systems may want to encode in the object label the relation of containment.

3.4 DIOs versus IDOs

In our quest for an identifier system for the largest archive of software source code ever built, we have thoroughly investigated the most popular digital identifier systems, and have been forced to strike them off the list of potential candidates one after the other: they all lacked one or the other of the key properties needed for a long-term universal source code archive that we discuss in Section 2.

We would like to offer an explanation for why our requirements were so difficult to satisfy in these commonplace identifier systems, which is really, in a nutshell, already contained in the following key remark from [24]:

> *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, it seems that all the systems of identifiers we surveyed were really designed with the objective of providing digital identifiers for *any kind* of object, including persons, organizations, concepts, books or poems, that have no canonical digital representation. These Digital Identifiers of Objects (or *DIOs*) make no assumptions about the nature of the object they represent, and hence they inherit all the epistemic issues of the traditional naming systems: the need for 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, to the point that the question of archiving identification information is considered an issue yet to be resolved [28]. This fact also explains why none of the systems of Table 1 supports reverse lookup. On the other hand, for Software Heritage we first of all need a system of identifiers for digital objects (or *IDOs*), as the objects we archive are all born digital, and have a canonical digital representation: a source code file is exactly the sequence of bytes that make it up.

Once we restrict ourselves to IDOs, we need only manipulate digital objects, and as a consequence all the properties that are difficult or impossible to satisfy in traditional systems of identifiers become feasible.

Indeed, relatively recent advances in Computer Science make it possible to design identifiers that a) do not rely on administrative care for uniqueness, non ambiguity and persistence, b) can be created and assigned without a central authority, and c) allow anybody to check independently the integrity of the corresponding digital object. These systems of identifiers leverage ideas and technology coming from cryptography [22] and operating systems [11, 23], and are now incorporated in most modern version control systems used by software developers, like Git [10]. Universally unique identifiers (UUID) are created without a central authority and are

[2] See the official list at https://www.crossref.org/06members/59conflict.html

[3] "there is no general guarantee that a URL... which at one time points to a given object continues to do so" *T. Berners-Lee et al. Uniform Resource Locators. RFC 1738*

[4] "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" [6]

considered unique (or quasi-unique) with a negligible collision probability [23]. And the same kind of intrinsic identifiers has been in use for decades in the field of forensic analysis, where the National Software Reference Library plays a prominent role [21].

This is why, despite the fact that the mission of Software Heritage is to build an archive, we moved away from the DIOs commonly used in archival projects and focused instead on the kind of IDOs nowadays widely used for software development.

4 SOFTWARE HERITAGE DATA MODEL

In order to better appreciate the specificities of the use cases described in Section 2 and how the proposed identifiers are going to address them, in this section we give a brief overview of the Software Heritage data model. Figure 2 provides an overview of the structure of the Merkle direct acyclic graph (DAG) [22] that lies at the heart of the Software Heritage archive [15].

At the bottom there are "file contents", that are uniquely identified by a cryptographic hash [16]: if the same file content is used in several software projects, we store it only once, and use only its hash as a unique identifier to link to this content from the different directories where it appears. Directories are also represented as text files, in a standard way, and can be identified with a cryptographic hash too, so duplicated directory structures get the same hash, and we can, again, store them only once. This process, typical of a Merkle DAG, goes on up to the root nodes of the graph.

Hence, when adding new content to the archive, if an object is already present it will not be stored again, only new links to it will be added. This no-duplication mechanism is efficient for storage and is also a robust mechanism to provide intrinsic and unique identifiers.

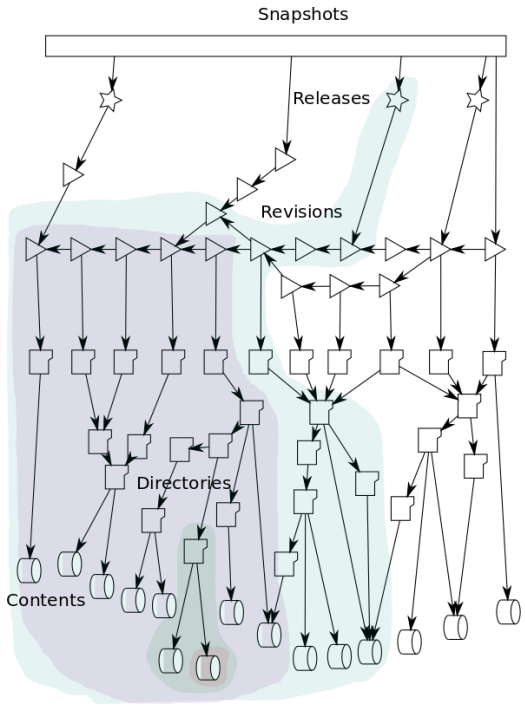

Figure 2: Software Heritage data model: a uniform Merkle DAG containing source code artifacts and their development history

Figure 3 gives a more detailed view of the Merkle DAG used by Software Heritage, zooming in on a real example of software artifacts stored in the archive. Each node in the diagram corresponds to an archived software artifact, produced as part of software development. The following kinds of artifacts are supported:

contents (AKA "blobs") the raw content of a file without its context, note that their names are context-dependent and stored as part of directory entries

directories a list of named directory entries, where each entry can point to content objects ("file entries"), revisions ("revision entries"), or transitively to other directories ("directory entries"). All entries are associated with the local name of the entry (i.e., a relative path without any path separator) and permission metadata and modification timestamps.

revisions (AKA "commits") a point-in-time snapshot in the development history of a project. A revision is made for each modification in the software development workflow, containing the root directory of the software artifact. The revision identifier is calculated with the cryptographic hash of all the metadata provided with the source code directory itself.

releases (AKA "tags") a revision that has been marked as noteworthy by a project with a specific, usually mnemonic, name (for instance, a version number). Each release points to a revision and might include additional descriptive metadata.

snapshots During software development, within a team, the work can be separated into branches to better collaborate. A branch is an autonomous line of development which facilitates work on new features and bug fixes or even just separation between an in-production branch and the development one. When collecting the software artifact, it is important to capture and identify the state of all visible branches during a specific visit, known as a snapshot.

This arrangement allows to store both specific versions of archived software (pointed to by release objects), their full development histories (following the chain of revision objects), and development states at specific points in time (pointed by snapshot objects) in a uniform data model.

It is important to remark that the structure of the Merkle tree where all the objects are injected is defined in a canonical way, and documented in the Software Heritage ingestion process. Hence, even if future version control system come of age, or other software development tools emerge, a given software project will always be represented in the same way inside the archive.

In addition to the content of the Merkle DAG, Software Heritage stores provenance information, as depicted in the top part of Figure 3. Each time a place that distributes software is visited, its full state is captured in a snapshot object (possibly reusing a previous object if the same state has been observed in the past) and a 3-way mapping between the place (usually as its URL), the time of the visit, and the snapshot object is added to an append-only record of crawling activities.

