To the Graduate Council:
I am submitting herewith a dissertation written by Mahmoud Jahanshahi entitled "Copy-Based Reuse and its Implications in Open Source Software Supply Chains." I have examined the final paper copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Audris Mockus, Major Professor

We have read this dissertation
and recommend its acceptance:

Audris Mockus

Jian Huang

Doowon Kim

Russell Zaretzki

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

To the Graduate Council:
I am submitting herewith a dissertation written by Mahmoud Jahanshahi entitled "Copy-Based Reuse and its Implications in Open Source Software Supply Chains." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Audris Mockus, Major Professor

We have read this dissertation
and recommend its acceptance:

Audris Mockus

Jian Huang

Doowon Kim

Russell Zaretzki

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Copy-Based Reuse and its Implications in Open Source Software Supply Chains

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Mahmoud Jahanshahi

05 2025

This dissertation is dedicated

To my beloved mother, Mahnaz, whose unwavering love and support have been my greatest source of strength and inspiration.

To my wonderful brother, Hamed, and my dear sister, Fatemeh, for their constant encouragement and steadfast support.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Audris Mockus, for his unwavering support, exceptional mentorship, and invaluable guidance throughout my research journey. His insights and encouragement have been instrumental in shaping both this dissertation and my growth as a researcher.

I am profoundly grateful to Dr. James Herbsleb and Dr. Bogdan Vasilescu for their collaboration and insightful contributions to this research. Their expertise and guidance have significantly enriched my work and broadened my perspectives.

I also extend my sincere appreciation to my dissertation committee members, Dr. Russell Zaretzki, Dr. Jian Huang, and Dr. Doowon Kim, for their thoughtful feedback and continuous support. Their critical insights and suggestions have greatly enhanced the quality and rigor of this dissertation.

I am deeply thankful to my co-authors, David Reid and Adam McDaniel, for their invaluable collaboration. Their contributions have played a pivotal role in refining my research, and our joint efforts have been instrumental in shaping the findings presented in this dissertation.

Finally, I acknowledge the generous support of the National Science Foundation through awards 1633437, 1901102, 1925615, and 2120429, which made this research possible.

Gegen den Positivismus, welcher bei den Phänomenen stehen bleibt, "es gibt nur Tatsachen", würde ich sagen: nein, gerade Tatsachen gibt es nicht, nur Interpretationen.

Against positivism, which halts at phenomena — "There are only facts" — I would say: No, facts is precisely what there is not, only interpretations.

- Friedrich Nietzsche

The Will to Power, §276 (Kröner ed.), §481 (trans. Kaufmann & Hollingdale)

Abstract

This dissertation investigates copy-based reuse in open source software (OSS) supply chains, emphasizing its identification, analysis, and potential impacts.

First, we develop a novel algorithm to identify copy-based reuse by detecting whole-file copying across the global OSS ecosystem. Leveraging the World of Code infrastructure, we generate a large-scale map of copy-based reuse instances, providing a foundation for future research and tool development to support reuse practices and mitigate associated risks.

Next, we analyze the prevalence, patterns, and motivations behind copy-based reuse. By integrating large-scale reuse detection with developer surveys, we find that copy-based reuse is widespread and varies by programming language, resource type, and project size. Popular projects drive substantial reuse activity, yet more than half of copied resources originate from small and medium-sized projects. Developers cite diverse motivations for copying code, including convenience and trust, while expressing a preference for package managers when feasible.

Our first case study examines the implications of copy-based reuse for OSS license compliance. We construct a copy-based code reuse network and quantify potential license noncompliance across the OSS ecosystem. Our analysis reveals that projects with permissive licenses, such as MIT and Apache, experience higher reuse rates, whereas copyleft licenses, like GPL, yield mixed effects. Alarmingly, 39.4% of reuse instances present a risk of noncompliance, particularly when license information is absent or ambiguous.

The second case study investigates the impact of copy-based reuse on LLM pretraining datasets. We propose an automated source code autocuration technique that utilizes OSS version histories to detect and filter outdated, buggy, and non-compliant code. Evaluating this approach on "The Stack" v2 dataset, we find that 17% of code samples have newer versions, with 17% of these updates addressing bugs, including known vulnerabilities (CVEs). Additionally, we identify serious compliance risks from misidentified blob origins, which introduce non-permissively licensed code into training datasets.

Collectively, this work provides novel insights and practical contributions to understanding and managing copy-based reuse in OSS supply chains. It offers foundational tools and datasets to advance research, informs policy on software licensing practices, and proposes methods to enhance the quality and compliance of AI model training datasets.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reuse refers to the practice of developing software systems from existing software rather than creating them from scratch Krueger (1992). Starting from scratch may demand more time and effort than reusing pre-existing, high-quality code that fits the required task. Developers, therefore, opportunistically and frequently reuse code Juergens et al. (2009). Programming for clearly defined problems often starts with a search in code repositories, typically followed by careful copying and pasting of the relevant code Sim et al. (1998).

The fundamental principle of Open Source Software (OSS) lies in its "openness", which enables anyone to access, inspect, and reuse any artifact of a project. This could significantly enhance the efficiency of the software development process. Platforms such as GitHub increase reuse opportunities by enabling the community of developers to curate software projects and by promoting and improving the process of opportunistic discovery and reuse of artifacts. A significant portion of OSS is intentionally built to be reused, offering resources or functionality to other software projects Haefliger et al. (2008), thus such reuse can be categorized as one of the building blocks of OSS. Indeed, developers in the open source community not only seek opportunities to reuse existing high-quality code, but also actively promote their own well-crafted artifacts for others to utilize Gharehyazie et al. (2017). Being widely

reused not only increases the popularity of the software project and its maintainers while providing them with job prospects Roberts et al. (2006), but also may bring new maintainers as well as corporate support.

Most commonly, code reuse refers to the introduction of explicit dependencies on the functionality provided by ready-made packages, libraries, frameworks, or platforms maintained by other projects (referred to as dependency-based or black-box reuse). Such external code is not modified by the developer and, generally, not committed into the project's repository but relied upon via a package manager. Copy-based reuse (or white-box reuse), on the other hand, refers to the case where source code (or other reusable artifacts) is reused by copying the original code and committing the duplicate code into a new repository. It may remain the same or be modified by the developer after reuse. We specifically focus on copy-based reuse in this study.

While it is generally accepted that programs should be modular Parnas (1972), with internal implementation details not exposed outside the module, copy-based reuse does exactly the opposite. OSS's copy-based reuse, where any source code file or even a code snippet can be reused in another project, may result in multiple, possibly modified instances of the same source code replicated across various files and repositories. These copies may undergo further changes during maintenance, leading to multiple different versions of the originally identical code existing in the latest releases of corresponding projects. Unifying such multiplicity of versions in copy-based reuse to refactor it into a single package that all these projects could depend upon may not always be a tractable problem.

Moreover, as this reuse process continues across various projects, possibly with some modifications, data related to the initial design, authorship, copyright status, and licensing could be lost Qiu et al. (2021). This loss could impede future enhancements and bug-fixing efforts. It might also diminish the motivation for original authors who seek recognition for their work and lead to legal complications for downstream users. These issues impact not only those who reuse the code but

also the software dependent on at least one package that involves reused code Feng et al. (2019).

As the landscape of Open Source Software (OSS) expands, tracing the origins of source code, identifying high-quality code suitable for reuse, and deciphering the simultaneous progression of code across numerous projects become increasingly challenging. This can pose risks, such as the spread of potentially low-quality or vulnerable code (e.g, orphan vulnerabilities Reid et al. (2022)).

1.1 Background

1.1.1 Reuse in Software Supply Chains

A software supply chain comprises various components, libraries, tools, and processes used to develop, build, and publish software artifacts. It covers all stages from initial development to final deployment, including proprietary and open source code, configurations, binaries, plugins, container dependencies, and the infrastructure required to integrate these elements. The software supply chain ensures that the right components are delivered to the right places and at the right times to create functioning software products. Software reuse is one form of the software supply chain that enhances efficiency, reduces costs, and mitigates the risks associated with developing new software from scratch.

In the context of open source software, reuse in software supply chains can be categorized based on how the open source components are integrated and utilized within software projects Mockus (2019a, 2022, 2023).

Dependency-based Reuse

Dependency-based reuse involves using open source libraries and packages as dependencies in a project. These dependencies are typically managed through package managers such as NPM for JavaScript, pip for Python, or Maven for Java. The

reliance on these dependencies can introduce vulnerabilities and risks if not properly managed Yan et al. (2021). A web application using the React library, which in turn depends on numerous other libraries is an example of reuse in this kind of supply chain.

Copy-based Reuse

Copy-based reuse is the type of reuse investigated in this work. In copy-based reuse, code from open source projects is copied directly into a project. For example, a developer might copy a utility function from an open source repository and integrate it into their own project. While this approach is quick, it can lead to challenges in maintaining and updating the copied code. It is essential to track and manage these copies to ensure they are secure and up-to-date Ladisa et al. (2023).

Knowledge-based Reuse

Knowledge-based reuse involves using knowledge and practices derived from open source projects without directly copying code or using dependencies. It includes the adoption of development methodologies, architectural patterns, and best practices from open source communities. For example, implementing a microservices architecture inspired by successful open source projects. While not explicitly detailed by many researchers, the concept of knowledge-based supply chains is inferred from broader discussions of open source influence on software development practices Zhao et al. (2021).

1.1.2 Associated Risks

While reuse can potentially reduce development costs, it is not always beneficial. It could introduce certain risks that might eventually escalate the overall costs of a project. These risks include, but are not limited to, security vulnerabilities,

compliance, and the spread of bugs or low-quality code Jahanshahi and Mockus (2024); German et al. (2009).

Security

The relationship between security and reuse can possess a dual-nature: a system can become more secure by leveraging mature dependencies, but it can also become more vulnerable by creating a larger attack surface through exploitable dependencies Gkortzis et al. (2021).

In the context of copy-based reuse, extensive code copying can lead to the widespread dissemination of potentially vulnerable code. These artifacts may reside not only in inactive projects (that are still publicly available for others to reuse and potentially spread the vulnerability further), but also in highly popular and active projects Reid et al. (2022).

Understanding the copy-based supply chain helps in identifying potential security risks and implementing appropriate safeguards Okafor et al. (2022). Therefore, detecting reused code aids in identifying and consistently patching these vulnerabilities across all affected systems Ladisa et al. (2023).

Compliance

Many open source licenses come with specific requirements that must be met. Unintentional reuse of code that is subject to intellectual property (IP) rights or licensing restrictions can lead to legal complications. Understanding the supply chain and detecting reused artifacts ensures compliance with licensing agreements and protects against IP infringements Liang et al. (2022); Zhao et al. (2021).

As software systems evolve, their licenses evolve as well. This evolution can be driven by various factors such as changes in the legal environment, commercial code being licensed as free and open source, or code that has been reused from other open source systems. The evolution of licensing can impact how a system or its parts can

be subsequently reused Jahanshahi and Mockus (2024). Therefore, monitoring this evolution is important Di Penta et al. (2010). However, keeping track of the vast amount of data across the entire OSS landscape is a challenging task, and as a result, many developers fail to adhere to licensing requirements An et al. (2017); German and Hassan (2009).

For example, investigating a subset of codes reused in the Stack Overflow environment revealed an extensive number of potential license violations An et al. (2017). Even when all license requirements are known, the challenge of combining software components with different and possibly incompatible licenses to create a software application that complies with all licenses, while potentially having its own, persists and is of great importance German and Hassan (2009). When individual files are reused, licensing information may be lost, and the findings of our study might suggest approaches to identify and remediate such problems.

Quality

Ensuring that all components of the supply chain meet quality standards is essential for the reliability and performance of the final product Boughton et al. (2024). Copied code that has not been thoroughly vetted and tested can introduce bugs and defects. By identifying and evaluating such reused code, organizations can ensure that it meets their quality standards Mockus (2019a).

Code reuse is not only assumed to escalate maintenance costs under specific conditions, but it is also seen as prone to defects. This is because inconsistent modifications to duplicated code can result in unpredictable behavior Juergens et al. (2009). Additionally, failure to consistently modify identifiers (such as variables, functions, types, etc.) throughout the reused code can lead to errors that often bypass compile-time checks and transform into hidden bugs that are extremely challenging to detect Li et al. (2006).

Apart from the bugs introduced through code reuse, the source code itself could have inherent bugs or be of low quality. These issues can propagate similarly to how

security vulnerabilities spread. The patterns of reuse identified in this study could potentially suggest strategies to leverage information gathered from multiple projects with reused code, thereby reducing such risks.

1.2 Research Objectives

This dissertation investigates the phenomenon of copy-based reuse in OSS supply chains, addressing the following core research questions:

1. How prevalent is copy-based reuse in OSS projects?
2. Can automated methods be developed to detect and analyze copy-based reuse at scale?
3. What are the motivations and practices of developers who engage in copy-based reuse?
4. How does copy-based reuse impact software licensing and compliance?
5. What are the broader implications of copy-based reuse in machine learning and large-scale software security?

To address these questions, this research introduces a novel method for detecting and analyzing copy-based reuse at scale. By applying this method to real-world OSS projects, this dissertation provides insights into how developers reuse code, the risks associated with such practices, and the potential solutions to improve its reliability.

1.3 Structure of the Dissertation

This dissertation is divided into two main parts: the first focuses on the understanding and analysis of copy-based reuse, while the second explores its applications.

Specifically, in the first part:

- **Chapter 2** details the methodology for constructing a large-scale dataset that identifies copy-based reuse in OSS projects. It describes data collection, preprocessing, and validation techniques.
- **Chapter 3** presents an in-depth analysis of copy-based reuse, examining its prevalence, common practices, and patterns in the OSS ecosystem.
- **Chapter 4** investigates developer perspectives through a survey, exploring their motivations, challenges, and practical considerations of copy-based reuse.

The second part of the dissertation applies the proposed method to practical challenges:

- **Chapter 5** explores how copy-based reuse impacts software licensing and presents an automated method to detect licensing inconsistencies across OSS projects.
- **Chapter 6** analyzes noncompliance issues, examining how copy-based reuse contributes to legal and security risks in OSS supply chains.
- **Chapter 7** extends the analysis to machine learning, investigating how copy-based reuse affects large-scale pretraining datasets and the implications for security and compliance in AI models.

Finally, **Chapter 8** concludes the dissertation by summarizing key findings, discussing limitations, and outlining future research directions.

1.4 Contributions

Understanding and addressing copy-based reuse is critical for ensuring sustainable and legally compliant OSS development. This dissertation provides a systematic framework for analyzing this phenomenon, shedding light on its implications for software supply chains, licensing, and security. The findings have significant relevance

for software developers, legal experts, and AI practitioners, guiding best practices for software reuse in an increasingly interconnected digital world. Specifically, this dissertation makes the following contributions to the field of software engineering and open-source software research:

1. **A Large-Scale Dataset for Copy-Based Reuse Detection:** The first dataset of its kind to systematically track copy-based reuse across OSS projects.
2. **Empirical Analysis of Copy-Based Reuse:** A comprehensive study of the prevalence, patterns, and developer motivations behind this practice.
3. **A Large-Scale Dataset for OSS License Identification:** The first dataset of its kind to systematically find license files with minor variations and map them to projects in which they reside across OSS.
4. **Automated Methods for Licensing and Compliance Detection:** Novel techniques for detecting license noncompliance in OSS.
5. **Security Implications in Machine Learning:** An exploration of how copy-based reuse introduces vulnerabilities in AI training datasets.
6. **Policy and Tooling Recommendations:** Practical suggestions for OSS communities, developers, and policymakers to improve compliance and security in software reuse.

1.5 Dissertation Publications

Each chapter of this dissertation corresponds to a separate published or submitted paper, with the exception of Chapters 3 and 4, which are published as a single paper. Chapter 6 has been submitted for publication but has not yet been accepted. The copyright for these publications is held by the respective publishers, and reproduction in this dissertation is in accordance with their policies.

Below are the details of these publications:

- **Chapter 2:** Jahanshahi, M. & Mockus, A. (2024, April). "Dataset: Copy-based Reuse in Open Source Software." In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)* (pp. 42-47). IEEE.

- **Chapters 3 & 4:** Jahanshahi, M., Reid, D., & Mockus, A. "Beyond Dependencies: The Role of Copy-Based Reuse in Open Source Software Development." Accepted in *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

- **Chapter 5:** Jahanshahi, M., Reid, D., McDaniel, A., & Mockus, A. "OSS License Identification at Scale: A Comprehensive Dataset Using World of Code." Accepted in *IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR 2025)*. IEEE.

- **Chapter 6:** Submitted to a conference, currently under review.

- **Chapter 7:** Jahanshahi, M. & Mockus, A. "Cracks in The Stack: Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets." Accepted in *Second International Workshop on Large Language Models for Code (LLM4Code 2025)*.

Across all these publications, I was responsible for the conceptualization of the research, data collection, analysis, and manuscript writing. My co-authors, David Reid and Adam McDaniel, contributed to data validation in some cases. Dr. Audris Mockus, as my advisor, provided guidance and mentorship throughout the entire research and publication process.

Chapter 2

Detecting Copy-based Reuse

Disclosure Statement

A version of this chapter was originally published as Jahanshahi and Mockus (2024):

Mahmoud Jahanshahi and Audris Mockus. 2024. **Dataset: Copy-based Reuse in Open Source Software**. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24)*. Association for Computing Machinery, New York, NY, USA, 42–47.

Available at: https://doi.org/10.1145/3643991.3644868

2.1 Introduction

This dataset seeks to encourage the studies of copy-based reuse by providing copying activity data that captures whole-file reuse in nearly all OSS. To accomplish that, we develop approaches to detect copy-based reuse by developing an efficient algorithm that exploits World of Code infrastructure: a curated and cross referenced collection

of nearly all open source repositories. We expect this data will enable future research and tool development that support such reuse and minimize associated risks.

A better understanding of code copying practices may suggest future research on approaches or tools that make productivity improvements even greater while, at the same time, helping to minimize inherent risks of copying. Specifically, we aim to provide a copy-based reuse dataset to enable further analysis of aspects concerning the extent and the nature of reuse in OSS and to provide information necessary to investigate approaches that support this common activity, make it more efficient, and safer.

First, we create a measurement framework that tracks all versions of source code (we refer to a single version as a `blob` in keeping with the terminology of the version control system git) across all repositories. The time when each unique `blob` b was first committed to each project P is denoted as $t_b(P)$. The first repository $P_o(b) = \text{ArgMin}_P\, t_b(P)$ is referred to as the the originating repository for b. Next, copy instances are identified via projects pairs: a project with the originating commit and the destination project with one of the subsequent commits producing the same blob $(P_o(b), P_d(b))$.

2.2 Contribution

To the best of our knowledge no curation system exists at the level of a `blob`, nor is there an easy way for anyone to determine the extent of copy-based reuse at that level and the introduced reuse identification methods (such as Kawamitsu et al. (2014a)) find reuse between given input projects and are not easily scalable to find reuse across all OSS repositories. The methods we use to identify reuse could, therefore, provide a basis for tools that expose these hard-to-obtain yet potentially important phenomenon.

Our dataset has two important aspects. *First*, we present the copying activity at the whole open source software ecosystem level. Previous provided datasets

normally focus on a specific programming language (e.g. Java as in Janjic et al. (2013)) and the data used in previous works investigating copying have as well mostly concentrated on a small subset of a specific community (e.g. Java language, Android apps, etc.) Heinemann et al. (2011); Haefliger et al. (2008); Mockus (2007); Hanna et al. (2012); Fischer et al. (2017); Sojer and Henkel (2010) or sampled from a single hosting platform (e.g. GitHub) Gharehyazie et al. (2017, 2019). Even research more comprehensive in programming language coverage Lopes et al. (2017) considered only a subset of programming languages and more importantly, used convenience sampling by excluding less active repositories Hata et al. (2021c,a). Furthermore, almost all research only focus on code reuse whereas our dataset tracks all artifacts whether they are code or other reusable development resources, such as images or documentation.

Second, copy-based reuse has not been as extensively investigated as the dependency-based reuse, e.g., Cox (2019); Frakes and Succi (2001); Ossher et al. (2010). Copy-based reuse is, potentially, no less important, but much less understood form of reuse. In fact, most of the efforts in copy-based reuse domain are focused on clone detection[1] tools and techniques Roy et al. (2009); Ain et al. (2019); Jiang et al. (2007); Hanna et al. (2012); White et al. (2016), not on the properties of files that are being reused or projects that produce or reuse artifacts. Clone detection tools and techniques usually take a snippet of code as input and then try to find similar code snippets in a target directory or an specific domain Inoue et al. (2021); Svajlenko et al. (2013) whereas in our dataset, we are finding all instances of reuse in nearly entirety of OSS.

The description and the curation methods of this dataset has not been published before. Furthermore, although the dataset is now publicly available through WoC[2], to the best of our knowledge, the data has not been used by authors or others in any published paper yet.

[1] identification of, often, relatively small snippets of code within a single or a limited number of projects

[2] It has been made available only recently

2.3 Methodology

We start by briefly outlining World of Code infrastructure we employed to create our dataset and then present the methods used to identify instances of copying.

2.3.1 World of Code Infrastructure

Finding duplicate pieces of code and all revisions of that code across all open source projects is a data and computation intensive task due to the vast number of OSS projects hosted on numerous platforms. Previous research on code reuse has, therefore, typically looked at a relatively small subset of open source software potentially missing the full extent of copying that could only be obtained with a nearly complete collection. World of Code (WoC) Ma et al. (2019, 2021) infrastructure attempts to remedy this by, on a regular basis, discovering publicly available new and updated version control repositories, retrieving complete information (or updates) in them, indexing and cross-referencing retrieved objects, conducting auto-curation involving author aliasing Fry et al. (2020) and repository deforking Mockus et al. (2020), and provides shell, Python and web APIs to support creation of various research workflows. The source code version control systems in WoC are collected from hundreds of forges and, after complete deduplication, takes approximately 300TB of disk space for the most recent snapshot we use for our dataset[3]. The specific objective of WoC is to support research on three kinds of software supply chains Ma (2018): technical dependency (traditional dependency-based package reuse), copy-based reuse, and knowledge flows Zhuge (2002); Ghobadi (2015); Lyulina and Jahanshahi (2021) (developers working on, and learning about, projects and then using that knowledge in their work on other projects).

WoC's operationalization of copy-based supply chains is based on mapping blobs (versions of the source code) to all commits and projects where they have been created. This implies that copy is detected only if the entire file is copied intact without

[3]version V

any modifications. Because of that, our dataset includes only the whole-file copying activity. This also means that different versions of the originally same file will be considered different objects since they are different blobs.

Specifically, WoC uses git object indexing via sha1 signature so that each association has to store only the sha1 of the object (in this case blob), and the actual content of each object is stored exactly once. When objects are extracted from a repository, WoC associates all extracted commits with that repository (the so called c2p map). Since a commit points to a tree and to its parent commit objects, the remaining objects in a repository can be easily derived by traversing versions and trees. WoC also computes the association between commits and blobs created by a commit (new versions of existing files or entirely new files) and makes it available via c2fbb map. The map lists all the instances where a blob corresponding to one of the files in the repository changed or a new file was created. In the former case, the blob corresponding to an earlier version of the file is also provided, making it possible to trace back or forth for earlier or newer versions of a blob.

Commits have attributes, such as time of the commit and author of the commit and these attributes can be accessed via c2dat map in WoC. A few more maps provided by WoC are also used in creating this dataset.

2.3.2 Project Deforking

To understand reuse across the entirety of open source software, it is important to identify distinct software projects. Git commits are based on a Merkle Tree structure, uniquely identifying modified blobs, and therefore, shared commits between repositories typically indicate forked repositories. As a distributed version control system (VCS), Git facilitates cloning (via git clone or the GitHub fork button), resulting in numerous repositories that serve as distributed copies of the same project. While this feature enables distributed collaboration, it also leads to many clones of the original repository Mockus et al. (2020).

To differentiate copy-based reuse from forking, we use project deforking map $p2P$ provided in WoC Mockus et al. (2020). Using community detection algorithms, this map provides a clearer picture of distinct projects by linking forked repositories p to a single deforked project P based on shared commits.

An advantage of this map over using the fork data from platforms like GitHub is that WoC's p2P map is based on shared commits, providing higher recall by not missing forks that did not occur through GitHub's forking option but rather through cloning the repository. Additionally, forks and clones hosted on different platforms cannot be traced easily, but the WoC map is platform-independent and does not have this constraint. Moreover, some forks may diverge significantly from the original repository but are still considered forks by hosting platforms. WoC's deforking algorithms use community detection via shared commits. If forks diverge substantially via maintenance after forking, the community detection algorithm would recognize them as distinct projects, which reduces false positives and increases precision.

Whenever we mention "project" in our paper, we are actually referring to a "deforked project" as defined here. This ensures that our discussions about reuse are based on unique instances of software development projects rather than duplicated efforts through forks.

2.3.3 Identification of reused blobs

Despite the key relationships available in WoC, we have to resolve several critical obstacles. We first need to identify the first time $t_b(P)$ each of the nearly 16B blobs landed in each of the almost 108M projects. We aim to minimize memory use and be able to run computations in parallel. First, we join c2fbb map[4] (that lists for each commit all the blobs it creates) with c2dat map (to obtain the date and time of the commit) and then with the c2P (which itself is the result of joining c2p with p2P maps) map to identify all projects containing that commit. WoC has each of the

[4]see https://github.com/woc-hack/tutorial for more information about WoC map naming convention

three maps split into 128 partitions[5] requiring us to run a sequence of two Unix join commands (first to join c2fbb and c2dat and then the result of that join with the c2P map) on each of the 128 partitions in parallel. The result is a new c2Ptb (commit, project, time, and blob) map stored in 128 partitions $(c^i, P, t, b) : i = 0, \ldots, 127$. To create the timeline for each blob we need to sort all that data by blob, time, and project. The list has hundreds of billions of rows (20B blobs often occurring in multiple commits and commits sometimes residing in multiple projects). We thus needed to break down the problem into smaller pieces to solve within a reasonable time frame. Specifically, we first split each partition (c^i, P, t, b) based on the blob into 128 sub-partitions, thus obtaining 128x128 partitions resulting from the original partitioning by commits and the secondary one by blobs $(b^j, t, P, c^i) : i, j = 0, \ldots, 127$. We then sort each of the 128x128 files by blob, time, and project (using Unix sort parameterized to handle extremely large files) and drop all but the first commit creating the blob for each project[6]. In the next step we merge 128 commit-based partitions for each blob-based partition using Unix sort with a merge option and drop all but the first commit of the blob to a project. Resulting in 128 blob-based partitions (b2tP map) $(b^j, t, P) : j = 0, \ldots, 127$ where we have only blob, time, and the deforked project that contain our desired timeline $t_b(P)$. Finally, the blob timelines are used to identify instances of copying $(t_b(P_o), t_b(P_d))$ (or, in the terminology of WoC, Ptb2Pt maps where the first project is originating[7] and the second project copied the blob – the blob was created at a later time). To accomplish this we first create a list of blob origination projects and times. A sweep over b2tP by keeping only the first time and the project associated with each b and excluding blobs associated with a single project[8] produces $(b^j, t, P_o) : j = 0, \ldots, 127$. We also store never reused

[5]Partitions are enumerated using the first seven bits of the sha1 representing the key — in this case commit — in order to obtain partitions of similar size. Each partition is a file sorted by the key and compressed.

[6]A blob is often copied within a repository.

[7]See section 2.5 for the limitations in identifying the originating project.

[8]Over 90% of the blobs belong to a single project, so excluding them reduces storage of the relations created downstream.

blobs $(b^j_{nc}, t, P_o) : j = 0, \ldots, 127$ (ones that are associated with only one project as identified during the sweep mentioned above). (b^j, t, P_o) partitions containing only originating project are then joined with (b^j, t, P) to obtain the cross-product $((b^j, t_o, P_o, t_d, P_d) : j = 0, \ldots, 127, P_o \neq P_d)$. Each of the resulting 128 partitions are then split via project name[9], into 128 sub-partitions and each sub-partition is then sorted by the originating project: $((P^i_o, t_o, P_d, t_d, b^j) : i, j = 0, \ldots, 127)$, then merging over blob-based partitions belonging to a single project-based partition. Resulting Ptb2Pt map contains all instances of blob copying: $(t_b(P^i_o), t_b(P_d))$ and is stored in 128 partitions $i = 0, \ldots, 127$ with each workflow step described above capable of being run as 128 parallel processes. The data flow digram of reuse identification is shown in Figure 2.1.

The initial step was to pinpoint the first instance, denoted as $t_b(P)$, when each of the approximately 16 billion blobs appeared in each of the almost 108 million projects. To this goal, first the c2fbb map[10] (which is the result of diff on a commit: commit file, blob, old blob and lists all blobs created by each commit) was joined with the c2dat map (full commit data) to obtain the date and time of each commit. The result was then joined with the c2P map (commit to project) to identify all projects containing that commit.

The result is a new c2btP map (commit to blob,, time, and Project). To create the timeline for each blob, all that data was sorted by blob, time, and project resulting in b2tP map (b, t, P) where we have only blob, time, and the deforked project that contain our desired timeline $t_b(P)$.

Finally, the blob timelines[11] were used to identify instances of reuse $(t_b(P_o), t_b(P_d))$ or Ptb2Pt map, where the first project is the originating project[12] and the second

[9] We use the first seven bits of the name's FNV digest Noll (2012) as it is faster and randomizes better short strings than sha1.

[10] See https://github.com/woc-hack/tutorial for more information about WoC map naming convention

[11] All but the first commit time creating the blob for each project were dropped as a blob is often reused within a repository.

[12] See section 2.5 for the limitations in identifying the originating project.

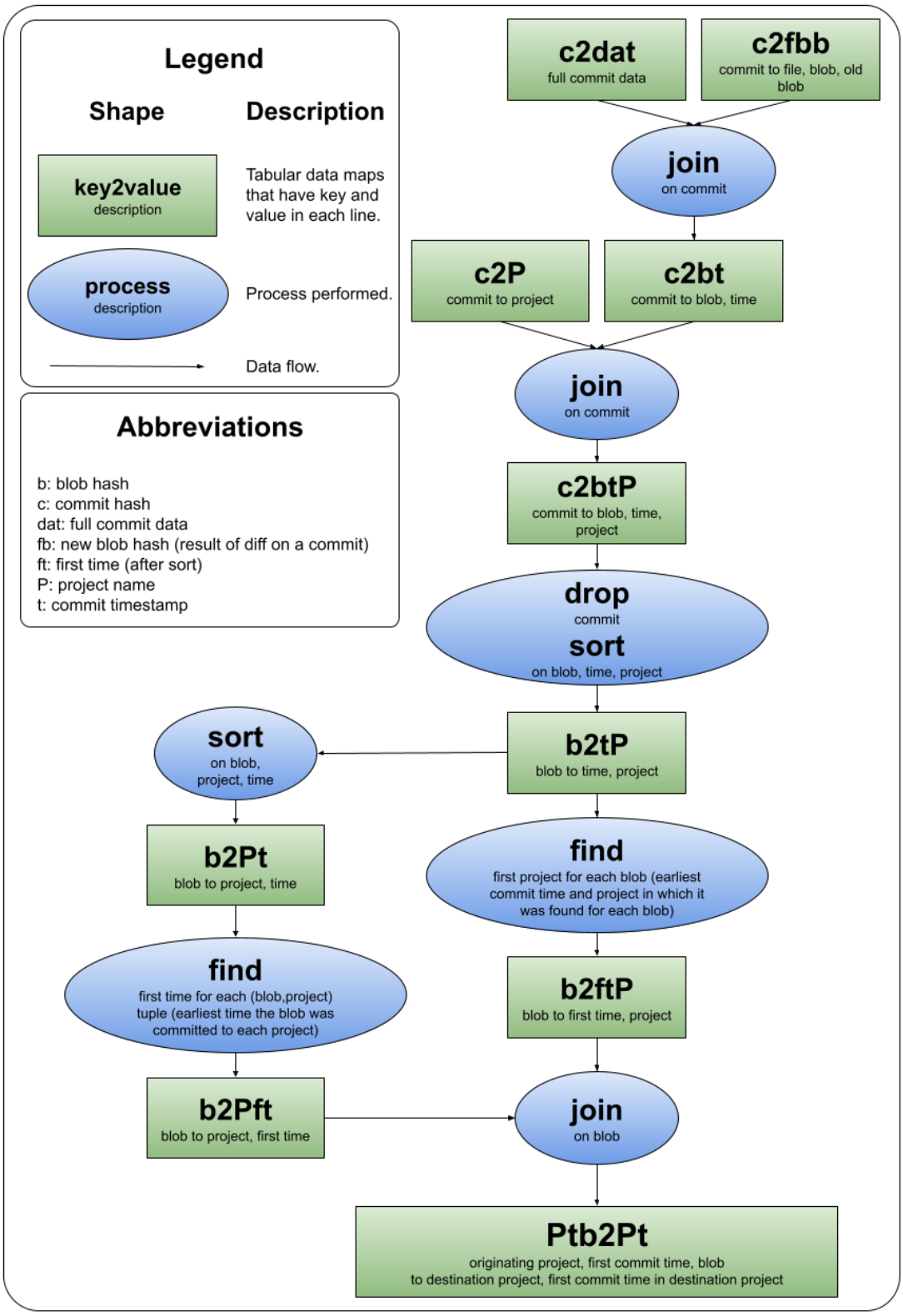


Figure 2.1: Reuse Identification Data Flow Diagram

project is the destination project of the reused blob, meaning the blob was created at a later time in this project. This resulting Ptb2Pt map contains all instances of blob reuse. The data flow of reuse identification is shown in Figure 2.1.

2.3.4 Time Complexity Analysis

To evaluate the complexity and time requirements of our methodology for identifying reuse, we analyze the time complexity of each step and provide a benchmark for execution time on a typical computer setup. The overall time complexity is dominated by the sorting operations involved in processing the large maps. Data preparation and joining involve merging the precalculated maps in WoC, namely the c2fbb, c2P, and c2dat maps. Since these maps are already sorted and split into 128 partitions, we can join them with a complexity of $128 \times O(l + m + n)$, where l, m, and n are the number of rows in the maps respectively. We then drop the commit hashes and sort the joined b2tP map based on blob, time, and project, which is the most computationally intensive step, with a complexity of $O(n \log n)$, where n is the total number of rows in the b2tP map. Identifying reuse instances, given that the data is already sorted by blob, has a complexity of $O(n)$, where n is the total number of copy instances.

Using a high-performance workstation as a benchmark (8-core processor at 3.5 GHz, 128 GB RAM, 2 TB SSD), we calculate the execution time for each step. Data preparation and joining, with a linear-time merge, primarily involve reading and writing large files. With a sequential read/write speed of approximately 500 MB/s for SSDs, joining the maps (total size around 128 billion rows) is expected to take roughly 1-2 hours. Sorting the created b2tP map, which requires external sorting of about 74 billion rows, necessitates multiple passes over the data. Based on empirical data, a modern external sorting algorithm with 8 cores can handle around 0.5 billion rows per hour. Hence, sorting this map would take approximately 148 hours. Identifying reuse instances, involving efficient I/O operations, is estimated to

take 4-6 hours. In total, the entire process is estimated to take approximately 153-156 hours, or about 6.5 days.

Detecting code reuse in finer granularity than blob-level, such as through syntax tree parsing or text similarity techniques, would offer a more comprehensive view of code reuse. However, these methods involve several computational challenges and resource constraints, making them impractical for our study.

Parsing the abstract syntax tree (AST) for each file to detect structural similarities involves several computational steps. First, each file must be parsed into its AST representation, which itself is an $O(n)$ operation where n is the total number of unique blobs. For our dataset of 16 billion blobs, this parsing step alone would be extremely resource-intensive. Following parsing, comparing each AST to identify potential reuse instances would require pairwise comparisons. The pairwise comparison complexity is $O(n^2)$, resulting in an infeasible $O((16 \times 10^9)^2)$ complexity.

Text similarity measures on the other hand, such as Levenshtein distance or cosine similarity, involve comparing each blob's contents with every other blob. These methods typically operate with a complexity of $O(n^2)$ for each pair of files, again resulting in an infeasible $O((16 \times 10^9)^2)$ complexity. Even with optimizations like locality-sensitive hashing or other approximation techniques, the scale of the data renders this approach impractical.

Given the significant computational complexity and resource requirements, detecting code reuse at a finer granularity than blob-level is not feasible for our study. Instead, we have chosen to focus on blob-level reuse detection, which provides a practical and scalable solution. While this approach is limited to detecting exact file copies, it ensures that the analysis remains within the bounds of available computational resources and time constraints, thereby enabling a thorough and efficient examination of code reuse in the OSS landscape.

2.4 Dataset

The created tables are stored on WoC servers. Each line of this dataset includes the originating repository (deforked repository), the timestamp of first commit including the blob in originating project, blob sha1, destination project (deforked repository) and the timestamp of first commit including the blob in destination project, all separated by semicolon.

```
format:
originating repo;timestamp;blob;
destination repo;timestamp
example:
MeigeJia_ECE-364;1514098666;
010000001b502dcb0fc8e89d4f854979c93503f8;
HaoboChen1887_Purdue;1598024605
```

This means blob 010000001b502dcb0fc8e89d4f854979c93503f8 was first seen in MeigeJia_ECE-364 repository at 1466402956 (Jun 20 2016) and was reused by HaoboChen1887_Purdue at 1551632725 (Mar 03 2019). Slash symbols are substituted with underscores in WoC repository naming convention. That is, MeigeJia_ECE-364 means github.com/MeigeJia/ECE-364. Furthermore, the project is hosted on GitHub unless the domain is mentioned at the beginning of the project name.

To get access to WoC infrastructure, the WoC registration form should be filled. This form can be found on WoC tutorial page[13]. There are no requirements for registration and any researcher can fill the form with a ssh key pair[14]. Upon gaining access, the data can be easily found and read at /da?_data/basemaps/gz/Ptb2PtFullVX.s with X ranging from 0 to 127 based on the 7 bits in the first byte of the blob sha1. The "V" in the name indicates that this dataset is based on WoC version V[15](the latest at the time of this work).

[13] https://github.com/woc-hack/tutorial
[14] https://www.ssh.com/academy/ssh/public-key-authentication
[15] https://bitbucket.com/swsc/overview

2.5 Limitations

Blob-level reuse Our dataset is at entire blob reuse granularity and does not capture the reuse of pieces of code that form only a part of the file. Thus blob-level reuse (despite being common) does not represent the full extent of all code reuse.

Notably, different versions of the same file would have different blobs as even if two versions differ by only one character, they still produce different file hashes (are different blobs). Thus blob reuse is not the same as file reuse. File reuse is, however, difficult to define precisely as it is not clear what files should be considered equivalent in distinct projects.

Commit time The reuse timeline (and identifying the first occurrence) of a blob is based on the commit timestamp. This time is not always accurate as it depends on the user's system time. We used suggestions by Flint et al. (2021a) and other methods to eliminate incorrect or questionable timestamps. We also used version history information to ensure time of parent commits do not postdate that of child commits.

Originating repository The accuracy of origination estimates can be increased by the completeness of data. Even if we assume that the WoC collection is complete, some blobs may have been originated in a private repository and then copied to a public repository, i.e., the originating repository in WoC may not be the actual creator of the blob. For example, a 3D cannon pack asset[16] was committed by 38 projects indexed by WoC. That asset, however, was created earlier in Unity Asset Store.

Copy instance A unique combination of blob, originating project and destination project may not always reflect the actual copy pattern because some destination projects may have copied the blob not from the originating project (e.g., for projects O, A, and B in blob creation order, project B may copy either from project O or A).

[16]https://assetstore.unity.com/packages/3d/props/weapons/stylish-cannon-pack-174145

Also, some blobs are not copied but are created independently in each repository, e.g, an empty string, or a standard template automatically created by a common tool. We use the list of such blobs provided by WoC Ma et al. (2019) to exclude them from all our calculations.

As was described in each paragraph, we took all the necessary steps to minimize the potential negative impact of these limitations and validated the curated data extensively to ensure its reliability within the boundaries of limitations.

Chapter 3

Beyond Dependencies

Disclosure Statement

A version of this chapter was originally published as Jahanshahi et al. (2024b):

Mahmoud Jahanshahi, David Reid, and Audris Mockus. 2025. **Beyond Dependencies: The Role of Copy-Based Reuse in Open Source Software Development**. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. Just Accepted (January 2025).

Available at: https://doi.org/10.1145/3715907

Replication package available at: https://zenodo.org/records/14743941

3.1 Introduction

In this chapter, our aim is to enable future research and tool development to increase efficiency and reduce the risks of copy-based reuse. We seek a better understanding of copy-based reuse by measuring its prevalence and identifying factors affecting the propensity to reuse. To identify reused artifacts and trace their origins, our method

exploits World of Code infrastructure. We begin with a set of theory-derived factors related to the propensity to reuse, sample instances of different reuse types, and survey developers to better understand their intentions. Our results indicate that copy-based reuse is common, with many developers being aware of it when writing code. The propensity for a file to be reused varies greatly among languages and between source code and binary files, consistently decreasing over time. Files introduced by popular projects are more likely to be reused, but at least half of reused resources originate from "small" and "medium" projects. Developers had various reasons for reuse but were generally positive about using a package manager.

Despite the sustained attention and potential benefits and risks associated with reuse, the exact scale, prevalent practices, and possible negative impacts related to OSS-wide reuse have not been thoroughly explored. This is primarily due to the formidable task of tracking code throughout the entirety of OSS Jahanshahi and Mockus (2024).

Gaining a more comprehensive understanding of reuse practices could guide future research towards developing methods or tools that enhance productivity while mitigating the inherent risks associated with reuse. Specifically, we aim to quantify several aspects concerning the extent and nature of reuse in OSS, providing information necessary to investigate approaches that support this common activity, making it more efficient and safer.

We use a measurement framework created by Jahanshahi and Mockus (2024) that tracks all versions of project artifacts, referred to as blobs[1], across all repositories.

In this approach, the first time each blob is committed to a repository is identified. The (repository, blob) tuples are then sorted based on the commit time of the first appearance of that unique blob in the repository. The repository with the earliest commit time is identified as the originating repository, and the person who made that commit is recognized as the creator of the blob. Reuse instances are then identified

[1]In alignment with the terminology used in the Git version control system, we use the term "blob" to refer to a single version of a file.

by pairing the originating repository with any subsequent repositories that commit the same blob.

Our work investigates how much and what kind of the whole-file reuse happens at the scale of OSS, with findings that could help guide future research and tool development to support this common but potentially risky activity. First, we show how the existing studies, by ignoring "small" and inactive projects, miss almost half of the code reused even by the "largest" and most active projects. There is a necessity for more in-depth study to fully comprehend how these abundant yet unseen "dark matter" projects contribute to reuse activity. Second, we theorize about and investigate empirically the properties of artifacts and originating projects that influence the likelihood of file reuse, addressing a key question that previous work, which has predominantly focused on copy detection techniques, has missed. To investigate historic reuse trends, we also introduce a time-limited measure of reuse. Our findings reveal several surprising patterns showing how copying varies with the programming language, properties of a blob, and originating projects. These insights could help prioritize and articulate further research and tool development that supports the most common reuse patterns.

In summary we ask the following research questions:

RQ1 How extensive is copying in the entire OSS landscape?

RQ2 Is copy-based reuse limited to a particular group of projects?

RQ3 Do characteristics of the blob affect the probability of reuse?

RQ4 Do characteristics of the originating project affect the probability of reuse?

3.2 Social Contagion Theory

Reusing code is an instance of technology adoption. One of the key questions we want to ask is what may affect the propensity of adopting (copying) a blob. Social

Contagion Theory (SCT) Christakis and Fowler (2013) is a widely used theory for examining dynamic social networks and human behavior in the context of technology adoption Angst et al. (2010); Samadi et al. (2016). In the field of software engineering, it has been used to explain how developers select software packages Ma et al. (2020).

We are using SCT to theorize about the dynamics of code reuse by conceptualizing it in terms of exposure, infectiousness, and susceptibility. SCT helps us frame our research questions by providing a structured way to analyze how code reuse spreads within the open source community. Specifically, we explore how developers become aware of reusable code, the inherent qualities of the code that make it more likely to be reused, and the characteristics of projects or developers that make them more likely to adopt reusable code. These dimensions guide the formation of our research questions, enabling us to systematically investigate the factors influencing reuse activity in open source software. The key value of SCT in our case is to help articulate factors affecting copy propensity via three dimensions:

1. *Exposure.* Exposure is an intuitive notion that in order to copy an artifact, you first have to learn about and find it.

2. *Infectiousness.* Infectiousness is the property of the artifact that affects its propensity to be reused.

3. *Susceptibility.* Susceptibility is the property of the destination project or developer that reflects how much benefit they would (or believe they would) derive by reusing the artifact.

First, for a blob (infectious agent) to be reused, a developer needs to become aware of it. In other words, it needs to be exposed to the open source community (population). Social coding platforms such as GitHub provide various crowd-sourced signals of project popularity. Developers may consider these characteristics of project popularity or health when choosing what resource to use Frakes and Terry (1996); Lozano-Tello and Gómez-Pérez (2002). These considerations suggest that developers

are more likely to be exposed to code in more popular or active projects. Therefore, we used project properties as a proxy for the likelihood of awareness. This primarily addresses RQ2 and RQ4 in this chapter.

The second concept of SCT, infectiousness, means that a highly virulent infectious agent is more likely to spread. In our context, this can be measured by the characteristics of the blob itself, corresponding to RQ3. Most of the literature on reuse has primarily focused on this aspect of the reused resource.

The final concept in our theory is susceptibility, which refers to the vulnerability of the target population to the infectious agent. In our case, this can be approximated by the characteristics of the target project (or author) that reuses the blob. For example, the use value, or how much the blob is needed in the project that copies it. These characteristics are, by definition, highly specific to the target project, making them more challenging to measure. We aim to shed more light on this aspect in chapter 4.

3.3 Related Work and Contributions

While the benefits and risks associated with code reuse seem tangible, the extent and types of reuse across the entirety of OSS remain unclear. To prioritize these risks and benefits, and explore methods to minimize or maximize them respectively, we employ the approach introduced in our previous work Jahanshahi and Mockus (2024). This method allows us to track copy-based reuse on a scale commensurate with the vast size of OSS. The scope of copying activity is not fully encompassed by previous studies based on convenience samples, as we will illustrate in the results section.

We are not aware of any other curation system that operates at the level of a blob or finer granularity, nor is there an easy way to determine the extent of OSS-wide copy-based reuse at that level. Methods for identifying reuse, such as the one introduced by Kawamitsu et al. (2014b), are designed to find reuse between specific input projects and do not easily scale to detect reuse across all OSS repositories Jahanshahi

and Mockus (2024). The methods we use to identify and characterize reuse could, therefore, serve as a foundation for tools that expose this difficult-to-obtain yet potentially important phenomenon Jahanshahi and Mockus (2024). We acknowledge that the actual extent of reuse is most likely much higher than what we find at blob-level granularity. Nevertheless, we believe the results we present will still be insightful, especially as the lower bound for the extent of copy-based reuse activity in the entirety of OSS.

We first differentiate copy-based reuse from related fields and then discuss our contributions.

3.3.1 Related Research Areas

To comprehensively understand copy-based reuse, it is essential to discuss two closely related fields: the clone detection and the clone-and-own practice. Following discussion will focus on differentiating copy-based reuse from dependency-based reuse, clone detection, and clone-and-own practices, situating these within the broader context of code reuse literature.

Code Reuse Analysis

Code Reuse Analysis encompasses techniques and practices that aim to maximize the efficiency and reliability of software development by leveraging existing code. Techniques such as static analysis, dependency analysis, and repository mining help identify reusable components within a codebase Koschke (2007). Through these methods, code reuse analysis seeks to reduce redundancy and enhance maintainability. Frakes and Kang (2005) show that systematic code reuse can significantly reduce development time and costs while improving software quality.

Clone Detection

Clone Detection is a technique within code reuse analysis for identifying similar or identical code fragments in a codebase. This process involves using tools to detect exact or slightly modified duplicates, which can then be refactored into reusable components. Techniques range from textual and token-based methods to more advanced semantic and abstract syntax tree (AST) analyses Roy and Cordy (2007); Svajlenko and Roy (2014). These methods focus on identifying code clones within constrained contexts, often limited to small code snippets within a few projects Svajlenko and Roy (2015). Clone detection helps in managing redundancy and maintaining code quality by highlighting areas where code can be simplified and reused Roy and Cordy (2007). The effectiveness of clone detection tools has been validated in various studies, showing significant improvements in software maintainability Kapser and Godfrey (2008).

Clone and Own

Clone and Own is a practice where existing software components are copied and modified to meet new requirements. This approach is often utilized in product line engineering and situations where rapid development is important. Clone-and-own allows developers to quickly adapt existing solutions but can lead to maintenance challenges due to the proliferation of similar, independently maintained code fragments Krueger (2001); Rubin and Chechik (2013). This practice, common in open source development, involves significant modifications and independent maintenance, often leading to divergent development paths German (2002); Blincoe et al. (2016).

While clone detection focuses on technical identification of code snippets, the clone-and-own practice highlights the importance of customization and independent management of forked projects. As the clone-and-own practice involves both technical customization and significant social factors, such as community engagement and

governance models, understanding these aspects is important for managing forked projects German (2002); Blincoe et al. (2016). Although clone-and-own supports the purpose of code reuse by facilitating quick adaptation, it often results in code duplication, complicating long-term maintenance. Research has shown that clone-and-own is prevalent in practice due to its simplicity and effectiveness in the short term Antoniol et al. (2004).

Copy-based Reuse

Copy-based reuse, a form of code reuse, involves copying existing code and potentially modifying it for use in new contexts. This method allows for rapid development but shares the maintenance challenges associated with clone-and-own, as duplicated code must be managed across different parts of the software. In summary, code reuse analysis encompasses techniques like clone detection to manage redundancy and practices like clone-and-own to adapt existing code for new purposes. While clone detection and code reuse analysis share the goal of improving code quality and maintainability by identifying and managing redundancy, clone-and-own focuses on rapid adaptation rather than efficient redundancy management, despite serving a similar purpose in promoting reuse. Both copy-based reuse and clone detection address code duplication but differ significantly in their methodologies and scopes. Copy-based reuse research, as exemplified by our work, provides a broader, ecosystem-level perspective, incorporating social aspects and the characteristics of entire projects. In contrast, clone detection focuses on the technical identification of code snippets within specific contexts, while the clone-and-own practice emphasizes customization and independent maintenance of forked projects.

3.3.2 Contributions

Our contribution in this work has three aspects as follows.

Accuracy

Our study leverages the World of Code (WoC) infrastructure to analyze reuse of nearly the entire open source software landscape. This allows the capture of the instances of copying that would be missed if only a subset of public repositories were to be analyzed. In contrast, previous studies often focused on samples of mostly "popular" repositories drawn from specific communities or subsets of programming languages. They either have mostly concentrated on a specific community (e.g. Java language, Android apps, etc.) Heinemann et al. (2011); Haefliger et al. (2008); Mockus (2007); Hanna et al. (2012); Fischer et al. (2017); Sojer and Henkel (2010)or only sampled from a single hosting platform (e.g. GitHub) Gharehyazie et al. (2017, 2019). This, consequently, prevented identification of all inter-community or out-of-sample copies.

Even research with more comprehensive programming language coverage such as study by Lopes et al. (2017) or studies by Hata et al. (2021d,b) analyze only a subset of programming languages and additionally use convenience sampling methods by excluding less active or "unimportant" repositories. As our results demonstrate, even inactive and "small" projects appear to provide many of the artifacts reused in OSS, even by the "largest" and most active projects.

Existing literature on code cloning primarily focuses on empirical studies, case studies, and tool evaluations. Empirical studies typically analyze code clones within specific projects or samples of open source software repositories. These datasets are large but not exhaustive of the entire OSS ecosystem. For example, studies by Juergens et al. (2009); Roy et al. (2009) examine hundreds to thousands of files or repositories, providing valuable but partial insights. Case studies offer in-depth analysis of cloning practices within individual projects or organizations, giving detailed context but limiting the scale to the specific cases under study. Tool evaluations involve benchmark studies of clone detection tools, evaluating their performance on curated datasets. While these studies contribute important information about tool effectiveness, they do not cover the entire OSS ecosystem.

Unlike studies that rely on selective sampling, our analysis encompasses nearly the entire open source software ecosystem, providing a broad and necessary foundation for understanding code reuse. This is a fundamental requirement for accurately tracking the origin of files within entire OSS, as it helps to uncover accurate trends and patterns that would be biased in analyses based on the samples of such data, offering a more accurate understanding of reuse practices.

Methodology and Focus

Copy-based reuse has not been explored as thoroughly as the dependency-based reuse (e.g., Cox (2019); Frakes and Succi (2001); Ossher et al. (2010)).

For example, Mili et al. (1995) have shown that dependency-based reuse can lead to more sustainable software architectures by promoting component-based design and reducing redundancy. Additionally, Brown and Wallnau (1998) demonstrated that by leveraging well-defined interfaces and reusable libraries, dependency-based reuse can significantly improve software maintainability and scalability. Nevertheless, very few, if any, similar analyses exist regarding copy-based reuse. Copy-based reuse is potentially no less important, but is a much less understood form of reuse Jahanshahi and Mockus (2024). Most studies in copy-based reuse domain focus on clone detection tools and techniques Roy et al. (2009); Ain et al. (2019); Jiang et al. (2007); Hanna et al. (2012); White et al. (2016) rather than on the characteristics of entire source code files that possibly make reuse more or less likely.

Furthermore, almost all studies we reviewed focus solely on source code reuse, whereas we track all artifacts, whether they are code or other reusable development resources Jahanshahi and Mockus (2024). By using the World of Code research infrastructure, which encompasses nearly the entire OSS ecosystem, we identified and analyzed copying activity at this scale for the very first time.

In contrast to clone detection, which primarily involves identifying similar code snippets within specific directories or domains Inoue et al. (2021); Svajlenko et al. (2013), our research addresses the broader context of entire files and diverse artifacts

across the OSS ecosystem, providing a more comprehensive understanding of reuse. Our method bridges the clone detection and clone-and-own approaches by detecting all instances of reuse, whether they are kept without any changes or modified after reuse, thereby encompassing both the technical and managerial aspects of code reuse.

In existing clone detection literature, several methods are employed to identify code clones. These methods include text-based, token-based, tree-based, and graph-based techniques. Text-based methods detect clones by comparing raw text, which is straightforward but can be less accurate due to variations in formatting. Token-based methods improve on this by converting code into tokens and detecting similarities at this more abstract level, enhancing accuracy but still being susceptible to variations in code structure. Tree-based methods parse the code into abstract syntax trees (ASTs) and identify clones by comparing these trees, providing a more structured and semantically meaningful detection. Graph-based methods further abstract code into control flow or data flow graphs, allowing for the detection of more complex and semantic clones Roy et al. (2009).

The clone and own literature primarily employs these detection methods to understand the broader landscape of code cloning. For example, Juergens et al. (2009) utilized a combination of these techniques to analyze cloning practices in software projects. These methods are effective in identifying different types of clones, such as exact, parameterized, and semantic clones, but they often focus on similarities and patterns rather than exact matches.

In contrast, our research employs a method focused on identifying reuse at the blob-level, specifically detecting if the exact versions of code have been copied. While it misses instances where a single code snippet has been copied, this approach does not rely on abstractions or patterns. This method involves obtaining hashes for all versions of the entire open source software ecosystem to detect identical code segments, ensuring that every version of code is tracked to its origin. This exhaustive and detailed approach allows for a comprehensive analysis of copy-based supply chains at the OSS level. Since software supply chains form a network over the entire OSS, it

is not feasible to study them by sampling projects: representative samples from large graphs are notoriously difficult to obtain (see, e.g., Leskovec and Faloutsos (2006)).

In addition to ensuring that the entire file has been copied and committed, our method easily scales to the entire OSS ecosystem as it avoids the need to look for similarities among tens of billions of versions by utilizing hashes. Traditional clone detection techniques would need to be substantially modified to work at this scale. We discuss some of the potential approaches in Section 8.3.1.

Influencing Factors and Social Aspects

Our study explores how the characteristics of OSS projects influence the propensity for their artifacts to be reused, examining their social aspects. Previously, the focus has been primarily on the desired functionality and the code itself Srinivas et al. (2014); Geisterfer and Ghosh (2006), but we also investigate the social aspects of this phenomenon in the open source community.

The literature on clone detection and our research both explore the social aspects of code reuse, but they do so from different perspectives and with varying emphases on social and technical factors. Existing literature on clone detection primarily focuses on the technical aspects of identifying code clones and understanding their impact on software maintenance and quality. For instance, studies by Juergens et al. (2009); Roy and Cordy (2007) delve into the reasons for code cloning, such as improving productivity, learning, and avoiding reimplementation of similar functionalities. These studies often highlight the technical motivations behind code cloning, such as reusability and rapid prototyping, but they also touch upon social aspects like collaborative development and knowledge sharing within teams. However, the primary emphasis remains on the technical detection and management of code clones.

In contrast, our research takes a broader view by examining how the characteristics of open source software projects influence the propensity for their artifacts to be reused. This includes a detailed analysis of both social and technical factors.

Our study explores the diverse motivations and implications of reuse in the OSS community, considering aspects such as project size, community engagement, and the collaborative nature of OSS development. By doing so, we highlight the importance of social dynamics in code reuse, including factors like community contributions, the reputation of projects, and the collaborative environment that fosters code sharing and reuse.

By examining these social and technical factors, our study provides a more comprehensive understanding of the motivations behind code reuse in the OSS community. We draw parallels to other factors influencing copy-based reuse, such as the ease of access to code, the open and collaborative nature of OSS projects, and the role of community support and documentation. This broader perspective allows us to highlight the diverse and sometimes conflicting motivations for code reuse, ranging from technical efficiency to social recognition and collaborative learning.

3.4 Methodology

To make the subsequent discussion precise, we first introduce a few definitions. The time when each unique blob b was first committed to each project P is denoted as $t_b(P)$. The first repository $P_o(b) = \text{ArgMin}_P\, t_b(P)$ is referred to as the originating repository for b (and the first author as the creator). Then project pairs consisting of a project with the originating commit and the destination project with one of the subsequent commits producing the same blob $(P_o(b), P_d(b))$ are identified as reuse instances. The reuse propensity (the likelihood that a blob will be copied to at least one other project) is then modeled based on the type of the file represented by the blob and the activity and popularity characteristics of the originating projects.

3.4.1 RQ1: How extensive is copying in the entire OSS landscape?

To investigate how widespread whole-file copying in OSS actually is, we first want to establish a baseline: what fraction of blobs were ever reused, and if reused, to how many downstream projects? Specifically, in RQ1, we are showing the number of blobs, originating as well as destination projects (deforked), and copy instances across the entire OSS ecosystem. These numbers are not estimates but the actual numbers calculated over the complete dataset.

3.4.2 RQ2: Is copy-based reuse limited to a particular group of projects?

One may argue that the results in RQ1 are not necessarily important, as only "small" projects may reuse code in a copy-based manner. To see if this is actually the case,

we randomly sampled 5 million reuse instances from each of the 128 files into which the data was divided, based on the first two bytes of the hash of blobs. This resulted in a total of 640 million instances for the analysis. This approach ensured that our sample was distributed across the entire dataset, capturing a diverse range of copy instances. The sample size of 640 million instances constitutes approximately 2.67% of the entire dataset. Although this is a small fraction of the data, it is sufficiently large to ensure the statistical reliability and representativeness of our analysis, as the large absolute size of the sample guarantees its statistical reliability according to the Central Limit Theorem.

Before going further, we need to define the qualitative and, more importantly, subjective terms of "small" and "big" projects with quantitative and justified measures. Crowston and Howison (2005) and Koch and Schneider (2002) have shown that project activity, as measured by commit frequency, is a strong indicator of project health and sustainability. Additionally, the use of stars as a metric is well-supported

in the literature, as they represent a form of user endorsement and are correlated with project visibility and perceived quality Ray et al. (2014). We choose these two metrics because both the number of commits and the number of stars are indicators of a project's activity and popularity. Commits reflect the ongoing development and maintenance efforts, which are important for the sustainability and evolution of a project. Stars, on the other hand, reflect the community's interest and endorsement, indicating the project's visibility and influence. These metrics are widely used in empirical software engineering research to evaluate the health and impact of open source projects Jiang et al. (2007); Borges et al. (2016).

We define projects with over 100 commits and 10 stars as "big" projects. The mean and 3rd quantile values for the number of commits in our dataset are 46 and 12, respectively. This aligns with established practices in the literature where thresholds are often set significantly above average to isolate highly active projects. By setting the threshold at more than double the mean, we ensure that only the top-performing projects are classified as big. Similarly, the threshold of 10 stars is set based on the mean of 2.33 and 3rd quantile value of 0 for stars. This indicates that the majority of projects receive few or no stars, reflecting their popularity and community engagement levels. By selecting projects with at least 10 stars, we focus on those with significant community recognition, capturing less than 1% of the dataset but representing the most influential projects.

The thresholds chosen for "small" group, on the other hand, are projects with no stars and fewer than 10 commits to ensure the projects are indeed small and inactive. This approach ensures that the small group, comprising 62% of projects, includes those with minimal activity and engagement, consistent with findings by Gousios and Spinellis (2012) that a large proportion of open source projects are relatively inactive. We consider all the other projects that do not fall into either the big or small categories as the "medium" group. The medium group captures the middle ground, excluding only the extremes, thus providing a balanced representation of the majority of active projects.

Using this taxonomy, we counted the number of unique blobs involved in these copy instances between groups. It should be mentioned that a blob can have several downstream projects that do not necessarily fall into the same group. Therefore, we considered the biggest downstream project for our analysis purposes. For example, if a blob originated in a medium project and was reused by both a big and a small project, we count it in the "medium to big" category.

Considering the biggest downstream project for each unique blob ensures that the most significant reuse instances are captured. This approach is supported by research indicating that the impact of code reuse is often determined by the size and activity of the downstream projects utilizing the code Mockus (2007); Weiss and Lai (1999). By focusing on the largest downstream project, we ensure that our analysis reflects the most substantial and influential reuse cases of a particular blob.

3.4.3 RQ3: Do characteristics of the blob affect the probability of reuse?

The third part of our research question (RQ1) focuses on the properties of reused artifacts. To address this, we obtained a large random sample of blobs comprising 1/128 of all blobs.

We have to point out that unlike RQ2, where we randomly sampled copy instances (meaning all the blobs involved were reused at least once), here we are sampling from the b2tP map that includes all blobs, whether they have been reused or not. Our dataset is divided into 128 files based on the first two bytes of the blob hash. Hash functions, by design, distribute input data evenly across the output space. The use of hash functions to divide data ensures a uniform distribution across the resultant files Mitzenmacher and Upfal (2017). By using one of these 128 files as our sample, and given the vast size of the dataset, we ensure that it is an unbiased representation of the entire dataset and that this sample size is sufficient to achieve high statistical power and accuracy in our analyses.

We then employed a logistic regression model with the response variable being one for reused blobs and zero for non-reused blobs.

Logistic regression is a robust statistical method used to model the probability of a binary outcome based on one or more predictor variables. It is widely used in empirical software engineering to understand factors influencing software development practices Hosmer Jr et al. (2013). By using logistic regression, we can quantify the effect of various predictors on the likelihood of a blob being reused.

In this research question, we are concerned with infectiousness based on our Social Contagion Theory. Specifically, we are looking for properties of artifacts that affect their propensity to be reused.

The first predictor in our model is the programming language of the blob. Different programming languages are associated with distinct package managers, development environments, and community cultures, which can influence reuse practices Bissyandé et al. (2013). For example, the ease of dependency management in languages like Python (via pip) or JavaScript (via NPM) might facilitate reuse more than in languages with less mature package management systems. Thus, including the programming language as a predictor helps capture these contextual differences. We anticipate that source code for programming languages such as C, which lack package managers, is likely to be copied more frequently than source code for languages with sophisticated package managers, such as JavaScript.

The second predictor is the time of blob creation. This factor helps account for temporal dynamics by indicating the period during which a blob was created, reflecting different reuse practices over time. We hypothesize that older blobs were more likely to be reused due to fewer available reusable artifacts in the OSS landscape at the time. However, the time of creation inherently includes the effect of a blob's availability duration $(t_b(P_d) - t_b(P_o))$, meaning older blobs have had more time to be discovered and reused. Previous research by Weiss and Lai (1999) indicates that the age and visibility of code artifacts influence their reuse.

To isolate and examine the influence of the creation period without the confounding effect of longer availability, we introduce the concept of time-limited reuse. By focusing on copies occurring within specific time intervals after the blob's creation, we remove the advantage of longer visibility and can better assess how the creation period itself influences reuse[2].

We evaluated both one-year and two-year intervals and found similar results. By evaluating both intervals and finding similar results, we enhance the robustness of our conclusions. To maintain conciseness and avoid repetition, we report the findings for the two-year interval. Reporting the two-year interval results provides a balance between sufficient observation time for reuse events and the practical need for concise reporting. Consequently, we excluded blobs created after May 1, 2020, ensuring that all blobs had at least two years to be potentially reused, providing a consistent time frame for analysis Weller and Kinder-Kurlanda (2016). This approach ensures that our findings are not skewed by varying availability periods.

The third predictor is whether the blob is a source code or a binary. We hypothesize that binaries, identified by their git treatment or file extensions like tar, jpeg, or zip, may exhibit different reuse patterns compared to source code. We expect that binary files, such as images, might be copied more often because they are easy to understand and reuse but difficult to recreate. Unlike other types of files, developers cannot easily extract specific parts or functionalities from binary files. That is, source code blobs are directly reusable and modifiable, whereas binaries might be reused as-is without modification. This distinction is important as it affects the ease or necessity of reuse Gabel and Su (2010). Therefore, when it comes to whole-file reuse, which is our definition of reuse in this work, we anticipated that binary blobs are more likely to be copied.

The last factor we hypothesize might affect the propensity of a blob to be reused is its size. The size of a blob can influence its reuse for several reasons. Larger blobs

[2]This definition is used solely for the purposes of our regression model and subsequent analysis. It is not applied in RQ1, RQ2

may contain more functionality, making them more attractive for reuse. Conversely, smaller blobs may be simpler to integrate into existing projects. Previous research by Capiluppi et al. (2003) and Mockus (2007) has indicated that the size of code artifacts can impact their maintainability, comprehensibility, and ultimately their reuse.

To investigate whether a difference exists between the sizes of copied and non-copied blobs, we exclude binary blobs from the analysis. The size of binary blobs is not comparable to the size of source code blobs due to their fundamentally different nature. Binary blobs often include compiled code, media files, or compressed archives, which do not provide a meaningful comparison to plain text source code in terms of size. Because of these differences, we did not incorporate blob size as a predictor in our logistic regression model. Including binary blobs could skew the results and lead to misleading conclusions. Instead, we perform a t-test to compare the sizes of copied blobs and non-copied blobs. The t-test is a robust statistical method used to determine whether there is a significant difference between the means of two groups Student (1908). By applying the t-test, we can rigorously assess whether blob size influences the likelihood of reuse.

3.4.4 RQ4: Do characteristics of the originating project affect the probability of reuse?

The fourth part of RQ1 concerns the chances of finding or being aware of a blob approximated by signals at the project level. This is the exposure factor in the Social Contagion Theory. To conduct this study, we use WoC's MongoDB project database to randomly sample one million projects, comprising nearly 1% of all projects indexed by WoC, to achieve a balance between statistical validity and computational feasibility. A sample size of one million is large enough to provide a representative snapshot of the entire population.

We then search the reuse instances $(t_b(P_o), t_b(P_d))$ in our Ptb2Pt map to determine if the project originated at least one reused blob. A logistic regression model with

the response variable being one if the project has introduced at least one reused blob (and zero otherwise) is then constructed. The predictors in the project-level model include the number of commits, blobs, authors, forks, earliest commit time, the activity duration of the project (the time between the first and the last commit in that project), the binary ratio (the ratio of binary blobs to total blobs), and the programming language. We also use the number of GitHub stars for each project as a predictor. This data in WoC (number of stars) is sourced from GHTorrent Gousios (2013).

The choice of these predictors for our model is based on the current literature on relevant project properties.

- *Number of Commits.* Number of commits is a strong indicator of project activity and maintenance. Koch and Schneider (2002) show that projects with higher commit frequencies tend to have more active development and are more likely to be reused due to their perceived reliability and continuous improvement.

- *Number of Blobs.* Number of blobs represents the volume of content and potential reusable components. Larger projects with more blobs are likely to offer more opportunities for reuse Mockus (2007). It can also indicate the project's complexity and modularity. Projects with more files may be more modular and provide more reusable components.

- *Number of Authors.* Number of authors reflects the collaborative nature of a project. Projects with more contributors tend to have diverse expertise, which supports innovation and decentralized communication, improving the development process Crowston and Howison (2005), and potentially increasing the likelihood of reuse.

- *Number of Forks.* Number of forks is a proxy for the project's popularity and community engagement. Projects with more forks are often viewed as valuable and trustworthy Tsay et al. (2014), increasing their reuse potential.

- *Earliest Commit Time and the Activity Duration.* Earliest commit time and the activity duration provide insights into the project's maturity and stability. Older and long-active projects are more likely to be well-established and reused Gamalielsson and Lundell (2014).

- *GitHub Stars.* GitHub stars is a form of social endorsement, indicating community approval and interest. Projects with more stars are likely to be considered high-quality and reliable, making them more attractive for reuse Borges et al. (2016).

- *Binary Ratio.* Binary ratio, defined as the ratio of binary blobs to total blobs, can impact the reuse potential of a project. Binary blobs, such as compiled code or media files, often indicate pre-packaged functionalities or resources that are ready for use. A higher binary ratio may suggest that a project provides ready-to-use components, which can facilitate reuse Mockus (2007).

Regarding language assignment, at the blob-level, WoC's b2sl map was used for blob language detection based on file extensions. This method is straightforward and effective for identifying the programming languages of individual blobs. Nevertheless, assigning a primary language to a project is more complex due to the use of multiple languages in most projects. WoC's MongoDB project database provides counts of files with each language extension, allowing us to pick the most frequent extension as the project's main language. For our study, we considered only a subset of blobs, specifically originating blobs (blobs first seen in OSS within the project), and assumed the most common language among these blobs as the project's primary language. This approach aligns with the practice of determining the dominant language based on primary contributions Vasilescu et al. (2015).

3.5 Results & Discussions

The numbers presented in this section are derived from version U[3] of WoC, which was the most recent version available at the time of this analysis.

3.5.1 RQ1: How extensive is copying in the entire OSS landscape?

We identified nearly 24 billion copy instances (unique tuples containing the blob and originating and destination projects) encompassing more than 1 billion distinct blobs. With approximately 16 billion blobs in the entire OSS landscape (as approximated by WoC), 6.9% of the blobs have been reused at least once, and each reused blob is copied to an average of 24 other projects (see Table 3.1).

Table 3.1: Basic Statistics of Reuse Instances

	Count	**Total**	**%**
Reuse instances	23,914,332,270	-	-
Blobs	1,084,211,945	15,698,467,337	6.9%
Originating projects	31,706,416	107,936,842	29.4%
Destination projects	86,483,266	107,936,842	80.1%

Nearly 32 million projects (about 30% of the nearly 108 million deforked OSS projects indexed by WoC) originated at least one reused blob. Over 86 million projects have copied these blobs, meaning 80% of OSS projects have reused blobs from another project at least once.

[3] https://bitbucket.com/swsc/overview

RQ1 Key Findings

1. We identified nearly 24 billion copy instances encompassing more than 1 billion distinct blobs.
2. 6.9% of all the blobs in the entire OSS have been reused at least once.
3. About 30% of all OSS projects originated at least one reused blob, and 80% of projects have reused blobs at least once.

The extensive reuse observed highlights the efficiency gains in OSS development, as projects benefit from existing code to accelerate development cycles and reduce costs. The widespread reuse also raises security concerns, as vulnerabilities in copied code can propagate across numerous projects. This necessitates improved vulnerability detection and management practices to ensure the integrity of reused code. Additionally, License violations due to improper code reuse can lead to legal challenges and compliance issues, underscoring the importance of clear licensing and adherence to open source policies. Furthermore, our identification of blob-level reuse, which only accounts for exact matches and not slight modifications, suggests that the actual extent of code reuse might be even higher. The findings advocate for the development of better tools and infrastructure to manage copy-based reuse, including automated detection of security and legal risks, and tools for maintaining code quality in reused components.

3.5.2 RQ2: Is copy-based reuse limited to a particular group of projects?

The numbers already demonstrate the prevalence of copy-based reuse in the OSS community. To understand how this reuse activity is distributed across different groups of projects, we constructed a contingency table as explained in the methods section. Each blob's originating project is unique and falls into one of three categories

(big, medium, and small). However, downstream projects are not unique and we consider the largest downstream project for each blob.

Our analysis revealed nearly 112 million unique blobs reused in our 640 million sample copy instances, with nearly 13 million of these blobs reused by at least one big project (see Table 3.2). This indicates that more than 11% of blobs are reused at least once by at least one big project, showing that copy-based reuse is not limited to small projects but is a widespread phenomenon in the OSS community.

Table 3.2: Blob Counts in Reuse Sample

		Biggest Downstream Projects			**Total**
		Big	Medium	Small	
Upstream Projects	Big	6,748,621	22,273,811	6,515,122	35,537,554 (31.8%)
	Medium	5,348,651	36,434,732	14,552,148	56,335,531 (50.3%)
	Small	691,644	10,151,838	9,231,618	20,075,100 (17.9%)
Total		12,788,916 (11.4%)	68,860,381 (61.5%)	30,298,888 (27.1%)	**111,948,185**

However, it is still unclear if these reused blobs are predominantly introduced by big projects. If this were the case, one could presume that these blobs are mostly of good quality and not error-prone, making the costs of managing and tracking code propagation through such reuse potentially outweigh the benefits. Sampling copy instances revealed that big projects are responsible for only about 30% of reused blobs, while the remaining 70% are introduced by medium and small projects. Specifically, nearly 18% of these blobs are introduced by small projects, with the remaining 50% coming from medium projects. Furthermore, even for big projects, almost 50%[4] of the blobs they reuse originate from medium and small projects (see Table 3.2). Therefore, it is evident that not only big projects serve as upstream sources for copy-based reuse. Indeed, many blobs introduced by medium and small projects are being widely reused.

Even if all widely reused blobs were exclusively introduced by big projects, copy-based reuse still requires management for several reasons. For example, security vulnerabilities may continue to spread even after the main project has fixed the issue Reid et al. (2022).

[4] $(5,348,651 + 691,644)/12,788,916$

RQ2 Key Findings

1. 32% of reused blobs originate from big projects, which comprise 1% of the total projects.
2. 18% of reused blobs originate from small projects, which make up 62% of the total projects.
3. 50% of reused blobs originate from medium projects, which represent 37% of the total projects.
4. Nearly 50% of blobs reused by big projects originate from medium and small projects, highlighting significant cross-category reuse.

Our findings demonstrate that a non-negligible portion of reused code in the OSS community comes from medium and small projects, challenging the assumption that high-quality code predominantly originates from large projects. This implies a diverse quality spectrum in reused code and underscores the importance of ensuring quality and security across all project sizes, as vulnerabilities in smaller projects can propagate widely. Tools that can track the origin and usage of blobs are essential to ensure timely updates and fixes across the OSS ecosystem, mitigating risks associated with vulnerabilities and outdated code. The widespread nature of code reuse across projects of all sizes, emphasizes the need for quality assurance, effective management, and community collaboration to maintain the health and sustainability of the OSS landscape.

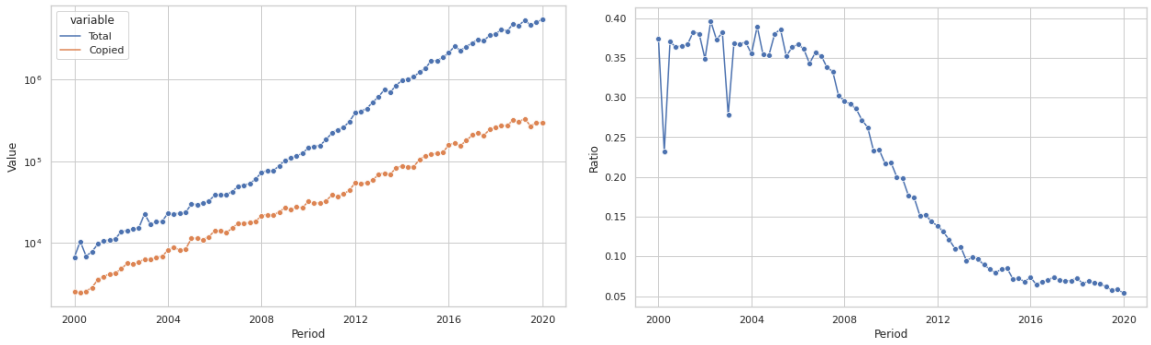
3.5.3 RQ3: Do characteristics of the blob affect the probability of reuse?

In this section, we first demonstrate the reuse trends, followed by the logistic regression model predicting the probability of a blob being reused. Additionally, we present the reuse propensity per language and show the difference in blob size between

reused and non-reused blobs. Finally, we discuss a case study using JavaScript as an example.

Reuse Trends

As explained in the methods section, we use a 2-year-limited copying definition in the RQ3 and RQ4 models and results. This means that we consider a blob reused only if it has been reused within 2 years of its creation. With this definition, 7.5% of blobs have been reused. Figure 3.1a shows the total counts of new blobs and copied blobs for each quarter since the year 2000[5]. Both counts exhibit rapid growth, although the growth in new blob creation appears to outpace that of copying. To investigate this difference, Figure 3.1b shows the reuse propensity measured via the reuse ratio (reused blobs divided by total blobs), confirming that new blob creation has outpaced copied blobs since 2006 when the ratio began to decline.



(a) Generated and Reused Blobs Trends (b) Reused to Generated Blobs Ratio Trend

Figure 3.1: Quarterly Reuse Trends

Logistic Regression Model

We expect the nature of the blob to affect its propensity to be reused. To test this hypothesis, we use a logistic regression model where the response variable is set to one if the blob has been copied at least once (i.e., has been committed in at least two

[5]The number of projects and blobs was much smaller before 2000.

projects) within two years of its creation, and zero otherwise. We used WoC definition of the programming language associated with each blob and categorized less common programming languages in the sample as "other". The descriptive statistics of the variables are presented in Table 3.3.

Table 3.3: Blob-level Model - Descriptive Statistics

Variable	**Statistics**			
Reused	Yes: 6,419,388 (7.5%)		No: 78,136,705 (92.5%)	
Language	JavaScript	Java	C	(Other)
(Counts)	11,122,849	4,579,458	3,460,733	65,393,053
Creation Time	5%	Median	Mean	95%
(Date)	7/29/2012	2/7/2018	5/28/2017	2/28/2020
Binary	Yes: 18,516,721 (21.8%)		No: 66,039,372 (78.2%)	

The sample dataset is predominantly composed of blobs written in JavaScript, with significant counts also in Java and C. Additionally, the distribution of blob creation time is provided, showing a median date of February 7, 2018. Furthermore, a notable proportion of the blobs, 21.8%, are binary.

The results of our logistic regression model are shown in Tables 3.4 and 3.5. The model shows that the coefficients for all predictors are statistically significant with p-values less than 0.0001, meaning they impact the probability of a blob being reused (see Table 3.4).

The ANOVA table (Table 3.5) provides insights into the significance of different variables. We see that all the predictors have p-value equal to zero, meaning that the null hypothesis[6] can be rejected. The null deviance is 45,438,151, which represents the deviance of a model with only the intercept. Adding the Binary variable reduces the deviance by 124,114, indicating its strong influence on reuse likelihood. The Creation Time variable further reduces the deviance by 830,322, highlighting its importance

[6]H0: The reduced model (without the predictor) provides a fit to the data that is not significantly worse than the full model (with the predictor). This suggests that the predictor does not significantly improve the model's fit.

Table 3.4: Blob-level Model - Coefficients

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-18.0293	0.0186	-967.07	$< 2 \times 10^{-16}$
Binary	0.4775	0.0010	460.16	$< 2 \times 10^{-16}$
Creation Time	0.8108	0.0010	828.34	$< 2 \times 10^{-16}$
C	0.7142	0.0017	426.32	$< 2 \times 10^{-16}$
C#	-0.1277	0.0033	-38.15	$< 2 \times 10^{-16}$
Go	0.3095	0.0065	47.74	$< 2 \times 10^{-16}$
JavaScript	-0.0832	0.0015	-56.21	$< 2 \times 10^{-16}$
Kotlin	-0.5606	0.0133	-42.02	$< 2 \times 10^{-16}$
ObjectiveC	0.0810	0.0066	12.30	$< 2 \times 10^{-16}$
Python	-0.0327	0.0030	-10.97	$< 2 \times 10^{-16}$
R	0.4070	0.0083	49.22	$< 2 \times 10^{-16}$
Rust	0.0879	0.0095	9.30	$< 2 \times 10^{-16}$
Scala	-0.6168	0.0123	-50.21	$< 2 \times 10^{-16}$
TypeScript	0.1827	0.0046	39.38	$< 2 \times 10^{-16}$
Java	0.0794	0.0019	42.37	$< 2 \times 10^{-16}$
PHP	0.3561	0.0024	151.14	$< 2 \times 10^{-16}$
Perl	0.7664	0.0082	92.95	$< 2 \times 10^{-16}$
Ruby	-0.4782	0.0044	-108.58	$< 2 \times 10^{-16}$

in predicting reuse. The "Language" variable also reduces the deviance by 230,614. Although these reductions might seem small relative to the null deviance, they are statistically significant given the large sample size and the high degrees of freedom involved.

Table 3.5: Blob-level Model - ANOVA Table

	Df	**Deviance**	**Resid. Df**	**Resid. Dev**	**p.value**
NULL			84,556,092	45,438,151.00	
Binary	1	124,114.20	84,556,091	45,314,036.80	$< 2 \times 10^{-16}$
Creation Time	1	830,322.63	84,556,090	44,483,714.17	$< 2 \times 10^{-16}$
Language	15	230,614.17	84,556,075	44,253,100.00	$< 2 \times 10^{-16}$

To assess the direction and the size of predictor effects, we need to go further. In a logistic regression model, a positive coefficient estimate indicates that as the predictor variable increases, the odds of the outcome occurring increase, while a negative coefficient estimate indicates that as the predictor variable increases, the

odds of the outcome occurring decrease. Since the coefficients represent the change in the log-odds of the outcome for a one-unit increase in the predictor, we transform these coefficients to odds ratios by exponentiating them to interpret the actual impact of each predictor. The odds ratio indicates how the odds of the outcome change with a one-unit increase in the predictor. The results are shown in Figure 3.2. This graph displays the odds ratios for various predictors in the logistic regression model at the blob level. An odds ratio greater than 1 indicates an increase in the likelihood of reuse, while an odds ratio less than 1 indicates a decrease.

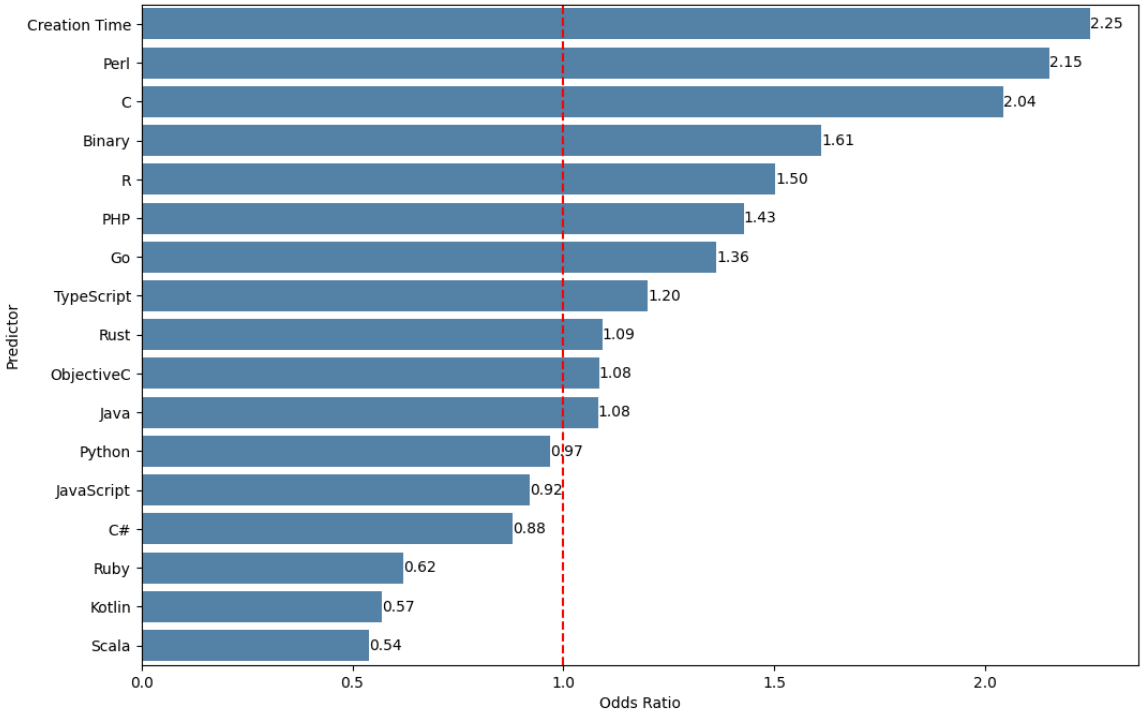


Figure 3.2: Blob-level Model - Logistic Regression Odds Ratios

The creation time has the highest positive coefficient. The time variable in the model represents the time elapsed from the blob's creation until current time, meaning that older blobs have higher time values. The positive coefficient indicates that newer blobs (with smaller time values) are less likely to be reused.

This is not because they have been visible for a shorter duration (as we controlled for this with the time-bound definition of reuse), but likely due to other factors we

hypothesized, such as fewer artifacts being available for reuse at the time of their creation.

Binary blobs show a significant increase in reuse likelihood with an odds ratio of 1.63. Given this confirmed effect, we calculated the reuse propensity for binary and non-binary blobs separately. The results showed that 9.5% of binary blobs were reused, compared to 7.0% of non-binary blobs in our sample.

Different programming languages show varied impacts on reuse likelihood. Blobs written in Perl, C, R, PHP, Go, TypeScript, Objective-C, Java, and Rust are more likely to be reused, with Perl showing the highest odds ratio. In contrast, blobs written in Kotlin, Scala, Ruby, C#, JavaScript, and Python are less likely to be reused, with Kotlin and Scala showing the most significant negative coefficients. This variability suggests that certain languages, perhaps due to their prevalence or specific use cases, are more conducive to code reuse.

Per-Language Propensity

Following our logistic regression results, which demonstrated that programming language is a statistically significant factor in reuse probability of a blob, we calculated the propensity to copy for each programming language, measured as the percentage of reused blobs within that language (see Table 3.6). The results show that blobs written in Perl have the highest propensity to be reused at 18.5%, indicating a strong tendency for code reuse among Perl developers. Conversely, Kotlin has the lowest propensity at 3.0%, suggesting minimal code reuse in this language. Languages such as C (15.2%) and PHP (9.9%) also show high reuse rates, while Python (6.4%), JavaScript (5.5%), and TypeScript (6.3%) have lower rates. Other languages like Java (7.8%), Go (7.9%), and R (9.8%) fall in the middle range, with moderate reuse rates.

Table 3.6: Blob-level - Propensity to Reuse

Language	Ratio	Language	Ratio	Language	Ratio
C	15.2%	ObjectiveC	8.4%	TypeScript	6.3%
C#	6.0%	Python	6.4%	Java	7.8%
Go	7.9%	R	9.8%	PHP	9.9%
JavaScript	5.5%	Rust	6.7%	Perl	18.5%
Kotlin	3.0%	Scala	3.8%	Ruby	5.1%

JavaScript Example

The role of programming language in reuse activity might have several underlying reasons, as previously discussed. One such reason is the presence of a reliable package manager. If true, improvements in a package manager should reduce the propensity to reuse an artifact. To examine this, we analyzed the timeline of the reuse ratio for JavaScript, shown in Figure 3.3. The figure indicates a sharper decrease in the slope around 2010, the year the NPM package manager was introduced. This downward trend continues until mid-2013, when the copying activity rate drops to around 7% and then levels off. This pattern supports the hypothesis that the introduction and adoption of NPM significantly reduced code reuse through copying.

However, it is important to note that this is just an illustration, and further research is needed to understand this phenomenon fully. Our current study was not focused on this aspect, so we did not conduct an in-depth analysis. Additional investigations with more data points and comparisons with other languages that have introduced similar improvements in their package management systems are necessary to confirm that the observed effect is not coincidental or specific to JavaScript alone.

Blob Size

The final predictor we hypothesized to affect the reuse probability of a blob was its size. To investigate whether there is a significant difference between the sizes of copied and non-copied blobs, we conducted a t-test comparing these sizes. Our analysis

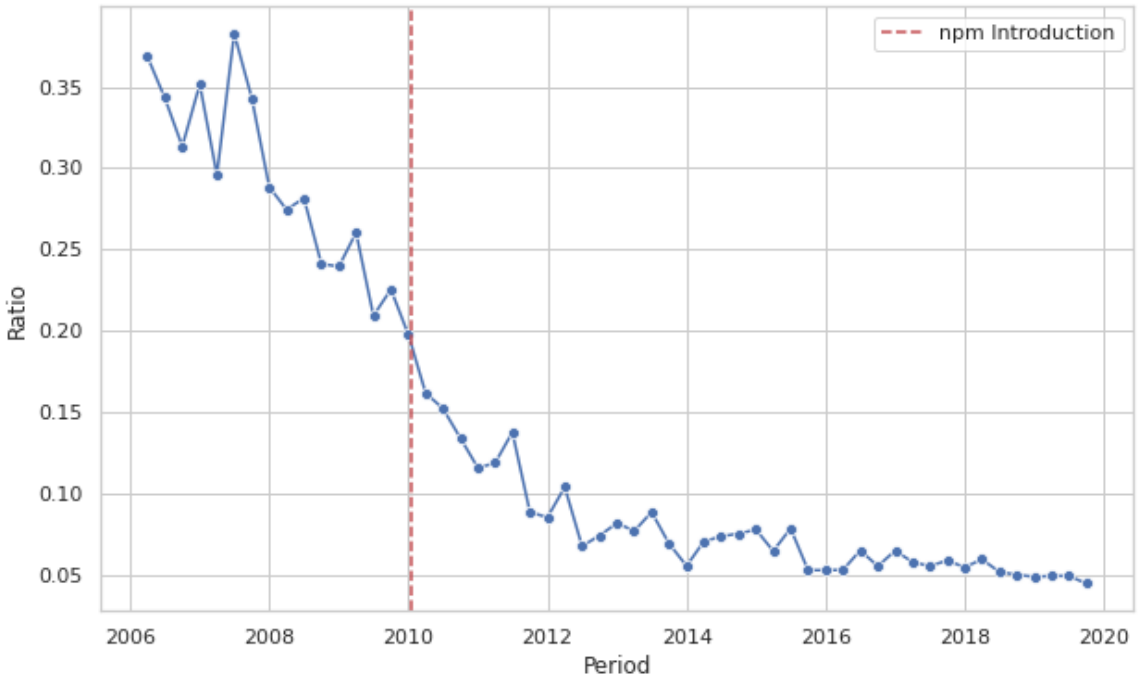


Figure 3.3: Reused Blobs to Total Generated Blobs Ratio Trend in JavaScript

revealed a significant difference (p-value ¡ 2.2e-16), indicating that, on average, copied blobs are smaller than non-copied blobs.

However, the effect varies by language. Specifically, per-language t-tests reveal that copied blobs are smaller in languages like JavaScript and TypeScript, larger in languages such as C and Python, and remain unchanged in Objective-C, as detailed in Table 3.7.

For example, in JavaScript, the t-value is -59.9, suggesting that copied blobs are significantly smaller, while in C, the t-value is 195.9, indicating that copied blobs are larger. Similar patterns are observed in other languages, with TypeScript showing a t-value of -35.9 (smaller copied blobs) and Python a t-value of -5.8 (also smaller copied blobs). Conversely, languages like Java (t-value 120.7) and PHP (t-value 28.6) show that copied blobs tend to be larger.

This variation highlights that the relationship between blob size and reuse propensity is complex and influenced by language-specific factors. While our findings

Table 3.7: Size Difference between Reused and non-Reused Blobs
(Positive t value means larger reused blobs.)

Language	**t value**	**p-value**	**Language**	**t value**	**p-value**
C	195.9	$< 2 \times 10^{-16}$	Rust	-7.8	$< 2 \times 10^{-16}$
C#	12.5	$< 2 \times 10^{-16}$	Scala	9.1	$< 2 \times 10^{-16}$
Go	15.5	$< 2 \times 10^{-16}$	TypeScript	-35.9	$< 2 \times 10^{-16}$
JavaScript	-59.9	$< 2 \times 10^{-16}$	Java	120.7	$< 2 \times 10^{-16}$
Kotlin	-14.5	$< 2 \times 10^{-16}$	PHP	28.6	$< 2 \times 10^{-16}$
ObjectiveC	0.7	0.430298	Perl	5.8	$< 2 \times 10^{-16}$
Python	-5.8	$< 2 \times 10^{-16}$	Ruby	-24.9	$< 2 \times 10^{-16}$
R	-7.6	$< 2 \times 10^{-16}$	Other	-364.9	$< 2 \times 10^{-16}$

demonstrate a general trend of smaller copied blobs, the differing patterns across languages suggest that other underlying factors may be at play.

RQ3 Key Findings

1. The reuse ratio is decreasing over time.
2. 7.5% of blobs have been reused within two years of creation.
3. Older blobs, when controlling for the confounding effect of increased visibility, are more likely to be reused.
4. Binary blobs are 63% more likely to be reused.
5. Programming languages significantly impact reuse likelihood. Blobs written in languages like Perl, C, R, PHP, Go, TypeScript, Objective-C, Java, and Rust are more likely to be reused, while those written in Kotlin, Scala, Ruby, C#, JavaScript, and Python are less likely to be reused.
6. The reuse ratio timeline for JavaScript shows a notable decrease in slope around the year the NPM package manager was introduced.
7. Copied blobs are generally smaller than non-copied blobs, but this is not consistent across different languages. The size difference varies by language, with reused blobs in C, Java, PHP, Go, C#, Scala, Perl, and Objective-C being larger than non-reused blobs, while in JavaScript, TypeScript, Ruby, Kotlin, Rust, R, and Python, the reused blobs are smaller than non-reused blobs.

The higher reuse propensity among binary blobs suggests that binaries are inherently more reusable, likely due to their compiled nature, which allows easy integration across projects. The lower reuse likelihood of newer blobs indicates a potential issue with the integration and acceptance of recent contributions, possibly due to rapid technological advancements and shifts in development practices. The significant impact of programming languages on reuse likelihood highlights the

importance of language-specific tools and ecosystems. Languages with higher reuse rates, such as Perl and C, benefit from mature ecosystems, while newer or niche languages like Kotlin and Scala show lower reuse rates, potentially due to smaller communities. The decline in JavaScript code reuse post-NPM introduction suggests that improved package management can reduce the need for direct code copying, promoting more modular and maintainable codebases.

Regarding blob size, the general trend indicates that smaller code artifacts are more reusable, likely due to their simplicity and ease of integration. However, this trend varies significantly across different programming languages. For example, in languages like JavaScript and TypeScript, copied blobs tend to be smaller, supporting the idea of writing concise and modular code to enhance reusability. In contrast, in languages like C and Python, copied blobs are often larger, suggesting that the nature and use cases of these languages might necessitate larger reusable components. This variation underscores the importance of understanding language-specific factors when considering code reuse management strategies.

3.5.4 RQ4: Do characteristics of the originating project affect the probability of reuse?

In this section, we first present the logistic regression model. We then demonstrate the per-language reuse propensity and compare it to blob-level results. Finally, we analyze binary blob reuse.

Logistic Regression Model

We applied a logistic regression model to determine the likelihood of a project introducing at least one reused blob. The response variable is binary: 1 if the project has introduced a reused blob, 0 otherwise. Descriptive statistics for the model variables are presented in Table 3.8. Consistent with blob-level data, the most frequent languages in our sample are JavaScript and Java.

Table 3.8: Project-level Model - Descriptive Statistics

Variable	Description	Statistics			
Reused	Project has at least 1 reused blob	Yes: 205,140 (33.7%)		No: 403,195 (66.3%)	
		5%	Median	Mean	95%
Blobs	Number of generated blobs	1	15	162.7	397
Binary	Binary blobs to total blobs ratio	0	0	0.1	0.6
Commits	Number of commit	1	5	57.0	84
Authors	Number of authors	1	1	2.5	3
Forks	Number of forks	0	0	1.5	1
Stars	Number of GitHub stars	0	0	3.4	2
Time	Earliest commit time	7/18/2013	3/26/2018	9/15/2017	3/3/2020
Activity	Total months project was active	1	1	2.5	8
Language	JavaScript	Java	Python	PHP	C	(Other)
(Counts)	86,065	43,172	40,503	24,659	22,258	391,678

Spearman's correlation analysis, suitable for the observed heavily skewed distributions, is presented in Table 3.9. The number of commits shows a high correlation with two other predictors: activity time (0.68) and the number of blobs (0.67). These high correlations indicate redundancy, as the number of commits does not add significant information beyond what is already captured by activity time and the number of blobs. This redundancy can lead to multicollinearity, potentially distorting the model's coefficients and reducing interpretability. Consequently, we remove the number of commits from the model, simplifying it without sacrificing explanatory power. All other correlations are below 0.52, which are not concerning.

Table 3.9: Project-level Model - Spearman's Correlations Between Predictors

	Blobs	**Binary**	**Commits**	**Authors**	**Forks**	**Stars**	**Time**	**Activity**
Blobs	1.00	0.46	0.67	0.34	0.22	0.22	0.09	0.52
Binary	-	1.00	0.18	0.12	0.06	0.05	0.02	0.14
Commits	-	-	1.00	0.45	0.27	0.26	0.05	0.68
Authors	-	-	-	1.00	0.32	0.22	0.05	0.38
Forks	-	-	-	-	1.00	0.48	0.14	0.28
Stars	-	-	-	-	-	1.00	0.13	0.28
Time	-	-	-	-	-	-	1.00	0.05
Activity	-	-	-	-	-	-	-	1.00

The results for the project-level logistic regression model are shown in Tables 3.10 and 3.11.

All the variables in the model have p-values less than 0.05, indicating that they are statistically significant in predicting the likelihood of a project introducing reused blobs (see Table 3.10). This demonstrates strong evidence against the null hypothesis, suggesting that these variables do have an effect on reuse.

Table 3.10: Project-level Model - Coefficients

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-4.79	0.16	-30.01	$< 2 \times 10^{-16}$
Blobs	0.61	0.00	228.94	$< 2 \times 10^{-16}$
Binary	0.77	0.02	40.09	$< 2 \times 10^{-16}$
Authors	0.09	0.01	8.24	$< 2 \times 10^{-16}$
Forks	0.31	0.01	27.72	$< 2 \times 10^{-16}$
Stars	0.06	0.01	7.19	6.61×10^{-13}
Time	0.10	0.01	12.00	$< 2 \times 10^{-16}$
Activity	0.07	0.01	10.48	$< 2 \times 10^{-16}$
C	-0.33	0.02	-19.60	$< 2 \times 10^{-16}$
C#	-0.30	0.02	-15.74	$< 2 \times 10^{-16}$
Go	-0.29	0.04	-7.70	1.33×10^{-14}
JavaScript	0.21	0.01	22.58	$< 2 \times 10^{-16}$
Kotlin	-0.23	0.05	-4.30	1.75×10^{-5}
ObjectiveC	-0.13	0.03	-3.63	0.000288
Python	-0.19	0.01	-14.78	$< 2 \times 10^{-16}$
R	-0.27	0.05	-5.93	3.04×10^{-9}
Rust	-0.48	0.07	-6.65	2.87×10^{-11}
Scala	-0.27	0.07	-3.79	0.000153
TypeScript	0.88	0.03	34.57	$< 2 \times 10^{-16}$
Java	-0.25	0.01	-20.90	$< 2 \times 10^{-16}$
PHP	0.29	0.01	19.59	$< 2 \times 10^{-16}$
Perl	-0.31	0.10	-3.20	0.001395
Ruby	0.63	0.02	33.18	$< 2 \times 10^{-16}$

Examining the ANOVA results (Table 3.11) provides further insight into the impact and significance of these predictors.

We see that all the predictors have p-value equal to zero, meaning that the null hypothesis can be rejected.

The deviance values in the ANOVA table indicate the reduction in model deviance when each predictor is included. For example, adding the number of blobs to the model reduces the deviance by 131,219.53, a substantial reduction that underscores its

important role in the model. These results confirm the importance of these predictors in explaining the variability in the likelihood of reuse.

Table 3.11: Project-level Model - ANOVA Table

	Df	**Deviance**	**Resid. Df**	**Resid. Dev**	**p.value**
NULL			608,334	777,660.48	
Blobs	1	131,219.53	608,333	646,440.95	$< 2 \times 10^{-16}$
Binary	1	662.94	608,332	645,778.01	$< 2 \times 10^{-16}$
Authors	1	926.69	608,331	644,851.32	$< 2 \times 10^{-16}$
Forks	1	2,084.02	608,330	642,767.30	$< 2 \times 10^{-16}$
Stars	1	63.77	608,329	642,703.53	1.44×10^{-15}
Time	1	156.98	608,328	642,546.54	$< 2 \times 10^{-16}$
Activity	1	139.31	608,327	642,407.24	$< 2 \times 10^{-16}$
Language	15	5,178.20	608,312	637,229.03	$< 2 \times 10^{-16}$

To understand the size and direction of the impacts, we look at the odds ratios inferred from the logistic regression coefficients. The odds ratio is calculated as the exponential of the coefficient. An odds ratio greater than 1 indicates a positive impact, while an odds ratio less than 1 indicates a negative impact. The results are shown in Figure 3.4.

The logistic regression analysis shows that several predictors significantly impact the likelihood of a project having a reused blob. TypeScript, Binary, Ruby, and Blobs have the strongest positive effects, indicating that increases in these variables substantially raise the odds of a project being reused. Other positive predictors include Forks, PHP, JavaScript, Time, Authors, Activity, and Stars, which also increase the likelihood, though to a lesser extent. Conversely, predictors like Rust, C, Perl, C#, Go, Scala, R, Java, Kotlin, Python, and Objective-C negatively impact the odds, suggesting that increases in these variables decrease the likelihood of a project introducing a reused blob.

When interpreting the time variable, it is important to note that since the earliest commit timestamp is represented as a number, we calculated the time elapsed from the earliest commit to the current date for better interpretability. A larger time value indicates an older earliest commit. The model shows that time has a positive

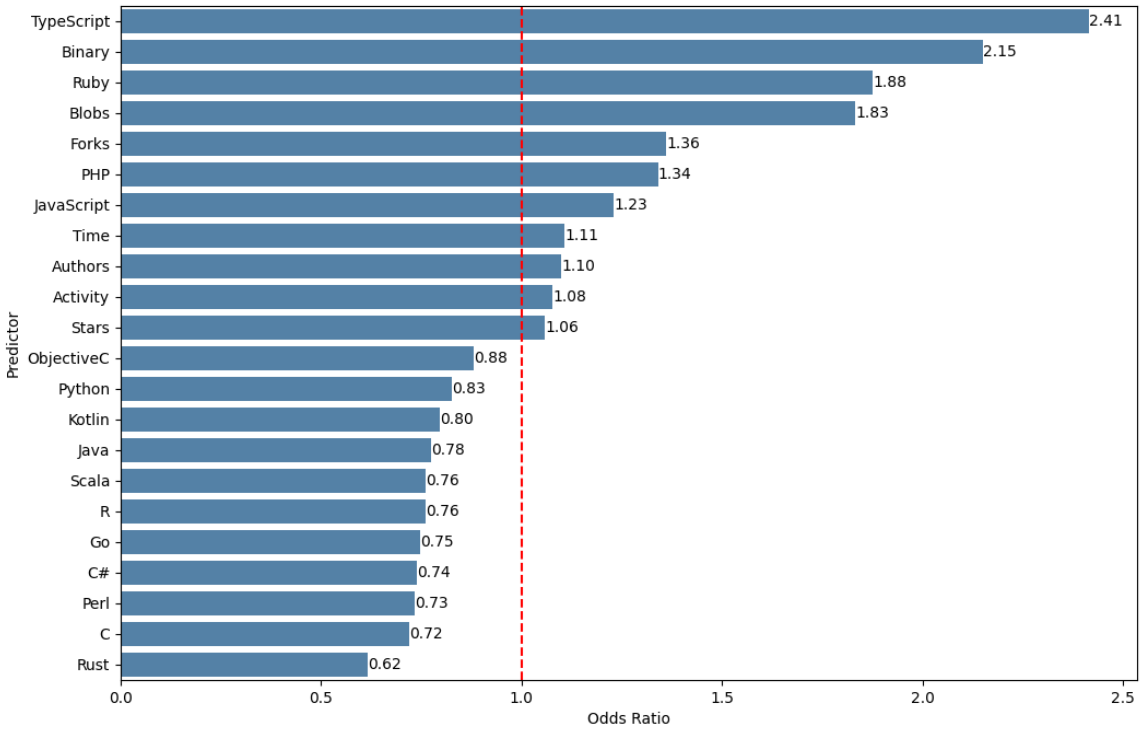


Figure 3.4: Project-level Model - Logistic Regression Odds Ratios

coefficient, suggesting that the older the earliest commit, the higher the probability of introducing reused blobs. This result could be influenced by two factors. First, at the blob-level model, we already observed that older blobs have a higher probability of being reused. Additionally, while the time-bound definition of reuse controls for the confounding effect of longer visibility at the blob level, it does not account for the longer visibility of the project itself. Therefore, the observed result might also be affected by the project's age, which implies longer visibility, even though the blob is reused within two years of its creation.

Per-Language Propensity

The project-level model highlights the significance of programming languages in the likelihood of a project introducing a reused blob. To explore this further, we calculated the percentage of projects in each language that have introduced reused blobs. From our previous analysis (RQ1), we know that approximately 29% of projects introduced

at least one reused blob. When using the time-bound definition of copying, this ratio increased to 33% in our sample. The results for each language are shown in Table 3.12.

Table 3.12: Percentage of Projects Introducing at Least One Reused Blob

Languages	**Ratio**	**Language**	**Ratio**	**Language**	**Ratio**
C	33.2%	ObjectiveC	40.0%	TypeScript	62.3%
C#	37.0%	Python	30.5%	Java	36.2%
Go	31.3%	R	28.5%	PHP	46.4%
JavaScript	41.2%	Rust	31.5%	Perl	29.9%
Kotlin	40.0%	Scala	36.0%	Ruby	51.2%

The ratio of projects that have introduced reused blobs varies significantly across different programming languages, offering new insights compared to the blob-level analysis. For example, projects dominated by TypeScript have the highest probability (62%) of introducing at least one reused blob. This finding is particularly interesting because, at the blob level, the propensity to copy in TypeScript was lower than average. This discrepancy suggests that TypeScript projects, acting as upstream in the language's supply chain, are less centralized. Developers in this language seem more inclined to incorporate code from various, possibly unknown, projects.

Other languages also show distinct patterns. For instance, Ruby projects have a high probability (51%) of reusing blobs, whereas Python projects have a lower probability (30.5%). This variation indicates that the likelihood of code reuse is strongly influenced by the primary language of the project, reflecting different practices and community norms across languages. These insights emphasize the importance of considering programming language when studying code reuse patterns in software projects.

To ensure these results are comparable to blob-level analysis, we calculated the copied blob ratio (copied blobs to total blobs) for each project and took the average of this ratio for projects in each language. An important difference here with the blob-level propensity is that at the blob level, language assignment was based on the

file extension of each blob, with binary blobs categorized as "Other". In this project-level analysis, the language of a blob is determined by the predominant language of the project it belongs to. For example, a Python-written blob in a C-dominated project is counted as a C blob. Similarly, binary blobs are assigned the language of the dominant language in their respective projects. The results of this new definition are shown in Table 3.13.

Table 3.13: Project-level - Propensity to Reuse

Language	**Ratio**	**Language**	**Ratio**	**Language**	**Ratio**
C	15.4%	ObjectiveC	9.5%	TypeScript	5.6%
C#	4.7%	Python	7.3%	Java	5.8%
Go	6.7%	R	7.2%	PHP	9.5%
JavaScript	8.8%	Rust	5.1%	Perl	21.2%
Kotlin	3.4%	Scala	3.5%	Ruby	5.3%

The propensity to copy varies when using this project-level definition compared to the blob-level definition (see Table 3.6).

For example, the propensity to copy in JavaScript-dominated projects is higher than for JavaScript blobs in general (8.8% vs. 5.5%). This indicates a greater likelihood of reuse within JavaScript projects compared to individual JavaScript blobs from various projects. This could be attributed to the modularity and strong reuse culture in the JavaScript ecosystem, where libraries and frameworks are frequently shared and integrated. JavaScript projects often incorporate multiple languages, such as HTML and CSS for web development or server-side languages for backend functionality, enhancing reuse through shared components. The evolution of JavaScript projects, involving various tools and libraries, also contributes to the higher reuse rate within the project context.

In Perl-dominated projects, the propensity to reuse is higher than for Perl blobs in general (21.2% vs. 18.5%). This suggests that blobs within Perl projects are more likely to be reused compared to individual Perl blobs from different projects. Perl's strong culture of code reuse and sharing, exemplified by the Comprehensive Perl Archive Network (CPAN), encourages the use and distribution of reusable code

modules. Perl projects often include a wide range of scripts and utilities shared across different applications, enhancing reuse. Furthermore, Perl's use in scripting, text processing, and system administration often requires the reuse of common patterns and libraries, contributing to the higher reuse rate within projects.

Conversely, R-dominated projects show a lower propensity to reuse compared to R blobs in general (7.2% vs. 9.8%). This implies that individual R blobs are more likely to be reused than blobs within R-dominated projects. R is primarily used for statistical computing and data analysis, where specific scripts and functions are reused across different analyses. However, R projects are often tailored to specific datasets and analyses, resulting in lower overall reuse within the project context. The specialized nature of many R projects, with unique data processing and analysis pipelines, limits reuse compared to individual reusable components like functions and libraries.

Java-dominated projects exhibit a lower propensity to reuse compared to Java blobs in general (5.8% vs. 7.8%). This indicates that individual Java blobs are more likely to be reused than blobs within Java-dominated projects. Java is widely used across various domains, and reusable components like libraries and frameworks are common across different projects. However, Java projects tend to be large and complex, with specific architectures and dependencies that may limit cross-project reuse. The high degree of customization and specificity in Java enterprise applications reduces the reuse rate within the project context compared to the reuse of individual Java blobs or libraries.

These analyses reflect the differing dynamics of code reuse in various programming ecosystems. Understanding these differences can help improve strategies for fostering code reuse and optimizing software development practices across different languages and project contexts.

Binary Blob Analysis

Although previous analyses indicated that binary blobs are more likely to be reused, we aimed to investigate whether this propensity varies across projects dominated by different programming languages. At the blob level, it was not feasible to ascertain the programming language of a binary blob. However, at the project level, such analysis becomes possible. Therefore, we examined the reused binary blob ratio (the percentage of reused binary blobs to total reused blobs) within each language and compared it to the binary blob ratio (the percentage of binary blobs to total blobs) within the same language, utilizing a t-test to identify any significant differences.

Consistent with the blob-level analysis, the reused binary blob ratio exceeds the general binary blob ratio across all programming languages, indicating a higher likelihood of reuse for binary blobs. This observation raises questions about language-specific differences in binary blob reuse. Specifically, we hypothesize that binary blobs are more frequently reused in certain languages compared to others. In other words, we want to know if identifying a reused binary blob allows us to infer that it is more likely to originate from projects written in particular languages.

Our findings confirm this hypothesis, as the proportion of reused binary blobs varies significantly among different programming languages. Nevertheless, we hypothesize that at least some of this difference stems from the general difference in binary blob ratios in different languages and is not limited to reuse. Our statistical tests reveal that the binary blob ratios indeed differ significantly across languages. Consequently, the ratio of reused binary blobs also exhibits significant variation among different languages, suggesting that this difference does not necessarily mean varying binary reuse practices among them.

We want to determine if the higher number of reused binary blobs in a certain language is solely due to the general prevalence of binary blobs in that language, or if some languages tend to reuse more binary blobs. To control for this confounding effect, we normalize the binary blob reuse ratio based on the total binary blob ratio.

Given the binary blobs ratio *br* in a project (binary blobs over total blobs), we defined the reused binary ratio *cbr* (binary reused blobs to total reused blobs) to binary ratio *br* metric. This metric (*cbr*/*br*) averaged 4.104 for all the projects in our sample. By using a linear regression with the project's primary language as a predictor, we obtained the results shown in Table 3.14[7].

$$m = \frac{cbr}{br} = \frac{cbc/cc}{bc/c}$$

m: normalized binary reuse metric

cbr: copied binary ratio

br: binary ratio

cbc: copied binary count

cc: copied count

bc: binary count

c: total count

Table 3.14: Reused Binary Blobs to Binary Blobs Metric

Language	**Metric**	**p-value**	**Language**	**Metric**	**p-value**
C	3.33	0.810722	Rust	6.06	0.422024
C#	4.92	0.025270	Scala	5.38	0.545028
Go	5.73	0.173372	TypeScript	5.17	0.063922
JavaScript	7.04	$< 2 \times 10^{-16}$	Java	4.91	0.000497
Kotlin	5.42	0.306698	PHP	4.49	0.035326
ObjectiveC	2.17	0.217673	Perl	3.32	0.975449
Python	2.19	0.005547	Ruby	3.51	0.951277
R	2.65	0.614773			

Our analysis reveals that the reused binary blobs to binary blobs metric varies across programming languages. Notably, C#, JavaScript, Python, Java, and PHP

[7]The complete coefficients and regression ANOVA tables are available in the online appendix.

exhibit statistically significant differences (p-value ¡ 0.05). In particular, JavaScript projects demonstrate a higher tendency to reuse binary blobs, while Python projects show a lower tendency. This suggests that in JavaScript-dominated projects, reusing binary blobs is likely more efficient and cost-effective than reusing code. Conversely, Python projects might benefit more from reusing code rather than binary blobs.

RQ4 Key Findings

1. Project properties significantly impact the probability of their blobs being reused, with binary ratio, number of blobs, forks, authors, activity duration, and stars having a positive impact.

2. Older projects are more likely to have introduced reused blobs.

3. Blobs residing in projects dominated by different programming languages have varying probabilities of reuse, with TypeScript, Ruby, PHP, and JavaScript having higher probabilities, and Rust, C, Perl, C#, Go, Scala, R, Java, Kotlin, Python, and Objective-C having lower probabilities.

4. On average, 33.7% of projects have introduced at least one reused blob, but this percentage varies significantly between languages, with TypeScript (62.3%) and Ruby (51.2%) having the highest propensity, and R (28.5%) and Perl (29.9%) the lowest.

5. The tendency to reuse binary blobs is much higher in JavaScript projects, while Python projects show a lower tendency.

The project-level analysis reveals that various factors significantly influence the likelihood of code reuse in open source software projects. Projects with more blobs, binary blob ratio, and longer activity tend to exhibit higher reuse rates. This aligns with our hypothesis that project health, activity, and popularity signals play an important role in promoting reuse.

The variation in reuse likelihood across different programming languages underscores the influence of language-specific ecosystems and practices, consistent with blob-level results. For instance, TypeScript and Ruby projects show the highest propensity for reuse, which may be due to their robust ecosystems and strong community practices that encourage code sharing and reuse. Conversely, languages like Python and Perl have lower reuse rates, suggesting different reuse dynamics and possibly a need for improved tools and practices to foster reuse. However, the impact between the blob's language and the language of the project it resides in differs. This suggests that the underlying factors behind these differences are not just technical aspects of the languages and their tools, but also their community culture and practices.

The significant reuse of binary blobs, particularly in languages like JavaScript, indicates that binary artifacts are valuable assets in software projects. This might be due to the efficiency and ease of integrating precompiled binaries compared to source code. However, the lower reuse rate of binary blobs in Python suggests that this language's ecosystem favors source code reuse, which could be due to its dynamic nature and the extensive use of interpreted scripts. These findings have important implications for the development and support of tools that facilitate reuse in different programming languages. For languages like JavaScript, where binary blob reuse is prevalent, enhancing asset libraries could be beneficial. In contrast, for languages like Python, where code reuse is more advantageous, improving code package managers would be more appropriate. This differentiation underscores the necessity for tailored support tools to optimize reuse practices in various programming environments.

These findings highlight the impact of project context on reuse patterns and suggest that different definitions and granularity levels can yield varying insights into code reuse behaviors.

3.6 Limitations

3.6.1 Internal Validity

Commit Time

The identification of the first occurrence and consequently building the reuse timeline of a blob is based on the commit timestamp. This time is not necessarily accurate as it depends on the user's system time. The dataset we utilized followed suggestions by Flint et al. (2021b) and other methods to eliminate incorrect or questionable timestamps. This increases the reliability of our reuse timeline. We also used version history information to ensure the time of parent commits does not postdate that of child commits Jahanshahi and Mockus (2024). This adds an extra layer of consistency and validation, further enhancing the accuracy of our data.

Originating Project

The accuracy of origination estimates is highly reliant on the completeness of data. Even if we assume that the World of Code (WoC) collection is exhaustive, it is possible that some blobs may have originated in a private repository before being copied into a public one. This means that the originating repository in WoC may not be the actual creator of the blob. This scenario suggests that even with a comprehensive dataset, there could be instances of code reuse that remain undetected, adding another layer of complexity to understanding the full extent of reuse across open source projects. For example, a 3D cannon pack asset[8] was committed by 38 projects indexed by WoC. However, that asset was originally created earlier in the Unity Asset Store Jahanshahi and Mockus (2024).

By utilizing the extensive WoC collection, we provide a broad and detailed analysis of code reuse, capturing a significant portion of open source activity even if some instances of private-to-public transitions are missed. Additionally, the examples we

[8] https://assetstore.unity.com/packages/3d/props/weapons/stylish-cannon-pack-174145

identified, such as the 3D cannon pack asset, highlight the practical implications and real-world relevance of our findings, demonstrating the robustness of our analysis despite potential data gaps. Our approach addresses the inherent challenges of tracking code origination and reuse, offering a framework that can be refined and expanded in future research to further improve accuracy and comprehensiveness.

Copy Instance

A unique combination of blob, originating project, and destination project might not always accurately represent the actual pattern of reuse. This is because some destination projects could potentially reuse the blob from a different source other than the originating project. For instance, if we have three projects—A, B, and C—in order of blob creation, project C might copy from either project A or B. Additionally, certain blobs are not reused but are created independently in each repository, such as an empty string or a standard template automatically generated by a common tool Jahanshahi and Mockus (2024). These blobs are excluded by using the list provided by WoC Ma et al. (2019).

Despite this limitation, our results remain significant. By recognizing the potential for indirect reuse and independently created blobs, we provide a more nuanced understanding of the reuse landscape, accounting for the complexity of code propagation across projects. Excluding independently created blobs and utilizing WoC's comprehensive list ensures that our analysis focuses on genuine reuse instances, enhancing the reliability of our findings.

3.6.2 External Validity

Blob-level Reuse

Our work focuses solely on the reuse of entire blobs, deliberately excluding the reuse of partial code segments within files. While blob-level reuse is common, it only covers a subset of the broader code reuse landscape. Blob-level reuse is more relevant to

scenarios where larger code blocks, consisting of entire files or even groups of files, are reused compared to statement or function-level reuse. This means that our results might have an implicit bias towards programming languages or ecosystems that rely more heavily on complete files, potentially overlooking reuse practices prevalent in languages that favor modular or snippet-based reuse.

This limitation also implies that different versions of the same file, even if they differ by just one character, generate different blobs due to distinct file hashes. Consequently, blob reuse does not equate to file reuse. Defining file reuse is challenging because it is difficult to determine what constitutes equivalence between files in different projects Jahanshahi and Mockus (2024). This could be a potential reason for the higher level of reuse in binary blobs, as they are relatively harder to modify.

Despite these limitations, our results remain significant for several reasons:

- **Prevalent Pattern**: By concentrating on entire blob reuse, we address a prevalent and impactful pattern in software development. This allows us to provide valuable insights into a substantial portion of code reuse practices.
- **Clarity and Precision**: Analyzing entire blobs offers a clear and precise method for identifying reuse, avoiding the ambiguity and complexity associated with defining partial file reuse. This clarity enhances the reliability of our findings.
- **Efficiency and Scalability**: Blob-level analysis is computationally efficient and scalable, enabling us to process large datasets and draw meaningful conclusions from extensive data. This scalability is important for comprehensive empirical studies.
- **Foundation for Future Research**: Our work lays the groundwork for future studies that can build on our findings to explore partial file reuse and other nuanced aspects of code reuse. By addressing a well-defined scope, we provide a solid foundation for subsequent research.

In summary, while our focus on blob reuse introduces certain limitations, it also provides clear, scalable, and impactful insights into code reuse practices. This targeted approach enables us to contribute valuable findings to the field, despite the inherent complexities of defining and analyzing file reuse. Although blob-level reuse is less granular than statement or method-level reuse, findings at the blob level would also apply to sub-blob-level analysis, which should adjust for blob-level reuse. Future studies are needed to investigate the extent to which different levels and types of code reuse overlap or differ.

3.7 Conclusions

In conclusion, our study highlights the non-negligible role of copy-based reuse in open source software development. By leveraging the extensive World of Code (WoC) dataset, we provided a comprehensive analysis of code reuse, revealing that a substantial portion of open source projects engage in this practice. Our findings indicate that 6.9% of all blobs in OSS have been reused at least once, and 80% of projects have reused blobs from another project. This widespread reuse emphasizes the efficiency gains in OSS development but also raises concerns about security and legal compliance.

The variation in reuse patterns across programming languages underscores the influence of language-specific ecosystems and practices. Moreover, the higher propensity for binary blob reuse suggests a need for tailored tools to support different types of reuse. Future research should focus on improving the accuracy and comprehensiveness of reuse detection and exploring the impact of partial file reuse.

The survey results further enrich our understanding of reuse practices. We found that many creators intended their resources for reuse, indicating a collaborative mindset among developers. Reusers generally found the reused blobs helpful. Despite these positive perceptions, reusers showed relatively low concern about potential bugs

and changes in the original files. This low level of concern could suggest either a high level of trust in the quality of the reused code or a lack of awareness of the associated risks. Additionally, the survey revealed a moderate interest in using package managers to handle changes to reused files. This indicates potential demand for tools that can streamline and manage code reuse more effectively.

Overall, our work provides insights into the patterns and factors affecting code reuse, advocating for better management and support tools to enhance the sustainability and security of OSS. By addressing the identified risks and leveraging the collaborative nature of the OSS community, we can improve code reuse practices and outcomes.

Chapter 4

Survey

Disclosure Statement

A version of this chapter was originally published as Jahanshahi et al. (2024b):

Mahmoud Jahanshahi, David Reid, and Audris Mockus. 2025. **Beyond Dependencies: The Role of Copy-Based Reuse in Open Source Software Development**. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. Just Accepted (January 2025).

Available at: https://doi.org/10.1145/3715907

Replication package available at: https://zenodo.org/records/14743941

4.1 Introduction

In this chapter, we want to answer this question: How do developers perceive and engage with copy-based reuse? To do so, we obtain responses from 374 developers about the code they have reused or originated. Most respondents write code with an explicit expectation that it will be reused. Developers reuse code for several reasons

and are not concerned with bugs in the reused code, but they are willing to use package managers for reused code if such tools were provided. Overall, we find that despite its questionable reputation due to inherent risks, code copying is common, useful, and many developers keep it in mind when writing code.

The research question in this chapter aims to triangulate the quantitative results from previous chapter and understand how developers perceive and engage with copy-based reuse. While quantitative research often focuses on metrics such as frequency, intensity, or duration of behavior, qualitative methods are better suited to explore the beliefs, values, and motives underlying these behaviors Castleberry and Nolen (2018).

Using a questionnaire for triangulation allows us to obtain self-reported data, which can confirm or challenge the quantitative findings. This method helps identify any discrepancies and provides a deeper understanding of participant behavior Denzin (2017). In our study, the questionnaire included a direct question ("Did you create or copy this file?") to gather self-reported data on whether participants copied the blob, offering a direct measure to compare against the quantitative results.

Additionally, based on the Social Contagion Theory (SCT), we hypothesize that the characteristics of the destination project and/or author influence reuse activity. However, treating all reusers the same could be problematic, as developers may have fundamentally different reasons for reuse. Motivations for reuse can vary widely based on individual needs, project requirements, and perceived benefits from the reused code Mockus (2007); Frakes and Fox (1995). Our primary focus was to understand these motivations to categorize different types of reuse, potentially providing more insight into measuring susceptibility for future research. By categorizing motivations, we aim to identify distinct patterns and factors influencing reuse behavior, facilitating the development of targeted strategies to enhance code reuse practices. This approach aligns with qualitative research methods that seek to explore complex phenomena through detailed, contextualized analysis Creswell and Creswell (2017).

4.2 Methodology

To gain insights into the motivations behind copy-based reuse, we conducted an online survey targeting both the authors of commits introducing reused blobs and the authors of commits in the originating repositories. The survey aimed to capture a range of experiences and perceptions related to copy-based reuse[1].

4.2.1 Survey Content and Questions

The survey included questions about the nature of the file, why it was needed, how it was chosen, and whether developers would use tools to manage reused files. General questions about the repositories and developers' expertise were also included. Notably, the question about the reason for needing the file was open-ended to capture unbiased and detailed responses about the motivations for reuse.

All the questions were optional, except for the very first one, which asked if the respondent had created or reused the file. We chose not to directly ask why did developers choose to copy to avoid provoking legal and ethical concerns about copy-based reuse. For this reason, instead, we asked: "Why was this file needed? How did it help your project?"[2].

Furthermore, we asked developers if the project in which the file resides was intended to be used by other people. Understanding whether creators intend for their resources to be reused helps assess the cultural and strategic aspects of OSS development. If a significant portion of creators design their code with reuse in mind, it indicates a collaborative ecosystem where resources are shared and built upon.

We also asked a series of Likert scale (on a scale from 1 to 5) questions as follows.

- **"To what extent did this file help you?"** - Gauging how helpful creators and reusers find the reused blobs provides quantitative data on the perceived

[1]The survey and its procedure was approved by our institutional review board, ensuring that it adhered to ethical guidelines for research involving human subjects.

[2]See online appendix for survey questions.

value of the reused code. Comparing the ratings between creators and reusers highlights any discrepancies or alignment in perceived usefulness.

- **"To what extent were you concerned about potential bugs in this file?"** - Investigating reusers' concerns about bugs in reused code sheds light on the perceived risks associated with this practice. Understanding the level of concern can indicate how much trust reusers place in the original code's quality.

- **"How important is it for you to know if the original file has been changed?"** - Understanding reusers' concerns about changes in the original files helps identify potential issues related to the stability and continuity of reused code. Frequent changes can disrupt the functionality of dependent projects.

- **"How likely would you use a package manager which could handle changes to this file if there was one?"** - Understanding the likelihood of reusers adopting a package manager if available provides insights into the demand for tools that can streamline and manage code reuse.

4.2.2 Sampling Strategy

To ensure a representative and comprehensive sample, we stratified the data along several dimensions. Stratified sampling ensures that all relevant subgroups are adequately represented in the survey, enhancing the generalizability of the findings Creswell and Creswell (2017). By considering multiple dimensions such as productivity, popularity, copying patterns, file types, and temporal aspects, we ensure a comprehensive analysis that captures the diversity of reuse behaviors in the OSS community:

- **Productivity and Popularity**: Based on the number of commits and stars, we differentiated between high and low productivity/popularity projects (similar to RQ1-b).

- **Copying Patterns**: We distinguished between instances where only a few files were copied versus multiple files, as these might indicate different reuse behaviors.
- **File Extension**: We included various file types and programming languages to capture a diverse range of reuse scenarios.
- **Temporal Dimensions**: We considered the blob creation time and the delay from creation to reuse to understand temporal patterns in reuse behavior.

4.2.3 Survey Design

For each copy instance, we targeted the author of the commit introducing the blob into the destination repository and the author of the commit in the originating repository[3]. This dual perspective allowed us to capture both the originator's and the reuser's viewpoints, offering a more comprehensive understanding of the reuse dynamics.

We conducted three rounds of surveys, progressively expanding the sample size and refining the questions based on feedback and preliminary results. We chose to conduct our survey in three steps to ensure a thorough and iterative approach to understanding developer motivations behind copy-based reuse.

1. We handpicked 24 developers (12 creators and 12 reusers) for an initial survey with open-ended questions. This round aimed to gather in-depth qualitative data and identify key themes. This small, purposive sample size allows for deep, exploratory insights, which are important for the initial stages of qualitative research Guest et al. (2006).

2. The survey was sent to 724 subjects (329 creators and 395 reusers) with a mix of open-ended and multiple-choice questions. This round helped validate and refine the themes identified in the first round. The increased sample size in this round provides more data to ensure that the themes and patterns observed

[3]Only if they had explicitly disclosed their email address on their public profile.

are not idiosyncratic but rather indicative of broader trends. This intermediate sample size balances the need for more extensive data while still allowing for qualitative depth Mason et al. (2010).

3. The survey was expanded to 8734 subjects (2803 creators and 5931 reusers), with most questions being multiple-choice to facilitate quantitative analysis, except for the open-ended question about the reason for needing the file. The large sample size in this final round ensures that the findings are statistically significant and generalizable across the broader population of developers involved in copy-based reuse. This sample size aligns with recommendations for achieving sufficient statistical power in survey research Krejcie and Morgan (1970).

The reason behind the seemingly random numbers of survey subjects in the three rounds is that after sampling our data, we had to perform data cleansing and preparation to reach the survey target audience. This process normally caused some samples to be removed. Initially, we chose sample sizes of 30, 1,000, and 10,000 respondents for the three rounds respectively, but after the data cleansing process, the actual numbers were lower.

4.2.4 Thematic Analysis

The thematic analysis allows us to systematically identify patterns and themes within qualitative data, providing deep insights into the reasons behind copy-based reuse Braun and Clarke (2006). To analyze the survey responses, we followed a structured thematic analysis process as outlined by Yin (2015):

1. **Compiling**: First author compiled all responses.

2. **Disassembling**: Each author individually analyzed and coded the responses to identify ideas, concepts, similarities, and differences Austin and Sutton (2014); Sutton and Austin (2015).

3. **Reassembling**: The coded responses were organized into meaningful themes by each author independently, focusing on identifying different types of reuse Braun and Clarke (2006).

4. **Interpreting and Concluding**: The authors discussed and compared the themes, clarifying and organizing them to ensure a coherent and comprehensive understanding. The final themes were then used to reclassify and interpret all survey responses.

4.3 Results & Discussions

Across three rounds, we received 247 complete responses from reusers and 127 from creators. There were also 360 and 178 partial responses, making the total of 607 and 305 responses from reusers and creators respectively. The results are shown in Table 4.1.

Table 4.1: Survey Participation

	Total	**Started**	**Completed**	**Response Rate**	**Completion Rate**
Creator	3,144	305	127	9.70%	4.04%
Reuser	6,338	607	247	9.58%	3.90%
Total	9,482	912	374	9.62%	3.94%

As will be discussed in Section 3.6.1, the identified originating repository might not always be the true creator of the blob. 39% of developers identified as creators reported reusing the blob from another source. Additionally, reusers might have obtained the blob from another reuser and not the original creator (see Section 3.6.1). Among the reusers who confirmed reusing the blob, 43% acknowledged the originating project as the source, 48% reported copying it from elsewhere, and 9% did not answer the question.

These findings provide important estimates: the fraction of reuse within open source software (OSS) is at least 61%, and the fraction of reuse from originating

projects is at least 43%. This data is essential for understanding the dynamics of code reuse within OSS, highlighting the significance of both direct reuse from original projects and secondary reuse through intermediate projects.

Furthermore, only 60% of those identified as reusers confirmed reusing the blob, while the remaining 40% claimed to have created it (see Table 4.2). This discrepancy can be attributed to several factors. First, some individuals might indeed be the original authors of the blob in the originating project, implying they have reused their own resources. Second, this gap could be explained by activities in private repositories (e.g., Developer A creates a file in a private repository, Developer B copies it to a public repository, and then Developer A reuses it in another public repository). Third, as mentioned in Section 4.2, concerns about potential licensing violations might have made many reusers uncomfortable admitting the reuse explicitly. Additionally, developers' faulty memory could play a role, especially for reuse instances that occurred a long time ago.

One potential area for further investigation could be examining the project owners and commit authors for each copy instance to gain a better understanding of this gap. However, this was not pursued further in this study as it was not the main focus. Exploring these factors in future research could provide deeper insights into the complexities of code reuse and attribution within open source software projects.

Table 4.2: Identified vs. Claimed Creators & Reusers

	Identified	Creators	Reusers	**Total**
Claimed	Creator	77 (61%)	99 (40%)	176
	Reuser	50 (39%)	148 (60%)	198
Total		127	247	374

Another dimension of the survey explored the intentions of creators for others to reuse their artifacts. Sixty-two percent of creators indicated that their resources were intended for reuse by others. When asked about the helpfulness of the particular blob on a scale from 1 to 5 (with 5 being the most helpful), reusers rated the average helpfulness at 3.81, while creators rated it at 4.24. This suggests that developers are

well aware of the reuse potential of their artifacts, even if the blob may be essential primarily for their own projects.

In the background sections, we discussed the risks associated with this type of reuse. We asked reusers if they were concerned about these risks as well. On a scale from 1 to 5 (with 5 being the most concerned), the average concern about bugs in the reused file was 1.83, and the average concern about changes in the original file was 2.35. Several factors might contribute to the low level of concern among developers, including trust in the original code's quality or confidence in their own testing processes. However, this lack of concern could facilitate the spread of potentially harmful code, even if the creator fixes the original code. The fact that reusers are not significantly worried about these risks amplifies the potential risk at the OSS supply chain level.

Next, we asked participants how likely they would be to use a package manager if one were available for the particular blob. On a scale from 1 to 5 (with 5 being the most likely), the average likelihood of using a package manager was 2.93. This indicates that although developers may not be very concerned about bugs or changes (potential improvements), many would still use such a tool if it were available. This suggests that "package-manager" type tools for refactoring or at least maintaining reused code might gain traction if developed. These results are shown in Table 4.3.

Table 4.3: Likert Scale Questions (Scale 1 to 5)

Question (audience)	**Responses**	**Average**	**Median**	**StdDev**
How helpful? (creators)	156	4.25	5	1.15
How helpful? (reusers)	185	3.82	4	1.32
Concern about bugs? (reusers)	185	1.85	1	1.33
Concern about changes in the original file? (reusers)	187	2.33	2	1.56
Likelihood of using a package manager? (reusers)	184	2.89	3	1.64

Finally, the thematic analysis of reasons for reuse, specifically responses to the question "why", revealed eight themes from the 162 responses we received (see

Table 4.4[4]). This analysis provides a nuanced understanding of the motivations behind code reuse, highlighting several key themes.

Table 4.4: Identified Reuse Themes

Theme	**Description**	**Frequency**
Demo	demonstration, test, prototype	14
Dependency	part of a library	11
Education	learning purposes	16
Functionality	specific functionality	39
Own	own reuse	2
Resource	image, style, dataset, license	30
Template	template, starting point, framework	14
Tool	parser, plugin, SDK, configuration	23

As expected, one of the main reasons for reuse was to provide specific functionality. This indicates that developers often reuse code to incorporate existing functionalities into their projects, saving time and effort in development, a practice well-documented in the literature Juergens et al. (2009). This underscores the importance of reusable components in efficient software development.

Another observed theme was the reuse of various resources, including datasets, instructions, license files, and graphical or design objects (e.g., PNG, JPEG, fonts, styles). This aligns with the significant reuse of binary blobs identified in RQ1. The inclusion of diverse resources indicates that developers often depend on readily available materials to enhance their projects' visual or functional aspects. While the literature acknowledges this practice, our findings suggest a slightly higher emphasis on resource reuse. This indicates that resource management might be more important for developers than previously thought.

Reusing tools such as parsers, plugins, SDKs, and configuration files was mentioned 23 times. This practice is noted for its practicality and efficiency in setting up development environments and ensuring consistency across projects. This

[4]Since survey participants were chosen through stratified sampling, these frequencies do not represent the actual data distribution.

highlights the role of auxiliary software components in streamlining development processes and providing necessary infrastructure or functionality.

Assignments, school projects, learning objectives, and similar concepts were another prominent theme. This emphasizes the role of code reuse in the software development knowledge supply chain, as developers reuse existing code to understand and learn new concepts.

Code reuse for demonstration, testing, and prototyping purposes was identified 14 times. This theme suggests that developers often reuse code to quickly create prototypes or test scenarios without focusing on the quality, security, or licensing of the reused code. The priority in these cases is to achieve rapid results. This aligns with the findings by Juergens et al. (2009), that developers often clone code to create prototypes and perform tests. Some of these quick prototypes, however, may end up as active projects.

Templates, starting points, and frameworks were mentioned 14 times. Developers often clone templates or frameworks to have a solid foundation for their projects, a practice supported by findings of Roy and Cordy (2007). This approach leverages existing structures to expedite development and ensure consistency.

Part of a library or dependency management was cited 11 times. This practice is highlighted in studies that emphasize the importance of managing dependencies within the development process, such as the study by Roy and Cordy (2007). Although checking in library files is not considered best practice, many developers do so to maintain specific versions and avoid potential issues with updates or changes. This conscious decision highlights a trade-off between best practices and practical needs.

Reusing one's own code was mentioned twice. The theme of "own reuse" where developers clone their own code for reuse in new projects, is less prominently featured in the literature compared to other reasons for code cloning. Developers clone their own code to ensure consistency, save time, and leverage previously written and tested code. This practice is practical and efficient, especially when developers are familiar

with the code and its functionality. However, the literature does not emphasize this reason as strongly. While studies acknowledge the broader concept of code reuse, their focus is more on reusing code from external sources, libraries, or for educational purposes Juergens et al. (2009); Roy and Cordy (2007). This discrepancy suggests that "own reuse" might be an underexplored area in existing research. It indicates that while developers recognize and practice it frequently, it may not be as thoroughly documented or emphasized in the academic literature. This gap highlights an opportunity for further investigation into how and why developers engage in "own reuse" and its impact on software development processes.

There were also 13 instances where responses were either incomprehensible or the respondent did not remember the file or the reason for reuse.

Key Findings

1. 39% of identified creators stated they reused the blob from another source.
2. Among reusers, 43% acknowledged the originating project (direct reuse), while 48% copied from elsewhere (indirect reuse).
3. Reuse within the OSS landscape is at least 61%.
4. 60% of reusers confirmed reuse; 40% claimed creation.
5. 62% of creators intended their resources for reuse.
6. Reusers are not very concerned about potential bugs or changes in the original file.
7. Reusers are willing to use a package manager if available.
8. Main reuse themes are: functionality, resources, tools, education, demo/testing/prototyping, templates, dependencies, and own reuse.

The findings reveal that a non-negligible portion of developers engage in copy-based reuse within the OSS community. This practice is common, with many reusers sourcing code not directly from the original creators but through intermediaries. Understanding these dynamics is important for improving the transparency and traceability of reused code, which could potentially enhance code quality and security.

The discrepancies between identified and claimed creators highlight complexities in attribution and ownership. Additionally, survey respondents' replies are not always accurate or true, which further complicates understanding the true origins of code. This gap underscores the need for better tracking mechanisms within repositories to accurately reflect code origins. Future research could delve deeper into these factors, offering insights that could inform policy and tooling improvements in OSS development.

Creators often intend their code to be reused, and both creators and reusers recognize the utility of such artifacts. This positive perception suggests that promoting reuse can be beneficial for the community, fostering collaboration and innovation. However, the difference in helpfulness ratings indicates that there might be room for improving the clarity and documentation of reusable code to better meet reusers' needs.

Despite the low concern about potential risks like bugs and changes, the moderate interest in package management tools suggests an opportunity for developing solutions that can help maintain and refactor reused code. Such tools could mitigate risks by providing updates and improvements in a managed manner, enhancing the overall reliability of reused code.

The thematic analysis of reuse motivations provides a comprehensive view of why developers opt for copy-based reuse. Reusing for specific functionality underscores the importance of modular and reusable code in software development. It also highlights the potential benefits of well-documented and easily integrable code components that can be readily reused by others.

This practice of including library files suggests a deliberate effort to maintain stability and avoid the uncertainties that might come with updates or changes. However, it also highlights a potential area for improvement in developer education and best practices, as well as the importance of tools that can help manage dependencies more effectively. These insights contribute to our understanding of the motivations behind code reuse and the practical considerations developers face in maintaining their projects.

While reusing for demo and testing can accelerate development and innovation, it also raises potential risks. Developers may inadvertently propagate vulnerabilities or violate licenses, leading to broader issues within the software supply chain. Highlighting the importance of balancing speed and security during testing phases can inform best practices and educational efforts.

Educational use underscores the educational value of code reuse. Reusing existing code allows learners to understand real-world applications and coding practices, fostering skill development. However, it also emphasizes the need for proper guidance and resources to ensure that educational reuse is done ethically and effectively. Encouraging educators to integrate lessons on best practices in code reuse can enhance the quality of learning and adherence to legal and ethical standards.

The proportion of no meaningful answers and not recalling the file, indicate that not all reuse instances are well-documented or remembered by developers. This lack of clarity can hinder the understanding and traceability of reuse practices. It highlights the need for better documentation and tracking mechanisms to ensure that the reasons and contexts for reuse are transparent and well-understood. Implementing such measures can improve the management of reused code and resources, reducing potential risks associated with undocumented reuse.

4.4 Limitations

4.4.1 Survey Response Rate

The relatively low response rate to our survey may have been due to the perception of the respondents that copying code is a sensitive subject. These concerns may have impacted the responses even in cases when developers chose to participate. It suggests that further work may be needed to design surveys that do not create such impressions.

Additionally, since many of these reuse instances happened a long time ago, developers might have forgotten about them. Therefore, it is important to conduct regular surveys to capture the experiences while developers still remember their practices.

Chapter 5

OSS License Identification at Scale

Disclosure Statement

A version of this chapter is accepted to be published as Jahanshahi et al. (2024a):

Mahmoud Jahanshahi, David Reid, Adam McDaniel and Audris Mockus. 2024. **OSS License Identification at Scale: A Comprehensive Dataset Using World of Code**. In *Proceedings of the 22st International Conference on Mining Software Repositories (MSR '25)*. Just Accepted (January 2025).

Replication package available at: https://zenodo.org/records/14279932

5.1 Abstract

The proliferation of open source software (OSS) and different types of reuse has made it incredibly difficult to perform an essential legal and compliance task of accurate license identification within the software supply chain. This study presents a reusable and comprehensive dataset of OSS licenses, created using the World of Code (WoC) infrastructure. By scanning all files containing "license" in their file paths,

and applying the approximate matching via winnowing algorithm to identify the most similar license from the SPDX and Open Source list, we found and identified 5.5 million distinct license blobs in OSS projects. The dataset includes a detailed project-to-license (P2L) map with commit timestamps, enabling dynamic analysis of license adoption and changes over time. To verify the accuracy of the dataset we use stratified sampling and manual review, achieving a final accuracy of 92.08%, with precision of 87.14%, recall of 95.45%, and an F1 score of 91.11%. This dataset is intended to support a range of research and practical tasks, including the detection of license noncompliance, the investigations of license changes, study of licensing trends, and the development of compliance tools. The dataset is open, providing a valuable resource for developers, researchers, and legal professionals in the OSS community.

5.2 Introduction

As the open-source software (OSS) ecosystem has expanded rapidly, it has given rise to a diverse array of projects, each characterized by different licenses and licensing practices. A fundamental value of OSS lies in the ability to reuse code, either through dependency management or by directly copying and potentially maintaining (vendoring) it. Many licenses impose specific requirements on code usage, such as the obligation to publish derived works under GPL licenses. The reuse supply chains are often complex and difficult to trace. Consequently, accurately identifying OSS licenses across the entire supply chain is crucial for understanding the legal frameworks that govern OSS distribution and use. Such understanding is crucial for ensuring license compliance, fostering collaboration, and mitigating risks within software supply chains. Despite the significance of OSS licensing, existing studies often fall short of covering the entire supply chain by focusing on specific ecosystems, subsets of projects, or lack essential attributes needed to identify timing and project information. Without this information, it becomes impossible to reconstruct the

dynamics or pinpoint the location of licenses within the supply chain Reid and Mockus (2023).

This work makes a step in addressing these challenges by compiling a reusable and comprehensive dataset of OSS licenses. To accomplish that we exploit the World of Code (WoC) Ma et al. (2021) that contains version history from a nearly complete collection of publicly accessible software projects. We start from all files that contain "license" in their file paths and discover over 10M blobs (distinct strings) associated with these files. For each we then find the most similar license from several "official" license collections. To accomplish that we use winnowing algorithm, a fingerprinting technique known for its ability to match text with minor variations, such as differences in formatting, even in cases where the text is embedded or has undergone slight modifications Serafini and Zacchiroli (2022). Our method successfully identifies and maps over 5.5 million distinct license blobs to known licenses, generating a project-to-license (P2L) map enriched with commit timestamps. Furthermore, we enhance our dataset by incorporating the previously published dataset by Gonzalez-Barahona et al. (2023).

This dataset fills critical gaps in the study of OSS licensing by providing: 1) a large-scale, cross-platform resource for analyzing license adoption, change, evolution, and compliance, 2) dynamic tracking capabilities through commit timestamps, enabling longitudinal studies of licensing practices, and 3) a foundation for developing tools and methods to address challenges in OSS license compliance and compatibility.

The dataset and its associated methodology have been designed with reusability and scalability in mind, ensuring that it can be readily adopted by researchers, practitioners, and legal professionals. By making the dataset openly available, we aim to foster new research in software engineering and contribute to better practices in the OSS ecosystem.

5.3 Related Work and Contributions

Understanding OSS licensing practices has been the focus of numerous studies, ranging from license identification to compliance analysis. These studies have contributed valuable insights but are often limited in scope, scale, or methodology.

5.3.1 Comprehensive Identification of License Blobs

Previous studies like Wu et al. (2024) and Xu et al. (2023) focus on explicit license declarations in metadata files, while others, such as Feng et al. (2019), use static analysis to detect embedded license texts in binaries. While these methods and datasets advance license text identification, they do not address partial matches or embedded license texts, which are common in OSS projects.

In contrast, our work leverages the winnowing algorithm, a robust fingerprinting method, to identify both partial and full matches of license blobs across millions of files, even when license texts are embedded or slightly modified. This approach enhances precision and ensures comprehensive identification, capturing both standard and non-standard licensing practices in OSS repositories.

5.3.2 Broad Scale and Scope of Analysis

Prior studies have often been limited in scope, focusing on specific platforms or datasets. Large-scale efforts have identified license files but overlooked contextual information, such as project associations or temporal data. For example, Zacchiroli (2022) introduced a dataset of 6.5 million blob-license text variant tuples (spanning 4.3 million unique blobs), enabling analyses of text diversity and NLP-based modeling of license corpora. However, their work focuses on cataloging text variants rather than linking licenses to their usage within projects. Similarly, Gonzalez-Barahona et al. (2023) documented 6.9 million blob-license tuples (representing 4.9 million unique

blobs), but the emphasis remained on cataloging rather than exploring connections to broader supply chain dynamics.

Our study expands the scope of previous research by analyzing the entire OSS landscape through the World of Code (WoC) infrastructure. We match over 5.5 million license blobs to known licenses and map them to specific OSS projects and their histories. This comprehensive project-to-license (P2L) mapping facilitates detailed tracking of licensing practices across platforms, bridging the gap between text-level variability and actionable project-level insights.

5.4 Methodology

5.4.1 World of Code Infrastructure

World of Code (WoC)[1] is an infrastructure designed to cross-reference source code changes across the entire OSS community, enabling sampling, measurement, and analysis across software ecosystems Ma et al. (2019, 2021). It functions as a software analysis pipeline, handling data discovery, retrieval, storage, updates, and transformations for downstream tasks Ma et al. (2021).

WoC offers maps connecting git objects and metadata (e.g., commits, blobs, authors) and higher-level maps like project-to-author connections, author aliasing Fry et al. (2020), and project deforking Mockus et al. (2020). We use WoC to identify all license blobs and their associated projects[2], employing the concept of deforked projects Mockus et al. (2020) to avoid biases from forks and duplicates.

5.4.2 License Blob Identification

We start by using the blob-to-filepath maps (b2f) in WoC to list all filepaths for each blob, specifically searching for those with "license" in their filepath. Using blob

[1] https://worldofcode.org
[2] Version V, latest at the time of this study.

hashes ensures that any license blob, even if associated with a "license" filepath in only a single project, will still be identified. Using blob-to-project maps (b2P), we then identify all projects containing that blob, which means that we do not require the blob to have the "license" filepath in every project. This ensures high recall in detecting potential license-related blobs by leveraging the collective metadata of public repositories. This approach resulted in over 10 million distinct potential license blobs.

Since there are relatively few known licenses, we anticipate that most of these blobs are similar licenses with minor variations, such as differences in whitespace, formatting, or non-essential information. The main challenge is matching these varied license blobs to known licenses.

We use licenses from the Open Source Initiative[3] and the Software Package Data Exchange (SPDX)[4], which include 103 and 635 licenses, respectively. To match the 10 million potential license blobs with these known licenses, we apply winnowing, an efficient local fingerprinting algorithm Schleimer et al. (2003).

Winnowing is a document fingerprinting technique often used in plagiarism detection. It generates fingerprints by sliding a window over hashed words in a document and selecting the smallest hash value in each window. This reduces the data needed for document representation, enabling faster and more memory-efficient comparisons while maintaining accuracy.

Using winnowing, we matched over 7 million potential license blobs to one of the known licenses (see Table 5.1). We assess the reliability of these matches by calculating a matching score, defined as the number of shared winnowing signatures divided by the total winnowing signatures between two files. This score, as shown in Equation 5.1, measures the similarity between the potential license blob and the known license, helping to verify the match's accuracy.

[3] https://github.com/OpenSourceOrg/licenses
[4] https://github.com/spdx/license-list-data

$$S = \frac{c(A \cap B)}{c(A \cup B)} \tag{5.1}$$

S**:** Matching score.

A**:** Set of signatures in document A.

B**:** Set of signatures in document B.

$c(X)$**:** Count function for the number of elements in set X.

Table 5.1: Potential License Blobs Matching Scores

	Count	**Percentage** (Relative to)	**Percentage** (Overall)
Potential Blobs	10,093,268	100%	100%
Winnowing	9,794,559	97% (Potential Blobs)	97%
Matched	7,167,046	73.2% (Winnowing)	71%
.....			
$S <= 0.2$	795,532	11.1% (Matched)	7.9%
$0.2 < S <= 0.4$	239,091	3.3% (Matched)	2.4%
$0.4 < S <= 0.6$	264,667	3.7% (Matched)	2.6%
$0.6 < S <= 0.8$	435,283	6.1% (Matched)	4.3%
$0.8 < S <= 1$	5,432,473	75.8% (Matched)	53.8%

We categorized matching scores into five groups: below 20%, 20-40%, 40-60%, 60-80%, and above 80%. As shown in Table 5.1, 97% of blobs generated winnowing signatures. We randomly sampled 30 blobs from the 3% that did not and manually confirmed they had no meaningful content. Of the 9.7 million blobs, 73% matched a known license (sharing at least one winnowing signature), with 75% of these matches scoring above 80%.

To assess match reliability, we sampled 20 blobs from each score group and manually compared them to the known license using 'diff'. Given the manual nature of the verification process, choosing 20 samples for each bucket provides a manageable workload while still offering a sufficient range of data to detect patterns

and inconsistencies. Our investigation revealed that matches in buckets with scores below 80% were not reliable enough, showing meaningful differences.

We then focused on scores above 80% and conducted another stratified sampling based on score range (80-85, 85-90, 90-95, 95-100) and the number of signatures (above/below 100). In each group, 20 matches were sampled. The differences fell into three main categories: 1) identical content with different formatting, 2) identical content with non-license text, and 3) identical content with additional clauses.

The second category was acceptable, as we do not claim a blob contains only the matched license. However, the third, with additional clauses, was concerning as it could alter the license's nature. Detailed results are in Table 5.2.

Table 5.2: Matching Score Samples

Signatures	**Score**	**Total Count (%)**	**Gr. 1**	**Gr. 2**	**Gr. 3**
<= 100	80-85	85,294 (1.6%)	17	3	0
	85-90	150,046 (2.8%)	17	3	0
	90-95	197,875 (3.6%)	20	0	0
	95-100	4,502,264 (82.9%)	20	0	0
	.....				
> 100	80-85	67,235 (1.2%)	10	9	1
	85-90	52,894 (1%)	17	2	1
	90-95	60,583 (1.1%)	18	2	0
	95-100	316,282 (5.8%)	20	0	0

We observed only two mismatches: one in the 80-85% range and one in the 85-90% range (both in the over 100 signatures group). Based on this, we determined that setting the threshold at 85% ensures reliable license identification. Above this threshold, critical mismatches—where additional clauses could alter the license—are extremely rare. Since over 90% of identified blobs had fewer than 100 winnowing signatures, the 85% threshold balances comprehensiveness and precision, capturing most valid matches while minimizing misleading results. This approach aligns with prior research emphasizing high similarity thresholds to reduce false positives in

textual matching (e.g., Kapitsaki et al. (2017)). As a result, 5,294,666 distinct blobs were matched with a known license.

For the remaining 2.5 million potential blobs with no matches, we randomly sampled 30 and manually investigated them. Only 5 contained license-related content, either mentioning a license name or linking to a license URL. The other 25 were unrelated to licenses.

5.4.3 Project to License Mapping

To create the project-to-license (P2L) map, we use the 5.5 million matched license blobs and join them with WoC's blob-to-time project (b2tP) map, which links blobs to the projects they were committed to, along with commit timestamps. This produces a table mapping each project to a known license and the time of the commit (see Figure 5.1).

However, a blob's presence in a project's latest status cannot be confirmed solely from commit history, as it might have been removed later. To address this, we use the project-to-last-commit (P2lc) and tree-to-objects (t2all) maps from WoC. The P2lc map links projects to their last commit at the time of the latest WoC update (Version V), allowing us to retrieve the list of all blobs in a project's current state by joining P2lc, c2dat (commit-to-tree), and t2all maps. This method not only provides all the times at which a blob was committed to a project but also verifies whether it still exists in the project.

The final table is saved as a semicolon-separated file containing three fields[5]:

$Project\_ID; License; Commit\_Time$

The $Commit\_Time$ field is in the "YYYY-MM" format and represents the commit timestamp when the license blob was committed to the project. This field may also have an "invalid" value, indicating that the commit timestamp was not valid (e.g.,

[5]For more information on accessing this data, please visit https://github.com/woc-hack/tutorial

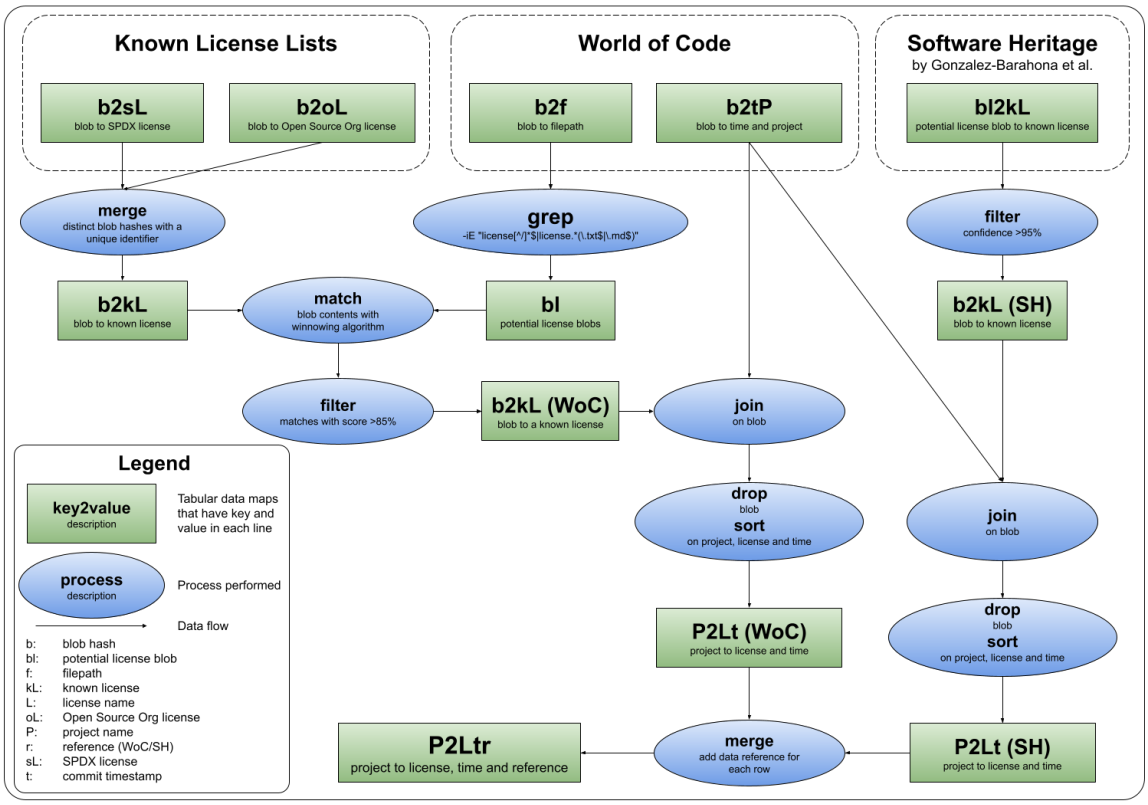


Figure 5.1: License Identification Data Flow Diagram

a future time due to discrepancies in the user's system time). Additionally, if the license blob was found in the latest status of a project, the time is "latest".

5.4.4 P2L Verification

For the Project-to-License (P2L) verification, we initially sampled 1,000 projects from approximately 130 million to evaluate the effectiveness of our license assignment methodology. This sample size was chosen to provide a statistically significant subset for manual verification while balancing the need for reliability with the practical constraints of manual inspection.

We stratified the sample into three groups: 1) Projects with matched licenses, where our automated process successfully matched license blobs to known licenses, 2) Projects with license blobs but no matched licenses, where license blobs were

identified, but no matching known license could be confirmed, and 3) Projects without any license blobs, where no license blobs were detected during the automated search.

This sampling approach was designed to cover a wide range of license detection scenarios, ensuring a comprehensive evaluation. Graduate students manually reviewed the sampled projects as part of a class assignment, focusing on verifying the license information. Out of the 1,000 sampled projects, we received meaningful responses for 580 projects, distributed as follows: 291 with matched licenses, 139 with license blobs but no matches, and 150 without any license blobs. The results are presented in Table 5.3.

Table 5.3: License Detection Confusion Matrix Across Stages

Stage	**Initial**		**Adjusted**		**Refined**	
	License	**No License**	**License**	**No License**	**License**	**No License**
Matched	210	81	210	31	210	31
Not Matched	22	267	22	267	10	267
Accuracy	82.24%		90.00%		92.08%	
Precision	72.16%		87.14%		87.14%	
Recall	90.52%		90.52%		95.45%	
F1 Score	80.31%		88.79%		91.11%	

Our license detection method demonstrated reasonable performance with an initial accuracy of 82.24%, precision of 72.16%, recall of 90.52%, and an F1 score of 80.31%.

However, several factors must be considered when interpreting these results: first, of the 81 projects identified as having matched licenses, 39 no longer exist on GitHub, preventing license verification, and second, in 11 projects, the license was absent in the latest status, which does not necessarily indicate a false positive, as the license could have been removed after an earlier commit. After excluding these cases, we are left with 31 false positives. Adjusting for these, our revised performance metrics show significant improvement: accuracy increases to 90.00%, precision to 87.14%, recall remains at 90.52%, and the F1 score rises to 88.79%.

For the 22 false negatives (where licenses were not detected), further investigation revealed that only 10 had a missed license blob, which was matched but fell slightly

below our 85% threshold. The remaining 12 projects only referenced a license (e.g., in the README) without including the actual license file in the repository, so they were not expected to be matched by our method. By excluding these 12 false negatives, which fall outside our method's intended scope, we can more accurately assess its performance. The recalculated metrics show an accuracy of 92.08%, precision of 87.14%, recall of 95.45%, and an F1 score of 91.11% (see Table 5.3).

5.4.5 Complementing Data

Although our P2L map already demonstrated strong performance in manual verification, we incorporated the previously published dataset by Gonzalez-Barahona et al. (2023) to enhance data comprehensiveness. Their dataset includes only blobs and their detected licenses using ScanCode Ombredanne (2022). We filtered data to blobs with license detection confidence 95% or higher and applied the same process described earlier to map these blobs to commits and projects, enabling us to determine the time and project in which each license was committed. The merged table (see Figure 5.1) includes a column indicating the license detection method for each entry: either our method (1-WoC) or the Software Heritage dataset method (2-SH) Gonzalez-Barahona et al. (2023).

5.5 Applications

The dataset described in this work provides a robust foundation for addressing key challenges in open source software (OSS) licensing. Below, we discuss use cases supported by the dataset and illustrate them with examples from ongoing research conducted by the authors, which are currently under review and cannot be cited directly.

5.5.1 Ensuring License Compliance

Managing license compliance is a critical issue in OSS, where licensing conflicts or noncompliance can lead to significant legal and ethical challenges. This dataset enables research into understanding and mitigating compliance risks. For instance, the dataset has been used to analyze how licensing conflicts arise from code reuse across OSS projects. These insights underscore the need for advanced compliance tools that leverage comprehensive project-to-license mappings to detect and address potential license violations.

5.5.2 Analyzing Licensing Trends and Practices

Understanding how OSS licenses are selected and evolve over time is essential for improving licensing practices and fostering innovation. The dataset supports large-scale analyses of license adoption trends, revealing patterns and the factors influencing license choices (e.g. Vendome et al. (2017)). For example, it has been used by the authors to explore the dynamics of license adoption, examining the role of social, technical, and ideological factors in shaping these decisions. The dataset's extensive coverage allows researchers to track the evolution of licenses within and across OSS ecosystems, providing actionable insights for developers and policy-makers.

5.5.3 Supporting Ecosystem Studies and Tool Development

The dataset's comprehensive project-to-license mapping has broad applicability in supporting ecosystem studies and tool development. Such applications include investigating how licensing practices influence collaboration and innovation in OSS communities, enabling the creation of automated tools for license verification, detecting noncompliance, recommending suitable licenses, and providing a resource for educating developers on licensing implications and best practices.

5.6 Limitations

Scope of License Identification The current methodology focuses on files explicitly named "license" or located in license-related directories, which may miss license information embedded in source code headers, build scripts, or files with unconventional names. These gaps particularly affect older or unconventional OSS projects. Expanding the search scope and using natural language processing (NLP) or pattern recognition could improve coverage. To partially address this, we incorporate the dataset by Gonzalez-Barahona et al. (2023) to enhance comprehensiveness.

Implicit Licensing Practices Implicit licensing practices, such as referencing licenses by name or URL in README files or documentation, are not captured, potentially leaving gaps for permissively licensed projects. Future work could parse these files to link references to known licenses.

Data Completeness and Noise Finally, while robust heuristics minimize errors, some non-license files may be misidentified, and legitimate licenses in non-standard formats could be excluded. Feedback mechanisms and automated quality checks could further enhance reliability.

Chapter 6

The Intersection of Copy-Based Reuse and License Compliance

Disclosure Statement

A version of this chapter is based on work that has been submitted for publication. The manuscript is currently under review.

This material is included in accordance with academic guidelines on thesis and dissertation reuse.

Replication package available at: https://zenodo.org/records/14061115

6.1 Abstract

As other creative work, source code is protected by copyright. The owner can license the work, e.g., to permit copy and other kinds of use, and even start legal proceeding against license violators. However, source code can be reused in subtle ways, e.g., via copying without explicit package manager dependencies, making it hard to reason about potential license noncompliance. Using the World of Code infrastructure approximating the entirely of open source software, in this paper we

create a copy-based code reuse network mapping direct copying across projects, and use it to quantify the extent of potential license noncompliance across the entire open source ecosystem. In addition, we estimate regression models to understand whether code copying is affected by the origin project's license, and, if so, how it varies with other project characteristics.

We find that code in repositories with permissive licenses, such as MIT and Apache, shows higher likelihood of reuse across programming languages. In contrast, copyleft licenses, like the GPL, exhibit mixed effects. Public domain licenses, despite their aim of allowing unrestricted use, are associated with lower likelihood of copy-based reuse. A widespread potential license noncompliance appears to accompany copy-based reuse, with 39.4% of project combinations at potential noncompliance risk, particularly when licenses are unclear or absent. Our findings reveal that only 2.43% of reuse detected through the copy-based network was discoverable via dependency analysis, highlighting the limitations of existing dependency-tracking tools in capturing copy-based reuse. This gap underscores the need for more advanced methods to ensure license compliance in open source projects, from nudging developers to set appropriate license templates to flagging potential noncompliance due to license changes across copy origin and destination projects.

6.2 Introduction

Open Source Software (OSS) plays a critical role in software development and distribution across various industries. A fundamental aspect of OSS is its licensing, which dictates how software can be reused, modified, and redistributed. OSS licenses are typically categorized into permissive licenses (e.g., MIT, Apache), copyleft licenses (e.g., GPL), weak copyleft licenses (e.g., LGPL), public domain licenses, and others with specific conditions (e.g., Creative Commons). Each type of license imposes distinct obligations on developers and users, making the choice of license a pivotal factor in determining the extent and manner in which a project's code can be reused.

Moreover, creative works, such as code, are protected by copyright by default if no license is specified. Despite these legal restrictions, code without a license is often copied in practice Vendome et al. (2018) and even used to train Large Language Models Xu et al. (2024).

This study aims to enhance understanding of the extent to which, and the contexts in which, different OSS license types affect software reuse in copy-based reuse networks, where artifacts are copied from one repository to another.

While prior research has primarily focused on dependency-based reuse, where projects formally declare dependencies on external libraries, copy-based reuse—where code is directly copied between projects—introduces unique challenges regarding license compliance and tracking, because there is typically no trace of the copying. Studies highlight that identifying the exact origin of reused OSS components remains a significant challenge, underscoring the need for more effective tools to track code provenance, particularly to ensure compliance with copyleft licenses Tuunanen (2021).

Although copy-based reuse is common in OSS development Jahanshahi et al. (2024b), it is often overlooked in studies that focus exclusively on dependencies managed through package managers Fendt and Jaeger (2019); Phipps and Zacchiroli (2020); German et al. (2010). While the decision to copy an artifact from an upstream project may be driven by factors largely unrelated to license compatibility, the type of license should still play a significant role, particularly if the license of the copied artifact is ultimately incompatible with that of the reusing project.

Specifically, we answer the following research questions:

- **RQ1**: How does the license type of the upstream project affect the probability of its artifacts getting copied?
- **RQ2**: How widespread is potential license noncompliance in copy-based reuse network?

We begin by reviewing the literature on code copying to identify key factors driving this phenomenon. Next, we use the World of Code (WoC) infrastructure,

which provides comprehensive, cross-referenced data on the global OSS ecosystem, to operationalize these factors and create a curated dataset of copying instances, including the licenses of both upstream and downstream projects. Finally, we fit a model that examines the probability of a project's artifacts being reused based on its license, while controlling for other contextual factors.

Our findings indicate that permissive licenses, such as MIT and Apache, are consistently associated with higher reuse rates across multiple programming languages. In contrast, copyleft licenses, like GPL, display more complex reuse patterns. While they are associated with higher rates of reuse in certain cases, such as in JavaScript projects, they are generally associated with lower reuse when factors like project size and activity are considered. Interestingly, projects under public domain licenses, which are intended to permit unrestricted reuse, tend to experience lower reuse rates. This suggests that legal uncertainties surrounding these licenses may deter developers from reusing the code.

One notable issue we uncovered is the prevalence of license noncompliance in copy-based reuse, especially when projects either lack a clear license or use incompatible licenses, posing legal risks for developers and organizations alike. License noncompliance in software reuse is not just a theoretical concern but has resulted in significant legal disputes in the software industry. A notable example is the *Jacobsen v. Katzer* case Shagall and Breithaupt (2008), wherein the court upheld the enforceability of open source licenses under copyright law. Jacobsen, the creator of the Java Model Railroad Interface (JMRI) project, sued Katzer for incorporating JMRI's code into commercial software without adhering to the terms of the project's Artistic License. The court's decision affirmed that violating open source license terms constitutes copyright infringement, emphasizing the legal obligations developers have when reusing code.

Another case illustrating the repercussions of license noncompliance involves the GPL-licensed *BusyBox* OSS project Software Freedom Law Center (2007). *BusyBox* developers filed lawsuits against several companies for distributing their software

within commercial products without complying with GPL terms. These companies failed to provide access to the source code and did not include the GPL license text with their products, both required under the GPL. The legal actions often resulted in settlements where the offending companies agreed to release the source code and comply with the GPL terms.

These real-world examples underscore the importance of understanding and adhering to license terms, especially in copy-based reuse where code is directly replicated between projects. Noncompliance not only exposes developers and organizations to legal risks but also undermines the collaborative ethos of the OSS community German et al. (2010). It can deter developers from contributing or reusing code due to fears of infringement, thereby stifling innovation and collaboration. Therefore, ensuring proper license compliance is essential for fostering trust and sustainability in open source software development.

Finally, our study reveals that traditional tools focused on dependency tracking fail to capture a substantial number of reuse cases occurring through direct code copying. This highlights the need for more sophisticated tools capable of detecting direct code copying at scale, to improve license compliance monitoring within the OSS ecosystem.

6.3 Related Work and Knowledge Gaps

6.3.1 Software Reuse

In open source software, the reuse within supply chains can be categorized based on how open source components are integrated and used in software projects Mockus (2019b, 2022, 2023).

Dependency-Based Reuse

This category involves incorporating open source libraries and packages as dependencies in a project. Package managers like NPM for JavaScript, pip for Python, or Maven for Java are typically used to manage these dependencies. If not properly overseen, reliance on these dependencies can introduce vulnerabilities and risks Yan et al. (2021).

Copy-Based Reuse (Our Focus)

In copy-based reuse, developers directly copy code from OSS projects, e.g., a utility function Jahanshahi et al. (2024b), into their own projects. While this approach is quick, it can lead to challenges in maintaining and updating the copied code. Therefore, it's essential to track and manage these copies to ensure they remain secure and up-to-date Ladisa et al. (2023).

Previous studies have identified several factors that influence the likelihood of a project's artifacts being reused through copy-based methods Jahanshahi et al. (2024b). One key factor is **project activity**, typically measured by the number of commits. Projects with a higher commit count are generally more active and frequently updated, making them attractive to developers seeking reliable and current code snippets Koch and Schneider (2002). Another important factor is **project size**, often indicated by the number of files. Larger projects tend to offer a broader range of functionalities and code examples, increasing the likelihood that other developers will find useful code for reuse. This extensive codebase provides a valuable resource for copy-based reuse Mockus (2007). The collaborative nature of a project also plays a role. Metrics such as the **number of authors** reflect the volume and diversity of expertise within a project's contributor base. Projects with more contributors tend to benefit from enhanced innovation and decentralized communication, which can improve the development process Crowston and Howison (2005) and increase the likelihood of reuse Jahanshahi et al. (2024b). Community engagement and

popularity, often approximated by metrics such as the **number of forks and stars** on platforms like GitHub, further explain reuse potential Tsay et al. (2014); Borges et al. (2016). Projects with more forks and stars are more visible and reputable within the developer community, increasing trust and making their code more likely to be reused Jahanshahi et al. (2024b). These indicators reflect community interest and endorsement, enhancing the project's appeal as a resource.

The **maturity and stability** of a project, assessed through its duration of activity, age, and activity fluctuations (burstiness), also correlate with its reuse potential Jahanshahi et al. (2024b). Mature projects with sustained activity over a long period are often viewed as stable and reliable. Consistent development without erratic bursts signals a well-maintained project, increasing the likelihood that its code will be reused Gamalielsson and Lundell (2014). Additionally, a project's community culture and technical characteristics—approximated by its **primary programming language**—play a significant role in explaining its reuse potential Jahanshahi et al. (2024b). Different programming languages vary in popularity, community support, and ecosystem maturity Bissyandé et al. (2013). Projects written in widely adopted languages such as Python, JavaScript, or Java are more accessible to a larger pool of developers, thus increasing the chances of their code being reused. Moreover, the programming language reflects the community's coding conventions, documentation practices, and collaboration norms, which can make the project more appealing for developers looking to incorporate its code into their own work.

Finally, the literature highlights that **permissive licenses**, such as MIT and BSD, are generally associated with higher reuse rates compared to restrictive licenses like GPL Kashima et al. (2011); Brewer (2012). Additionally, a **delay in license adoption** for a project might increase the chances of its artifacts being reused as the absence of a clear license can create ambiguity, leading developers to assume permissibility, thus fostering reuse even if unintended by the project maintainers. However, these conclusions are based on simple statistical analyses that do not account for the critical factors influencing reuse discussed earlier. Therefore, it is

possible that the observed effect of licensing on reuse is not as strong as suggested, or that other variables may be driving these patterns. A more comprehensive analysis—one that controls for these additional variables—is necessary to determine whether licensing independently influences reuse or if the previously-reported results are mostly shaped by other project characteristics. Towards answering **RQ1**, we posit two concrete hypotheses:

- **Hypothesis (H1a)**: Projects using permissive licenses, when controlling for other context factors, have a higher likelihood of their artifacts being reused via copying.
- **Hypothesis (H1b)**: Projects using restrictive licenses, when controlling for other context factors, have a lower likelihood of their artifacts being reused via copying.

6.3.2 Open Source Licenses

There are many licenses for open source code, each with its own requirements and restrictions.

Permissive licenses, such as MIT and Apache-2.0, typically allow for extensive reuse with few restrictions. They usually require only attribution and permit integration with other license types, offering significant flexibility Laurent (2004). In contrast, **copyleft licenses**, such as the GPL, require that any derivative work be distributed under the same license. Noncompliance can occur if copyleft-licensed code is combined with code under a non-copyleft license without adhering to the copyleft terms. For example, incorporating GPL-licensed code into proprietary software without releasing the combined code under the GPL would violate the license Stallman (2002). This principle ensures that all modifications and derivative works remain free, preserving software freedom Lessig (2004). **Weak copyleft licenses**, such as the LGPL, are less restrictive than full copyleft licenses. They permit linking with proprietary software without requiring the entire work to be

open sourced, as long as the LGPL-covered components remain modifiable and separable. However, it's important to carefully consider the terms to avoid violations, particularly regarding modification and distribution Rosen (2005). **Conditional open licenses**, including many Creative Commons licenses, offer specific conditions for use. For example, CC-BY licenses require attribution, while CC-BY-SA licenses require derivative works to be licensed under the same terms. These licenses can include share-alike clauses, which impact how code can be distributed, especially if combined with other licenses with different terms. While these licenses are more commonly used for creative works than software, they can still impact code distribution. **Public domain** and **license-free** software code generally impose no restrictions on reuse, as they are not protected by copyright. Works in the public domain can be freely used, modified, and distributed. Finally, projects with **no explicit license** (not to be confused with license-free) present significant legal risks. By default, all rights are reserved under copyright law, meaning that reuse, modification, or distribution may be restricted without the author's explicit permission Välimäki (2005). This lack of clarity can lead to potential legal issues, as the permissions for using the software are not clearly defined.

6.3.3 Open Source License Compliance

License compatibility is a critical concern in OSS development. Projects often encounter significant difficulties when integrating components with conflicting licenses Di Penta et al. (2010). Ensuring compliance with open source licenses is also a major concern for companies incorporating OSS into their products. German et al. (2010) emphasized the need for auditing OSS distributions to ensure adherence to license terms, especially in scenarios where components with varying licenses are integrated. Wu et al. (2024) conducted a large-scale empirical analysis on the usage of open source licenses, highlighting the practices and challenges developers face. Their findings revealed frequent misunderstandings and misapplications of licenses,

especially in large-scale projects. Cui et al. (2023) created a tool called DIKE to detect license conflicts in over 16,000 popular free and OSS software, finding that over 25% had conflicts. In addition, their study suggests that these conflicts often arise from misinterpretations of license terms and the challenges of handling multi-license environments. Finally, Mathur et al. (2012) conducted an empirical study on license violations resulting from code reuse across 1,423 projects, uncovering numerous instances of license incompatibilities.

In addition, many developers involved in OSS projects do not fully understand the implications of the licenses they use. Almeida et al. (2019, 2017) revealed gaps in developers' knowledge of licensing issues, which can result in non-compliance, particularly in complex projects that integrate multiple OSS components. Moraes et al. Moraes et al. (2021) and Qiu et al. Qiu et al. (2021) focused on the JavaScript ecosystem, investigating the effects of multi-licensing and license violations related to dependencies. Their findings show that the complex network of dependencies in JavaScript projects frequently results in unintentional license violations, highlighting the need for improved dependency management practices. Feng et al. Feng et al. (2019) investigated license violations in large-scale binary software, revealing that many projects unintentionally breach license terms due to the complexities involved in binary distribution. Finally, Papoutsoglou et al. (2022) examined licensing questions on Stack Exchange sites, their results showing that many developers find it challenging to grasp licensing terms, leading to frequent inquiries about compliance and compatibility issues.

Studies have also demonstrated that a project's declared license is not always reliable German et al. (2010); Reid and Mockus (2023); Wolter et al. (2023). For example, in a study of OSS projects on GitHub, Wolter et al. (2023) discovered that in approximately 50% of the projects analyzed, the top-level declared license did not fully reflect all the licenses present within the project, emphasizing the importance of improved education and automated tools for ensuring compliance. Moreover, Wu et al. (2015) found instances where the license of a source code file was altered after

being copied, both by the original author of the code, and by the reuser; the latter likely constitute a license violation.

The complexities of OSS licensing are further heightened by the widespread practice of copy-based code reuse, which can lead to unintended license violations Jahanshahi et al. (2024b). Managing license compliance in these scenarios is crucial for maintaining the integrity of open source projects. Jahanshahi et al. (2024b) showed that 80% of OSS projects have practiced copy-based reuse, including large and popular projects. They also demonstrated that a significant portion of the reused artifacts originate from small, lesser-known projects. Given the widespread prevalence of copy-based reuse and the complexities of tracking the origins of artifacts, we anticipate a high potential risk of license noncompliance in this type of reuse. This issue becomes even more critical considering that copy-based reuse is generally overlooked both by prior research and practitioners, thereby increasing the overall risk for the OSS community. Towards answering RQ2, we hypothesize that:

- **Hypothesis (H2a)**: Copy-based reuse carries a high risk of license noncompliance due to compounded complexities in tracking artifact origins.
- **Hypothesis (H2b)**: By overlooking copy-based code reuse, we are missing a significant portion of license noncompliance issues in open source software.

6.3.4 Our Study vs Prior Work

Our work offers a comprehensive and practical approach to identifying and addressing potential licensing issues arising from copy-based reuse in open source software, and it distinguishes itself from prior research in several ways:

Comprehensive Identification of Licenses

Most studies, including those by Wu et al. Wu et al. (2024) and Xu et al. Xu et al. (2023), rely heavily on explicit license declarations in metadata files. Others, like

Feng et al. Feng et al. (2019), use static analysis of binaries to detect embedded license texts. However, these approaches can miss licenses that are not explicitly declared or are located in less conventional directories. In contrast, our work analyzes a comprehensive dataset Jahanshahi et al. (2024a) created by exhaustively scanning the entire OSS landscape (as reflected in the World of Code Ma et al. (2019)) for files containing the word "license" in their filepath. This includes not only standard license files but also any file that may contain licensing information, ensuring no (obvious) potential license data is overlooked.

Scale and Scope of Analysis

Previous studies often concentrate on specific platforms (e.g., GitHub), particular package manager ecosystems (e.g., NPM), or a narrow range of licenses (e.g., OSI-approved), leading to a partial approach to license detection and analysis. For instance, the work by Feng et al. (2019) maps binary code to source code, detecting instances where code is directly incorporated into binary software. While theoretically feasible, this approach encounters significant scalability challenges due to the substantial processing power required for large-scale analysis. The computational demands of binary-to-source mapping render it impractical for use across the entire open-source ecosystem, especially when dealing with diverse binaries and platforms. In contrast, our work examines the entire open-source ecosystem, offering a more comprehensive, cross-platform perspective on licensing violations. By focusing on scalable methods that encompass various licenses, package managers, and code reuse practices, our approach addresses the scale limitations of prior studies, while providing a more practical solution for detecting license violations across the open-source landscape. Moreover, our approach is not limited to code reuse; it can identify reuse across various types of artifacts, including documentation, configuration files, and other non-code components. This capability offers a more comprehensive perspective on reuse and the associated licensing challenges.

Controlling for Project Context

Compared to prior research, our work reflects a more nuanced analysis of the relationship between software licensing and code reuse. Unlike earlier studies that primarily used bivariate statistical correlations Kashima et al. (2011); Brewer (2012), we use a more sophisticated methodology that accounts for covariates, such as project size, community activity, and programming language. By controlling for these factors, our work provides a clearer understanding of whether licensing type—permissive versus restrictive—independently influences reuse probability. This allows us to re-examine the claims made in prior studies and offers more robust insights into the impact of licensing on OSS reuse.

Analysis of License Violations in Copy-based Reuse

While many studies have explored license conflicts, few have employed a copy-based reuse network approach to understand the reuse patterns and potential violations and they often focus only on dependency-based reuse networks. As shown recently Jahanshahi et al. (2024b), copy-based reuse is prevalent and contributes significantly to reuse practices in OSS. Our research uses the copy-based reuse network to identify potential license violations due to license incompatibilities and reuse patterns, providing a novel perspective on how licenses interact across repositories. This not only reveals license conflicts but also traces their origins, facilitating targeted resolutions and ensuring compliance across the software ecosystem.

6.4 Methodology

6.4.1 World of Code Infrastructure

World of Code (WoC) Ma et al. (2019) is an infrastructure developed to cross-reference source code change data across the entire OSS community, enabling sampling, measurement, and analysis both within and across software ecosystems Ma et al.

(2019, 2021). Essentially, WoC functions as a software analysis pipeline, handling data discovery and retrieval, storage and updates, as well as the transformations and augmentations required for subsequent analytical tasks Ma et al. (2021).

WoC provides various maps that link git objects and metadata (e.g., commits, blobs, authors) to each other. It also offers more advanced maps, such as project-to-data connections (e.g., project-to-author), author aliasing Fry et al. (2020), and project deforking maps Mockus et al. (2020). In our study, we use WoC's project-to-license (P2L) map Jahanshahi et al. (2024a), which shows the licenses committed to each project in its most recent state (Version V of WoC, updated in March 2024). Additionally, we apply the concept of deforked projects, as introduced by Mockus et al. (2020), to minimize potential biases caused by forks and duplicates of the same project. Throughout this paper, the term "project" refers to these deforked projects unless stated otherwise.

6.4.2 Copy-based Reuse Network

In the context of OSS development, analyzing code reuse is essential for understanding the propagation of software components and the associated licensing implications. Traditionally, the literature has primarily focused on dependency-based reuse, where the relationships between projects are analyzed based on declared package-manager dependencies, such as libraries or frameworks included in a project. While dependency-based analysis provides valuable insights into how projects rely on external components, it often overlooks the more granular aspect of direct code copying, which can occur independently of formal dependencies. Such practices are common in OSS projects but often remain undetected in dependency-based analyses, as shown by Jahanshahi et al. (2024b). By mapping these direct copies, a copy-based reuse network provides a comprehensive view of code propagation, highlighting the actual flow of code between projects.

In the realm of license compliance, dependency-based analysis often focuses on the licenses of declared dependencies. However, license obligations are not limited to these formal dependencies. Copy-based reuse, particularly when undetected, can lead to unintentional license violations. By mapping direct code copying, a copy-based reuse network allows for the identification of potential licensing conflicts that may arise from incorporating code with incompatible license terms, when the code wasn't part of a declared dependency.

To track this kind of reuse, WoC offers the Ptb2Pt map, which lists reused blobs (i.e., file versions) along with the creator, reuser, and the time each project first committed that blob Jahanshahi and Mockus (2024). This map is created by sorting the timestamps of all commits creating a blob, with the project associated with the earliest commit identified as the creator. Projects with any subsequent commits are then identified as reusers of that blob.

Next, since we are interested in a project-level analysis and since projects may reuse many blobs from one another, we further aggregated the data based on unique combinations of upstream and downstream projects, counting the number of reused blobs between these projects for each combination. The total number of unique upstream-downstream project combinations was 1,815,996,757. Given our focus on potential license noncompliance, we excluded all instances of code reuse where the same entity (account) owns both the source and target projects. This further reduced the data down to 1,788,541,220 combinations, indicating that about 1.5% of reuse instances occurred between projects with the same owner.

Furthermore, given that the distribution of copied blob counts between projects is heavily right-skewed, we analyze potential noncompliance within the reuse network in two distinct modes to gain better insights. First, we consider **complete reuse**, including any instance where **at least one blob** has been copied in our analysis. Second, we refine the data to focus on reuse instances where **at least ten blobs** have been copied between upstream and downstream projects, as a proxy for more deliberate and **substantial reuse**.

6.4.3 Potential License Noncompliance

Noncompliance can manifest in various ways, often resulting in substantial legal and operational risks. For example, it can occur when there are conflicts or misunderstandings regarding the terms and conditions of these licenses. To better understand the associated risks, we categorize the outcomes of license combinations into three levels: *No Issues*, *Potential Issue - Low Risk*, and *Potential Issue - High Risk*.

No Issues This category covers situations where combining different licenses does not create legal or practical issues. Projects under these licenses can be freely integrated, modified, and redistributed without concern for restrictive terms. For instance, public domain and permissive licenses, such as the MIT or Apache 2.0 licenses, generally impose minimal restrictions. These licenses are designed to encourage widespread use and modification, making them highly compatible with other licenses. Their permissive nature ensures that they do not impose additional restrictions on combined works, allowing for seamless integration with other projects Laurent (2004); Rosen (2005).

Potential Issue - Low Risk These combinations produce minor or manageable incompatibilities, such as attribution, notice preservation, or compliance with specific conditions. For example, weak copyleft licenses, such as the LGPL, allow linking with proprietary software, provided that modifications to the LGPL-covered code remain open-source. This flexibility reduces the likelihood of significant legal issues when combined with other licenses. Similarly, licenses such as the Mozilla Public License (MPL) require modified files to be distributed under the same license but allow linking with other code, thus posing only minor issues Fitzgerald (2006).

Potential Issue - High Risk These combinations can create substantial legal or practical obstacles. These issues typically arise from strict copyleft provisions or other

Table 6.1: License Reuse Matrix and Potential Noncompliance Scenarios

From \ To	**Permissive**	**Copyleft**	**Weak Copyleft**	**Conditional Open**	**Public Domain**	**No License**
Permissive	No	No	No	No	No	Low
Copyleft	High	No	High	High	High	High
Weak Copyleft	Low	No	No	Low	Low	High
Conditional	Low	High	High	High	Low	High
Public Domain	No	No	No	No	No	No
No License	High	High	High	High	High	High

incompatible conditions that limit the redistribution, modification, or integration of the software. For instance, strong copyleft licenses, such as the GPL, require that any derivative works be licensed under the same terms. This requirement can conflict with other licenses, especially those that are more permissive or do not allow for relicensing under the GPL's terms. Such incompatibilities can prevent the distribution of combined works, necessitating careful consideration and potentially complex legal negotiations Stallman (2002); Moglen (2001).

The matrix in Table 6.1 outlines various reuse scenarios and the corresponding risks of license noncompliance.

We use this rationale in **RQ2** to identify and categorize potential license noncompliance in our copy-based reuse network. We use projects' latest status licenses for this examination. Since both upstream and downstream projects may have multiple licenses, we evaluate all combinations of possible noncompliance to test hypothesis **H2a**. However, there is an aggregation design decision here: how to aggregate possible noncompliance combinations of licenses *with different risk levels* for the same pair of upstream–downstream projects? We consider two options. For a **high sensitivity** approach, we select the highest risk level combination of licenses for a given pair of upstream–downstream projects. Conversely, for a **low sensitivity** approach, we select the lowest risk level combination.

$$\text{Compliance}_{\text{A,B}} = \begin{cases} \max\left(\text{risk}(L_{A_i}, L_{B_j})\right): & \text{High Sen.} \\ \min\left(\text{risk}(L_{A_i}, L_{B_j})\right): & \text{Low Sen.} \end{cases}$$

where:

- L_{A_i}: Each license of Project A,
- L_{B_j}: Each license of Project B,
- $\text{risk}(L_{A_i}, L_{B_j})$: Incompatibility risk level between license L_{A_i} and license L_{B_j}.

For brevity, we present and discuss only the low-sensitivity results below, but include the high-sensitivity results in our replication package, for completeness.

6.4.4 Copy-based vs. Dependency-based Reuse

To test our hypothesis **H2b**, we compare the reuse instances captured via copy-based network with dependency-based network. For this analysis, we focus on high-risk categories in low-sensitivity mode with 10 or more reused blobs—our least conservative scenario—to quantify how often noncompliance is detectable through conventional methods (package manager analysis) versus cases that require copy-based detection. To keep the analysis tractable we selected a sample of 50,000 unique upstream-downstream project pairs from our dataset. Using a stratified sampling, we proportionally selected from each of the 16 high-risk categories, which together represent a total of 82 million projects. To ensure adequate representation of smaller categories, a minimum sample size of 1,000 was enforced, even when the proportional size was smaller. This approach ensures sufficient representation from smaller categories while maintaining overall proportionality. Our final sample included a total of 57,341 project combinations.

Next, we used the maps provided in WoC, which detail all import and export statements in every blob for each commit. By analyzing these maps, we identified

all import/export statements within the projects in our sample[1]. We then matched these statements between upstream and downstream projects to determine if they share any declared dependencies (i.e., the downstream project imports a package that the upstream project exports).

6.4.5 Regression Model

In **RQ1**, we investigate whether the upstream project's license type affects the likelihood of its artifacts getting reused, testing hypotheses **H1a** and **H1b**. Since the response variable is binary (1 if the project has introduced at least one reused blob, 0 otherwise), a logistic regression model is used. It is the standard approach for binary outcomes and enables us to estimate the probability of reuse from various predictors Agresti (2012).

Stratified Sampling

Given the scale and diversity of OSS projects, we employed a stratified sampling approach to ensure that our regression model accurately represents the OSS landscape Thompson (2012). Projects were divided into strata based on six key variables: number of commits, blobs, authors, forks, active months, and earliest commit time. These variables reflect project size, activity, and history, all of which are likely to influence our outcome variables, as discussed in Sec. 6.3 above. The strata were defined as follows: number of commits (fewer than 500, 500–2000, and more than 2000), number of blobs (fewer than 10,000 and more than 10,000), number of authors (one author, 2–10 authors, and more than 10 authors), number of forks (no forks and at least one fork), and active months (fewer than three months and more than three months). Additionally, we categorized projects into four historical eras based on their earliest commit time: the Foundational Era (before 1998), the Dot-com Boom and OSS Expansion (1998–2010), the Maturation and Mainstream Adoption

[1]Analyzed languages: Java, JavaScript, Python, R, Rust, Scala, C#, Go, Groovy, Kotlin, and Perl.

phase (2010–2018), and the Modern Era with a Community Focus (2019–present). This stratification resulted in 288 unique bins. We sampled projects from each bin, yielding a final dataset of approximately half a million projects. While some bins contained fewer projects than anticipated due to uneven distribution, this approach ensures that our sample is representative of the broader OSS ecosystem, allowing for robust and generalizable conclusions from our analyses.

Predictors

Checking for correlations among predictors is crucial in regression models, as multicollinearity—strong correlations between predictors—can distort the results and reduce reliability Dormann et al. (2013). To manage multicollinearity, we applied a 0.6 correlation threshold. Variables with correlations exceeding this threshold indicate overlapping information, and removing them helps mitigate multicollinearity while retaining the most important predictors and their portion of explained variance Vatcheva et al. (2016). The descriptive statistics for the remaining variables are provided in Table 6.2.

Table 6.2: Regression Model - Descriptive Statistics

Variable	**Description**	**Statistics**			
Reuse	Introduced at least 1 reused blob	Yes:	444,144 (77.62%)	No:	128,029 (22.38%)
		5%	**Median**	**Mean**	**95%**
EarliestCommit	Time since the earliest commit	05/08/2006	07/05/2017	01/31/2016	11/23/2021
LatestCommit	Time since the latest commit	04/30/2011	02/17/2020	03/15/2019	04/28/2023
CoreAuthors	Authors with 80%+ of commits	1	2	8.62	17
Forks	Number of forks	0	0	27.66	48
Commits	Number of commits	2	155	2,982.63	5,770
Files	Number of files	5	1,820	17,295.57	59,939.60
AdoptDelay	Earliest commit to license adoption (days)	0	0	133	751
Burstiness	(Latest - Earliest) / Active months	0	1	1.87	6.37

Language	JavaScript	C/C++	Python	Java	PHP	Ruby	(Remaining)
Counts (%)	221,588 (38.72%)	82,551 (14.43%)	53,468 (9.34%)	50,372 (8.80%)	44,952 (7.86%)	18,592 (3.25%)	100,650 (17.59%)

License	No License	Permissive	Copyleft	Weak Copyleft	Conditional Open	Public Domain
Counts (%)	263,974 (46.13%)	148,320 (25.92%)	60,925 (10.65%)	43,143 (7.54%)	30,933 (5.41%)	24,878 (4.35%)

While we removed highly correlated numerical variables to avoid multicollinearity, this approach cannot be directly applied to categorical variables. Therefore, we included interaction terms between two categorical variables—license type and

programming language—in our model to better capture the combined effect of these factors on reuse probability. This approach allows us to account for potential interactions between these variables, offering a more nuanced understanding of how different license types may influence reuse within the context of specific programming languages.

Additionally, we applied sum contrasts for these two predictors, also known as effect coding, where each level of the predictor is compared to the overall mean of all levels. This method allows for a more balanced interpretation of coefficient estimates, by contrasting each category with the overall mean rather than a specific reference category. In sum contrasts, the coefficients for all levels, including the intercept, sum to zero, ensuring that one level's coefficient is determined by the others, thereby maintaining balance and enhancing interpretability in the model.

6.5 Results and Discussion

6.5.1 RQ1 - Regression Model

Our Findings

To establish a baseline, we first modeled the probability of reuse based solely on the project's license type, without considering other potential factors. This initial model showed a significant relationship between license type and reuse likelihood. Specifically, projects with permissive, copyleft, or weak copyleft licenses were more likely to have their artifacts reused, while those with public domain licenses were less likely to be reused.

To assess the impact of the variables with significant coefficients, we examine the odds ratios derived from the logistic regression coefficients. An odds ratio greater than 1 signifies a positive impact, whereas an odds ratio less than 1 indicates a negative impact. Figure 6.1 presents the odds ratios along with their corresponding 95% confidence intervals.

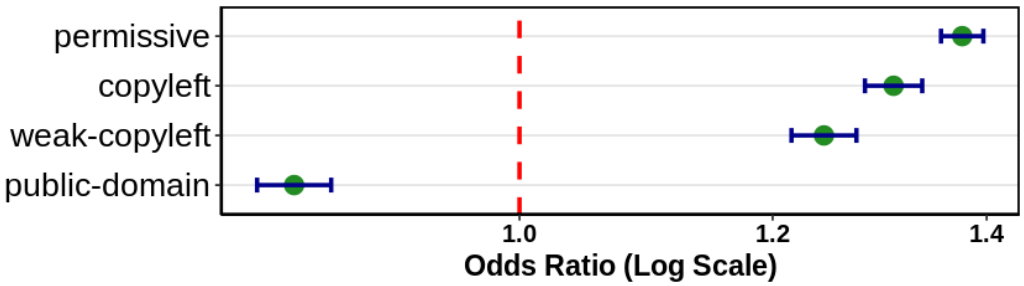


Figure 6.1: Simple Model - Odds Ratios and 95% Confidence Intervals.

Based on these findings, hypothesis **H1a** is partially supported. Projects with permissive licenses, such as MIT and BSD, have higher reuse rates; however, those with public domain licenses do not follow this pattern. Similarly, hypothesis **H1b** receives partial support: while restrictive licenses generally exhibit a lower probability of reuse compared to permissive licenses, they unexpectedly show higher odds of reuse than public domain licenses.

Recall, this initial model does not account for other potential factors that may influence reuse. Consequently, while the preliminary results provide valuable insights, they may be confounded by unconsidered variables. To address this limitation, we introduce a second model incorporating additional control variables, which allows for a more precise analysis of the true impact of license type on artifact reuse.

Table 6.3 presents the ANOVA results for this model, showing that all predictors have highly statistically significant coefficients (p-values close to zero; not surprising given our sample size), and allowing for a comparison of relative explanatory power of each variable (the Deviance column). Almost all control variables had the hypothesized effects, except for burstiness, which seems to be encouraging reuse; however, its deviance is relatively low. The regression coefficient estimates are also shown in this table for non-categorical variables[2]. Note, while a categorical variable may be significant in the model based on ANOVA results, indicating it contributes meaningfully, the coefficients for some individual *levels* of the variable can still be

[2]The p.value ($Pr(>|z|)$) for all this variables are close to zero ($< 2.2e^{-16}$).

Table 6.3: ANOVA Table and Regression Coefficients

	Df	**Deviance**	**Pr(>Chi)**	**Coefficient**
EarliestCommit	1	5,396	$< 2.2e^{-16}$	$5.60e^{-01}$
LatestCommit	1	30,987	$< 2.2e^{-16}$	$-1.49e^{-01}$
CoreAuthors	1	8,143	$< 2.2e^{-16}$	$2.56e^{-01}$
Forks	1	7,994	$< 2.2e^{-16}$	$4.05e^{-01}$
Commits	1	23,749	$< 2.2e^{-16}$	$2.57e^{-01}$
Files	1	65,912	$< 2.2e^{-16}$	$2.80e^{-01}$
AdoptionDelay	1	662	$5.75e^{-146}$	$2.05e^{-02}$
Burstiness	1	128	$1.42e^{-29}$	$6.68e^{-02}$
Language	11	7,710	$< 2.2e^{-16}$	Cat.
License	5	874	$1.20e^{-186}$	Cat.
Language:License	55	1,799	$< 2.2e^{-16}$	Cat.

insignificant. This suggests that, although the variable as a whole impacts the outcome, not every category within it shows a statistically significant effect.

Similarly to the previous model, Figure 6.2 displays the odds ratios and their corresponding 95% confidence intervals for the significant license variables. When additional control variables such as programming language and its interaction with license types are introduced into the model, the results reveal a more nuanced understanding of how these license types influence software reuse. Significant results are observed only in specific combinations of license types and programming languages.

For permissive licenses, Python, C/C++, and JavaScript projects exhibit an odds ratio greater than 1, indicating an increase in reuse. The positive impact of permissive licenses is significant only for these three programming languages, while other languages do not show statistically significant effects.

The first model suggested that public domain licenses are negatively associated with reuse, and the second model confirms that this effect is significant only for JavaScript and Ruby, with no notable impact in other languages. This finding implies that public domain licenses may lack the legal incentives or protections that

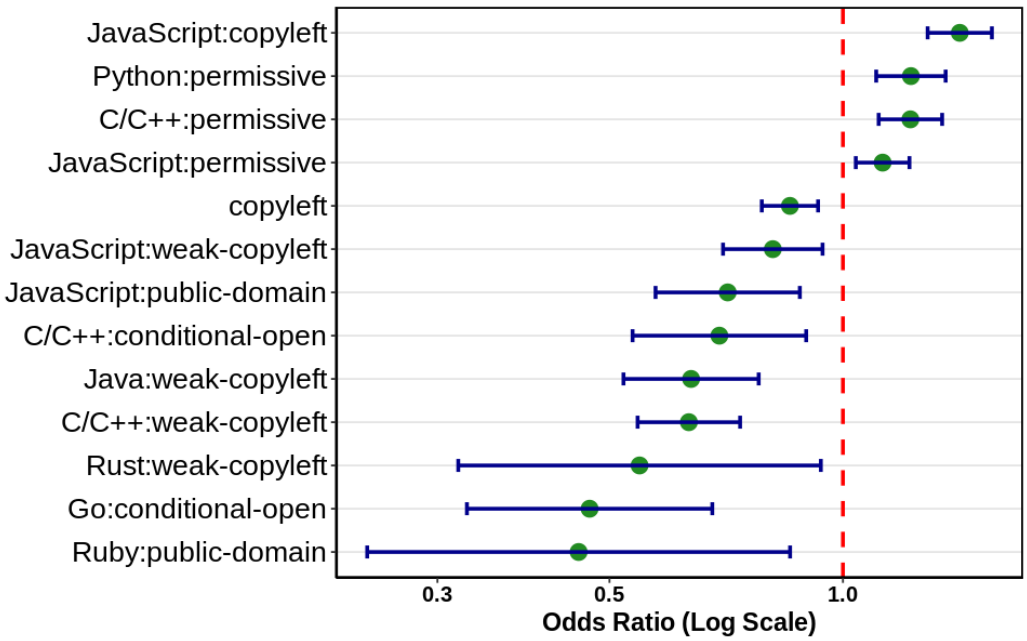


Figure 6.2: Full Model - Odds Ratios and 95% Confidence Intervals.

developers value, making them less attractive for promoting reuse in certain contexts. Hypothesis **H1a** is therefore partially supported.

Projects using permissive licenses show increased reuse in Python, C/C++, and JavaScript, but this effect is not significant in other languages, suggesting that permissive licenses enhance reuse only in specific environments. Furthermore, public domain licenses do not generally impact reuse odds, but reduce the likelihood of reuse in JavaScript and Ruby, contrary to the expectation that more permissive licenses encourage reuse, and thus in contrast to **H1a**.

Several factors may contribute to this unexpected result. One possibility is **legal uncertainty**; the concept of dedicating works to the public domain is not consistently recognized across jurisdictions. In some countries, authors cannot fully waive their copyright, leading to ambiguities that might deter developers from reusing public domain code. Additionally, the **absence of explicit permissions** can create confusion. Although public domain status implies freedom of use, developers and organizations may prefer licenses that clearly state permissions and limitations, such

RQ1 Key Findings

1. Permissive licenses have the strongest positive impact on reuse, particularly in Python, C/C++, and JavaScript projects. (H1a)
2. Public domain licenses show a negative association with reuse, specifically in Ruby and JavaScript projects. (H1a)
3. Copyleft licenses show mixed results: they are beneficial for reuse in specific contexts, such as JavaScript, but generally have a negative effect on reuse when controlling for other factors. (H1b)
4. Weak copyleft licenses reduce reuse only in Rust, C/C++, and Java projects when other factors are considered. (H1b)
5. The influence of license type on reuse is highly dependent on programming language, indicating that license effectiveness varies significantly across different language ecosystems.

as the MIT or BSD licenses, which provide explicit legal reassurances. The perceived **lack of explicit disclaimers or warranties** in public domain software might also make it appear riskier, particularly for commercial use. By contrast, permissive licenses typically include clauses limiting liability and disclaiming warranties, thereby offering additional protections. **Community trust and familiarity** may also play a significant role. Established permissive licenses are widely recognized and trusted, whereas public domain licenses may not enjoy the same level of familiarity or acceptance, leading developers to favor more well-known licensing options.

For copyleft licenses, the overall effect is negative. However, JavaScript projects under such licenses exhibit an odds ratio greater than 1, suggesting that the effect of copyleft licenses varies significantly depending on the language. Weak copyleft licenses also show negative impacts on reuse for JavaScript, Java, C/C++, and Rust projects. These findings suggest that hypothesis **H1b** is also partially supported. Although copyleft licenses generally reduce the probability of reuse, this is not the case for all programming languages. Moreover, weak copyleft licenses reduce reuse only in specific languages.

Implications

A key takeaway is that the choice of license for a project has a substantial impact on the likelihood of its artifacts being reused. This effect varies across different license types and programming languages, highlighting nuanced relationships between license choice, programming language, and reuse behavior. This indicates that developers and contributors should be mindful of how their choice of license can influence the adoption and reach of their work.

One of the most unexpected findings is that public domain licenses, designed to allow free and unrestricted reuse, have a negative effect on reuse. This is concerning because the intent behind these licenses is to eliminate barriers, yet the data suggest the opposite. The negative association of public domain licenses with reuse indicates that the OSS community may need to address this unintended outcome. One way forward is to enhance awareness and education about public domain licensing, clarifying the legal protections and reuse rights it offers. Clearer guidance on how public domain licenses differ from other open source licenses, particularly regarding legal clarity and potential liability, could benefit OSS contributors, especially newcomers. The community might also consider providing stronger legal frameworks or support around public domain licenses to reduce uncertainties and hesitations. Project maintainers may also reconsider using public domain licenses if their primary goal is to maximize reuse. The data suggest that permissive licenses may be more effective in promoting reuse.

In conclusion, while the OSS movement encourages reuse and collaboration, these results show that the choice of license plays a crucial role in determining whether a project achieves those goals. The community must be attentive to the barriers that certain licenses, such as public domain, may unintentionally create and take steps to provide better education, support, and legal frameworks to ensure that the intentions behind these licenses are effectively realized in practice.

6.5.2 RQ2 - Noncompliance

Our Findings

As discussed above, we report only the results of our low-sensitivity aggregation here (i.e., considering the lowest-risk pairs of licenses for a given upstream–downstream pair of projects). Figures 6.3 and 6.4 summarize our findings for the two flavors of reuse we consider (complete reuse, with at least one shared blob, and substantial reuse, with 10 or more blobs).

At least 1 Reused blob Figure 6.3 highlights the top 10 categories of license combinations between upstream and downstream projects, showcasing the most frequent pairings. The pie chart illustrates the distribution of project tuples across three categories: no issues, high-risk potential, and low-risk potential for license noncompliance.

The results indicate that a significant majority (55%) of upstream-downstream license combinations fall into the high-risk category. The most common high-risk scenario occurs when neither the upstream nor downstream projects have a license, accounting for 605 million project tuples. This creates legal uncertainty regarding reuse, modification, and distribution rights. Other high-risk combinations within the top 10 involve cases where one project lacks a license, such as no-license to permissive. Even when the upstream project has a clear license, the absence of a downstream license introduces ambiguity and potential legal challenges.

On the positive side, 30% of the tuples present no issues, such as permissive to permissive combinations, where both upstream and downstream projects are clearly licensed, minimizing legal risk. Low-risk combinations make up 14%, including cases like permissive to no-license, which involves some legal uncertainty but is less risky than high-risk scenarios.

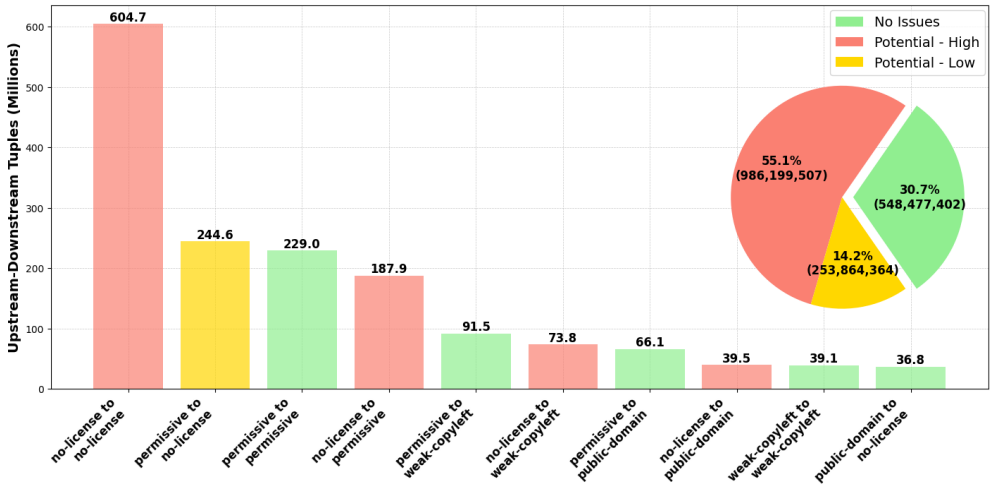


Figure 6.3: Top 10 License Types - 1 Reused Blob, Low Sensitivity

At least 10 Reused blobs The total number of reuse instances (unique combinations of upstream and downstream projects) drops significantly from 1.816 billion to 212 million after applying the constraint of at least 10 reused blobs—a reduction of approximately 88%. This sharp decline indicates that the majority of earlier reuse cases involved fewer than 10 blobs, suggesting that much of the initial reuse was minimal or partial. This reduction highlights that a significant portion of copy-based reuse in the open source ecosystem is small-scale or potentially superficial, involving limited sharing between projects, with fewer instances of deeper, substantial dependencies. By focusing on reuse instances involving at least 10 reused blobs, the data now captures more meaningful relationships, wherein downstream projects are more closely integrated with upstream codebases.

Although the number of high-risk combinations decreases proportionally from the earlier results, they still account for 39% of the remaining reuse instances (see Figure 6.4). This indicates that even in cases of more substantial reuse, issues related to licensing or lack of clear licensing persist. However, the increase in the no-issues category to 48%, primarily driven by permissive to permissive license reuse, suggests that when significant reuse occurs, clearer licensing tends to be in place, especially for permissive licenses.

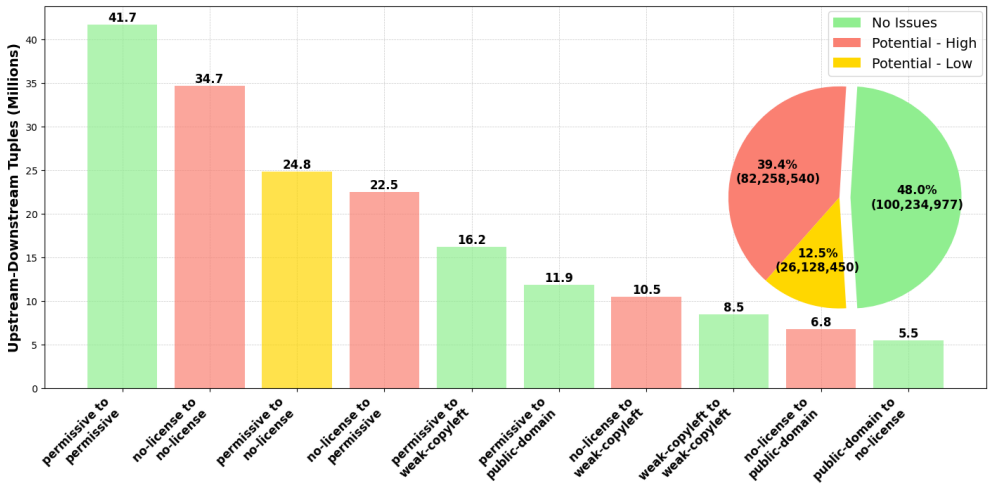


Figure 6.4: Top 10 License Types - 10 Reused Blobs, Low Sensitivity

Overall, these findings support our hypothesis **H2a** and underscore the critical importance of proper licensing.

Copy-based vs. Dependency-based Reuse The results of comparing reuse detected via copy-based network and dependency-based network are presented in Table 6.4.

The results highlight a significant limitation in current dependency detection tools, showing that the percentage of code reuse detected through declared dependencies is remarkably low across all categories. Despite analyzing over 57,000 project combinations, the overall detection rate of code reuse through formal dependency relationships was only 2.43%. This suggests that traditional methods relying on package managers, which track declared imports and exports between projects, are insufficient for capturing most instances of code reuse, supporting our hypothesis **H2b**.

Implications

These findings have significant implications for the open-source community, particularly in relation to license compliance and code reuse detection. The results reveal that

Table 6.4: Reuse Detectable by Dependency Relationship

License Type	Sample Size	Decl. Dep.	Percent
no-license-2-no-license	21,102	499	2.36%
no-license-2-permissive	13,670	346	2.53%
no-license-2-weak-copyleft	6,357	94	1.48%
no-license-2-public-domain	4,107	93	2.26%
copyleft-2-no-license	1,105	48	3.35%
conditional-open-2-conditional-open	1,000	20	2.00%
conditional-open-2-copyleft	1,000	20	2.00%
conditional-open-2-no-license	1,000	40	4.00%
conditional-open-2-weak-copyleft	1,000	18	1.80%
copyleft-2-conditional-open	1,000	19	1.90%
copyleft-2-permissive	1,000	36	3.60%
copyleft-2-public-domain	1,000	36	3.60%
copyleft-2-weak-copyleft	1,000	14	1.40%
no-license-2-conditional-open	1,000	34	3.40%
no-license-2-copyleft	1,000	43	4.30%
weak-copyleft-2-no-license	1,000	36	3.60%
Total	**57,341**	**1,396**	**2.43%**

a majority of upstream-downstream project combinations are classified as high-risk for potential noncompliance, underscoring a persistent issue in open-source software development. The high occurrence of high-risk cases, especially in projects with no license, highlights a potential legal vulnerability that could impact the sustainability and collaboration within the open-source ecosystem.

This findings also call for more advanced detection techniques that go beyond traditional dependency analysis. Tools that can detect code reuse through copying are essential for identifying non-compliance with licensing terms. The low detection rates across the board demonstrate that current tools are not capable of providing a complete picture of how code is reused, and more comprehensive approaches are necessary to ensure effective license compliance monitoring.

RQ2 Key Findings

1. A significant portion of upstream-downstream project combinations are classified as high-risk for potential license noncompliance, leading to considerable legal uncertainties regarding reuse, modification, and distribution rights. (H2a)
2. The most common high-risk potential noncompliance scenario involves projects lacking any license, underscoring a legal vulnerability within the open-source community and highlighting the urgent need for consistent and clear licensing practices.
3. Dependency tracking is inadequate for detecting most instances of code reuse, highlighting the need for more granular detection methods capable of identifying copy-based reuse that would enable more accurate license compliance monitoring in open-source projects. (H2b)

6.6 Limitations

6.6.1 Internal Validity

Project to License Map

The project to license map (P2L) in WoC relies on detecting license files in repositories, assuming licenses are always recorded in dedicated files.Nevertheless, licenses might appear in README or source files, leading to underreporting or misclassification. This suggests that results should be interpreted cautiously, and additional manual verification may be needed for a more accurate understanding of license noncompliance.

License Scope

Assigning a license to an entire OSS project can introduce challenges, as the license may not uniformly apply to all components. Projects often incorporate third-party libraries, modules, or contributions that come with their own distinct licenses, which may conflict with or restrict the applicability of the main project license. Thus, while

the project may be licensed under a specific open-source framework, that license may only cover certain parts, with other components subject to different licensing terms.

Dependency-Based Reuse

One limitation in comparing copy-based reuse with dependency-based reuse is that some projects use dynamic or implicit imports, where dependencies are loaded at runtime or through unconventional methods that may not be captured by a straightforward export-import analysis. This can result in certain dependencies, which package managers can detect, being overlooked, exposing gaps in our approach. Nonetheless, our methodology is conservative, as we track dependencies over time rather than focusing solely on the latest version. By excluding any reuse instance that was detectable through dependencies at any point in the project's history, we provide a more thorough view of potential dependency-based reuse. This approach reduces the risk of missing past dependencies that may have been removed or modified in subsequent versions, delivering a more inclusive analysis of reuse instances. However, this conservatism may also lead to attributing reuse to dependencies that no longer exist, slightly skewing the results toward historical dependency detection.

6.6.2 External Validity

Copy-Based Reuse

While emphasizing copy-based reuse offers valuable insights into license compliance, we recognize the significant role of dependency-based reuse within the broader reuse network. Focusing solely on copy-based reuse may overlook certain aspects of how dependencies are integrated into a project. Conversely, dependency-based reuse can miss critical instances where code is directly copied between projects, which is equally crucial in identifying potential noncompliance. Thus, while this work prioritizes copy-based reuse, it serves to complement—rather than replace—the understanding gained

from analyzing dependency-based reuse, together providing a more comprehensive view of compliance.

6.7 Conclusions

Our study shows that the choice of open-source license plays a significant role in influencing the likelihood of reuse. Permissive licenses consistently encourage reuse across a variety of programming languages, while copyleft and weak copyleft licenses exhibit more context-specific effects, sometimes limiting reuse depending on the language and environment. Despite offering unrestricted reuse, public domain licenses were linked to a negative impact on reuse, likely due to legal uncertainties. Our findings also emphasize the importance of detecting copy-based reuse, as traditional dependency-based approaches often fail to capture the full scope of reuse, especially when explicit dependencies are not declared. This highlights the need for more advanced detection methods to improve license compliance monitoring in the open-source ecosystem. Moreover, projects without clear licenses continue to present significant legal risks, underscoring the need for more consistent and transparent licensing practices within the open-source community.

Chapter 7

Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets

Disclosure Statement

A version of this chapter is accepted to be published as Jahanshahi and Mockus (2025):

Mahmoud Jahanshahi and Audris Mockus. 2025. **Cracks in The Stack: Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets**. In *Proceedings of the second International Workshop on Large Language Models for Code. (LLM4Code '25)*. Just Accepted (January 2025).

Replication package available at: https://zenodo.org/records/14175945

7.1 Abstract

A critical part of creating code suggestion systems is the pre-training of Large Language Models (LLMs) on vast amounts of source code and natural language text, often of questionable origin, quality, or compliance. This may contribute to the presence of bugs and vulnerabilities in code generated by LLMs. While efforts to identify bugs at or after code generation exist, it is preferable to pre-train or fine-tune LLMs on curated, high-quality, and compliant datasets. The need for vast amounts of training data necessitates that such curation be automated, minimizing human intervention.

We propose an automated source code autocuration technique that leverages the complete version history of open-source software (OSS) projects to improve the quality of training data. The proposed approach leverages the version history of all OSS projects to: (1) identify training data samples that have ever been modified, (2) detect samples that have undergone changes in at least one OSS project, and (3) pinpoint a subset of samples that include fixes for bugs or vulnerabilities. We evaluate this method using "The Stack" v2 dataset, comprising almost 600M code samples, and find that 17% of the code versions in the dataset have newer versions, with 17% of those representing bug fixes, including 2.36% addressing known CVEs. The clean, deduplicated version of Stack v2 still includes blobs vulnerable to 6,947 known CVEs. Furthermore, 58% of the blobs in the dataset were never modified after creation, suggesting they likely represent software with minimal or no use. Misidentified blob origins present an additional challenge, as they lead to the inclusion of non-permissively licensed code, raising serious compliance concerns.

By deploying these fixes and addressing compliance issues, the training of new models can avoid perpetuating buggy code patterns or license violations. We expect our results to inspire process improvements for automated data curation, a critical component of AI engineering, with the potential to significantly enhance the quality and reliability of outputs generated by AI tools.

7.2 Introduction

Large Language Models (LLMs) are already employed by popular tools such as GitHub Copilot and have a significant impact on how people interact with computing resources. LLM code-generation tools appear to increase productivity Ziegler et al. (2022), are easy to access with little or no cost on popular coding platforms, and generated code is rapidly spreading ("GitHub Copilot is behind an average of 46% of a developer's code" Zhao (2023)). Quality control of this code, however, is severely lacking in the LLM-based Software Supply Chain (SSC). LLMs are trained on vast amounts of source code and natural language text that are of questionable origin and quality. The output generated by LLMs, therefore, often contains bugs, vulnerabilities, or license violations that are copied or reused to train other LLM models, thus propagating the problem. Hubinger et al. (2024) showed that LLMs can introduce vulnerabilities and this behavior is extremely difficult to change via fine-tuning. It is reasonable to assume that at least part of that buggy output may be attributed to the buggy files used to train LLMs. While existing approaches use AI to detect the most common insecure coding patterns Zhao (2023), but many vulnerabilities do not fit such simple patterns. It is widely accepted that the size and quality of training corpus are essential for good performance of the models, yet common curation techniques, such as number of stars or forks, appear ineffective Allal et al. (2023). Independent of the intended coding tasks, a large body of training data is necessary for LLMs to be effective. As poor quality training data can reduce the quality of LLM-based tools, improving the state of art in source code training data curation is an important task that would impact all downstream efforts. It is worth noting that source code is often included in training data for natural language models as well. For example, the natural language collection in Laurençon et al. (2022) has hundreds of gigabytes of source code and collection described in Gao et al. (2020) nearly 100GB.

Previous work found instances of vulnerable or license-violating code in open source training datasets. This shows that by taking information from version control systems, it is feasible to identify vulnerable, buggy, or license-violating code and replace it with fixed versions Reid et al. (2022); Reid and Mockus (2023).

In summary, it is essential to exclude problematic code from LLM training datasets, or, at least, to flag it as high risk.

The goals of this work is to investigate the quality of the source codes that are used to train LLMs and to develop automated approaches to improve it. Specifically, we propose a simple and effective way to identify (and fix) several types of problematic source code that is used to train LLMs.

In a nutshell, we leverage the fact that a file's content may undergo numerous changes over its lifetime, with some of these changes being bug fixes. By identifying cases where a file in the training data has been modified and updated, we can recommend these newer versions as replacements for older versions in the training dataset. In order for this approach to work, we have to go across repository boundaries and consider versions (and their history) in all public repositories, i.e., Universal Version History (UVH) Reid and Mockus (2023). World of Code (WoC) research infrastructure Ma et al. (2019, 2021) provides capabilities to accomplish such an arduous task as described in Section 7.4.

Our primary contributions are: 1) an approach to identify potentially vulnerable, buggy, or not heavily used source code in public LLM training datasets; 2) an approach to identify potential license violations in these datasets; and 3) evaluation of the approach on the largest public curated code LLM training dataset the Stack v2 Lozhkov et al. (2024). We also articulate how code LLM's represent a novel type of software supply chains and suggest that never-modified code may indicate its low use and untested quality and that should be taken into account when constructing training datasets.

In the remainder of the paper Section 7.3 discusses curated training datasets used for evaluation, relevant key concepts of software supply chains, how LLM-generated

coder represents a novel type of software supply chain, and key features of WoC used in this study. Section 7.4 describes our approach in detail. Section 7.5 presents and discusses our findings.

7.3 Background

7.3.1 Types of Software Source Code Supply Chains

Software supply chain concept is helpful for assessing risks, as in traditional supply chains. However, software supply chains have substantially different nature from traditional supply chains. In particular, three types of software source code[1] supply chains have been previously identified Mockus (2019b). The most common, or Type I SSC is represented by code (runtime) dependencies. For example, an import statement in Java or include statement in C programming languages. The two primary risks for downstream projects in this scenario are: insufficient upstream maintenance, where bugs and vulnerabilities remain unresolved, and overly aggressive maintenance, where upstream changes disrupt downstream code Xavier et al. (2017).

Type II SSC involves copied code, a common practice in open-source software where code is shared publicly Jahanshahi et al. (2024b), allowing anyone to copy or fork it (within licensing requirements). While breaking changes are no longer a risk in Type II SSC, the absence of upstream maintenance becomes inevitable, as the code is now maintained within the destination project.

Type III SSC involves knowledge transfer where developers learn procedures techniques and tools by working in one project and then apply some of what they learned elsewhere. While learning, in general, is a good thing, some quality practices or API usage may introduce bugs or vulnerabilities that, if adopted by developers, are then spread by these developers to other projects.

[1]We explicitly exclude various ways binary software is delivered as, for example in Solar Winds hack.

The current state of the industry in source code SSCs is to capture dependencies based on package managers (Type I SSCs) and to rely on the "official" directories such as NVD and package managers to identify the security and licensing attributes. As was shown in Reid et al. (2022); Reid and Mockus (2023), rampant code copying enabled and encouraged by OSS results in massive orphan vulnerabilities and licensing violations that cannot be detected by existing approaches.

7.3.2 The Promise and Challenges of Large Code Datasets

Large-scale code datasets are invaluable for advancing AI-driven code solutions, such as automated code generation, bug detection, and refactoring. These datasets provide extensive repositories of programming languages, styles, and structures, enabling large language models (LLMs) to learn complex coding patterns and generalize across diverse coding tasks. By leveraging such data, AI models significantly improve in generating, completing, and correcting code, which supports developers in accelerating the software development cycle and reducing costs Allamanis et al. (2018); Lozhkov et al. (2024).

However, maintaining the quality and integrity of these large datasets poses several challenges, often underexplored in research. Duplication, for instance, can lead to redundancy, creating biases and reducing model diversity. Version control is another critical challenge, as datasets sourced from dynamic platforms like GitHub may frequently change; without careful version tracking, models risk learning outdated or deprecated practices. Provenance tracking is essential for maintaining the contextual relevance and reliability of data, allowing users to trace the origins and evolution of code snippets. Additionally, licensing complexities arise, as open-source code often comes with a range of permissive and restrictive licenses. Properly handling these licensing issues is crucial to ensuring lawful usage, especially in commercial settings Gunasekar et al. (2023).

LLMs introduce a novel type (Type IV) of Software Supply Chains that manifest by relationships between the LLM-generated code and the code used to train the LLM models. LLMSSCs, similar to Type II SSCs, are conceptually copying the code (including its bugs) in the training data but in a way that obfuscates the origin. The full scope of risks posed by Type II copy-based SSCs has yet to be studied in depth.

7.3.3 The Stack v2 Dataset

To evaluate our approach we use a large open source dataset intentionally curated for training code LLMs: the Stack v2 Lozhkov et al. (2024). "The Stack v2 contains over 3B files in 600+ programming and markup languages. The dataset was created as part of the BigCode Project , an open scientific collaboration working on the responsible development of Large Language Models for Code (Code LLMs). The Stack serves as a pre-training dataset for Code LLMs, i.e., code-generating AI systems which enable the synthesis of programs from natural language descriptions as well as other from code snippets."

This dataset is widely adopted in AI and software development due to its extensive multi-language coverage and permissive licensing, enabling use in both academic and commercial contexts. The Stack (v2) fosters open collaboration, supporting model training across diverse coding ecosystems and advancing tools for software automation and analysis Gunasekar et al. (2023).

In addition to the full dataset, the Stack v2 has several deduplicated versions. the-stack-v2-dedup is near-deduplicated, the-stack-v2-train-full-ids is based on the the-stack-v2-dedup dataset but further filtered with heuristics and spanning 600+ programming languages. Finally, the-stack-v2-train-smol-ids is based on the the-stack-v2-dedup dataset but further filtered with heuristics and spanning 17 programming languages. We evaluate our fixing approach on the **full** and **smol** (maximally deduplicated) datasets[2].

[2]For more details on the dataset and the deduplication process, refer to the Stack v2 documentation: https://huggingface.co/datasets/bigcode/the-stack-v2

7.3.4 Motivation for This Study

Evaluating large code datasets is essential to address the intricacies of version security and licensing, which collectively impact the reliability and ethical compliance of large language models (LLMs) for code.

Security Vulnerabilities and Bugs

The security implications of large datasets are significant, especially in the context of outdated or vulnerable code. If models are trained on datasets containing undetected security flaws, these vulnerabilities may persist in model outputs, increasing the risk of insecure code suggestions. This issue is particularly concerning for code used in sensitive applications, where even minor security oversights can lead to substantial risks and exploitation potential. Security-focused dataset evaluation is therefore vital to prevent models from inadvertently embedding insecure practices into their code outputs Pearce et al. (2022).

Given the large-scale, open-source nature of The Stack v2 dataset, it is likely to contain instances of vulnerable and buggy code. This hypothesis (**H1**) is based on the prevalence of "orphan vulnerabilities" in open-source projects, as described by Reid et al. (2022), where vulnerabilities in copied code persist even after they are patched in the original source. In large datasets aggregated from numerous repositories, code reuse without consistent patching introduces security risks, as outdated or unpatched code versions may proliferate across projects, spreading known vulnerabilities further.

- **Hypothesis 1 (H1)**: The Stack v2 dataset is likely to contain instances of vulnerable and buggy code.

Legal Considerations

Maintaining licensing integrity is fundamental for the lawful and ethical deployment of code-based AI. The provenance and licensing of code samples in these datasets must be meticulously tracked to prevent legal risks associated with licensing

misrepresentation or inaccurate attributions. Open-source projects often involve significant code reuse, which can lead to fragmented metadata or altered licensing information as code is copied across projects. Proper licensing ensures that the models' outputs respect open-source constraints, which is crucial for both research and commercial applications. Without rigorous checks, models might generate code based on improperly licensed data, exposing end-users to compliance issues and potential litigation. Ensuring that datasets uphold licensing integrity not only fosters ethical AI but also protects users from unforeseen legal complications Gunasekar et al. (2023).

Due to the prevalence of "copy-based reuse" in open-source development, as explored by Jahanshahi et al. (2024b), we hypothesize (**H2**) that The Stack v2 dataset contains instances of misidentified code origins. While the dataset has metadata identifying the project from where each source code file was obtained, that file may have been copied from another project that has a different or even incompatible license. This form of reuse, where source code is directly copied into new projects, often results in fragments with altered or lost metadata, which complicates the ability to accurately trace their provenance. This lack of provenance tracking can lead to legal and ethical issues in AI applications for code. Without accurate metadata, models may inadvertently generate code with improper licensing, exposing users to potential compliance issues. Misidentification of code origins in datasets like The Stack v2 is particularly risky for industry applications, as it challenges the trustworthiness and lawful deployment of LLM4Code models in commercial environments.

- **Hypothesis 2 (H2)**: The Stack v2 dataset is likely to contain instances of misidentified code origins that are prone to license violation.

7.3.5 Contributions

The primary contributions of this paper focus on addressing data quality and compliance concerns within The Stack v2 dataset. The paper aims to enhance the

understanding and reliability of large code datasets by providing the following key contributions.

Assessment of Security and Reliability

We introduce a novel methodology for identifying source code that may be potentially vulnerable, contain bugs, or exhibit minimal usage in real-world applications. Our approach uniquely incorporates version control history to track and analyze the evolution of source code, focusing on identifying newer versions of files that indicate updates, bug fixes, or refinements over time. By examining commit histories and versioning patterns, we can detect files that have undergone improvements or corrections, flagging older versions as potentially vulnerable or buggy. This historical perspective provides insight into code stability and usage trends, allowing us to differentiate actively maintained, reliable code from outdated, less robust sections.

Analysis of Code Provenance and Licensing Accuracy

We conduct a detailed examination of code provenance to evaluate licensing accuracy and the origins of code snippets within the dataset. By tracking the source and licensing status of code entries, we provide a comprehensive assessment of compliance with open-source licensing requirements. This contribution is particularly important for models deployed in industry, where legal and ethical use of data must be assured.

Evaluation on Large-Scale Code Dataset

To validate the effectiveness of our approach, we perform a comprehensive evaluation on the largest publicly curated code LLM training dataset, Stack v2. This dataset serves as an ideal benchmark due to its scale and diversity. By applying our methodology to Stack v2, we can assess the robustness of our techniques in identifying potentially vulnerable or outdated code segments, accurately tracking version histories, and verifying licensing compliance across a large and varied dataset.

This evaluation establishes the applicability and scalability of our contributions to real-world, large-scale code datasets, reinforcing the value of our work in supporting the development of secure, high-integrity LLM training corpora.

7.4 Methodology

To address big data-related aspects of the proposed work, we leverage WoC research infrastructure Ma et al. (2019, 2021) for open source version control data. This data includes a vast majority of public open source projects and provides access to petabytes of data that includes versions of source code, information on time, authorship, and exact changes made to the source code over the entire activity history of most participants in OSS.

7.4.1 Key Concepts

The proposed method for identifying issues in training data leverages unique capabilities of WoC. In particular, WoC's ability to cross-reference and track the history of code versions across nearly all public repositories, along with its curated data that addresses complex challenges like repository deforking Mockus et al. (2020) and author ID aliasing Fry et al. (2020), makes this approach feasible.

We use a simple example to demonstrate the tracing and cross-referencing capabilities of WoC. Suppose we take a single sample b (version or, in git terms, blob) of source code from any training (or test) data. We can calculate git SHA-1 [3] for this sample. All further calculations use git SHA-1 and do not require the content.

For a blob b to materialize in a version control repository, it has to be created by a commit c. Git commits include the time of the commit, commit message, SHA-1 of the parent commit(s) and SHA-1 of the tree (folder). WoC, by comparing the trees[4]

[3]Git SHA-1 is simply a SHA-1 calculated on the string (representing the content) with prepended string "blob SIZE\0" where SIZE is the length of the content.

[4]WoC contains over 20B blobs.

of the commit and its parent(s) determines all the modifications to the project done by the commit. Specifically, in case any of the project's files are modified, it extracts the tuple (b_o, b_m) representing the old and the new version of the file. These pairs are associated with the commit and its other attributes, like time, author and commit message.

Suppose there is a commit, $c_t(b_o, b_m)$, which addresses a vulnerability v in project P. This commit, c, modifies a file f at time t, where the original version of the file is represented by the blob b_o and the modified version by b_m. WoC's cross-referencing allows us to identify all repositories containing b_o or b_m, all relevant commits, their parent and child commits, and the authors and projects associated with these commits.

Typically, we need a repository and a commit to identify what files were changed, their content before and after the change, as well as the parent commit. By collecting and cross-referencing nearly all open source data, WoC allows us not only to go forward in version history (see child commits), but also to identify all commits that either created or modified a particular version of the file. To identify problems with the LLM training data, we will first match it to blobs or commits in WoC. Both the Stack and the Stack v2 contain versions of the files (blobs) and their git SHA-1 digests. We, therefore, just need the list of SHA-1 digests to match them to blobs in WoC. We further assume that if there exists at least one commit that modifies b_o, and its commit log message contains keywords (described below) indicating that it is a fix, then that blob is buggy. Similarly, if the commit indicates that it fixes a vulnerability, we assume that modified blob contains vulnerability.

7.4.2 Identifying Potential Noncompliance

The Stack dataset provides information on repositories and their identified licenses for all blobs. Since code reuse through copying is common among developers Jahanshahi et al. (2024b), accurately tracing the originating projects for each blob can be

challenging. WoC addresses this by offering a map Jahanshahi and Mockus (2024) that, for blobs found in multiple projects, sorts them by the commit time of each blob's creation, allowing us to identify its first occurrence and the repository where it was initially committed. By comparing this origin information from WoC with the data in the Stack, we can verify whether the originating repository of each blob has been accurately identified.

If the origin identified by WoC does not match the origin listed in the Stack data, we then analyze the licenses associated with both the WoC-identified originating repository and those detected by the Stack. Using WoC's license map Jahanshahi et al. (2024a), we compare this information with the Stack's license data to identify potential instances of license noncompliance.

7.4.3 Sampling

We used a $\frac{1}{128}$th sample for certain quantitative analyses to balance computational feasibility with representativeness. The sampling was based on SHA-1 hashes of the blobs and commits, which ensures that the selection process is effectively random. This approach maintains statistical robustness while significantly reducing the computational overhead of processing the entire dataset.

7.5 Results and Discussions

7.5.1 Hidden Vulnerabilities

As described in Section 7.4, we first extract git SHA-1 for all blobs in the Stack v2 (*full*) and the-stack-v2-train-smol-ids (*smol*) datasets. The former has 582,933,549 and the latter has 87,175,702 unique blobs. The total number of blobs in WoC version V3 (extracted at about the same time as the Stack v2) has over 26B blobs, or almost 45 times more blobs than the full version and 300 times more than the small deduplicated version.

Starting from these two lists of blobs[5] we first obtained two maps to commits: the first map links blobs to commits creating the blob (including the previous version of the file), while the second map links to commits that modified the file, thereby creating a new blob, as described in the previous section. Not all blobs could be mapped to commits, as a small fraction did not appear in either map. This could be due to certain code versions being created without a publicly accessible version history or missing corresponding commits or trees in WoC.

Table 7.1 summarizes the blob counts for two evaluation datasets, based on a $\frac{1}{128}$th random sample determined by the SHA-1 hash of each blob. These counts can be extrapolated to the full dataset by multiplying by 128.

From Table 7.1, we observe that approximately 2.5% of the blobs could not be linked to any commits. Among the remaining blobs, 62% and 55% represent files that were created without preceding blobs, i.e., they are the initial versions. Of these, only 5.5% and 4.6% had a newer version, meaning the majority were created but never modified. Since the first version of frequently executed source code is rarely error-free, this lack of updates suggests the code was likely not used in practice, raising concerns about its overall quality.

Furthermore 17.3% and 10.2% of the blobs have a subsequent version(s). These versions are likely fixing existing bugs, vulnerabilities, make code compatible with newer versions of libraries, or add new functionality. Since the next version of the code is known, it would make sense to replace the versions of the training data with updated versions.

We further analyze the blobs that have been updated. Using the methodology described in Mockus and Votta (2000), we identify likely bug fixes by searching

[5]The second list had only 26% overlap with the first list instead of being a strict subset of the first.

Table 7.1: Counts in the blob sample

		full		smol	
		count	% (row)	count	% (row)
1	**Total**	**4,553,119**		**680,917**	
2	Missing	115,239	2.53 (1)	16,533	2.42 (1)
3	**Have an old version**	1,622,641	35.63 (1)	287,412	42.20 (1)
4	**First version**	2,813,171	61.78 (1)	376,719	55.32 (1)
5	No new version	2,658,805	94.51 (4)	359,380	95.39 (4)
6	**Have a new version**	788,059	**17.30** (1)	69,346	**10.18** (1)
7	Found new versions	1,462,363	-	111,453	-

for terms *fix*, *bug*, *issue*, *patch*, *error*, *resolve*, *correct*, *problem*, and their common variations, as well as *cve* in the commit messages[6].

The results are shown in Table 7.2. It summarizes the counts for two evaluation datasets, based on a $\frac{1}{128}$th random sample determined by the SHA-1 hash of each commit that introduces a new version for a blob in the Stack dataset. These counts similarly can be extrapolated to the full dataset by multiplying by 128.

Among the 5,068,635 blobs with newer versions, we find that 17.31% and 14.36% of the blobs were updated by a fix commit. If we extrapolate the results, we see that in total, 101M blobs in the current *full* Stack v2 database (representing 17.30% of all blobs in it) can be updated to newer versions and 17.31% of these new versions are bug fixes. For the *smol* dataset, we have 9M (representing 10.18% of all blobs in it) that can be updated to newer versions and 14.36% of those are bug fixes. While deduplication reduced the proportion of buggy samples, millions of them still remain and can be easily fixed.

[6] `grep -iwE 'fix | fixes | fixing | bug | bugs | issue | issues | patch | patches | error | errors | resolve | resolved | resolving | correct | corrects | corrected | correcting | problem | problems | debug | debugs | debugged | debugging | cve'`

Table 7.2: Counts in the new version commit sample

		full		smol	
		count	% (row)	count	% (row)
1	Commits	835,699		104,782	
2	Blobs	5,068,635		279,652	
3	New versions	5,657,384		307,362	
4	Fix commits	137,091	16.40 (1)	13,628	13.00 (1)
5	Fix blobs	877,811	**17.31** (2)	40,168	**14.36** (2)
6	Fix new versions	935,587	16.53 (3)	41,222	13.41 (3)
7	CVE commits	845	0.61 (4)	83	0.60 (4)
8	CVE blobs	20,765	**2.36** (5)	756	**1.88** (5)
9	CVE new versions	20,561	2.19 (6)	809	1.96 (6)
10	**Distinct CVEs**	**851**		**78**	

Table 7.3: CVE counts in complete smol dataset

	CVE commits	CVE blobs	Distinct CVEs
Count	11,907	19,944	6,947

Finally, we checked how many code sample have fixes to known vulnerabilities. To do that we searched for the regular expression representing CVE "cve-[0-9]+-[0-9]+" and found that 2.36% and 1.88% of the fixes in our sample relate to a known CVE.

Due to the important nature of known vulnerabilities, we further analyzed the complete *smol* dataset—that is supposed to be most reliable version of the Stack v2—to find blobs that have a newer version with fixes to known CVEs. The results are shown in Table 7.3. We found that 19,944 blobs in the *smol* dataset have newer versions that fixing a known CVE. These samples were changed by 11,907 commits that mentioned 6,947 distinct CVEs in their commit message.

In summary, despite careful curation and employment of sophisticated heuristics, even the clean version of the Stack v2 dataset contains millions of unfixed versions of the code, including thousands of unfixed vulnerabilities that supports our first hypothesis (**H1**).

Key Findings 1

1. 17.30% and 10.18% of blobs in the *full* and *smol* datastes, respectively, have newer versions, out of which 17.31% and 14.36% are bug fixes.

2. 61.78% and 55.32% of blobs are the first version created, out of which 94.51% and 95.39% have no newer versions, meaning they were created but never modified, suggesting low quality.

3. There are 19,944 blobs in the clean and deduplicated version of the Stack v2 (*smol*) that have a newer version were a known security vulnerability is being fixed.

4. In total, 6,947 known CVEs has been found in the *smol* dataset.

7.5.2 Potential Noncompliance

The Stack v2 dataset consists of code that is either licensed under permissive terms or lacks a specified license. To address potential licensing concerns, the Stack v2 allows authors to opt out of inclusion in the dataset. It is important to note that code without a license is distinct from unlicensed code. From a copyright perspective, code without a license defaults to "all rights reserved" U.S. Copyright Office (2021), which raises significant concerns about the inclusion of such code in this dataset.

As detailed in Section 7.4.2, we analyzed blobs within the dataset that were reused across multiple OSS projects, as identified through WoC Jahanshahi and Mockus (2024). For each blob, we determined its originating project—the project with the earliest commit timestamp containing that blob—and cross-referenced it with the corresponding project in the Stack dataset. The results are shown in Table 7.4.

The results indicate that 15.49% and 11.30% of blobs were reused at least once. Furthermore, in 67.42% and 61.78% of instances, the originating projects identified by the Stack dataset differ from those identified by WoC. This highlights the inherent complexity of tracing the origins of code reused through copy-and-paste. WoC's

Table 7.4: Reused blobs and their origin

		full		smol	
		count	% (row)	count	% (row)
1	**Total**	**582,933,549**		**87,175,702**	
2	Reused	90,303,809	15.49 (1)	9,848,987	11.30 (1)
3	Same	29,432,636	32.59 (2)	3,764,702	38.22 (2)
4	Different	60,871,173	67.41 (2)	6,084,285	61.78 (2)

Table 7.5: Reused blobs with different origins and their licenses

	Stack v2	WoC	full		smol	
			count	% (row)	count	% (row)
1	**Different Origin**		**60,871,173**		**6,084,285**	
2	**Same License**		**38,410,728**	**63.10 (1)**	**4,418,289**	**72.62 (1)**
3	no license	no license	26,604,621	69.26 (2)	3,269,149	73.99 (2)
4	permissive	permissive	11,806,107	30.74 (2)	1,149,140	26.01 (2)
5	**Different License**		**22,460,445**	**36.90 (1)**	**1,665,996**	**27.38 (1)**
6	permissive	no license	10,257,891	45.67 (5)	721,920	43.33 (5)
7	no license	permissive	9,309,959	41.45 (5)	658,085	39.50 (5)
8	no license	restrictive	1,868,500	8.32 (5)	193,358	11.61 (5)
9	permissive	restrictive	1,024,095	4.56 (5)	92,633	5.56 (5)

ability to perform such identification stems from its comprehensive coverage of nearly all open-source projects and their version histories.

Since cases with misidentified origins present a potential risk of license noncompliance, we conducted a further investigation into the blobs with differing identified origins. The detailed results of this analysis are presented in Table 7.5.

The results reveal that 36.90% and 27.38% of the blobs with misidentified origins have licenses that differ from those identified in the Stack dataset. These discrepancies fall into four distinct categories. In the first case, the Stack identifies the license as permissive, while WoC identifies no license. In the second, the Stack identifies no license, but WoC identifies a permissive license. The third case involves the Stack identifying no license, while WoC identifies a restrictive license. Finally, in the fourth

case, the Stack identifies a permissive license, but WoC identifies a restrictive license. Among these, the second scenario does not pose a compliance risk and may even be advantageous, given the problematic nature of reusing code without a license, as previously discussed. However, the first scenario still raises some concerns. The third and fourth scenarios are particularly concerning as they indicate a high risk of license noncompliance due to the blobs originating from projects with restrictive licenses.

In summary, our analysis reveals that even the smaller version of the Stack dataset contains hundreds of thousands of blobs originating from projects with restrictive licenses, raising significant legal compliance concerns for any LLM trained on this dataset. These findings provide strong support for our second hypothesis (**H2**).

Key Findings 2

1. 15.49% and 11.30% of blobs in the *full* and *smol* datasets, respectively, have been reused at least once. Among these, 64.41% and 61.78% have origins that were misidentified.
2. 36.90% and 27.38% of blobs with misidentified origins have licenses that differ from those identified in the dataset.
3. 12.88% and 17.17% of blobs with differing licenses are subject to a restrictive license, presenting a significant risk of noncompliance.

7.6 Limitations

7.6.1 Internal Validity

Impact of Buggy Code Removal on Model Outputs

Eliminating all buggy code from pre-training or fine-tuning datasets does not guarantee that the resulting LLM will generate bug-free code. However, it is reasonable to assume that some generated code may replicate buggy patterns observed in the training data. Therefore, removing bugs from the training data, especially

through a low-cost approach like ours, is a sensible step toward improving the model's output quality.

WoC Dataset Coverage

Some code may originate outside public version control systems or may simply not be included in WoC's collection. However, as demonstrated with the Stack v2 dataset, only 2.5% of blobs could not be linked to commits already present in WoC, indicating that this is a relatively minor issue.

Blob Updates and Quality

While updating blobs to newer versions eliminates known bugs, it can occasionally introduce new and unknown bugs. However, in most projects, only a small proportion of bug fixes result in new issues or fail to address the intended bugs. Consequently, applying fixes generally enhances the overall quality of the training data.

Rebasing and Metadata Loss

Our approach relies on git SHA-1 hashes to track blobs, which ensures that content-based identification is robust to rebasing. However, rebasing may obscure certain metadata, such as precise commit lineage, which could limit the ability to fully trace the historical context of some blobs.

Commit Keyword Usage for Fix Identification

Not all commits containing the keywords we used represent bug fixes, nor do all bug fixes include these keywords in their commit messages. Despite this, applying all changes, not just those identified as fixes, is likely necessary. These keywords and similar ones have been widely used in prior research to identify changes related to bug fixes. In our validation of 20 randomly selected commits, only three (15%) were found not to be clearly bug fixes.

Reliability of CVE Detection

Our method successfully identified thousands of CVEs in the Stack v2 dataset, leveraging commit messages as a primary indicator. However, this approach relies on the presence of explicit references to CVEs in commit messages, which may not comprehensively capture all vulnerabilities. For instance, CVEs that were not documented in commit messages or introduced through transitive dependencies might be missed. Future work could address this limitation by conducting a manual review of a representative sample or validating the method against additional datasets to evaluate recall more comprehensively.

7.6.2 Construct Validity

Impact of Dataset Vulnerabilities on Model Outputs

This study assumes that vulnerabilities and flaws in training datasets may influence the quality and security of model outputs. While this assumption aligns with logical inference and prior research on LLM behavior, direct empirical validation of this relationship is currently lacking and represents an important avenue for future research.

Never-Modified Code Assumption

While we suggest that never-modified code may indicate low use or untested quality, this is based on logical inference rather than direct empirical evidence. Future studies are needed to validate whether unmodified code consistently correlates with lower reliability or usability in practice.

Blob Origin Identification

Identifying the origin of a blob is not always possible, particularly for blobs that did not originate in open-source projects. Accurate identification requires comprehensive

access to all project data. However, the extensive coverage provided by WoC significantly reduces this risk.

License Applicability Assumption

The licensing assumption for a blob is based on the identified license of the project from which it originated. However, not all files within a project necessarily fall under the project's overarching license, as some files may have distinct individual licenses.

7.6.3 External Validity

New Bugs and Iterative Updates

Even if all known bugs are addressed at time t, new bugs will inevitably be discovered at time $t + 1$. Therefore, regular updates are necessary. Fortunately, the approach outlined here can be automated, allowing it to be efficiently applied to each new version of the WoC dataset.

Updating to Latest Versions

The updated version of a blob may not always represent the latest available version. As a result, the process may need to be repeated iteratively until the most recent fix is applied. The median timestamp of the commits updating blobs was June 2020, indicating that these updates were available well before the creation of the Stack v2 dataset in 2024.

7.7 Conclusions

Processes to ensure provenance, security, and compliance in SSCs are essential. This project sets the stage for future work on the curating LLM training data and provide several insights and interventions that can improve on the current state of the art.

Several notable observations emerge from our analysis. First, the largest open-source training dataset, Stack v2, contains only a small fraction of all publicly available source code versions. These datasets could be significantly enhanced by incorporating intelligently selected data from comprehensive sources like WoC. Second, between 10% and 20% of the versions have updates, even though the WoC dataset version V3 is contemporaneous with Stack v2. Third, a substantial portion of the training data includes files with known bug fixes. While newer versions may incorporate updated APIs or additional features, applying these bug fixes is crucial to prevent LLMs from being trained on buggy code. Fourth, such fixes can be leveraged to train or align LLMs that specialize in generating changes or fixes. Fifth, training datasets should prioritize heavily or moderately modified code, which often has fewer bugs, rather than relying heavily on pristine, first-version code that dominates many existing datasets. Finally, misidentified code origins have resulted in non-permissive code being included in these datasets, raising compliance concerns.

Beyond improving the curation practices for LLM training data, this work also introduces the concept of the LLM supply chain, highlighting its similarities to and differences from traditional software supply chains.

While our primary focus has been on data curation for code LLMs, the insights generalize to any scenario involving version-controlled data.

Chapter 8

Conclusions & Future Work

8.1 Summary of Findings

This dissertation has systematically investigated copy-based reuse in Open Source Software (OSS) supply chains, shedding light on its prevalence, motivations, and broader implications. The findings demonstrate that while copy-based reuse is a common practice among developers, its unregulated nature leads to legal, security, and maintainability risks.

The first part of the dissertation provided a foundation for understanding copy-based reuse:

- Chapter 2 presented the methodology for dataset construction, involving large-scale data collection and the development of heuristics to detect copy-based reuse. This dataset serves as a crucial resource for further empirical studies in this domain.

- Chapter 3 analyzed reuse patterns, showing that copy-based reuse is widespread but unevenly distributed, often influenced by project size, programming language, and licensing conditions. The chapter also identified recurring clusters of reused code, including cases where modifications introduce security vulnerabilities.

- Chapter 4 examined the developer perspective, revealing that many practitioners engage in copy-based reuse for practical reasons such as performance optimization and ease of integration, while others are unaware of the legal and security risks associated with this practice.

The second part of the dissertation applied the detection method to real-world challenges:

- Chapter 5 explored its implications for license detection in OSS projects, demonstrating how license inconsistencies arise from untracked copy-based reuse.
- Chapter 6 investigated noncompliance issues and their consequences, showing that projects with copied code frequently violate original licensing terms, leading to potential legal disputes and forced re-licensing.
- Chapter 7 extended the method to machine learning, identifying noncompliant and vulnerable code in Large Language Model (LLM) pretraining datasets, underscoring an overlooked risk in AI development.

8.2 Implications

8.2.1 For Developers

Copy-based reuse enables developers to save time and effort by leveraging existing code. However, it introduces risks such as maintenance fragmentation, security vulnerabilities, and outdated dependencies. To address these challenges, developers should adopt tools and practices to track reused code, ensure compliance with licensing requirements, and mitigate risks associated with unverified code quality.

Fostering a practice of systematically reviewing and documenting reused code not only enhances its reliability and maintainability, but also contributes to the overall

sustainability of software projects. Additionally, staying informed about updates to reused code and integrating these updates promptly can further reduce risks associated with outdated or insecure components.

8.2.2 For Businesses

Businesses that rely on open source software must proactively address the inherent risks of copy-based reuse, including security vulnerabilities and potential non-compliance with licensing terms. Investing in robust tools for tracking and maintaining reused code is critical to safeguarding the software supply chain. This effort should encompass implementing workflows for regularly updating and reviewing reused components.

Moreover, businesses should actively support smaller open source projects that provide valuable code contributions. Such support not only enhances the quality and reliability of business-critical software, but also fosters goodwill and collaboration within the open source community. By taking these steps, businesses can effectively mitigate risks while strengthening the ecosystem upon which they rely.

8.2.3 For the Open Source Community

The open source community plays an important role in ensuring the safe and effective reuse of code. By promoting best practices for ethical and secure reuse, such as adopting standardized licensing and improving quality benchmarks, the community can minimize risks and build trust in shared resources. Equally important is supporting small and medium-sized projects that contribute significantly to the reusable code base. Providing mentorship, funding, and collaboration opportunities can bolster the overall open source ecosystem, fostering innovation and cooperation across projects.

Additionally, establishing centralized repositories or resources that facilitate traceability and offer detailed metadata on provenance, authorship, and licensing can

streamline the reuse process and mitigate associated risks. These efforts collectively enhance the reliability, sustainability, and scalability of open source software.

8.2.4 For Researchers and Educators

Researchers have a unique opportunity to investigate finer-grained reuse patterns, such as instances involving slight modifications or partial reuse, to better understand the factors influencing reuse and its long-term impact on software quality and security. Such insights can guide the development of tools and methodologies that promote safe and effective reuse practices.

Educators should integrate lessons on ethical reuse practices, licensing compliance, and dependency management into software engineering curricula. By leveraging real-world case studies and addressing practical challenges, such as balancing development speed with security concerns, educators can equip future developers to navigate the complexities of software reuse responsibly. This approach will help ensure that the next generation of software professionals actively supports the sustainability and growth of open source ecosystems.

8.2.5 For OSS Platform Maintainers

Platforms like GitHub and GitLab are well-positioned to enhance practices surrounding copy-based reuse. Improving traceability mechanisms to preserve provenance, authorship, and licensing metadata is essential for minimizing risks such as unintentional license violations and outdated dependencies. Integrating features for automated detection of license conflicts, dependency vulnerabilities, and changes in reused code can further empower developers to manage their projects efficiently and securely.

Additionally, platforms can offer educational resources and in-platform guidance to encourage best practices for reuse and compliance. By fostering a culture of informed and collaborative reuse, platform maintainers can contribute significantly to the long-term sustainability and resilience of the open source ecosystem.

8.3 Future Work

8.3.1 Code-Snippet Granularity

We discussed in methodology section that going to a finer granularity than blob-level to detect code reuse is not practically feasible. Nevertheless, there are approaches that can make this a relatively more tractable problem. Specifically, hashing the abstract syntax tree (AST) for each code snippet (such as classes or functions) in a blob and mapping blobs to these hashes could potentially make finer-grained code reuse detection more feasible.

Assuming an average of k code snippets for each of the 16 billion blobs, the parsing and hashing operation has a complexity of $O(n)$, resulting in $O(16 \times 10^9 \times k)$. We can then perform a self-join on the created map of blob to syntax tree hash (b2AST) using the AST hash as the key. The self-join complexity depends on the number of unique hashes and their distribution. In the worst case, if every blob had unique hashes, the join operation would approach $O((16 \times 10^9 \times k)^2)$. However, the join complexity would typically be significantly less if there are many common hashes. A more realistic estimate assumes that the number of unique AST hashes h is much smaller than the total number of entries in the b2AST map, making the join complexity closer to $O(h \times 16 \times 10^9 \times k)$. This join, although potentially large, can be more feasible than pairwise comparisons of entire blobs due to the more efficient handling of common hashes.

By examining code reuse at the granularity of code snippets, we could potentially uncover a far more intricate network of reuse. This approach might reveal patterns and practices that are not noticeable when looking solely at whole-file or blob-level reuse. Although this increased complexity is challenging to manage, it offers valuable opportunities for a more comprehensive analysis of reuse Jahanshahi and Mockus (2024).

8.3.2 Dependency-Based Reuse

In this work, we aimed to demonstrate the prevalence and importance of copy-based reuse. To gain a comprehensive understanding of code reuse, it is important to analyze both copy-based and dependency-based reuse. Each type of reuse reveals different aspects of how software developers leverage existing code in their projects. By studying them side by side, we can paint a more complete picture of the extent and nuances of reuse in software development. Ignoring one in favor of the other would provide an incomplete narrative Jahanshahi and Mockus (2024).

8.3.3 Upstream Repository

As highlighted in the limitations section, we currently lack precise knowledge about the source from which a repository reuses a file. We tend to assume it is from the originating repository in all instances of copying. However, this assumption may not capture the real-world complexity of reuse. To enhance our understanding of how developers identify suitable repositories for reuse, we could potentially leverage meta-heuristic algorithms or artificial intelligence techniques. These advanced methods might enable us to predict the actual source of reused artifacts in each instance of copying with greater accuracy Jahanshahi and Mockus (2024).

8.3.4 Open Source Software Supply Chain Network

Directed Acyclic Graphs (DAGs) have been instrumental in clone detection and reuse literature due to their ability to model and analyze complex relationships and dependencies between various software components. In the context of copy-based reuse, the dataset created using the World of Code (WoC)[1] infrastructure can be leveraged to construct DAGs that represent the flow and reuse across different repositories.

[1]For more information about how to access this data, please visit: https://github.com/woc-hack/tutorial.

The dataset's detailed tracking of blob copies, including their origins and destinations, provides a rich source of data to map these relationships accurately. By drawing DAGs, researchers can visualize and analyze the propagation of reused blobs, identifying critical nodes (projects or blobs) that play a central role in the reuse network. This visualization helps in understanding the structure and dynamics of reuse, highlighting patterns such as the most reused blobs, the central projects in the reuse network, and potential vulnerabilities or licensing issues propagating through these reused blobs.

DAGs can reveal how reuse spreads across projects, helping to identify which projects are the primary sources of reusable blobs and how code flows between different projects. By mapping out the reuse network, it is possible to pinpoint critical points where vulnerabilities or licensing issues could propagate, allowing for targeted interventions to mitigate these risks. Understanding the reuse network also aids in developing better tools and practices for managing code quality and ensuring that reused code is maintained and updated consistently across all projects that use it.

Studies on large-scale clone detection such as Sajnani et al. (2016) and Koschke (2007) provide foundational methodologies for leveraging DAGs in these contexts. These methodologies can be adapted and extended using our dataset to enhance the understanding of copy-based reuse in open source software development.

8.3.5 Security Vulnerability Detection Tools

Reused code can propagate vulnerabilities across multiple projects Reid et al. (2022). For instance, if a security flaw exists in a reused blob, it can potentially affect all projects that include this blob. Analyzing the reuse patterns can help identify critical points where vulnerabilities might spread and allow for proactive mitigation measures. There have been notable incidents where widespread code reuse led to security breaches. For example, the Heartbleed bug in OpenSSL had far-reaching

impacts due to the extensive reuse of the affected code across numerous projects. Future research can focus on developing automated tools that scan reused code for known vulnerabilities and suggest patches. This proactive approach can enhance the security posture of software systems.

8.3.6 Compliance Detection Tools

Reused code may carry licensing obligations that need to be respected. Failure to comply with these obligations can lead to legal disputes and financial penalties. By understanding reuse patterns, organizations can ensure they meet licensing requirements. There have been instances where companies faced legal challenges due to improper reuse of code with restrictive licenses. For example, using GPL-licensed code in a proprietary software without complying with GPL terms has led to lawsuits. Developing tools that automatically check for license compliance when code is reused can help organizations avoid legal pitfalls. These tools can flag potential issues and provide guidance on how to resolve them.

8.3.7 Survey

Surveying projects' owners to verify our noncompliance alarm accuracy might also prove useful especially for research purposes.

8.3.8 Code Quality Enhancement Tools

Reused code may not always meet the quality standards of the adopting project. Ensuring that reused code adheres to best practices and coding standards is essential for maintaining overall code quality. Poorly written code can lead to maintenance challenges and degraded performance in adopting projects. Future work can focus on creating tools that assess the quality of reused code and suggest improvements. These tools can analyze code for adherence to coding standards, detect code smells, and recommend refactoring.

8.3.9 Package Managers

Developing package managers tailored for different programming languages and communities can be highly beneficial. These managers can offer more relevant and effective support for managing code reuse in specific environments. Additionally, enhancing existing package managers with features such as reuse tracking, version control, and automated updates can improve development efficiency and reduce the associated risks of code reuse.

8.3.10 Autocuration Tool for LLM pretraining Datasets

A promising direction for future work is the development of automated curation tools specifically designed to enhance the quality of datasets used for pre-training large language models (LLMs) for code, such as Stack v2. Building on the cost-efficient approach introduced in this paper, these tools could automatically identify and apply patches for known fixes or vulnerabilities, ensuring that the datasets include secure and reliable code. They could also locate and update blobs to their latest versions, minimizing the inclusion of outdated or buggy code. Furthermore, the tools could enhance license compliance by automatically detecting and removing code with non-permissive licenses, ensuring that only code with appropriate licensing is included in the dataset. The feasibility of such automation is demonstrated by the scalability and efficiency of our approach in handling large-scale datasets. By automating these tasks, the proposed tools would streamline the iterative updates required for maintaining high-quality training data, ensuring practicality and cost-effectiveness in preparing datasets for LLM pre-training.

8.3.11 Community Engagement

Engaging with open source communities to develop tools and practices that address the unique needs of different ecosystems, and collaborating with these communities, can ensure widespread adoption and effectiveness. Continuously gathering user

feedback and iterating on the tools to enhance their functionality and usability is also important. This iterative process helps create robust and reliable tools that meet the evolving needs of software developers.

8.4 Conclusions

This dissertation has provided an in-depth examination of copy-based reuse in open source software (OSS), exploring its prevalence, motivations, and risks across a vast ecosystem of projects. By leveraging the extensive World of Code (WoC) infrastructure and integrating insights from developer surveys, our work sheds new light on how and why code is replicated and adapted, and how these practices impact the legal, security, and maintainability dimensions of software.

The findings reveal that copy-based reuse is remarkably common. While it can offer considerable efficiency benefits—reducing development time, facilitating rapid prototyping, and fostering collaboration—it also presents significant unregulated risks. Our large-scale analysis confirms that a substantial fraction of OSS projects contain reused blobs, often without clear provenance or licensing metadata, creating uncertainty for both developers and organizations. Furthermore, this work underscores how license noncompliance and security vulnerabilities can propagate through the supply chain unnoticed, potentially jeopardizing entire ecosystems that rely on shared code.

Across the chapters, several recurrent themes and insights emerged:

1. **Prevalence and Patterns of Copy-Based Reuse**. We showed that reuse varies by programming language, project size, and license type, indicating a complex landscape of developer habits and community norms. Importantly, binary blob reuse appears to be more prevalent than is often assumed, emphasizing the need for tools and practices that handle diverse file formats.

2. **Developer Motivations and Perspectives**. The developer survey revealed that practical considerations—such as ease of integration and performance optimizations—often override concerns about licensing and security. At the same time, many creators explicitly welcome reuse, consistent with an ethos of open collaboration. This nuanced view reinforces the idea that developers frequently operate under time pressures and knowledge gaps, rather than intentional disregard for legal and security best practices.

3. **Legal and Licensing Implications**. Our analysis of licensing patterns demonstrated how permissive licenses generally facilitate reuse, while more restrictive licenses limit it in certain contexts. However, the negative or unclear impact of public domain licensing indicates that ambiguous legal contexts can deter reuse, highlighting the need for more consistent and transparent licensing standards. This work also evidenced how reliance on dependency-based detection alone can miss substantial portions of copy-based reuse, leading to hidden legal risks.

4. **Security and Compliance in Supply Chains**. Chapters focusing on noncompliance and vulnerabilities revealed how projects that incorporate copied code can inadvertently carry forward bugs or violate license terms, leaving them exposed to legal or reputational harm. Even more critically, when vulnerabilities reside in widely reused blobs, they may compromise entire ecosystems. The consequences extend to machine learning (ML) and Large Language Model (LLM) pretraining datasets, where outdated or insecure code can pollute training corpora.

5. **Implications for OSS Platforms, Businesses, and the Community**. Given the central role of platforms like GitHub and GitLab, the importance of facilitating traceability, automated license checks, and educational resources cannot be overstated. Businesses, too, must invest in workflows that track and maintain reused components to avoid costly noncompliance. Likewise, the open

source community can champion ethical, secure, and transparent practices for shared code, thereby preserving OSS's collaborative spirit while safeguarding its integrity.

6. F**uture Directions**. Beyond the scope of current methods, this dissertation points to fine-grained detection at the snippet level, holistic dependency-based analysis, and advanced heuristics or AI-driven approaches for tracing the true "upstream" of reused code. These enhancements can help tackle the complexity of partial file reuse and further refine our understanding of how code moves through global software networks. The emergence of LLM supply chains underscores the growing need for robust curation, where automated tools can ensure high-quality and legally compliant data for training code models.

Collectively, these findings emphasize that copy-based reuse is both a powerful enabler of innovation and a significant source of unseen risk. Addressing these challenges will require coordinated efforts among developers, businesses, researchers, educators, platform maintainers, and the broader OSS community. By developing better detection tools, clearer license guidelines, and enhanced educational programs, the ecosystem can harness the benefits of reuse while minimizing its drawbacks.

Ultimately, the work presented in this dissertation serves as an evidence-based foundation for rethinking open source practices. As software continues to be built upon layer after layer of shared code, the imperative grows for proactive and collaborative management. It is our hope that these insights—supported by rigorous empirical data and grounded in real-world developer perspectives—will spur ongoing dialogue, research, and practical initiatives that fortify the OSS supply chain. In doing so, we can realize the full promise of open source software as a secure, sustainable, and freely shared global resource.

Bibliography

Agresti, A. (2012). *Categorical data analysis*, volume 792. John Wiley & Sons. 123

Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. (2019). A systematic review on code clone detection. *IEEE access*, 7:86121–86144. 13, 34

Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., et al. (2023). Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*. 140

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37. 143

Almeida, D. A., Murphy, G. C., Wilson, G., and Hoye, M. (2017). Do software developers understand open source licenses? In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 1–11. IEEE. 114

Almeida, D. A., Murphy, G. C., Wilson, G., and Hoye, M. (2019). Investigating whether and how software developers understand open source software licensing. *Empirical Software Engineering*, 24:211–239. 114

An, L., Mlouki, O., Khomh, F., and Antoniol, G. (2017). Stack overflow: A code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293. IEEE. 6

Angst, C. M., Agarwal, R., Sambamurthy, V., and Kelley, K. (2010). Social contagion and information technology diffusion: The adoption of electronic medical records in us hospitals. *Management Science*, 56(8):1219–1241. 28

Antoniol, G., Di Penta, M., and Merlo, E. (2004). An automatic approach to identify class evolution discontinuities. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 31–40. IEEE. 32

Austin, Z. and Sutton, J. (2014). Qualitative research: Getting started. *The Canadian journal of hospital pharmacy*, 67(6):436. 81

Bissyandé, T. F., Thung, F., Lo, D., Jiang, L., and Réveillere, L. (2013). Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE. 41, 111

Blincoe, K., Sheoran, J., Goggins, S., Petakovic, E., and Damian, D. (2016). Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30–39. 31, 32

Borges, H., Hora, A., and Valente, M. T. (2016). Predicting the popularity of github repositories. In *Proceedings of the The 12th international conference on predictive models and data analytics in software engineering*, pages 1–10. 39, 45, 111

Boughton, L., Miller, C., Acar, Y., Wermke, D., and Kästner, C. (2024). Decomposing and measuring trust in open-source software supply chains. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 57–61. 6

Braun, V. and Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101. 81, 82

Brewer, J. V. (2012). *The Effects of Open Source License Choice on Software Reuse*. PhD thesis, Virginia Tech. 111, 117

Brown, A. W. and Wallnau, K. C. (1998). The current state of cbse. *IEEE software*, 15(5):37–46. 34

Capiluppi, A., Lago, P., and Morisio, M. (2003). Characteristics of open source projects. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 317–327. IEEE. 43

Castleberry, A. and Nolen, A. (2018). Thematic analysis of qualitative research data: Is it as easy as it sounds? *Currents in pharmacy teaching and learning*, 10(6):807–815. 77

Christakis, N. A. and Fowler, J. H. (2013). Social contagion theory: examining dynamic social networks and human behavior. *Statistics in Medicine*, 32:556–577. 28

Cox, R. (2019). Surviving software dependencies: Software reuse is finally here but comes with risks. *Queue*, 17(2):24–47. 13, 34

Creswell, J. W. and Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications. 77, 79

Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. 38, 44, 110

Cui, X., Wu, J., Wu, Y., Wang, X., Luo, T., Qu, S., Ling, X., and Yang, M. (2023). An empirical study of license conflict in free and open source software. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 495–505. IEEE. 114

Denzin, N. K. (2017). *The research act: A theoretical introduction to sociological methods*. Routledge. 77

Di Penta, M., German, D. M., Guéhéneuc, Y.-G., and Antoniol, G. (2010). An exploratory study of the evolution of software licensing. In *Proceedings of the 32nd*

ACM/IEEE International Conference on Software Engineering-Volume 1, pages 145–154. 6, 113

Dormann, C. F., Elith, J., Bacher, S., Buchmann, C., Carl, G., Carré, G., Marquéz, J. R. G., Gruber, B., Lafourcade, B., Leitão, P. J., et al. (2013). Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46. 124

Fendt, O. and Jaeger, M. C. (2019). Open source for open source license compliance. In *Open Source Systems: 15th IFIP WG 2.13 International Conference, OSS 2019, Montreal, QC, Canada, May 26–27, 2019, Proceedings 15*, pages 133–138. Springer. 107

Feng, M., Mao, W., Yuan, Z., Xiao, Y., Ban, G., Wang, W., Wang, S., Tang, Q., Xu, J., Su, H., et al. (2019). Open-source license violations of binary software at large scale. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 564–568. IEEE. 3, 94, 114, 116

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. (2017). Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE symposium on security and privacy (SP)*, pages 121–136. IEEE. 13, 33

Fitzgerald, B. (2006). The transformation of open source software. *MIS quarterly*, pages 587–598. 120

Flint, S. W., Chauhan, J., and Dyer, R. (2021a). 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)*. 23

Flint, S. W., Chauhan, J., and Dyer, R. (2021b). 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)*, pages 85–96. IEEE. 71

Frakes, W. and Terry, C. (1996). Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435. 28

Frakes, W. B. and Fox, C. J. (1995). Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–ff. 77

Frakes, W. B. and Kang, K. (2005). Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 31(7):529–536. 30

Frakes, W. B. and Succi, G. (2001). An industrial study of reuse, quality, and productivity. *Journal of Systems and Software*, 57(2):99–106. 13, 34

Fry, T., Dey, T., Karnauch, A., and Mockus, A. (2020). A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits. In *Proceedings of the 17th international conference on mining software repositories*, pages 518–522. 14, 95, 118, 148

Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. 42

Gamalielsson, J. and Lundell, B. (2014). Sustainability of open source software communities beyond a fork: How and why has the libreoffice project evolved? *Journal of systems and Software*, 89:128–145. 45, 111

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., et al. (2020). The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*. 140

Geisterfer, C. M. and Ghosh, S. (2006). Software component specification: a study in perspective of component selection and reuse. In *Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'05)*, pages 9–pp. IEEE. 36

German, D. M. (2002). The evolution of the gnome project. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, pages 20–24. 31, 32

German, D. M., Di Penta, M., and Davies, J. (2010). Understanding and auditing the licensing of open source software distributions. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 84–93. IEEE. 107, 109, 113, 114

German, D. M., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G. (2009). Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90. IEEE. 5

German, D. M. and Hassan, A. E. (2009). License integration patterns: Addressing license mismatches in component-based development. In *2009 IEEE 31st international conference on software engineering*, pages 188–198. IEEE. 6

Gharehyazie, M., Ray, B., and Filkov, V. (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)*, pages 291–301. IEEE. 1, 13, 33

Gharehyazie, M., Ray, B., Keshani, M., Zavosht, M. S., Heydarnoori, A., and Filkov, V. (2019). Cross-project code clones in github. *Empirical Software Engineering*, 24(3):1538–1573. 13, 33

Ghobadi, S. (2015). What drives knowledge sharing in software development teams: A literature review and classification framework. *Information & Management*, 52(1):82–97. 14

Gkortzis, A., Feitosa, D., and Spinellis, D. (2021). Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 172:110653. 5

Gonzalez-Barahona, J. M., Montes-Leon, S., Robles, G., and Zacchiroli, S. (2023). The software heritage license dataset (2022 edition). *Empirical Software Engineering*, 28(6):147. 93, 94, 102, 104

Gousios, G. (2013). The ghtorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE. 44

Gousios, G. and Spinellis, D. (2012). Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE. 39

Guest, G., Bunce, A., and Johnson, L. (2006). How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82. 80

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., et al. (2023). Textbooks are all you need. *arXiv preprint arXiv:2306.11644*. 143, 144, 146

Haefliger, S., Von Krogh, G., and Spaeth, S. (2008). Code reuse in open source software. *Management science*, 54(1):180–193. 1, 13, 33

Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. (2012). Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer. 13, 33, 34

Hata, H., Gaikovina Kula, R., Ishio, T., and Treude, C. (2021a). Research artifact: The potential of meta-maintenance on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 192–193. 13

Hata, H., Kula, R. G., Ishio, T., and Treude, C. (2021b). Research artifact: The potential of meta-maintenance on github. In *2021 IEEE/ACM 43rd International*

Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 192–193. IEEE. 33

Hata, H., Kula, R. G., Ishio, T., and Treude, C. (2021c). Same file, different changes: The potential of meta-maintenance on github. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 773–784. IEEE Press. 13

Hata, H., Kula, R. G., Ishio, T., and Treude, C. (2021d). Same file, different changes: the potential of meta-maintenance on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 773–784. IEEE. 33

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B., and Irlbeck, M. (2011). On the extent and nature of software reuse in open source java projects. In *International Conference on Software Reuse*, pages 207–222. Springer. 13, 33

Hosmer Jr, D. W., Lemeshow, S., and Sturdivant, R. X. (2013). *Applied logistic regression*. John Wiley & Sons. 41

Hubinger, E., Denison, C., Mu, J., Lambert, M., Tong, M., MacDiarmid, M., Lanham, T., Ziegler, D. M., Maxwell, T., Cheng, N., et al. (2024). Sleeper agents: Training deceptive llms that persist through safety training. *arXiv preprint arXiv:2401.05566*. 140

Inoue, K., Miyamoto, Y., German, D. M., and Ishio, T. (2021). Finding code-clone snippets in large source-code collection by ccgrep. In *Open Source Systems: 17th IFIP WG 2.13 International Conference, OSS 2021, Virtual Event, May 12–13, 2021, Proceedings 17*, pages 28–41. Springer. 13, 34

Jahanshahi, M. and Mockus, A. (2024). Dataset: Copy-based reuse in open source software. In *2024 IEEE/ACM 21st International Conference on Mining Software*

Repositories (MSR), pages 42–47. IEEE. 5, 6, 11, 26, 29, 30, 34, 71, 72, 73, 119, 150, 154, 165, 166

Jahanshahi, M. and Mockus, A. (2025). Cracks in the stack: Hidden vulnerabilities and licensing risks in llm pre-training datasets. *arXiv preprint arXiv:2501.02628*. 138

Jahanshahi, M., Reid, D., McDaniel, A., and Mockus, A. (2024a). Oss license identification at scale: A comprehensive dataset using world of code. *arXiv preprint arXiv:2409.04824*. 91, 116, 118, 150

Jahanshahi, M., Reid, D., and Mockus, A. (2024b). Beyond dependencies: The role of copy-based reuse in open source software development. *arXiv preprint arXiv:2409.04830*. 25, 76, 107, 110, 111, 115, 117, 118, 142, 146, 149

Janjic, W., Hummel, O., Schumacher, M., and Atkinson, C. (2013). An unabridged source code dataset for research in software reuse. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 339–342. 13

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE. 13, 34, 39

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009). Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE. 1, 6, 33, 35, 36, 85, 86, 87

Kapitsaki, G. M., Kramer, F., and Tselikas, N. D. (2017). Automating the license compatibility process in open source software with spdx. *Journal of systems and software*, 131:386–401. 99

Kapser, C. J. and Godfrey, M. W. (2008). "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692. 31

Kashima, Y., Hayase, Y., Yoshida, N., Manabe, Y., and Inoue, K. (2011). An investigation into the impact of software licenses on copy-and-paste reuse among oss projects. In *2011 18th Working Conference on Reverse Engineering*, pages 28–32. IEEE. 111, 117

Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., De Roover, C., and Inoue, K. (2014a). 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*, pages 305–314. 12

Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., De Roover, C., and Inoue, K. (2014b). 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*, pages 305–314. IEEE. 29

Koch, S. and Schneider, G. (2002). Effort, co-operation and co-ordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42. 38, 44, 110

Koschke, R. (2007). Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 30, 167

Krejcie, R. V. and Morgan, D. W. (1970). Determining sample size for research activities. *Educational and psychological measurement*, 30(3):607–610. 81

Krueger, C. (2001). Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*, pages 282–293. Springer. 31

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183. 1

Ladisa, P., Plate, H., Martinez, M., and Barais, O. (2023). Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE. 4, 5, 110

Laurençon, H., Saulnier, L., Wang, T., Akiki, C., Villanova del Moral, A., Le Scao, T., Von Werra, L., Mou, C., González Ponferrada, E., Nguyen, H., et al. (2022). The bigscience roots corpus: A 1.6 tb composite multilingual dataset. *Advances in Neural Information Processing Systems*, 35:31809–31826. 140

Laurent, A. M. S. (2004). *Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software.* " O'Reilly Media, Inc.". 112, 120

Leskovec, J. and Faloutsos, C. (2006). Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636. 36

Lessig, L. (2004). How big media uses technology and the law to lock down culture and control creativity. *Retrieved December*, 5:2004. 112

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192. 6

Liang, L., Wu, X., Deng, J., and Lv, X. (2022). Research on risk analysis and governance measures of open-source components of information system in transportation industry. *Procedia Computer Science*, 208:106–110. 7th International Conference on Intelligent, Interactive Systems and Applications. 5

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. (2017). Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28. 13, 33

Lozano-Tello, A. and Gómez-Pérez, A. (2002). Baremo: how to choose the appropriate software component using the analytic hierarchy process. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 781–788. 28

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al. (2024). Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*. 141, 143, 144

Lyulina, E. and Jahanshahi, M. (2021). Building the collaboration graph of open-source software ecosystem. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 618–620. 14

Ma, Y. (2018). Constructing supply chains in open source software. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 458–459. IEEE. 14

Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., and Mockus, A. (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)*, pages 143–154. IEEE. 14, 24, 72, 95, 116, 117, 141, 148

Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R., and Mockus, A. (2021). World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data. *Empirical Software Engineering*, 26(2):1–42. 14, 93, 95, 118, 141, 148

Ma, Y., Mockus, A., Zaretzki, R., Bradley, R., and Bichescu, B. (2020). A methodology for analyzing uptake of software technologies among developers. *IEEE Transactions on Software Engineering*, 48(2):485–501. 28

Mason, M. et al. (2010). Sample size and saturation in phd studies using qualitative interviews. 81

Mathur, A., Choudhary, H., Vashist, P., Thies, W., and Thilagam, S. (2012). An empirical study of license violations in open source projects. In *2012 35th annual IEEE software engineering workshop*, pages 168–176. IEEE. 114

Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE transactions on Software Engineering*, 21(6):528–562. 34

Mitzenmacher, M. and Upfal, E. (2017). *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press. 40

Mockus and Votta (2000). Identifying reasons for software changes using historic databases. In *Proceedings 2000 international conference on software maintenance*, pages 120–130. IEEE. 151

Mockus, A. (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)*, pages 7–7. IEEE. 13, 33, 40, 43, 44, 45, 77, 110

Mockus, A. (2019a). Insights from open source software supply chains (keynote). In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 3, New York, NY, USA. Association for Computing Machinery. 3, 6

Mockus, A. (2019b). Insights from open source software supply chains (keynote). In *ESEC/FSE 2019*, page 3, New York, NY, USA. Association for Computing Machinery. 109, 142

Mockus, A. (2022). Tutorial: Open source software supply chains. 3, 109

Mockus, A. (2023). Securing large language model software supply chains. ASE'23 LLMs in Software Engineering. 3, 109

Mockus, A., Spinellis, D., Kotti, Z., and Dusing, G. J. (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*, pages 513–517. 14, 15, 16, 95, 118, 148

Moglen, E. (2001). Free software matters: Enforcing the gpl, ii. *Column in LinuxUser Magazine (August 2001)*. 121

Moraes, J. P., Polato, I., Wiese, I., Saraiva, F., and Pinto, G. (2021). From one to hundreds: multi-licensing in the javascript ecosystem. *Empirical Software Engineering*, 26(3):39. 114

Noll, L. C. (2012). Fowler/noll/vo (fnv) hash. *Accessed Jan*. 18

Okafor, C., Schorlemmer, T. R., Torres-Arias, S., and Davis, J. C. (2022). Sok: Analysis of software supply chain security by establishing secure design properties. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 15–24. 5

Ombredanne, P. (2022). Scancode toolkit. https://aboutcode.org/scancode/. Accessed 2022-01-25. 102

Ossher, J., Bajracharya, S., and Lopes, C. (2010). Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 130–140. IEEE. 13, 34

Papoutsoglou, M., Kapitsaki, G. M., German, D., and Angelis, L. (2022). An analysis of open source software licensing questions in stack exchange sites. *Journal of Systems and Software*, 183:111113. 114

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058. 2

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE. 145

Phipps, S. and Zacchiroli, S. (2020). Continuous open source license compliance. *arXiv preprint arXiv:2011.08489*. 107

Qiu, S., German, D. M., and Inoue, K. (2021). Empirical study on dependency-related license violation in the javascript package ecosystem. *Journal of Information Processing*, 29:296–304. 2, 114

Ray, B., Posnett, D., Filkov, V., and Devanbu, P. (2014). A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165. 39

Reid, D., Jahanshahi, M., and Mockus, A. (2022). The extent of orphan vulnerabilities from code reuse in open source software. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2104–2115. 3, 5, 48, 141, 143, 145, 167

Reid, D. and Mockus, A. (2023). Applying the universal version history concept to help de-risk copy-based code reuse. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–12. IEEE. 93, 114, 141, 143

Roberts, J. A., Hann, I.-H., and Slaughter, S. A. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984–999. 2

Rosen, L. (2005). Open source licensing. *Software Freedom and Intellectual Property Law*. 113, 120

Roy, C. K. and Cordy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68. 31, 36, 86, 87

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495. 13, 33, 34, 35

Rubin, J. and Chechik, M. (2013). A survey of feature location techniques. 31

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168. 167

Samadi, M., Nikolaev, A., and Nagi, R. (2016). A subjective evidence model for influence maximization in social networks. *Omega*, 59:263–278. 28

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. 96

Serafini, D. and Zacchiroli, S. (2022). Efficient prior publication identification for open source code. In *Proceedings of the 18th International Symposium on Open Collaboration*, pages 1–8. 93

Shagall, Y. and Breithaupt, E. (2008). Jacobsen v. katzer: Federal circuit affirms economic interest of open source copyright holder. *Harvard Journal of Law & Technology*. Accessed: 2024-09-27. 108

Sim, S. E., Clarke, C. L., and Holt, R. C. (1998). Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 180–187. IEEE. 1

Software Freedom Law Center (2007). On behalf of busybox developers, sflc files first ever u.s. gpl violation lawsuit. Accessed: 2024-09-27. 108

Sojer, M. and Henkel, J. (2010). Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, 11(12):2. 13, 33

Srinivas, C., Radhakrishna, V., and Rao, C. G. (2014). Clustering and classification of software component for efficient component retrieval and building component reuse libraries. *Procedia Computer Science*, 31:1044–1050. 36

Stallman, R. (2002). *Free software, free society: Selected essays of Richard M. Stallman*. Lulu. com. 112, 121

Student (1908). The probable error of a mean. 43

Sutton, J. and Austin, Z. (2015). Qualitative research: Data collection, analysis, and management. *The Canadian journal of hospital pharmacy*, 68(3):226. 81

Svajlenko, J., Keivanloo, I., and Roy, C. K. (2013). Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 16–22. IEEE. 13, 34

Svajlenko, J. and Roy, C. K. (2014). Evaluating modern clone detection tools. In *2014 IEEE international conference on software maintenance and evolution*, pages 321–330. IEEE. 31

Svajlenko, J. and Roy, C. K. (2015). Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE. 31

Thompson, S. K. (2012). *Sampling*, volume 755. John Wiley & Sons. 123

Tsay, J., Dabbish, L., and Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th international conference on Software engineering*, pages 356–366. 44, 111

Tuunanen, T. (2021). Tool support for open source software license compliance: The first two decades of the millennium. *JYU dissertations*. 107

U.S. Copyright Office (2021). *Circular 1: Copyright Basics*. Library of Congress. Accessed: January 5, 2025. 154

Välimäki, M. (2005). *The rise of open source licensing: a challenge to the use of intellectual property in the software industry*. Turre publishing. 113

Vasilescu, B., Serebrenik, A., and Filkov, V. (2015). A data set for social diversity studies of github teams. In *2015 IEEE/ACM 12th working conference on mining software repositories*, pages 514–517. IEEE. 45

Vatcheva, K. P., Lee, M., McCormick, J. B., and Rahbar, M. H. (2016). Multicollinearity in regression analyses conducted in epidemiologic studies. *Epidemiology (Sunnyvale, Calif.)*, 6(2). 124

Vendome, C., Bavota, G., Penta, M. D., Linares-Vásquez, M., German, D., and Poshyvanyk, D. (2017). License usage and changes: a large-scale study on github. *Empirical Software Engineering*, 22:1537–1577. 103

Vendome, C., German, D. M., Di Penta, M., Bavota, G., Linares-Vásquez, M., and Poshyvanyk, D. (2018). To distribute or not to distribute? why licensing bugs matter. In *Proceedings of the 40th International Conference on Software Engineering*, pages 268–279. 107

Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc. 40, 41

Weller, K. and Kinder-Kurlanda, K. E. (2016). A manifesto for data sharing in social media research. In *Proceedings of the 8th ACM Conference on Web Science*, pages 166–172. 42

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE. 13, 34

Wolter, T., Barcomb, A., Riehle, D., and Harutyunyan, N. (2023). Open source license inconsistencies on github. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–23. 114

Wu, J., Bao, L., Yang, X., Xia, X., and Hu, X. (2024). A large-scale empirical study of open source license usage: Practices and challenges. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 595–606. IEEE. 94, 113, 115

Wu, Y., Manabe, Y., Kanda, T., German, D. M., and Inoue, K. (2015). A method to detect license inconsistencies in large-scale open source projects. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 324–333. IEEE. 114

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE. 142

Xu, S., Gao, Y., Fan, L., Liu, