Efficient Prior Publication Identification for Open Source Code

Daniele Serafini, Stefano Zacchiroli

Efficient Prior Publication Identification for Open Source Code

Daniele Serafini[1] and Stefano Zacchiroli[2][0000-0002-4576-136X]

[1] University of Turin, Italy
me@danieleserafini.eu
[2] LTCI, Télécom Paris, Institut Polytechnique de Paris, France
stefano.zacchiroli@telecom-paris.fr

Abstract. Free/Open Source Software (FOSS) enables large-scale reuse of preexisting software components. The main drawback is increased complexity in software supply chain management. A common approach to tame such complexity is *automated open source compliance*, which consists in automating the verification of adherence to various open source management best practices about license obligation fulfillment, vulnerability tracking, software composition analysis, and nearby concerns.
We consider the problem of auditing a source code base to determine which of its parts have been published before, which is an important building block of automated open source compliance toolchains. Indeed, if source code allegedly developed in house is recognized as having been previously published elsewhere, alerts should be raised to investigate where it comes from and whether this entails that additional obligations shall be fulfilled before product shipment.
We propose an efficient approach for prior publication identification that relies on a knowledge base of known source code artifacts linked together in a global Merkle direct acyclic graph and a dedicated discovery protocol. We introduce `swh-scanner`, a source code scanner that realizes the proposed approach in practice using as knowledge base Software Heritage, the largest public archive of source code artifacts. We validate experimentally the proposed approach, showing its efficiency in both abstract (number of queries) and concrete terms (wall-clock time), performing benchmarks on 16 845 real-world public code bases of various sizes, from small to very large.

Keywords: open source · software supply chain · prior art · source code scanning · license compliance · open compliance

1 Introduction

Free/Open Source Software (FOSS) guarantees, among other fundamental user freedoms, the ability to build upon existing FOSS components when creating new software applications of any kind. After several decades of constant growth this has led to present-day massive reuse of FOSS components. A recent analysis [34] by a major industry player in the field of mergers and acquisitions

(M&A) software audits reports that 99% of code bases audited in 2019 contained open source software components, with 70% of all audited code being itself open source.

"But there ain't no such thing as a free [software] lunch", as the saying goes. To fully reap the benefits of massive open source software reuse, the integration of FOSS components into enterprise development processes requires proper management of the (open source) software supply chain [16]. In particular, attention should be devoted to component selection and validation [31,21], licence obligation fulfillment [11], and tracking known security vulnerabilities in reused components. Most of these concerns are in fact *not* specific to open source software, but the extent to which FOSS enables software reuse makes most development teams face them first and foremost when dealing with open source software.

As a whole, these concerns are referred to as *Open Source Compliance (OSC)*, which consists in adhering to all obligations and best practices for the proper management of FOSS components. Note that open source *license* compliance (or OSLC [11]) is just a part of OSC, albeit an often-discussed one due to the variety and complexity of software licensing [20,2]. The state-of-the-art industry approach for managing the complexity of OSC—known as *continuous open source compliance* [26]—is to automate as much as possible the verification of adherence to all obligations and best practices for FOSS component management and integrate them into continuous integration (CI) toolchains [23].

Source code scanners play a fundamental role in open source compliance toolchains. They are run on local code bases (during CI builds or otherwise) to identify which code parts are known FOSS components v. in-house unpublished code, determine the applicable licenses for the open source parts [10,13], break down components into where they come from (known as Software Composition Analysis [25], or SCA), produce software bills of materials [32,9] (or SBOMs), and automatically detect license incompatibilities [18] or violations of other best practices (optionally making CI build fails).

In this paper we focus on the first among these problems: *prior publication identification*. Given a local source code base to audit, we aim to efficiently identify which parts of it have been published before, according to a reference knowledge base of previously published code. Slightly more formally: we aim to partition the audited source code artifacts into two non-overlapping sets $known \uplus unknown$. The *known* partition contains code that is, according to the reference knowledge base, known to have been published before (possibly, but not necessarily, under a FOSS license); whereas the *unknown* partition contains code that is supposed to have been written in-house, and hence never published before. Determining prior publication efficiently is of practical importance because it helps continuous OSC toolchains to "fail fast" [15]. Indeed, if one can quickly determine that supposedly in-house code has in fact been published before, that is often reason enough to raise an alert, e.g., by making a CI build fail. That will in turn trigger further investigation, usually by Open Source Program Office [21] (OSPO) staff, to determine where the unknowingly reused code comes

from and whether additional overlooked obligations (legal or policy) concerning it need to be fulfilled before product shipment.

Contributions. With this paper we contribute to improve the state-of-the-art of open source compliance toolchains for large software systems as follows:

- We propose an efficient approach for prior publication identification based on: (1) a source scanner running locally on the code base under audit; (2) a (remote or local) knowledge base of known source code artifacts, indexed as a global Merkle DAG [22]; (3) a discovery protocol between the two, called *layered discovery*, that minimizes the amount of artifact identifiers whose known/unknown status has to be queried from the knowledge base.
- We introduce `swh-scanner`, a novel source code scanner that realizes the proposed approach in practice, establishing its feasibility. As its default knowledge base `swh-scanner` uses Software Heritage [1], the largest public archive of publicly available source code artifacts, having archived and indexed (at the time of writing) 12 billions unique source code files and 2.5 billions unique commits from more than 180 million development projects. Alternative knowledge bases can be used instead of Software Heritage, e.g., to better cope with inner source [33,4] use cases. `swh-scanner` is open source software developed at `https://forge.softwareheritage.org/source/swh-scanner/` and distributed via PyPI under the name `swh.scanner` (see Section 3).
- We validate experimentally the proposed approach, by analyzing 16 845 real-world public code bases of various sizes, from a handful up to 2 million source code files and directories. Benchmark results show that the proposed approach is efficient in terms of how many source code artifacts have to be looked up from the knowledge base w.r.t. its total size (15.4% on average). Benchmarks also show that `swh-scanner` is efficient enough for both interactive use and CI integration. 95% of the tested code bases can be scanned in less than 1 second using Software Heritage as knowledge base (99% in less than 4.9 seconds), with a mean scan time of 0.34 seconds.

In the context of open source supply chain management, *open compliance* [19,8] refers to the goal of pursuing compliance by only using open technology, including open source software, open data information, and open access documentation (including standard specifications). Open compliance helps with reducing lock-in risks towards service providers and helps with establishing trust in the scanning tools when they need to run on sensitive code bases. As a byproduct of the chosen approach, and when Software Heritage is used as knowledge base, `swh-scanner` is the first open-compliance-compliant source code scanner, for the specific purpose of prior publication identification.

2 Approach

The problem we aim to solve can be stated as follows. The scanner takes as input: (1) a local code base to audit, i.e., a source code tree rooted at a "root"

directory on the filesystem that (recursively) contains all relevant source code files, and (2) a knowledge base ("KB" for short) capable of answering queries about whether individual source code files or directories are known to have been previously published or not, based solely on their content—so that, for instance, once a given version of a `hello.c` file has been observed in a given Git repository, it will be considered to be *known* no matter in how many different directories or repositories it appears in the future.

The scanner produces as output a $known \uplus unknown$ partition of the input code base, where each audited source code file belongs to either the *known* or *unknown* set. An input file will belong to the *known* partition if and only if it is reported as "known" by the KB. Note that, the output being a *partition* of the input, it also holds that $known \uplus unknown$ is equal to the full set of scanned files.

A couple of caveats are worth noting. First, the input code base can contain duplicate files, or even duplicate directories. As the known/unknown determination by the knowledge base depends only on their content, all different occurrences of the same file (or directory) in the input source tree will belong to the same output partition. Second, when all files contained in a source directory belong to the same partition (say, *known*) we can say as a shorthand that the directory itself belongs to that partition, but the final partitioning can always be described as the set of all files contained recursively in the root directory.

2.1 Knowledge base

Without any additional information about the structure of scanned source code artifacts, the best one can do to establish the $known \uplus unknown$ partition is to query the KB for *all* individual files. Doing so can incur significant costs for large code bases. For example, version 5.9.1 of the Linux kernel contains 327 441 files and it is going to be just *a* part of mixed FOSS/proprietary code bases for IoT devices that use Linux as embedded operating system. This naive approach of querying the status of all source code files and directories is our baseline of (non) efficient prior publication identification.

The centerpiece of the proposed approach is a Merkle [22] DAG (Direct Acyclic Graph) that links together all source code artifacts known to the KB. In Merkle structures node labels are not chosen, but computed as strong cryptographic identifiers based only on the content of each node and, for non-leaf nodes, on the identifiers of their children. State-of-the-art distributed version control systems (DVCSs) such as Git [12] already rely on Merkle structures, as do P2P filesystems like IPFS [3] and Distributed Ledger Technologies (DLTs).

Merkle structures enjoy properties which are useful for efficient comparison of structured data. In particular it holds that if a *complete* Merkle structure (i.e., one with no outgoing dangling links from any node in it) contains a given node, then it also contains all of its descendants, leaves or otherwise.

A generic data model for representing source code artifacts commonly stored in VCSs has been introduced by Software Heritage (SWH) [7] for long-term archival needs; it is shown in Figure 1. The SWH data model supports individual source code files (or "*blobs*" in SWH jargon), *directories* (i.e., source code trees),

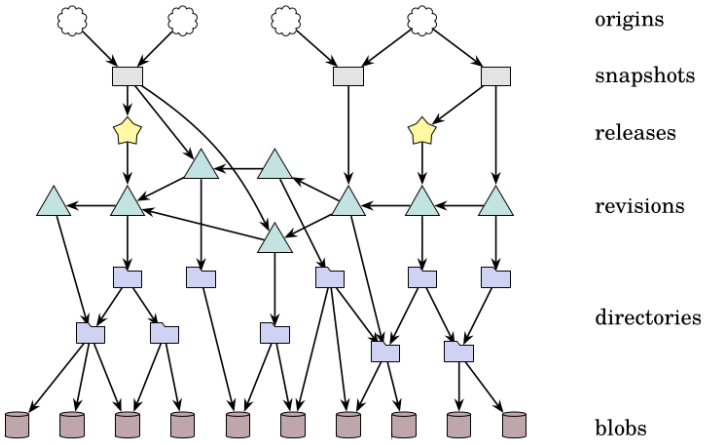

Fig. 1. Data model of the global corpus of public software development: a Merkle DAG (Direct Acyclic Graph) linking together deduplicated source code artifacts.

commits (or "*revisions*"), *releases* (i.e., commits annotated with mnemonic labels like "2.0"), and repository *snapshots* (i.e., the full state of repositories, keeping track of where each branch was pointing at archival time). *Origin* nodes represent software distribution places, such as public Git repositories identifier by URLs, and act as the graph roots pointing *into* the Merkle DAG.

As node identifiers Software Heritage, and by extension `swh-scanner`, relies on *SWHIDs* (SoftWare Heritage IDentifiers) [6], which are standardized intrinsic textual identifiers that embed a cryptographic checksum (a SHA1, in SWHID version 1) and node type information. Examples of a blob and directory SWHID identifier are: `swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2` and `swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505`. SWHIDs can be resolved via the SWH archive Web UI and various other public resolvers.[3]

We require the KB to index source code artifacts using a Merkle DAG, to support at least directory and blob nodes, and to behave consistently w.r.t. Merkle properties when answering known/unknown queries. In particular, if the lookup of the known status of a directory returns *known*, then the knowledge base must also return *known* for all source code files and directories (recursively) contained in it. On the other hand the proposed approach does not *depend* on SWH specifically. We have built `swh-scanner` (see Section 3) using the Software Heritage archive due to its availability and large coverage of public code, but the proposed approach is applicable to any KB respecting the desired Merkle properties. Node types other than files and directories, while not strictly needed for code tree scanning, can also be exploited if available (see Section 6).

[3] For details see `https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html`.

Algorithm 1 *Layered discovery*: efficient prior publication identification of known source code artifacts

```
1: procedure LayeredDiscovery(root, kb)
2:     known, unknown ← ∅, ∅                                  ▷ output partition
3:     Q ← ∅                                                  ▷ queue of nodes to visit
4:     Q.enqueue(root)
5:     while Q ≠ ∅ do                                         ▷ main loop
6:         node ← Q.dequeue()
7:         if kb.knows(node.id) then                          ▷ knowledge base lookup
8:             known ← known ∪ {node}
9:             if node.type = directory then
10:                                  ▷ known directory, mark descendants as known
11:                known ← known ∪ visit(node)
12:            end if
13:        else                                               ▷ unknown node
14:            unknown ← unknown ∪ {node}
15:            if node.type = directory then
16:                                          ▷ unknown directory, dig further
17:                for child ∈ children(node) do
18:                    Q.enqueue(child)
19:                end for
20:            end if
21:        end if
22:    end while
23:    return ⟨known, unknown⟩                                ▷ return partition
24: end procedure
```

2.2 Discovery protocol

Once a knowledge base is available, a source code scanner can efficiently determine the $known \cup unknown$ using the *layered discovery* protocol detailed in Algorithm 1. It takes as first input *root*, the root node of a Merkle tree that corresponds to the code base to audit, indexed in the same way used by the knowledge base to index known code[4] and containing directory and blob nodes. Note that, due to the fact that Merkle structures are built bottom-up, the scanner needs to read the entire local code base to obtain the root node before starting Algorithm 1. The second input is *kb*, a handle to the knowledge base that can be queried to obtain the known/unknown status of a given node identifier.

The following additional notation is used in Algorithm 1: nodes have two attributes, $n.type$ returning the node type (blob or directory) and $n.id$ returning its Merkle identifier; $visit(n)$ is a function returning all the (recursive) descendants of node n, including itself, in an arbitrary order; $children(n)$ returns the direct (non-recursive) descendants of a given node; Q is a FIFO queue, equipped with the usual $Q.enqueue(n)$ and $Q.dequeue()$ methods. The knowledge base is

[4] In particular, Merkle node identifiers shall be computed in the exact same way between the knowledge base and the scanner.

equipped with a single *kb.knows*(*id*) method that returns a boolean indicating whether it knows about a given node or not, based on its Merkle identifier.[5]

Layered discovery proceeds by performing a BFS (breadth-first search) visit of the local source tree, querying the knowledge base and updating the (initially empty) *known*/*unknown* partition as it goes. Once a known node is encountered, knowledge base querying can stop, because all nodes in its subgraph must be known to the knowledge base as well, due to Merkle properties. This allows to prune potentially large subgraphs, minimizing querying, which is expected to be costly, as it will usually happen over the network. However, note that when querying stops, the entire subgraph should still be added to the *known* partition; this step can be done locally without further interacting with the knowledge base.

Runtime complexity. In the worst case scenario, when *kb* does not know any node of the local code base, layered discovery has a time-complexity of $2 \cdot O(V+E) = O(V+E)$ (one bottom-up visit of the source tree to build the Merkle DAG plus one top-down visit to determine the *known*$\cup$ *unknown* split), where V and E are respectively the nodes and edges of the audited code base as a Merkle DAG; the scanner will perform $O(V)$ knowledge base lookups using *kb.knows*().

In the best case scenario, when *kb* knows the entire code base, complexity is $O(V+E)$ with a single call ($O(1)$) to *kb.knows*(*root*).

We will verify experimentally in Section 4 that, using Software Heritage as a knowledge base, we are often close to the best case scenario and also that, independently from the chosen knowledge base, this approach is significantly more efficient than the baseline. We will also see that runtime in practice is good enough for both interactive and CI integration use cases. Before that, let us see how `swh-scanner` implements in practice the proposed approach.

3 Design and implementation

Figure 2 shows the architecture of `swh-scanner` as a C4 container diagram. In this section we describe the role of each component in the architecture, throughout the execution of a typical `swh-scanner` use case.

On the left of Figure 2 a compliance engineer of some organization is interested in establishing the prior art status of a local source code tree. `swh-scanner`, on the top-right of the figure, is free/open source software, released under the GNU GPL license version 3 or above, implemented in Python and distributed via PyPI. It can be installed using the `pip` package manager as follows:

```
$ pip install swh.scanner
```

The compliance engineer can then run `swh-scanner` on the input source tree:

```
$ swh scanner scan SRC_ROOT/
```

[5] Hash collisions are possible, but we assume that the chosen cryptographic hash function is strong enough for the target domain.

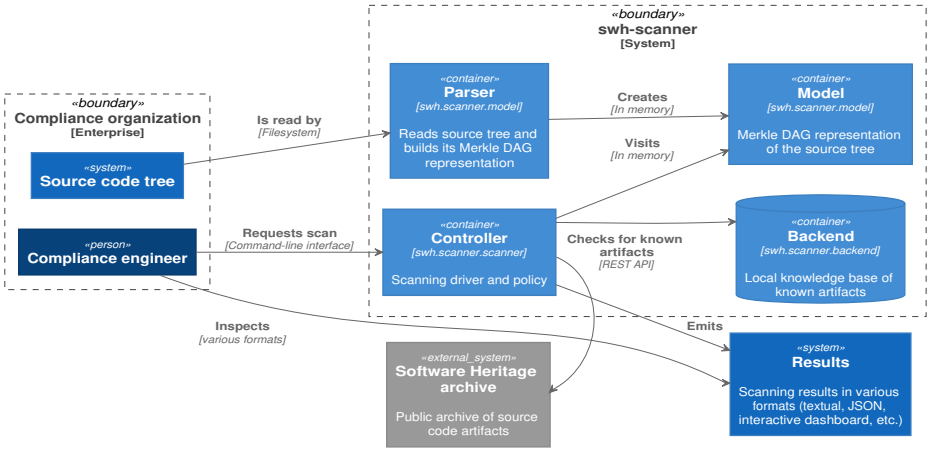

Fig. 2. Architecture of `swh-scanner` as a C4 container diagram.

where `SRC_ROOT` is the root directory of the source code tree for which we want to determine prior art. Upon invocation `swh-scanner` will first build, using the *Parser* component, an in-memory *Model* of the source tree as a Merkle DAG structure compatible with the one described in Section 2.1.

The *Controller* component will then run layered discovery (see Algorithm 1) on the source tree model to determine the $known \cup unknown$ partition. `swh-scanner` also implements alternative, user-selectable discovery protocols (e.g., scan directory nodes first, scan file nodes first, random scanning, etc.) in addition to layered discovery, which is the default and most efficient protocol.

By default `swh-scanner` directs KB queries—corresponding to *kb.knows()* method invocations in Algorithm 1—to the Software Heritage public REST API, which provides a dedicated `/known` endpoint[6] capable of returning the known/unknown status of several SWHIDs at once. Alternatively, a local knowledge base of known source code artifacts, identified by SWHIDs, can be operated locally. In such a scenario `swh-scanner` can be pointed to the non-default, local KB via the `web-api` configuration setting. The companion command `swh scanner db serve -f KB.sqlite` will serve a `swh-scanner`-compatible KB (on a configurable host/port) using the `KB.sqlite` database as source of truth for known SWHIDs. More complex setups are possible by providing a custom implementation of the `/known` endpoint; for instance, in inner source [33] scenarios one might want to check first a local KB and then fallback to SWH.

At the end of scanning the Controller emits *Results* in various user-selectable formats: textual output for manual inspection (*à la* recursive `ls`, with annotations of which parts of the tree are known/unknown), machine-parsable JSON reports, or an interactive HTML dashboard to drill down into scanning results. For example, a sample run of `swh-scanner` on a locally modified version of a

[6] `https://archive.softwareheritage.org/api/1/known/doc/`

Linux kernel source tree, requesting detailed output in JSON format (e.g, for further automated processing), could look like this (excerpt):

```
$ time swh scanner scan -f json /srv/src/linux/kernel
{
  [...]
  "/srv/src/linux/kernel/auditsc.c": {
    "known": true,
    "swhid": "swh:1:cnt:814406a35db163080bbf937524d63690861ff750"
  },
  "/srv/src/linux/kernel/backtracetest.c": {
    "known": true,
    "swhid": "swh:1:cnt:a2a97fa3071b1c7ee6595d61a172f7ccc73ea40b"
  },
  "/srv/src/linux/kernel/bounds.c": {
    "known": true,
    "swhid": "swh:1:cnt:9795d75b09b2323306ad6a058a6350a87a251443"
  },
  "/srv/src/linux/kernel/bpf/": {
    "known": false,
    "swhid": "swh:1:dir:fcd9987804d26274fee1eb6711fac38036ccaee7"
  },
  "/srv/src/linux/kernel/capability.c": {
    "known": true,
    "swhid": "swh:1:cnt:1444f3954d750ba685b9423e94522e0243175f90"
  },
  [...]
}
0,53s user 0,61s system 145$
```

4 Experimental validation

In order to validate the proposed approach for efficient prior publication identification of open source code artifacts we have used `swh-scanner` to analyze 16 845 public code bases.

We initially selected 20 000 public Git repositories from the Software Heritage dataset [27] whose sizes, measured as the number of commits in the repository, uniformly distributed on a log scale. This gave us a varied project sample in terms of project age and activity. We then cloned each of those projects from their public repository URLs. It is important to notice that we explicitly did not retrieve the projects from the SWH archive to allow active projects to have more code and commits w.r.t. the last archived version in SWH, e.g., due to archival lag, which is a common scenario in compliance use cases: the prior art knowledge base used at scan time is not necessarily up-to-date w.r.t. the state of public code in the real world.

Clones were performed using `git clone -depth 1` to retrieve only the most recent commit of each repository, which will constitute one *code base* to be scanned. We successfully cloned 16 845 repositories. The other repositories were no longer available from their original hosting places and have been ignored.

We then run `swh-scanner`[7] on each code base keeping track for each invocation of: n. of *SWHIDs looked up* from the KB (equivalently: n. of *kb.knows(id)*

[7] Specifically, we used the `swh-scanner` version identified by SWHID `swh:1:rev:979d7c803a1478c1e65a6cf8a827c16a746e3aa1`, archived by SWH.

calls in Algorithm 1); *elapsed real time* for scanning; *code base size* as the total number of files and directories it contains. (equivalently: n. of nodes in the Merkle DAG model). Scanned projects have sizes ranging from small code bases of a few files+directories (mean: 3160, median: 132) up to medium and very large code bases (90% percentile: 4198, 95%: 10 283, max: 2.54 millions files+directories).

On each code base we have run `swh-scanner` using different knowledge bases. The scenario denoted as `known-swh` consists of using the live Software Heritage archive as knowledge base at the time of scanning; it best represents a real compliance engineer using `swh-scanner`. Other scenarios, denoted `known-0`,`known-10`,...,`known-100`, are simulations of different knowledge bases knowing from none (0%, or `known-0`) to all of (100%, or `known-100`) the files and directories encountered in *all* tested code bases by 10% increments.

To obtain the simulated knowledge bases we first mined from all code bases the identifiers (as SWHIDs) of all contained files and directories. This set constitutes the `known-100` knowledge base: any node that could ever be queried when scanning any code base will be reported as "known" by this KB. To obtain the `known-90` we proceeded as follows: randomly mark 10% files (which correspond to leaves in the Merkle DAG) as unknown; then visit the Merkle DAG backward from leaves to roots marking all encountered directory nodes as unknown as well. The visit ensures that Merkle properties are respected by the simulated KBs: if a given file is reported as unknown, no directory (recursively) containing it should be reported as known, because if the directory had been encountered in the wild, all its (recursive) content would have been too. Iterating this process by 10% increments we produced all simulated KBs up to `known-10`. (`known-0` is trivial to produce: it is the KB that always answers "unknown".)

Practically, experiments for the `known-swh` KB were run using default scanner settings (which make `swh-scanner` query the live SWH archive), whereas other `known-*` cases by operating local KBs using `swh scanner db serve` (discussed in Section 3) and pointing the scanner to them.

Figure 3 shows experimental results about the number of files and directories looked up from the KB using *kb.knows(id)*, for various KBs. For the sake of readability results are split between two charts, one for small code bases (above) and one for big ones (below). The size line is our baseline, corresponding to a discovery scenario where the scanner has to lookup all artifacts (files + directories = size) from the KB. Indeed, the `known-0` scenario, i.e., a KB that knows no artifact, is identical to the baseline (and hidden below it in the charts).

The real-world scenario of `known-swh`, where we have queried the live SWH archive performs much better than the baseline, looking up only an increasingly marginal fraction of scanner artifacts. On average as little as 15.4% of the input files and directories need to be looked up, with a median of 2.3% and a 75% percentile of 20% nodes of the input code base looked up. Among simulated scenarios, the only one outperforming `known-swh` is `known-100`, for a KB that knows all artifacts and hence always need a single lookup of the root directory node. Other simulated KB scenarios increasingly approach the baseline: the less the KB knows, the closer they get to it, i.e., the worse they perform.

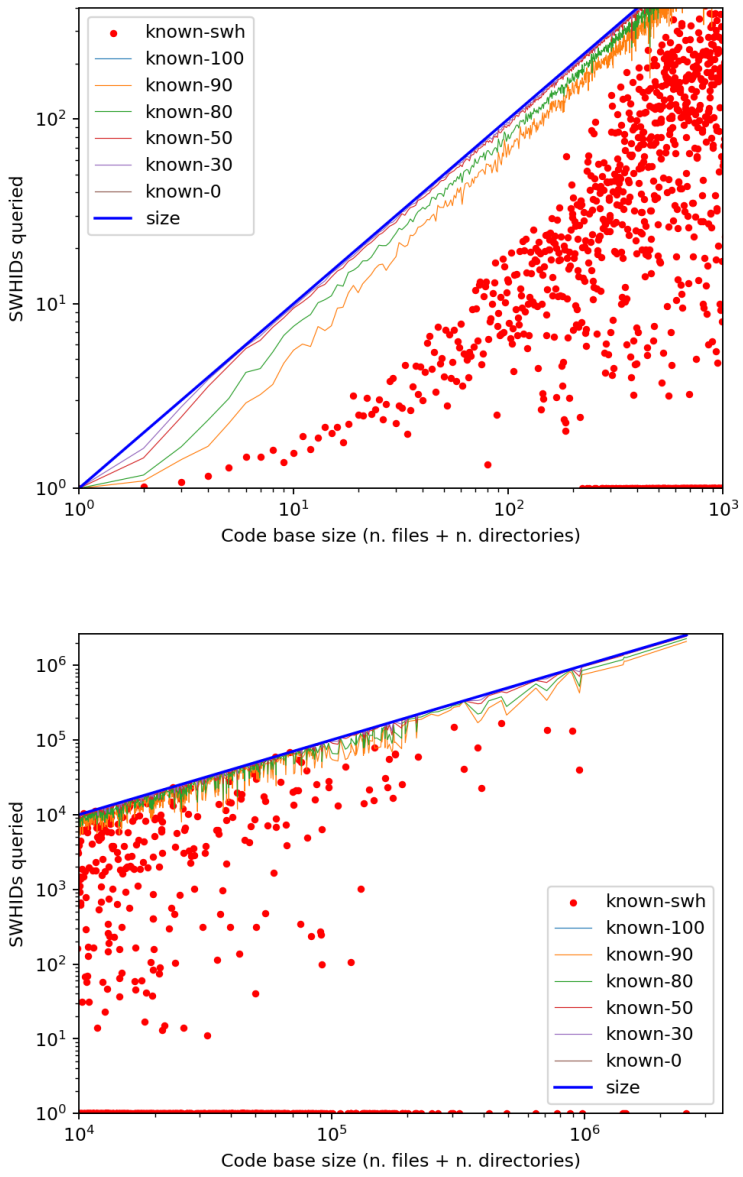

Fig. 3. Amount of knowledge base lookups performed for determining the *known* ⊎ *unknown* partition of public code bases of various sizes (small ones above, large ones below) using various knowledge bases: `known-swh` for the live Software Heritage archive, `known-100` for 100% of the code base files and directories known, down to `known-10` and `known-0` for 10% and 0% artifacts known, respectively.

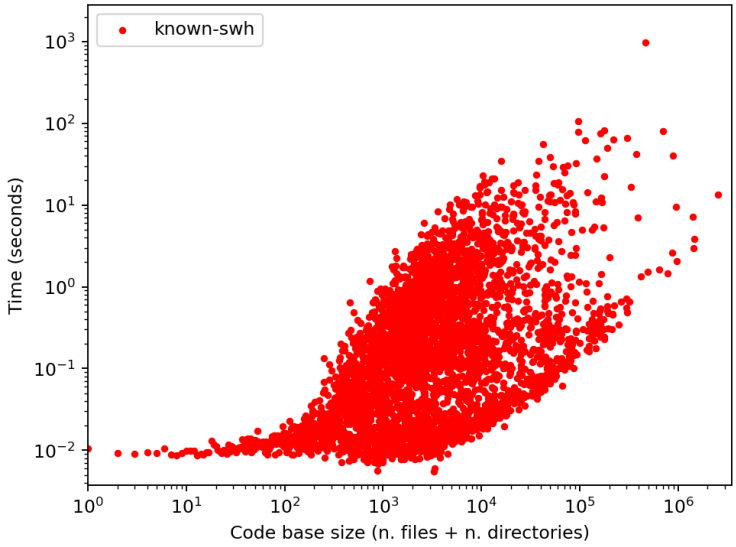

Fig. 4. Elapsed real time (seconds) for determining the *known* ⊎ *unknown* partition of public code bases of various sizes using Software Heritage as remote knowledge base. Mean: 0.34 seconds; 95% percentile < 1 second; 99% < 4.9 seconds.

These results show that, in terms of lookup efficiency, the proposed approach beats the baseline and does so by far when using the SWH archive as KB. But what about the *practical* efficiency and viability of `swh-scanner` as a tool for the task? We answer this question by showing in Figure 4 the elapsed real time for scanning all analyzed code bases.

Timing benchmarks show that `swh-scanner`, when used with SWH as remote knowledge base over the network, is efficient enough for both interactive use, e.g., by a compliance engineer, and integration into CI/CD workflows for continuous open source compliance. 95% of the tested code bases can be scanned in less than 1 second using Software Heritage as knowledge base (99% in less than 4.9 seconds), with a mean scan time of 0.34 seconds. Aside from a single outlier (1 code base out of 16 845 projects, which took ≈ 15 minutes to scan 0.5 million files/directories) even the largest code bases in our sample, up to 2 million files/directories, were scanned in less than 2 minutes.

5 Related Work

We have introduced an efficient approach to detect prior publication of open source code artifacts, implemented it in `swh-scanner`, and verified experimentally its efficiency. To the best of our knowledge this is the only scanner for

FOSS compliance that is open source itself, leverages Merkle properties to improve scanning efficiency and uses Software Heritage as an open data knowledge base to determine prior publication of source code files and directories. While the underlying problem seems to have received little attention in the research literature a number of industrial code scanning tools exist, for the purpose of (semi-)automating the verification of compliance with FOSS license obligations and/or security best practices.

The tooling landscape[8] conducted by the Open Source Tooling Group and the OpenChain [5] curriculum[9] provide a good overview of existing tools to support automated governance of FOSS supply chains, including tools that adhere to the open compliance principle [8] (see Section 1). State-of-the-art license scanners in the field are FOSSology [17], and ScanCode (discussed in [25] together with other FOSS tools for Software Composition Analysis). Zooming out from license detection *per se*, several tools are used in the compliance landscape to manage the workflow of vetting open source component before production use, such as Eclipse SW360[10] as component inventory manager and the OSS Review Toolkit (ORT)[11] that provides a customizable pipeline for continuous compliance [26].

SCANOSS[12] has recently announced an open data knowledge base (OSSKB) to accompany its (also open) scanning tool. Its coverage is comparable in size to Software Heritage, but the indexing technique is different. OSSKB has finer granularity (see Section 6 for a discussion of this point) than `swh-scanner`, relying on winnowing [29] for approximate matches on individual source code files. On the other hand the SCANOSS scanner does not rely on Merkle structuring to prune code base parts that do not need scanning. In-depth quantitative benchmarking of the two approaches constitutes interesting future work.

6 Discussion

Scanning commits and other artifact types. We focused our discussion on the scanning of source code files and directories, because that corresponds to both the state-of-the-art in terms of artifact types and to what `swh-scanner` supports today. But in fact all artifact types supported by the SWH data model (Figure 1)—and in particular commits, releases and snapshots, not discussed in the paper—can be supported via the same approach. Minimal changes would be needed in the discovery protocol, roughly speaking to treat all other non-leaf nodes similarly to how directories are handled. This flexibility is likely to become increasingly relevant in the future, as relevant FOSS projects (e.g., the Linux kernel) are starting to discuss[13] the possibility that their *entire Git development history* might correspond best to the *complete and corresponding source* (CCS)

[8] `https://github.com/Open-Source-Compliance/Sharing-creates-value/`
[9] `https://github.com/OpenChain-Project/curriculum/`
[10] `https://www.eclipse.org/sw360/`
[11] `https://github.com/oss-review-toolkit/ort`
[12] `https://www.scanoss.com/`
[13] `https://lore.kernel.org/linux-spdx/YqILppVZUrD19M6D@ebb.org/`

that should be made available to users for compliance with the terms of the GNU GPL. In such a scenario it will become important for compliance engineer to determine the prior publication of entire Git repositories; the proposed approach will fit well and efficiently such novel use cases.

Scanning granularity. Whereas scanning commits, release, etc. goes in the direction of (conceptually) larger artifacts, one can also increase scanning granularity and scan *within* source code files, e.g., to support prior art detection at level of individual snippets contained in a larger source code file. Various techniques exist to support this in source code scanners, including plagiarism detection (like Winnowing [29], mentioned in Section 5), locality-sensitive hashing (like TLSH [24]), or source code parsing followed by code clone detection [30].

While the proposed implementation, based on `swh-scanner` + SWH as KB, stops at file granularity with a notion of file equality based on cryptographic hashes—and hence will *not* recognize as *known* a source code file where a single byte has been altered w.r.t. a previously published version of it—the proposed approach is granularity-agnostic. The Merkle structure of Figure 1 can be extended to have as leaves file parts, such as code snippets or lines (SLOCs), rather than files. That would cause an increase in the size of the KB, but will not substantially alter the approach efficiency, because large known sub-parts of public code bases will be fully detected as known based on hashes that are high up in the Merkle structure. To the best of our knowledge no attempt of building a Merkle structure of public code at a granularity finer than individual files and at the scale of SWH (= tens of billions source code files) has ever been attempted.

Enriching scan results with additional information. Industrial source code scanners generally offer as output more information than mere known/unknown information, as `swh-scanner` does. The latter not being a marketed product, we do not consider this a significant limitation: `swh-scanner` is meant to be a research prototype showing how the specific sub-problem of determining prior publication for FOSS artifacts can be solved efficiently using a Merkle-structured open data knowledge base. At the same time it is important to discuss how *compatible* the proposed approach is with "joining" additional information to scanning output.

Once the $known \cup unknown$ is identified, computed SWHIDs can be used as unique keys to lookup additional information about scanned artifacts that can then be included in scanning results. Typical example of additional information returned by code scanners for open compliance are: licensing information (already available from SWH itself, detected using FOSSology [17]), software composition analysis [25] decomposition (which would need to be computed separately), software provenance [14] information (that can be tracked at the scale of SWH [28]), and known vulnerability information (available from public CVE databases, but currently lacking an open data CVE↔SWHID mapping). Once these information become available from third-party KBs, extending `swh-scanner` to look them up and join them with scanning results would be a simple matter of programming.

6.1 Threats to validity

Our experimental benchmarks have been conducted on public code bases rather than on *private code bases*. We have simulated the ignorance by the knowledge base of encountered artifacts, but there is no guarantee that matches the reality of in-house code bases encountered in the real world. It is difficult to do better while at the same time preserving experiment reproducibility (on public code that anyone can retrieve and experiment with). It would nonetheless be interesting to experiment with `swh-scanner` in an industrial compliance engineering setting. We are aware of large companies integrating `swh-scanner` in their licence compliance toolchains, but no rigorous empirical experiment has been conducted yet.

The correctness of `swh-scanner` depends on the fact that the answers returned by the KB are Merkle-consistent. In particular it must hold that when the KB answers *known* for a non-leaf node (e.g., a directory), all its descendants must be *known* as well. In specific corner cases this property might not hold for the SWH archive. For example, corrupted objects from some VCS repositories might not have been archived or legal takedown actions might have forced the archive to poke "holes" into the Merkle structure. Given the Merkle DAG can only be built bottom-op, arguably, holes do not invalidate the fact that the content that used to be there should be reported as *known*; after all *it has been observed* in the past, only to disappear later. A more satisfying answer is technically possible, but requires engineering a more complex Merkle-based accounting of "holes"; doing so is beyond the scope of this paper. In quantitative terms the problem is negligible when using SWH as KB. It is also a KB-specific issue which does not impact the validity of the approach as a whole.

7 Conclusion

We introduced an efficient approach to determine prior publication of open source code artifacts based on a Merkle-structured knowledge base (KB) and implemented it in the open source `swh-scanner` tool, which uses the Software Heritage archive as KB. By scanning 16 845 code bases we have experimentally validated the efficiency of the proposed approach and tool, both in intrinsic terms (calls between scanner and KB) and in terms of wall-clock time.

Several steps remain as future work. Alternative discovery protocols are possible: they should be designed, modeled, and benchmarked to determine if further efficiency improvements are practically viable. Merkle structuring can also be improved to better cater for real-world data losses on the KB side, finer artifact granularity, and approximate artifact matching.

Acknowledgements The authors would like to thank Guillaume Rousseau for providing the project sample used in the experiments described in Section 4 and, more generally, for insightful discussions about `swh-scanner`.

References

1. Abramatic, J.F., Di Cosmo, R., Zacchiroli, S.: Building the universal archive of source code. Communications of the ACM **61**(10), 29–31 (Sep 2018). https://doi.org/10.1145/3183558
2. Almeida, D.A., Murphy, G.C., Wilson, G., Hoye, M.: Do software developers understand open source licenses? In: Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017. pp. 1–11. IEEE Computer Society (2017). https://doi.org/10.1109/ICPC.2017.7
3. Benet, J.: IPFS - content addressed, versioned, P2P file system. CoRR **abs/1407.3561** (2014), `http://arxiv.org/abs/1407.3561`
4. Capraro, M., Riehle, D.: Inner source definition, benefits, and challenges. ACM Computing Surveys (CSUR) **49**(4), 67 (2017)
5. Coughlan, S.: Standardizing open source license compliance with OpenChain. Computer **53**(11), 70–74 (2020)
6. Di Cosmo, R., Gruenpeter, M., Zacchiroli, S.: Identifiers for digital objects: the case of software source code preservation. In: Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018 (2018). https://doi.org/10.17605/OSF.IO/KDE56
7. Di Cosmo, R., Zacchiroli, S.: Software Heritage: Why and how to preserve software source code. In: Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017 (Sep 2017), `https://hal.archives-ouvertes.fr/hal-01590958/`
8. Fendt, O., Jaeger, M.C.: Open source for open source license compliance. In: Open Source Systems - 15th IFIP WG 2.13 International Conference, OSS 2019. IFIP Advances in Information and Communication Technology, vol. 556, pp. 133–138. Springer (2019). https://doi.org/10.1007/978-3-030-20883-7_12
9. Gandhi, R.A., Germonprez, M., Link, G.J.P.: Open data standards for open source software risk management routines: An examination of SPDX. In: Proceedings of the 2018 ACM Conference on Supporting Groupwork, GROUP 2018, Sanibel Island, FL, USA, January 07 - 10, 2018. pp. 219–229. ACM (2018). https://doi.org/10.1145/3148330.3148333
10. Germán, D.M., Manabe, Y., Inoue, K.: A sentence-matching method for automatic license identification of source code files. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering. pp. 437–446. ACM (2010). https://doi.org/10.1145/1858996.1859088
11. Germán, D.M., Penta, M.D.: A method for open source license compliance of java applications. IEEE Softw. **29**(3), 58–63 (2012). https://doi.org/10.1109/MS.2012.50
12. Git community: Git version control system (2005), `https://git-scm.com/`, retrieved 09 April 2018
13. Gobeille, R.: The FOSSology project. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008. pp. 47–50. ACM (2008). https://doi.org/10.1145/1370750.1370763
14. Godfrey, M.W.: Understanding software artifact provenance. Science of Computer Programming **97**, 86–90 (2015)
15. Gray, J.: Why do computers stop and what can be done about it? In: Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986. pp. 3–12. IEEE Computer Society (1986)

16. Harutyunyan, N.: Managing your open source supply chain-why and how? Computer **53**(6), 77–81 (2020). https://doi.org/10.1109/MC.2020.2983530
17. Jaeger, M.C., Fendt, O., Gobeille, R., Huber, M., Najjar, J., Stewart, K., Weber, S., Wurl, A.: The FOSSology project: 10 years of license scanning. IFOSS L. Rev. **9**, 9 (2017)
18. Kapitsaki, G.M., Kramer, F., Tselikas, N.D.: Automating the license compatibility process in open source software with SPDX. J. Syst. Softw. **131**, 386–401 (2017). https://doi.org/10.1016/j.jss.2016.06.064
19. Koltun, P.: Free and open source software compliance: An operational perspective. IFOSS L. Rev. **3**, 95 (2011)
20. Lindberg, V.: Intellectual property and open source: a practical guide to protecting code. O'Reilly Media, Inc. (2008)
21. McAffer, J.: Getting started with open source governance. Computer **52**(10), 92–96 (2019). https://doi.org/10.1109/MC.2019.2929568
22. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Proceedings of Advances in Cryptology - CRYPTO '87. pp. 369–378 (1987). https://doi.org/10.1007/3-540-48184-2_32
23. Meyer, M.: Continuous integration and its tools. IEEE Softw. **31**(3), 14–16 (2014). https://doi.org/10.1109/MS.2014.58
24. Oliver, J., Cheng, C., Chen, Y.: Tlsh–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. pp. 7–13. IEEE (2013)
25. Ombredanne, P.: Free and open source software license compliance: Tools for software composition analysis. Computer **53**(10), 105–109 (2020). https://doi.org/10.1109/MC.2020.3011082
26. Phipps, S., Zacchiroli, S.: Continuous open source license compliance. Computer **53**(12), 115–119 (2020). https://doi.org/10.1109/MC.2020.3024403
27. Pietri, A., Spinellis, D., Zacchiroli, S.: The Software Heritage graph dataset: public software development under one roof. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019. pp. 138–142 (2019). https://doi.org/10.1109/MSR.2019.00030
28. Rousseau, G., Di Cosmo, R., Zacchiroli, S.: Software provenance tracking at the scale of public source code. Empirical Software Engineering **25**(4), 2930–2959 (2020). https://doi.org/10.1007/s10664-020-09828-5
29. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. pp. 76–85. ACM (2003). https://doi.org/10.1145/872757.872770
30. Shobha, G., Rana, A., Kansal, V., Tanwar, S.: Code clone detection—a systematic review. Emerging Technologies in Data Mining and Information Security pp. 645–655 (2021)
31. Spinellis, D.: How to select open source components. Computer **52**(12), 103–106 (2019). https://doi.org/10.1109/MC.2019.2940809
32. Stewart, K., Odence, P., Rockett, E.: Software package data exchange (SPDX) specification. IFOSS L. Rev. **2**, 191 (2010)
33. Stol, K.J., Fitzgerald, B.: Inner source–adopting open source development practices in organizations: a tutorial. IEEE Software **32**(4), 60–67 (2014)
34. Synopsis: 2020 open source security and risk analysis report (OSSRA). Tech. rep., Synopsis (2020), `https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf`, accessed 2020-04-15