Applying the Universal Version History Concept to Help De-Risk Copy-Based Code Reuse

Anonymous Author(s)

ABSTRACT

The ability to easily copy code among open source projects makes it difficult to comply with the need to determine the provenance of code essential for cybersecurity and for complying with the licensing terms. Such provenance encompasses the exact origin of each component and its license, and various qualities of the component, such as absence of vulnerabilities and high likelihood of future maintenance. With the aim to address these challenges, we created an approach supported by a tool prototype, UVHistory, that links each piece of source code to all projects where it resides and, also, to its version histories in all these projects. This combined version history of a file from all open source projects we refer to as universal version history. We exemplify UVHistory via scenarios illustrating how it can help developers identify bugs and vulnerabilities and verify that license terms are not violated. Specifically, using UVHistory, developers can find the origin of a file including the open source repository where it originated, follow the evolution of the file over time and across different repositories, identify which authors have worked on a file, and read all the log messages for any modifications to that file in any repository. We also evaluate UVHistory in two contexts: to identify license non-compliance and to find instances of unfixed vulnerabilities. We find that in active and popular projects both problems are common and anyone can easily identify them using our approach.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

code reuse, software security, vulnerability, license

ACM Reference Format:
Anonymous Author(s). 2023. Applying the Universal Version History Concept to Help De-Risk Copy-Based Code Reuse. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Version control systems were a major advance in software engineering by automating storing and making accessible a complete history of the source code within a repository. The rapid growth in open source software (OSS) and its widespread use made obvious the need to create Universal Version History (UVH) (a full version history of a file across all repositories and revision histories where either parents or descendants of that file reside). More recently, the concept of a UVH appears to be articulated in "Improving the Nation's Cybersecurity" [42] order with the key component of "Enhancing Software Supply Chain Security." Specifically, "maintaining accurate provenance (i.e., origin) of software, providing a Software Bill of Materials, and ensuring and attesting to the integrity and provenance of open source software used within any portion of a product." This recent presidential order highlights an issue, software supply chain security, that has long been known to be important for various reasons but which is often overlooked. In addition to the security implications explicit in the presidential order, knowing the software provenance and providing a software bill of materials is also vital for ensuring compliance with license requirements, finding additional useful features that may be available in different versions of the software, improving code quality problems, and addressing aspects of developer reputation. The realization of this idea, however, depends on the ability to collect, clean, curate, and integrate VCS data from over a hundred million of open source repositories and remained out of reach for many years.

In this paper, we present an approach to produce a universal version history which links files across multiple repositories and multiple repository hosting platforms to construct a single history by tracing the version of a single file across all repositories and revision histories where either parents or descendants of that file reside. We then show how this approach can reduce the risks of copy-based code reuse.

Copying code for reuse in other projects is widespread [15, 20, 21, 29]. Much of the work on software supply chain issues, such as security and license management, focuses on software dependencies. A software dependency is generally considered an external component (such as a library or package) that is used within a project. When the external component is copied and committed into a project's repository, it is no longer an external component but rather is now part of the project. This approach is sometimes called clone-and-own [11] [12] [32] or vendoring [48] [3]. The cloned component is clearly part of the supply chain, but is often overlooked because it is considered part of the core project rather than a dependency once it is committed into the project's repository. This clone-and-own method can cause problems with code maintenance because of the lack of information about the connection between the clone and the original. In fact, as we show below, even the originating projects sometimes do not contain public security vulnerability fixes implemented in projects that copied the code.

Much of our work focuses on projects in languages like C and C++ because they do not have a standard package manager system. When using a package manager, it is easier to find the origin of

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

the code and any known vulnerabilities or license issues. However, many projects using languages with good package managers don't take advantage of the package manager. Therefore, we also look at languages like Java, JavaScript, Python, etc, that do have standard package manager systems.

In order to understand and address the risks associated with unknown provenance in open source software, we would first like to create a tool that is able to automate the process of producing a universal version history of a file across repositories and even across repository hosting platforms. Such a tool, if widely deployed, would inform developers about potential problems that might exist in code that they would like to reuse. Second, we want to determine if unknown provenance causes problems in real-world open source projects. For example, are there potential license violations or security vulnerabilities that are unknown to developers wishing to reuse code from another project? We find a large class of instances where even the most advanced licence violation detection tools can not (and do not) work because it is not possible to find such violations without UVH. Third, we would like to see if constructing a universal version history can help mitigate some of the problems caused by unknown provenance. Forth, we want to determine if our proposed tool detects different problems than popular dependency management products and software composition analysis tools. Fifth, we would like to see if our tool can produce useful results in a reasonable amount of time.

To establish the feasibility of producing the universal version history in general and in being able to address the issues related to code copying, we introduce a prototype tool, UVHistory, which automates the process of finding the universal version history of a file across all open source repositories. We build on the World of Code [23] infrastructure to discover and report the complete history of code in any language from a nearly complete collection of open source software. The results can be used to look for new features or functionality available in other revisions, look for security vulnerabilities reported in other revisions, find which developers have worked on the code throughout its evolution, look for license requirements that may have been lost as the code propagated, and more. Also, different revisions can be compared with public sources, such as the National Vulnerability Database, for known vulnerabilities or other bugs. This version history tracks changes to a specific file even when the file is copied to a new and possibly unrelated repository, allowing a developer to trace changes over time and across different repositories and different repository hosting platforms.

Our work makes the following contributions:

- We propose a computationally feasible approach to produce a universal version history which links source code by content and its modification history across multiple repositories and hosting platforms.
- We present a prototype tool, UVHistory, that implements our proposed approach efficiently over the nearly complete collection of open source software in World of Code. Our tool is the first application of the universal version history concept using such a large collection of open source software.
- We evaluate the application of the universal version history concept to address two specific software engineering problems identified in this paper. We show that producing a universal version history can help mitigate problems with potential license violations and security vulnerabilities caused by copy-based code reuse.
- We show that the declared license of a project cannot always be trusted, and we show how our tool can help identify those projects with incorrect copyright and license information.

The rest of the paper is organized as follows. We start with some examples and usage scenarios that motivate our work in sections 2. We present the universal version history concept in section 3 and our tool in section 4. We show the research questions in section 5, the evaluation in section 6, and we explore related work in section 7. We present limitations in section 8, look at future work in section 9, and conclude in section 10.

2 APPLICATION SCENARIOS

Source code with unknown history introduces risks such as the possibility that the code could have been modified in ways that unintentionally introduce security vulnerabilities or other bugs, that intentionally include malware, or that violate license terms. In this section, we describe in more detail specific scenarios where the universal version history concept and associated tools help de-risk copy-based code reuse.

2.1 Security Vulnerabilities

When a security vulnerability is discovered in open source software, it is typically documented in the Common Vulnerabilities and Exposures (CVE) system [41] maintained by The Mitre Corporation. Developers know to look at the CVE system for possible security vulnerabilities. However, when a vulnerable file has been copied to other projects, those other projects may not be listed in the CVE entry. When code with security vulnerabilities is cloned, the target project may inherit the vulnerability. When the vulnerability is found and fixed in the original project, the fix may not be propagated to the clone, especially when the target project does not maintain a link to the parent. Reid et al. [34] coined the term "Orphan Vulnerability" to refer to these kinds vulnerabilities that exist in copied code even after they are fixed in another project. Our proposed tool would aid developers in knowing the origin and history of the code, which would allow them to learn about the reported vulnerabilities in the original project.

It is also possible that a vulnerability is found and fixed in a file copied from an original project, but the fix is not back patched into the original project. Woo et al. [44] reported an example of this in the jpeg-compressor project [35]. CVE-2017-0700 [40] describes a vulnerability in the Android System UI that allows remote code execution. The file jpgd.cpp, which is the source of the vulnerability, was copied from the jpeg-compressor project. The vulnerability was discovered and reported in the Android source code [2], and a CVE was created. However, the vulnerability was not reported and not fixed in the jpeg-compressor project, which is the original source of the vulnerable file in Android. Therefore, developers who copy and reuse the Android code can easily find the vulnerability and the patch. However, developers who copy the jpeg-compressor project are not easily able to find out about that vulnerability, which was found and fixed in a derivative work. Clearly, it is not safe to assume that the original source is the best or most secure version. Our research aims to aid developers in finding not only the origin, but all revisions of a file across all open source repositories.

2.2 License Compliance

Another concern about copied code is license requirements. If a developer wishes to reuse software, it is important to understand the license terms of the original code. Trusting the declared license terms is not always safe. In some cases, the complete license information is not copied with the code. We found many such cases. As just one example, libofa [27] contains license information in a file named COPYING in the top level directory. But the license information is not included in every source code file. Our UVHistory tool found several cases where projects copied the source files from libofa without copying the file that contains the license information. The result is that a developer who copies the copied code without knowing the origin will not have the license information, resulting in a potential license violation. It is important to understand the origin of the code being copied in order to comply with the license terms.

The 2021 Open Source Security and Risk Analysis report (OSSRA) [39] indicates that 65% of the codebases audited in 2020 have license conflicts with open source software. Knowing the license requirements can be complicated when a developer reuses an open source project which itself reuses components from other projects. In some cases, the original license information is not propagated with the code, which means that the necessary knowledge required to answer the license question is not available to the developer, making it practically impossible to comply with the license requirements. Therefore, finding the history of open source code is critical to understanding and complying with license requirements of reused code. The tool we present later in this paper, UVHistory, traces a file to its origins which helps identify missing license information.

2.3 Additional Scenarios

Our paper focuses on addressing security and licensing concerns. The universal version history concept can be valuable in other areas as well, including identifying other code flaws, such as defects, incompatibilities, or even missing functionality. It may even support better author attribution considered an important motivation for open source developers [1]. We briefly describe those scenarios here, but leave in-depth study of these areas for future work. Finding the complete file history, including all ancestor and descendant code, can be valuable for finding additional useful features available for a piece of software. Source code is often copied into different projects and then improved for use in that project. These useful enhancements or fixed bugs would often be valuable to other projects, but maintainers of other projects are often unaware of them. An old clone may be missing the latest enhancements/fixes that could add improved quality, functionality, performance, or other benefits.

Widely copied code may indicate its high utility or other aspects of quality. Knowing which developers have worked on the specific code in question can help build trust in that code. It may also serve as an indicator of popularity for other developers who may benefit from the widely used functionality implemented in such code. As such, a tool such as the one we propose here could serve as a component of a code recommender system. The tool could also be used to identify the developers who create such widely used code and help increase their reputation, direct support, or other resources, to motivate such production.

3 UNIVERSAL VERSION HISTORY CONCEPT

The concept of a universal version history(UVH) is to track the evolution of a source code file across multiple repositories. It is the documented history of a file that has been modified and potentially copied across different repositories which may be hosted on different repository hosting platforms. This documented history includes verifiable information about revisions to the file, dates of those revisions, log messages for every revision, and the chain of custody (who wrote the code, who revised the code, and what projects included the code).

It is worth defining UVH more precisely and comparing to a common version control system such as git. The essential entities are versions of the source code (blobs). In git, each blob can be associated with all versions (commits) of the repository where it is present, and each version may be associated with one or more filenames (including the full path from the root of the repository). The blobs associated with such file with a pathname can be used to determine a version history of a file (and git provides several heuristics methods on how to do that: the lack of determinism of file history arises with merges). A particular commit “creates” a blob if either there was no such file in the previous version (parent commit(s)) or if the previous blob was different. We can thus use the time of the first commit creating a blob (the same blob can be created multiple times in different folders or even in the same folder) to obtain the time when blob was introduced to a repository. Furthermore, each time a blob is created by a commit, we link it to old blob: a blob (if any) that exists in the parent commit for the same filename. Hence within a repository is simply a graph linking each blob to the “old blob” and to commits that created it (including all commit attributes such as time, author, commit message, and the pathname of blob-associated file). Notice that it is a bit different from version control systems such as CVS or SVN, where versions of individual files are tracked. Notice that the resulting graph has several distinct kinds of nodes (e.g., blobs and commits), and multiples types of links (e.g., blob to old blob, commit to parent commit, and blob to creating commit). In UVH, we simply add one more type of node: a repository. Each repository is linked to all commits within that repository and, transitively, to all blobs contained therein. Blobs, on the other hand, are linked to all commits in all repositories. The first creation time for a blob is defined the same way. We thus can identify the original commit and original author for every version of the source code in WoC. In addition to the time of the commit, a partial temporal order based on the old-new blob relation is available. Section 8 discussed in more detail how we handle potentially inaccurate time recorded in git commits. Notably, as we expand the scope of UVH across repositories we loose some aspects of a sequence. For example, let blob *a* be first created in repository *A*, then in *B*, and lastly in *C*. In such case, without additional information (who did the commit and what other blobs were created) it would be impossible to determine if repository *C* got the blob from repository *A* or from repository *B*. For some applications, determining if the blob came from *A* or *B* is not as important as identifying licenses, vulnerabilities, or other attributes of the blob that may vary among these various repositories. In cases where knowing the true origin is critical, the UVH reduces the search space required so that manual inspection is feasible. The tool cannot know the origin for sure, but can point to the earliest commit into a public repository.

The universal version history (especially in combination with techniques employed in Mining Software Repositories field) can be used to find or infer other information about the code such as copyright notices and license information, the reputation of the authors, the quality of the code, what coding standards were used, the use of (or lack of) secure coding standards, what vulnerabilities have been reported, what test methods were used, what security assessments and were done, the location where it was developed and modified, the trustworthiness and reliability of the code, and the likelihood that the project will continue to be maintained.

4 THE UVHISTORY TOOL

UVHistory is our tool for automating the process of tracking changes across all open source repositories and their version histories by operationalizing the universal version history concept introduced in section 3. It supports the study of issues concerning code reuse in real-world open source projects.

4.1 Infrastructure: World of Code

Before describing our tool, we need to introduce the infrastructure, World of Code, on which our tool is built. World of Code [24] is a nearly complete collection of all publicly available open source software. Software source code is periodically collected from many sites including GitHub, Bitbucket, SourceForge, and many others. The software is then curated and stored using methods that allow for efficient searching of the very large amount of source code collected. World of Code, which aims to support research in software engineering, currently contains over 20 billion Git objects with over 100 million unique public repositories (not including forks or empty repositories) [45].

This World of Code infrastructure, with its extensive collection of open source software, allows us to find code duplication across projects where no link to the origin exists on a scale that is not possible without this kind of infrastructure.

4.2 UVHistory

UVHistory takes as input the contents of a file and finds all duplicate versions of that file or any revision of that file across all of the open source software available in WoC.

UVHistory specifically looks for source code from open source projects that is copied and committed into different projects. This approach is different than most existing tools for identifying vulnerabilities and licenses which look for external libraries that are linked in or look at package management systems for dependencies.

Some systems tie into the build process and detect any libraries or other third-party dependencies. But these approaches miss code that is copied and committed into the source code repository without any link to the original project. Our tool is different than other tools in that it is specifically designed to find these kind of file-level copy dependencies that have no link to the original project and are therefore missed by existing tools.

Since World of Code archives source code repositories over time, we are able to trace the history even to projects which are no longer available on public source code hosting platforms.

4.3 Algorithm

For simplicity we assume that our algorithm is provided one or more sha1 hashes computed via the method used by Git. A user of our tool may, instead, provide an entire repository or a specific file name within a repository. In that case we have simple scripts that collect either the complete set of blobs in a repository or a complete set of blobs associated with a particular filename. In any case, we start with one or more sha1 hashes as input. These hases correspond one or more blobs, which is our seed list from which to start finding more files in the universal version history.

Next, we use WoC's blob to old blob mapping recursively to find all ancestor blobs. Similarly, we use WoC's old blob to blob mapping recursively to find all descendant blobs. For each blob, we use WoC's blob to commit mapping to find all commits containing any of the blobs that have been found. We now have all commits for all revisions of the file across all source code repositories. The commit gives us the time, author, pathname, and log message for that revision of the file. From the commit, using WoC's commit to project mapping (c2P), we find all projects which contain a revision of the given file. Now we have the information needed to construct a link to that revision of the file on the repository hosting platform (such as GitHub, Bitbucket, SourceForge, etc). The final output of the tool shows the complete history of the file with all ancestor and descendant revisions across all repositories. The history includes the author of each revision, the date it was committed, the log message of the commit, the link to the original source of the project of which that revision is a component if the project is still publicly available and accessible, and a link to the specific revision of the file on the hosting platform (if available).

For each blob, we sort all of the commits for that blob in date order. We then sort the blobs by the date of the earliest commit of that blob. The date we use is the date in the Git commit. It is possible that the date in the Git commit is not correct. We look for obvious discrepancies; for example, values of 0 or dates that are in the future are clearly not correct. In addition, if we see a date that is before Git was released in 2005, we flag it as suspicious. It is possible that the early date is correct, as it may be a file that was migrated from a different source code control system such as SVN or CVS. We also flag any dates prior to 1990 when CVS was introduced, as it is somewhat unlikely that any date prior to the introduction of CVS is accurate. We cannot guarantee that the date in the Git commit is correct, which we note in the limitations section.

When identifying projects in the universal version history, we want to find distinct projects. GitHub projects often have many forks, sometimes tens of thousands. Most of those forks are not independent projects; many were only created for the purpose of issuing pull requests to the original project. Showing tens of thousands of related forks makes it far more difficult to find the useful information. Using the community detection algorithm described in [26], WoC maps each Git repository to a central repository which is expected to represent the same project. This mapping in WoC allows us to create a list of deforked projects. UVHistory displays the deforked projects on the main page and then includes a link to a secondary page that contains a list of all projects, including (possibly irrelevant) forks.

4.4 Output

Our final output contains:

- A list of all blobs in the universal version history. The list is sorted by date in reverse chronological order. The blobs are named by the sha1 hash of the blob as computed by Git.
- For each blob, a list of all commits of which this blob is a part. The commits are named by the sha1 hash of the commit as computed by Git. The commit information includes the author, time of commit, and the commit log message.
- For each commit, a list of all of the projects where this commit was applied, the full pathname of the file, and the URL linking to the file on the source code hosting platform (note that the link may be dead if the project is no longer publicly accessible).

5 RESEARCH QUESTIONS

In this section, we present four research questions and the research methods used to address each of the questions. Our study is designed to show the relevance of the problem presented and also to evaluate the effectiveness of our proposed solution. The results are presented in section 6.

Our goal is to see if constructing a universal version history across repositories and across hosting platforms can help solve the class of problems presented earlier in this paper. We specifically look at two of the problems mentioned earlier: potential license violations and security vulnerabilities. The other issues are similar and can likely be solved in a similar way, but we leave those for future work.

RQ1: Can the declared license in an open source software be trusted?

The aim of this question is to see if it is common in real-world open source software projects for code to be copied from other projects without the correct copyright and license information being retained and without a clear link back to the original project where the copyright and license information can be found. When code is copied, is the correct license information copied along with it, or if not, is the correct license readily available. We are specifically looking for real-world projects, not toy projects or student assignments.

If we can trust that the license information provided in projects is correct, then finding the universal version history is not necessary to be able to properly understand and comply with the license terms. If, however, we find that there are frequent license violations due to missing or incorrect license information in popular open source projects, then we will conclude that it is worth our effort to find ways to mitigate the problem. The specific problem we want to mitigate is the problem of copying code without knowing or without having an easy way to find the correct license terms for that code. We want to determine if there is real-world benefit in a tool to help mitigate this problem.

Our research method to answer RQ1 was an exploratory study designed to see the extent of of the problem. We examined open source projects which have copied code from popular open source projects to see if the correct copyright and license information was propagated along with the copied code. We developed some tools to select a set of projects that are likely to contain license problems. We then manually inspected that subset.

RQ2: Can our UVHistory tool, by constructing a universal version history, help identify projects with missing copyright and license information and help find the correct information for the given code?

It is important, when reusing software, to comply with the license terms. One cannot comply with terms of which one is not aware. Reusing software without knowing the correct license terms can cause someone to infringe intellectual property without being aware of the infringement. We want to see if our tool can help developers identify when correct license terms are missing and help them find the correct license.

Most open source licenses require the copyright and license information to be retained. Cases where projects copy code, but omit the copyright and license information, is a clear violation of the license.

We used a case study to answer RQ2. We studied two cases where license information was not properly propagated. The two case were selected from results of the study for RQ1. The case study method allowed us to look in-depth at two specific projects.

RQ3: Can our UVHistory tool, by constructing a universal version history, help identify projects which are subject to security vulnerabilities that have been found and fixed in another project but which still persist unknowingly in other projects.

Previous research [34] [44] [8] has identified this as a real problem in popular real-world projects. Due to the seriousness of this problem, a tool that could help mitigate this problem would have value.

We answered RQ3 with another case study. This case study examined a case we introduced section 2.1 as one of the motivating examples. Again, the case study method allowed us to have an in-depth look at a project. This time, the project studied contained a known security vulnerability propagated through code reuse.

RQ4: Is the performance of UVHistory such that it can run in reasonable time on commodity hardware for source code files in typical open source projects?

In order to have practical value, the tool needs to be able to produce results in reasonable time on reasonably affordable hardware.

Our final RQ is addressed with a simple study to examine the performance of our prototype tool on common projects using specific hardware.

6 EVALUATION

We evaluate our method and tool by answering each of the four research questions, and we present the results in this section.

6.1 RQ1: Can the declared license be trusted?

To answer our first research question, we searched for cases where code was copied from one project to another, but the copyright and license information was not copied. We were not looking for projects that used package managers or linked to external libraries, but only cases of code cloned from one project and committed into the repository of another project.

We selected a small subset of projects to investigate in more detail. We randomly selected 100 source code projects from GitHub which include a top-level file named LICENSE.txt containing a license that requires copies to retain the copyright notice, had no reference to copyright or license information in the individual source code files, and had more than 1000 stars. We chose projects with a top-level LICENSE.txt file because it is common for projects to put the license information in a single file in the top-level directory of a repository and not duplicate the license in every source code file. LICENSE.txt is one common filename used for the license file. When the license information is not included in every source code file, it is easy for a developer to copy a copyrighted source code file without copying the relevant license information. We limited our selection to repositories with more than 1000 stars so that we would find popular projects [4] that are likely to have files that are copied into other projects. The selected projects were composed of projects in a variety of languages and using a variety of licenses.

For each of the 100 projects, we used our UVHistory tool to trace the history of one of the files in the project to find other projects which had copied code from the original project. We then checked those other projects to see if the copyright and license information had been propagated to the new project. In the few cases where there were more than 500 clones of a project, we limited our search to the first 500. Because of the manual work involved in investigating each license, we had to limit the scope. We looked at 100 original projects and up to 500 clones of each of those 100 original projects.

Our procedure for finding out if the projects containing cloned code also contained the proper license was as follows: First, we used UVHistory to find projects containing copies of the code in question. Next, we used a tool we developed (also layered on top of World of Code) to find all licenses used in a project. The tool used the winnowing algorithm [37] to find the most similar license from among 1862 licenses provided by spdx[1] for each blob in a project. The winnowing algorithm relies on extracting a collection of signatures from text and matching them among documents (blobs in the project and blobs representing licenses). We compared the known license in the original project to the licenses found by our tool to see if there was a match. If there was an exact match, then clearly the license information was correctly propagated with the code. If there was not a match, we manually inspected the project to make sure that, in fact, the correct license was not included. We also checked to see if the project was still publicly available, as World of Code will still have information about removed projects but we only care about projects that are currently publicly available. If neither our tool nor our manual inspection found a match, then we conclude that the correct license information is not properly included.

[1]https://spdx.org/licenses

In 76 of the 100 projects, we found at least one case where code from that project had been cloned to another project. In 54 of the 100 projects, we found at least one case where another project had copied the code but had not copied the copyright and license information and did not include an obvious link back to the original project where the copyright and license information could be found. In total, we found 3,431 projects which had cloned code from one of the original 100 projects (Note that our 500 project limit reduced that total). We found that 1,132 of those projects did not properly retain the copyright and license information.

The answer to RQ1 is clearly no, the declared license cannot always be trusted. License violations caused by license omission are common in real world projects since we found a high percentage of popular projects where code is copied but the license and copyright information are not retained (as required by the license) and there is no link from the copied project back to the original project where the license can be found. These non-compliant projects are publicly available, which means that someone might very well copy and use the code without being aware that they are violating the license terms.

The answer to RQ1 suggests that future work involving a large scale empirical study concerning license omissions in cloned code would be valuable.

6.2 RQ2: Can UVHistory help with license compliance issues?

Based on the answer to RQ1, someone who wants to reuse code from one of these non-compliant projects would have no easy way to know the license terms that must be followed unless they obtained some additional information. Our second research question considers whether our UVHistory tool can provide the additional information necessary to ensure compliance. To answer this question, we look in detail at two specific projects chosen from the ones identified in the section above.

The first project chosen was AIOHTTP[2], an asynchronous HTTP client/server framework. We chose AIOHTTP because it has over 12,000 stars on GitHub, indicating it is a very popular project likely to be copied, and because it has the Apache license which is a very common license which requires the copyright notice to be retained in all copies. We have already discovered, in answering RQ1 above, that there are projects which reuse AIOHTTP without following the license terms that require attribution. To answer RQ2, we want to find out if UVHistory can confirm that proper license terms are followed or, if not followed, find the correct license for the project. We start with projects that we know, from our study of RQ1, do not comply with the license requirement to include the copyright notice and do not provide a clear link to the original project that contains the correct license terms. There are many cases of projects that use AIOHTTP without retaining the copyright notice as required. We pick just one, Hackathon-Torrent[3], to show that UVHistory is able to identify the correct license and copyright notice that should be included with any reuse of this project. Running UVHistory on a source code file in the Hackathon-Torrent project, we find the AIOHTTP project as the project in the universal version history

[2]github.com/aio-libs/aiohttp

[3]github.com/AdoenLunnae/Hackathon-Torrent/

with the earliest date. Following the link produced by the tool takes us to the AIOHTTP project on GitHub where the license and copyright information is very clearly available. This means that someone wishing to reuse the Hackathon-Torrent project could very easily, by looking at the universal version history produced by UVHistory, find the correct license and copyright information that is missing from Hackathon-Torrent.

The second project chosen was VirtualXposed[4]. We chose this project because it is widely copied and because it has a commercial license. The commercial license is particularly problematic when copied into open source projects, especially when the license information is not propagated with the copy. We traced the code from the VirtualXposed project to it's origin, which is VirtualApp[5]. VirtualApp's README is very clear that in order to use this software you must purchase a license. However, VirtualXposed includes the GPL license in it's LICENSE.txt file, which would make it appear that it is available under GPL, but that is not completely correct since it also include commercially licensed code. VirtualXposed has more than 15,000 stars on GitHub and more than 2,000 forks, indicating that it is a widely used and copied project. Following the history of the project produced by UVHistory, we find several copies of this code that include an open source license such as Apache or no license at all. Without a tool like UVHistory, there would be no way to know that these copies of VirtualApp are restricted by a commercial license. Some examples of projects that do not propagate the commercial license follow. We only list a few examples, there are many more than what we have listed here. VirtualDump[6] contains copies of some of the code that originated in VirtualApp, but it does not include any license information or any link back to the original VirtualApp project. YCVaHelpTool[7], which also uses code from VirtualApp, includes the Apache license in a file named LICENSE. There is no mention of the VirtalApp or the commercial license, leaving a developer wishing to reuse the code assuming that it is available through the Apache license. Following the universal version history to the origin again leads to the correct license and copyright information.

The answer to RQ2 is yes, the universal version history can help identify missing license information and find the original project containing the correct license information. We demonstrated that our UVHistory tool can effectively find the original project, allowing developers wishing to reuse code to be able to find the correct license information.

We contacted the project maintainers of these projects to report license issues.

6.3 RQ3: Can UVHistory help identify projects with security vulnerabilities?

To answer RQ3, we follow a similar procedure as for RQ2, except we look at projects with known security vulnerabilities rather than potential license violations.

In section 2.1, we used a security vulnerability in a jpeg compression library as a motivating example for this work. That case was particularly challenging because the fix for the vulnerability was not in the original project from where the code came, but rather in a project that had reused the vulnerable code and then fixed it. Thus finding the origin is not enough, we also need to look at other projects in the history. The specific example we look at, Entropia Engine++[8], is a cross-platform game and application development framework. With recent commits and a number stars it appears to be a reasonably active and popular project. It reuses the vulnerable file jpgd.cpp. The header comment in that file references the jpeg-compressor project from where the code was copied. A developer wishing to reuse Entropia Engine++ could easily know from where it was copied. However, the CVE identifying the vulnerability lists the Android System UI[9] where the vulnerability was fixed. There is no clear way for the developer reusing it to know that it in fact contains the vulnerable version of jpgd.cpp. This is where UVHistory proves its value. By finding the universal version history using UVHistory, we are able to see not only the original jpeg-compressor project, but also other projects which reuse it, including Android. Searching the universal version history for common strings like "CVE" or "vulnerability" finds hints about potential problems. In this example, we find 2 hits when searching for "vulnerabil": "38889eb Fix series of JPEG vulnerabilities by xxxxx" and "890381c Fix security vulnerability by xxxxx"[10], both from the Android project. This allows a developer wishing to reuse Entropia Engine++ to find the potential vulnerability CVE-2017-0700 by searching the universal version history. Commit 38889eb fixes this vulnerability.

We contacted the project maintainers of the project with the cloned vulnerability to let them know about the issue. The issue was fixed on June 28, 2022 by updating to a new version of jpeg-compressor, so the project is no longer vulnerable.

6.4 RQ4: Is the UVHistory prototype feasible?

Our final research question considers performance. We want to understand if it is feasible to effectively identify code history across repositories on such a large scale.

Our tests were performed on a machine with Intel(R) Xeon(R) Gold 6148 CPUs running at 2.40GHz. We limited our program to 16 threads running in parallel to limit the load on the machine which is in heavy use by multiple users. As a prototype tool, UVhistory is not optimized for performance. Increased parallelism and other enhancements would improve performance.

By leveraging the World of Code infrastructure, which has already curated the data and stored useful information in a database which can be efficiently searched, we are able to produce results relatively quickly. We looked at the timing on the 100 cases selected for RQ1. Our timing results varied greatly based on how many different projects contain a cloned version of the file in question. Most of the projects in our 100 cases had less than 500 clones. The elapsed time for a case with 187 cloned projects was 9 minutes. The worst case, which found clones in well over 10,000 projects, took just under 3 hours. Thus we conclude for RQ3 that, yes, UVHistory is able to finish in practical time.

[4]github.com/android-hacker/VirtualXposed
[5]github.com/asLody/VirtualApp
[6]github.com/LiveSalton/VirtualDump
[7]github.com/yangchong211/YCVaHelpTool
[8]github.com/SpartanJ/eepp
[9]source.android.com/security/bulletin/2017-07-01#system-ui
[10]Author names redacted for privacy

6.5 Evaluation of Existing Tools

The goal of our tool is similar to that of Software Composition Analysis (SCA) tools, but our methods are different and therefore help developers find issues not found by SCA tools. SCA tools identify the open source software in a codebase in order to find security, license compliance, and code quality issues. In this section, we identify existing tools, describe a test case we set up to test those tools, and then present the results of the test.

Current open source SCA tools that detect license compliance issues look at licenses that are explicitly declared in a project being reused through code clones or through a package manager. They trust the declared license in a project or source code file. What they fail to find are cases where code is copied from project to project multiple times, and sometimes modified, without the license information also being copied. The history is lost, making it impossible to find the original license. Commercial tools are harder to evaluate. Some tools claim to find clones from a large collection of open source software, but we do not have access to that collection and cannot evaluate its completeness. Most tools appear to trust the declared license without searching for the origin of the cloned code. We tested some of those tools, both open source and commercial, and present the results below.

Similarly, with vulnerabilities, current open source tools fail to trace the history as a file is modified and copied across repositories, and therefore often miss vulnerable code that has been copied from a known vulnerable project to a different project. Our research shows that cases like this are common, and that our tool can help identify these cases. Again, commercial tools are harder to evaluate. Most appear to have the same limitations. We tested several using a our example project containing a cloned vulnerability.

We created a small test case example project where we built a very simple HTTP client and server using code cloned code from a vulnerable version of aiohttp[11] (a project which we identified when we collected data for RQ1). We cloned only the directory that contains the source code, but we did not clone the top-level directory, which contains the License.txt file. We added an MIT license for our example project. Our project cloned v3.7.3 of aiohttp, which is subject to the vulnerability described in CVE-2021-21330. Anyone wanting to reuse our project would assume everything in the repository is available under the MIT license. It is not immediately clear that parts of the project are actually subject to a different license. Additionally, the project contains a known vulnerability, but our project is not listed in any CVE entry. This example project mimics real-world cases that we found in many open source repositories.

Popular free dependency checker tools such as GitHub Dependency Graph [16], Dependabot [10], Google Open Source Insights [18], and OWASP Dependency-Check [30] rely on supported package ecosystems that use a supported file format because they rely on the packaging information to find the dependencies. This means that languages like C and C++, which don't have a standard package management system, are not well supported by these kinds of tools. Even projects using languages that have popular package management systems sometimes copy and commit the code into their own repositories rather than using the package management system. In our tests using our example project, none of these 4 tools detected the license or security issue. This is as expected since our example uses cloned code rather than a package manager.

[11]github.com/aio-libs/aiohttp

We next tested two commercial SCA tools: FOSSA[12] and Snyk[13]. We chose those two because they were listed in "The Forrester Wave"[14] 2021 Q3 Report as having strong market presence, and they have free downloadable trials available. We did not look at commercial tools that do not provide a free download of a trial version. While it is harder to know the exact capabilities of closed source tools, the public documentation and trial versions give us good insight.

FOSSA traditionally relied on package manager information to find license compliance issues. They recently announced (on November 1, 2022) support for "vendored code" (what we in our introduction call "clown-and-own or vendoring"). We tried out their free version (which supports license compliance but not vulnerability management) on our example project. Using FOSSA's new option –experimental-enable-vsi (which enables vendored source code identification), FOSSA did not detect the missing license information from the cloned file we inserted.

Snyk provides tools which address both security vulnerabilities and license issues. Synk's free version does not support license compliance, so we signed up for their 14 day free trial, which supports both vulnerability management and license compliance. We ran the test with our example project described above and Synk did not report the license violation or the security vulnerability.

Our tool's purpose is to help developers find the provenance (history and chain of custody) of a file, which can help them find security and license issues. We make no claim that our tool competes with these very impressive SCA tools. We only claim that it can, in some specific cases, help a developer find an issue that SCA tools miss.

7 RELATED WORK

7.1 Universal History

Early work on finding a complete version history was conducted by Chang and Mockus [5]. They looked for cases where directories of source code contain many files with the same names and then compared those files to find clones. The matching files and their version histories were used to construct the file history. In follow-up work [6], they proposed a large-scale copy detection and validation process and improved reuse detection. Mockus [25], using the same algorithm, found significant large scale code reuse where many files were copied. At the time of their work, there were no complete collections of open source code like World of Code, which limited their work to a small number of repositories and only worked when multiple files in a directory were duplicated and the filenames did not change. They concluded that there was still a challenge to scale the work to very large numbers of open source repositories [6]. World of Code provides the infrastructure to meet that challenge, which is the goal of this work.

[12]https://fossa.com
[13]https://snyk.io/
[14]forrester.com/policies/forrester-wave-methodology/

7.2 Large Scale Software Archives

World of Code [45] and Software Heritage [38] provide large scale code archives. Our tool is built on World of Code, which we described earlier. In this section, we look at related work that uses Software Heritage.

Software Heritage Graph Dataset [31] links together source code file contents, which allows duplicate code to be found across projects, much like what is provided by the World of Code data maps that we use. What they do not include is the linkage of the history of each file within a project to all other projects containing any version of the file. This linkage is what UVHistory provides.

Provenance work by Rousseau et al [36] using Software Heritage looks at occurrences of the "exact same file content." They specifically state that they do not look at "predecessors or successors in a given development history" and that that is "outside the scope of the present work". The strength of our work, and much of the effort to produce it, comes from tracing the full history by following the predecessors and successors, thus giving us a complete history that follows the evolution of a file as it changes over time, not just instances of exact copies.

7.3 Tracking Code Changes

Kawamitsu et al. [21] proposed a technique to find which file revision a copied file comes from in another project for the purpose of keeping copies up-to-date. They aimed to identify which revision of a file was reused and how that file was modified over time. Their method only looked at project pairs to find files that were copied from one project to the other, but it cannot handle a large number of projects. Ishio et al. [20] expanded on the idea of tracking code changes by taking a set of source files in C/C++ and Java and finding files that are likely to include the original version of the file. They look at a relatively small subset of projects compared to what is available in World of Code. They note that tracking file changes across repositories is tedious. We further expand code change tracking by using World of Code's massive collection of projects to track modifications to files in any language across a nearly complete collection of open source software.

7.4 Finding File Origin

Xia et al. [47] looked at reuse of third-party code and found that 18.7% of the projects studied copied only the source file but no companion files like readme or changelog files; therefore, the version information and links back to the original project are lost. This is particularly relevant to our study of license terms as the license and copyright information is often only in the companion files. They also discovered that third-party code is sometimes mixed with other third-party code, making it even harder to trace each file back to its original project.

Inoue et al. [19] designed and implemented a tool that used source code search engines to take source code fragments and find sets of cloned code fragments in order to track the history of the code. Limitations of those search engines, such as only allowing keywords and/or code attributes as their inputs or not allowing automated queries, posed challenges to the tool. The source code search engines they used (Koders, Google Codesearch, and SPARS/R) are no longer available. We use World of Code, which is currently actively maintained.

Davies et al. [9] introduced a method to reduce the search space when looking for the origin of a piece of code in cases where a direct link to the origin is not clearly available. Once the search space is reduced, manual inspection or other expensive methods can be used to identify the origin from the reduced set. They demonstrated their method on a collection of Java files.

Godfrey et al. [17] pointed out that it is becoming increasingly important to determine the origin of software in cases where code is cloned into a new project with no clear link to the origin, but that effective techniques for finding such code provenance do not yet exist. We aim to help fill the gap that they identified.

Woo et al. [44] proposed an approach to find the original software where a vulnerability originated. They noted that many CVE [41] reports do not give the correct origin of the vulnerability. Finding the true origin can help mitigate further propagation of the security risk. Their method uses function-level clone detection methods, which can be more precise, but not as efficient at large scale as the file-level clone detection we use. They only used about 10,000 projects, and only from GitHub, for their evaluation.

7.5 Security Issues

Davies et al. [8] performed manual license and security audits in real-world applications and found potential legal and security issues in some of the studied applications.

Kula et al. [22] looked at Java projects that use a dependency management tool and found that 81.5% of projects in their study still have outdated dependencies, many with security vulnerabilities. They also, through surveys, found that 69% of the developers were not aware of the vulnerability. We hypothesize that the number of outdated cloned copies of files that have no link back to the origin would be even higher.

Chen et al. [7] designed and implemented a machine learning system to help identify which libraries in open source dependencies contain vulnerabilities listed in the National Vulnerability Database (NVD) [28]. It relies on package management systems including Maven Central, npmjs.com, and PyPi.

7.6 License Compliance Issues

German et al. [14], through an empirical study of license issues in open source projects, show instances of incompatible licenses when open source code is reused in different projects. They found that there are often mismatches between the declared license of a package and the license of the source code within the package, and also incompatibilities between packages contained within one project. They note that auditing of license issues is "quite complex" and suggest that improving automation is this area would be beneficial. This kind of automation improvement is exactly the aim of our work.

Wu et al. [46] looked at license inconsistencies within large projects. In their conclusion and future work section, they say "These problems highlight the need for a method to find and maintain provenance between applications". Our work, using World of Code, looks for inconsistencies across all open source projects as they suggest.

Wolter et al. [43] found that the license declared at the top-level of the repository does not always match the license found in source code files.

Qiu et al. [33] looked at dependency-related license violation and report a relatively small number of dependency-related violation in npm. The small number is in part because permissive licenses are more common in npm. Our work looks at code clones rather than dependencies.

The work cited in this section finds license inconsistencies. That is similar to our work, but what we are looking at is slightly different. We are looking at cases where the license information is not retained when code is cloned from one project to another, possibly multiple times, and there is no clear link back to the original project. Without being able to trace the history of the file across repositories, someone reusing one of these projects with missing license information would have no way to know that they are violating the license when they reuse the code.

8 LIMITATIONS

Our UVHistory tool uses the vast source code archive in World of Code to find clones of open source code. The tool will not find clones of code that are not included in the World of Code data.

If a source code file has been identified as containing a security vulnerability, the project using that file might be subject to the vulnerability. However, the project might not be using the vulnerable file in a vulnerable way. Our tool can help identify if vulnerable code is included in a project, but cannot identify whether it is used in a vulnerable way.

The tool trusts the timestamp and author information in the Git commit. There are occasional cases where that information is not correct. Flint et al. [13] demonstrated that while timestamps are usually accurate, there are unusual cases where the timestamp is not correct. We used the reported suggestions and additional techniques, like identifying unreasonable (empty, too old or two new) and inconsistent (parent commit occurring after the child commit) time stamps to weed out some of the problematic commits.

Additionally, the first commit may have borrowed from a source that is not in World of Code (like Stack Overflow), in which case our tool will not find the true origin, but rather find the first place where it was committed into an open source repository.

The tool only looks at file-level copying. It will not detect snippets of code that are included in a file. It will also not detect a copy if a developer copies a file and modifies it before committing to the new repository. See section 9, Future Work, for ideas about how these issues could be addressed.

9 FUTURE WORK

In this study, we came across a couple of interesting questions that we think would be worth studying. The jpeg compressor issue was fixed and CVE entry created in a well funded Android project. The fix was not quickly put into the less funded jpeg-compressor project. Is there a difference in the number of vulnerabilities in better funded projects vs less funded projects? Also, what is the prevalence of commercially licensed code being copied into open source projects?

While the current tool only looks for file-level duplication, the tool could be expanded to also look for duplicated code fragments. Finding the origin of code fragments would also be useful in addressing the challenges discussed in this paper. The current tool, in order to scale to near the entirety of open source software, uses hash matching. While we cannot escape hash matching to work at this scale, it would be possible to match on multiple hashes: from the original content (as described), from the tokenized content, from content with removed comments, from computed ASTs, vulnerability fixing diffs, etc. Matching the sets of tokens or ASTs would yield many false positives, but more precise similarity measures can be employed for the matched sets since the number of comparisons would be tiny compared to the entire collection of open source software. Similarly, locality-sensitive hashing (LSH) could be employed. Additions to the underlying World of Code infrastructure would be required to support these kinds of enhancements.

10 CONCLUSION

In this paper, we articulate the concept of universal version history and argue for its usefulness in the context of the entirety of open source software. We introduce a prototype tool, UVHistory, that leverages the World of Code infrastructure to collect information about the source code and other artifacts to help better understand and manage widespread copying of source code. We demonstrate the value of the universal version history concept by finding evidence of negative affects of reuse, including reuse of outdated code that contains known vulnerabilities or other bugs, is missing useful features, or has different license restrictions. Our UVHistory tool helps automate the production of the universal version history of source code by tracing code among repositories and enables finding the origins and version history for any source code file. We have shown the potential of our approach by demonstrating a solution in two different contexts which have practical relevance: license compliance and security vulnerabilities.

11 DATA AVAILABILITY

To encourage independent verification or replication, we have made the source code for our prototype tool and the data we generated for the study available. These artifacts are available at https://figshare.com/s/aac2c1a68f976f20b860. Replication requires access to World of Code, which is freely available at worldofcode.org.

REFERENCES

[1] S. Amreen, A. Karnauch, and A. Mockus. 2019. Developer Reputation Estimator (DRE). In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1082–1085.

[2] Android. 2017. *Android Security Bulletin—July 2017* . https://source.android.com/security/bulletin/2017-07-01

[3] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 42 (jul 2021), 56 pages. https://doi.org/10.1145/3447245

[4] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[5] Hung-Fu Chang and Audris Mockus. 2006. Constructing Universal Version History. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 76–79. https://doi.org/10.1145/1137983.1138002

[6] Hung-Fu Chang and Audris Mockus. 2008. Evaluation of Source Code Copy Detection Methods on Freebsd. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (Leipzig, Germany) *(MSR '08)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/1370750.1370766

[7] Yang Chen, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2020. Automated Identification of Libraries from Vulnerability Data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 90–99. https://doi.org/10.1145/3377813.3381360

[8] Julius Davies. 2011. Measuring Subversions: Security and Legal Risk in Reused Software Artifacts. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1149–1151. https://doi.org/10.1145/1985793.1986025

[9] Julius Davies, Daniel German, Michael Godfrey, and Abram Hindle. 2013. Software Bertillonage. In *Empirical Software Engineering*. https://doi.org/10.1007/s10664-012-9199-7

[10] Dependabot. 2021. *Github Dependabot*. https://github.com/dependabot

[11] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 391–400. https://doi.org/10.1109/ICSME.2014.61

[12] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 665–668. https://doi.org/10.1109/ICSE.2015.218

[13] Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. 2021. Escaping the Time Pit: Pitfalls and Guidelines for Using Time-Based Git Data. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*.

[14] Daniel M. German, Massimiliano Di Penta, and Julius Davies. 2010. Understanding and Auditing the Licensing of Open Source Software Distributions. In *2010 IEEE 18th International Conference on Program Comprehension*. 84–93. https://doi.org/10.1109/ICPC.2010.48

[15] M. Gharehyazie, B. Ray, and V. Filkov. 2017. Some from Here, Some from There: Cross-Project Code Reuse in GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 291–301.

[16] Github. 2021. *About the dependency graph*. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph

[17] Michael W. Godfrey, Daniel M. German, Julius Davies, and Abram Hindle. 2011. Determining the Provenance of Software Artifacts. In *Proceedings of the 5th International Workshop on Software Clones* (Waikiki, Honolulu, HI, USA) *(IWSC '11)*. Association for Computing Machinery, New York, NY, USA, 65–66. https://doi.org/10.1145/1985404.1985418

[18] Google. 2021. *Open Source Insights*. https://deps.dev/

[19] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. 2012. Where does this code come from and where does it go? - Integrated code history tracker for open source systems. In *2012 34th International Conference on Software Engineering (ICSE)*. 331–341. https://doi.org/10.1109/ICSE.2012.6227181

[20] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue. 2017. Source File Set Search for Clone-and-Own Reuse Analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 257–268.

[21] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, and Katsuro Inoue. 2014. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 305–314. https://doi.org/10.1109/SCAM.2014.17

[22] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (01 Feb 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[23] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus. 2019. World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 143–154.

[24] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering* 26 (2021). https://doi.org/10.1007/s10664-020-09905-9

[25] A. Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. 7–7.

[26] Audris Mockus, Diomidis Spinellis, Zoe Kotti, and Gabriel John Dusing. 2020. A Complete Set of Related Git Repositories Identified via Community Detection Approaches Based on Shared Commits. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 513–517. https://doi.org/10.1145/3379597.3387499

[27] MusicIP. 2007. *musicip-libofa* . https://code.google.com/archive/p/musicip-libofa/

[28] National Institute of Standards and Technology. 2021. *National Vulnerability Database*. http://nvd.nist.gov

[29] J. Ossher, H. Sajnani, and C. Lopes. 2011. File cloning in open source Java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 283–292.

[30] OWASP. 2022. *OWASP Dependency-Check*. https://owasp.org/www-project-dependency-check/

[31] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The Software Heritage Graph Dataset: Public Software Development Under One Roof. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 138–142. https://doi.org/10.1109/MSR.2019.00030

[32] Francisca Pérez, Manuel Ballarín, Raúl Lapeña, and Carlos Cetina. 2018. Locating Clone-and-Own Relationships in Model-Based Industrial Families of Software Products to Encourage Reuse. *IEEE Access* 6 (2018), 56815–56827. https://doi.org/10.1109/ACCESS.2018.2873509

[33] Shi Qiu, Daniel M. German, and Katsuro Inoue. 2021. Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem. *Journal of Information Processing* 29 (2021), 296–304. https://doi.org/10.2197/ipsjjip.29.296

[34] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510216

[35] Richard Geldreich. 2020. *richgel999/jpeg-compressor* . https://github.com/richgel999/jpeg-compressor

[36] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. 2020. Software provenance tracking at the scale of public source code. In *Empirical Software Engineering*. https://doi.org/10.1007/s10664-020-09828-5

[37] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 76–85.

[38] Software Heritage. 2022. *Software Heritage*. https://www.softwareheritage.org

[39] Synopsys Technology. 2021. *2021 Open Source Security and Risk Analysis*. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html?intcmp=sig-blog-ossra1

[40] The MITRE Corporation. 2017. *CVE-2017-0700*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0700

[41] The MITRE Corporation. 2021. *Common Vulnerabilities and Exposures (CVE)*. https://cve.mitre.org/

[42] The White House. 2021. *Executive Order 14028 on Improving the Nation's Cybersecurity*. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity

[43] Thomas Wolter, Ann Barcomb, Dirk Riehle, and Nikolay Harutyunyan. 2022. Open Source License Inconsistencies on GitHub. *ACM Trans. Softw. Eng. Methodol.* (dec 2022). https://doi.org/10.1145/3571852

[44] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3041–3058. https://www.usenix.org/conference/usenixsecurity21/presentation/woo

[45] World of Code. 2022. *World of Code*. https://worldofcode.org/

[46] Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. 2015. A Method to Detect License Inconsistencies in Large-Scale Open Source Projects. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 324–333. https://doi.org/10.1109/MSR.2015.37

[47] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. 2014. Studying Reuse of Out-dated Third-party Code in Open Source Projects. *Information and Media Technologies* 9, 2 (2014), 155–161. https://doi.org/10.11185/imt.9.155

[48] Théo Zimmermann. 2020. *A First Look at an Emerging Model of Community Organizations for the Long-Term Maintenance of Ecosystems' Packages*. Association for Computing Machinery, New York, NY, USA, 711–718. https://doi.org/10.1145/3387940.3392209