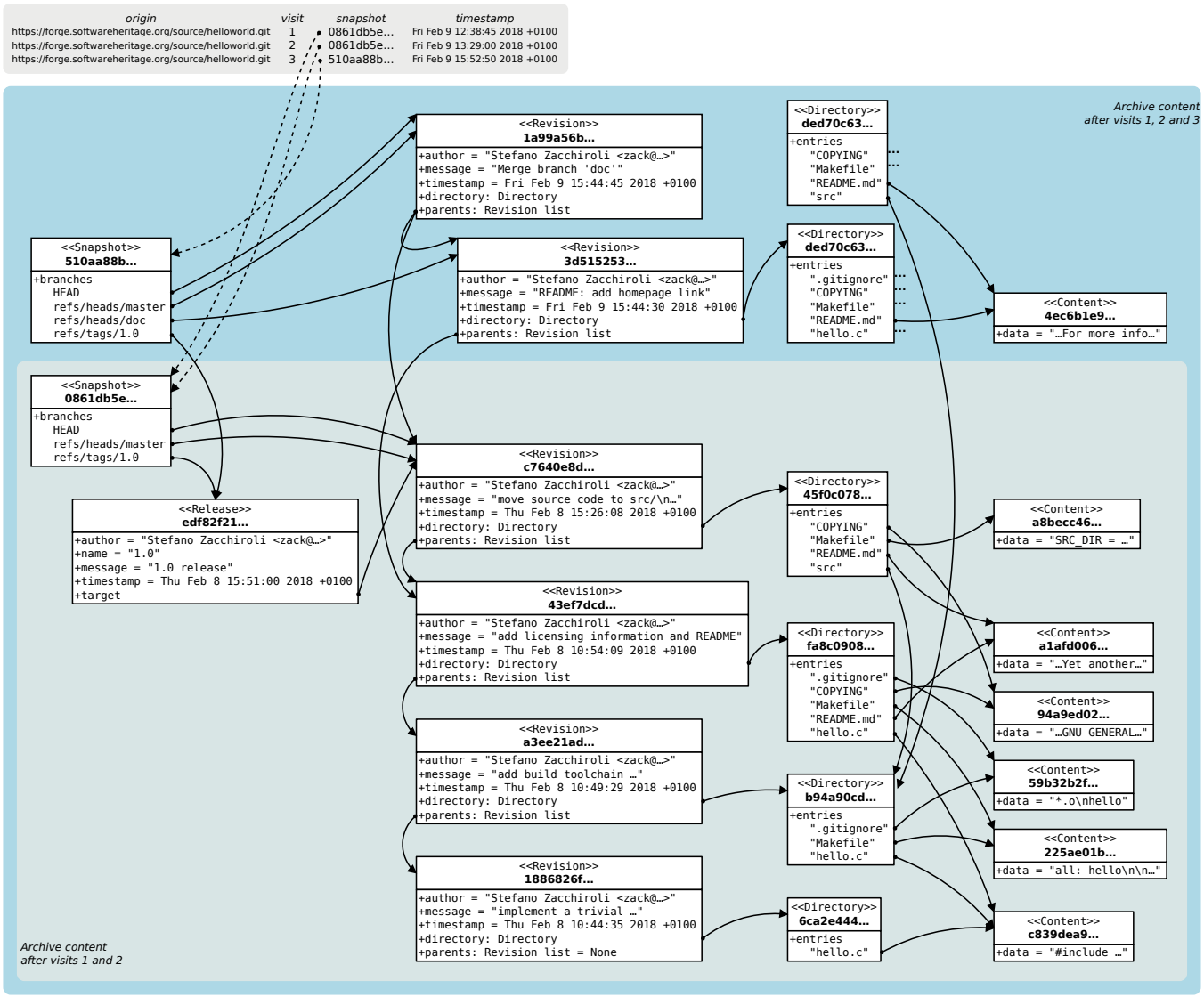

Figure 3: Software Heritage data model: a more detailed view

5 SOFTWARE HERITAGE IDENTIFIERS

As a software source code universal archive, Software Heritage needs to store and identify a variety of software artifacts that together compose a software product in source code form. One of the particular properties of source code is that it is massively duplicated as it was observed in [15]. With the Merkle DAG structure discussed in Section 4, each node is an object that can be identified, while deduplication is built-in. If an object is already represented in the archive it will not be recreated, only the new link to it or the new timestamp when it was seen will be kept. This no-duplication mechanism is efficient with storage but it is also a robust mechanism to provide intrinsic and unique identifiers.

To each object present in the Software Heritage archive we associate an *intrinsic identifier* computed through cryptographic hashes. These identifiers are guaranteed to remain stable over time, and we use them to provide the *persistent identifiers* for the content of the archive. Their syntax, meaning, and usage is described below. Note that they are identifiers and not URLs, even though a URL-based resolver for Software Heritage persistent identifiers is also provided.

A persistent identifier can point to any software artifact (or "object" at each level of granularity) available in the Software Heritage archive, as detailed in Section 4: contents, directories, revisions, releases, snapshots.

For each object we provide an intrinsic, type-specific object identifier that is embedded in its persistent identifier, as described below. Object identifiers are strong cryptographic hashes computed on the entire set of object properties to form a Merkle structure where each node is labeled with the identifier and provides a secure and efficient verification of the objects [22].

Table 2: EBNF grammar of Software Heritage persistent identifiers

```
<identifier> ::= "swh" ":" <scheme_version> ":" <object_type> ":" <object_id> ;
<scheme_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" ;
```

5.1 Syntax

Syntactically, persistent identifiers are generated by the `<identifier>` entry point of the EBNF grammar given in Table 2.

5.2 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 scheme version (currently `1`) is the current version of this identifier scheme; future editions will use higher version numbers, possibly breaking backward compatibility (but without breaking the resolvability of identifiers that conform to previous versions of the scheme).

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

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

The actual object pointed to is identified by the intrinsic identifier `<object_id>`, which is a hex-encoded (using lowercase ASCII characters) SHA1 [16] computed on the content and metadata of the object itself.[5]

5.3 Git compatibility

Intrinsic object identifiers for contents, directories, revisions, and releases are, at present, compatible with the Git way of computing identifiers for its objects. A Software Heritage content identifier will be identical to a Git blob identifier of any file with the same content, a Software Heritage revision identifier will be identical to the corresponding Git commit identifier, etc. This is not the case for snapshot identifiers as Git doesn't have a corresponding object type. Git compatibility is incidental and is not guaranteed to be maintained in future versions of this scheme (or Git), but is a convenient feature for developers, for the time being.

[5]See https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html for more details.

5.4 Examples

The identifiers below are all interesting examples of what the Software Heritage identifiers look like.

They are resolved by the Software Heritage browsing pages available at:
`https://archive.softwareheritage.org/<identifier>`

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.

5.5 Contextual information

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

Table 3: EBNF grammar of complementary contextual information

```
<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 *)  ;
```

to extend the identifier with the optional elements below using the dedicated syntax presented in Table 3:

- the software origin where an object has been found/observed
- the line number(s) of interest, usually within a content object

The semi-colon (;) is used as a separator between the object identifier and other contextual information. Each piece of contextual information is specified as a key/value pair, using the equal sign (=) as a separator. The extended contextual elements should be added in the following manner:

software origin a URL where a given object has been found or observed in the wild and used by Software Heritage to ingest the object into the archive.

line numbers a single line number or a line range, two numbers separated with the hyphen (-). Note that line numbers are purely indicative and are not meant to be stable, as in some degenerate cases (e.g., text files which mix different types of line terminators) it is impossible to resolve them unambiguously.

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[6] 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.

We plan to extend these optional contextual attributes in the future, as required to cover more use cases.

6 VALIDATION

The use of cryptographic hashes to produce unique identifiers for digital objects is now commonplace in version control systems used by programmers, but the underlying assumptions of this approach deserve to be spelled out.

A hash signature, like the SHA1 we use in Software Heritage, has a fixed length of 160 bits, and of course it can only provide 2^{160} different identifiers. While this number is mind-bogglingly larger than any number of digital artifacts we might want to ever archive, the fact that the identifier is not assigned sequentially from a central authority, but computed from the digital object itself means that there is the possibility of a *collision*: the same hash computation performed on two different digital objects ending up in the same signature. The question is what the probability of such a collision is and what to do in case one actually occurs.

Good cryptographic hashes are almost uniformly distributed, so that a classical mathematical calculation based on the birthday paradox shows that the probability of an *accidental* collision can be approximated by:[7]

$$1 - e^{\frac{-k(k-1)}{2N}}$$

where N is the number of all possible different hash values, and k is the number of different objects hashed. Taking the case of the Software Heritage archive, where we use SHA1 as a hash, we have $N = 2^{160}$, and $k = 10^{10}$ counting all kinds of objects. This gives a probability of an accidental collision of 1 in 10^{28}. Even if the size of the archive were to grow to one thousand times bigger, the odds of a collision would still be 1 in 10^{22}, which is negligibly small.[8]

A different threat are *malicious attacks*, which are now feasible, but at a steep computing cost of over 6000 years of processor time.[9] To counter this problem, in Software Heritage we store multiple hashes internally for each object (with SHA1 being in fact the weakest of them), which allows us to spot both accidental collisions and attacks on a particular hash function.

Hence, the chances of assigning the same identifier to two different objects are negligible, and largely inferior to the chances of human error in maintaining a resolver database.

We can now validate the Software Heritage identifier scheme presented in the previous section by performing a self-assessment exercise, reviewing it against the properties discussed in Section 3.

uniqueness the identifier is computed from the content itself, using a deterministic process, so each object can be given only one identifier.

non-ambiguity giving the same identifier to two different objects is equivalent to finding a collision in a cryptographic hash [16]. As discussed above, the chances of finding a collision are negligible.

persistence the Software Heritage archive guarantees that nothing will be deleted intentionally and it will undertake the task to perpetually maintain each identifier schema version, even when it will be labeled obsolete (in the case of collisions on the SHA1 hashes).

integrity using a cryptographic hash as identifier ensures that modifications to the object, however minimal, would yield

[6] See https://en.wikipedia.org/wiki/Quake_III_Arena

[7] See for example https://en.wikipedia.org/wiki/Universally_unique_identifier#Collisions

[8] It is estimated that the odds of a meteor landing on your house are of 1 in 10^{17}!

[9] See https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html for a discussion.

a different identifier with an extremely high probability. A user can recompute the object's identifier and verify it is indeed the right object.

no middle man the link between an object and 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.

abstraction (opacity) the identifier schema presented in this article does not expose any piece of information that is subject to change over time.

gratis (free of charge) the identifiers calculated and maintained by Software Heritage are free and there are no other costs related to the creation and attribution of the identifiers.

Considering the above assessment, we can conclude that the Software Heritage identifier schema provides an identifier system for digital objects (IDO) that satisfies all the requirements we have identified for our use cases, *except one*.

Indeed, if we use only the IDOs to reference an object from the archive, we have no means, looking only at the identifier, to know which original software development platform it has been obtained from. This would result in the loss of context information necessary to trace future evolutions of the software artifact.

This is why we provide the optional contextual attributes described in Section 5.5 to store, among others, the piece of information describing the particular origin from which the object has been obtained. The Software Heritage resolver currently understands these optional contextual attributes and exploits them to show the identified object in the specified context.

7 CONCLUSION

Starting from the requirements for persistent identifiers for software source code artifacts, we have analysed in depth the properties of systems of identifiers and introduced an essential distinction between digital identifiers of an object (DIOs) and identifiers of digital objects (IDOs). By focusing on IDOs, we have shown that the cryptographic hashes widely used in software development can be used as identifiers of digital objects. With this option, several seemingly impossible properties, like the independence from resolvers and the possibility of independently verifyed object integrity become feasible. We have then described the Software Heritage IDO schema that we now use in production to identify over 7 billion different objects in the project archive and shown that it satisfies all stated requirements. We believe that this approach is appropriate not only for software artifacts but also for other digital objects and ensures long-term guarantees: uniqueness, persistence, and digital object integrity.

Acknowledgements. The authors are grateful to the anonymous reviewers for their constructive comments.

REFERENCES

[1] [n. d.]. What is an ISBN? ([n. d.]). https://www.isbn-international.org/content/what-isbn [Online; accessed February 28th 2018].

[2] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. 2018. Building the Universal Archive of Source Code. *Commun. ACM* 61, 10 (Sept. 2018), 29–31. https://doi.org/10.1145/3183558

[3] Smith AM, Katz DS, and Niemeyer KE. 2016. Software citation principles. *PeerJ Computer Science* 2:e86 (2016). https://doi.org/10.7717/peerj-cs.86

[4] David Anderson. 2015. The digital dark age. *Commun. ACM* 58, 12 (2015), 20–23.

[5] William Y. Arms. 2001. Uniform Resource Names: Handles, PURLs, and Digital Object Identifiers. *Commun. ACM* 44, 5 (May 2001), 68–. https://doi.org/10.1145/374308.375358

[6] Alapan Arnab and Andrew Hutchison. 2006. Verifiable Digital Object Identity System. In *Proceedings of the ACM Workshop on Digital Rights Management (DRM '06)*. ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/1179509.1179514

[7] Michel Castagné. 2013. Consider the Source: The Value of Source Code to Digital Preservation Strategies. *SLIS Student Research Journal* 2, 2 (2013), 5.

[8] Vinton G Cerf. 2011. Avoiding" Bit Rot": Long-Term Preservation of Digital Information [Point of View]. *Proc. IEEE* 99, 6 (2011), 915–916.

[9] J. Charles. 1997. Web interests tangle over DNS proposal. *IEEE Software* 14, 4 (July 1997), 100–105. https://doi.org/10.1109/MS.1997.595968

[10] Git community. 2005. Git version control system. (2005). https://git-scm.com/ retrieved 09 April 2018.

[11] Wikipedia contributors. 2018. Plan 9 from Bell Labs — Wikipedia, The Free Encyclopedia. (2018). https://en.wikipedia.org/w/index.php?title=Plan_9_from_Bell_Labs&oldid=832417303 retrieved 09 April 2018.

[12] Crossref. 2017. DOI Fees. (2017). https://web.archive.org/web/20180129114723/https://www.crossref.org/fees/ Online; retrieved 09 April 2018.

[13] Sam Cumming. 2016. Names. In *The Stanford Encyclopedia of Philosophy* (fall 2016 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[14] J Davidson. 2006. Persistent Identifiers. *DCC Briefing Papers: Introduction to Curation. Edinburgh: Digital Curation Centre. Handle: 1842/3368.* (2006). http://www.dcc.ac.uk/resources/briefing-papers/introduction-curation

[15] 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.*

[16] D Eastlake 3rd and Paul Jones. 2001. *US secure hash algorithm 1 (SHA1).* RFC 3174. RFC Editor. https://doi.org/10.17487/RFC3174

[17] International DOI Foundation. 2015. Factsheet: DOI System and Internet Identifier Specifications. (2015). Online; retrieved 09 April 2018.

[18] The California Digital Library. 2001. Archival Resource Key. (2001). http://n2t.net/e/ark_ids.html

[19] Brian Matthews, Arif Shaon, Juan Bicarregui, and Catherine Jones. 2010. A framework for software preservation. *International Journal of Digital Curation* 5, 1 (2010), 91–105.

[20] Julie A. McMurry, Nick Juty, Niklas Blomberg, Tony Burdett, Tom Conlin, Nathalie Conte, Mélanie Courtot, John Deck, Michel Dumontier, Donal K. Fellows, Alejandra Gonzalez-Beltran, Philipp Gormanns, Jeffrey Grethe, Janna Hastings, Jean-Karim Hériché, Henning Hermjakob, Jon C. Ison, Rafael C. Jimenez, Simon Jupp, John Kunze, Camille Laibe, Nicolas Le Novère, James Malone, Maria Jesus Martin, Johanna R. McEntyre, Chris Morris, Juha Muilu, Wolfgang Müller, Philippe Rocca-Serra, Susanna-Assunta Sansone, Murat Sariyar, Jacky L. Snoep, Stian Soiland-Reyes, Natalie J. Stanford, Neil Swainston, Nicole Washington, Alan R. Williams, Sarala M. Wimalaratne, Lilly M. Winfree, Katherine Wolstencroft, Carole Goble, Christopher J. Mungall, Melissa A. Haendel, and Helen Parkinson. 2017. Identifiers for the 21st century: How to design, provision, and reuse persistent identifiers to maximize utility and impact of life science data. *PLOS Biology* 15, 6 (06 2017), 1–18. https://doi.org/10.1371/journal.pbio.2001414

[21] Steve Mead. 2006. Unique File Identification in the National Software Reference Library. (2006). https://www.nist.gov/sites/default/files/draft-060530.pdf National Institute of Standards and Technology.

[22] 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)*, Carl Pomerance (Ed.), Vol. 293. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[23] R. Salz P. Leach, M. Mealling. 2005. A Universally Unique IDentifier (UUID) URN Namespace. (2005). https://doi.org/10.17487/RFC4122

[24] Norman Paskin. 2010. Digital object identifier (DOI) system. *Encyclopedia of library and information sciences* 3 (2010), 1586–1592.

[25] David S. H. Rosenthal. 2017. The medium-term prospects for long-term storage systems. *Library Hi Tech* 35, 1 (2017), 11–31. https://doi.org/10.1108/LHT-11-2016-0128

[26] Leonard J. Shustek. 2006. What Should We Collect to Preserve the History of Software? *IEEE Annals of the History of Computing* 28, 4 (2006), 110–112. https://doi.org/10.1109/MAHC.2006.78

[27] S. Sun, L. Lannom, and B. Boesch. 2003. *Handle System Overview.* RFC 3650.

[28] Douglas Thain, Peter Ivie, and Haiyan Meng. 2015. Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness? *Proceedings of the International Conference on Digital Preservation (iPRES)* (2015). https://doi.org/doi:10.7274/R0CZ353M

[29] the Contributors to the Decentralized Identifiers (DIDs). 2018. Decentralized Identifiers (DIDs) v0.9. (2018). https://w3c-ccg.github.io/did-spec/ Online; Draft Community Group Report 02 April 2018.

[30] Herbert Van de Sompel and Andrew Treolar. 2014. A perspective on Archiving the Scholarly Web. *Proceedings of the International Conference on Digital Preservation (iPRES)* (2014), 194–198.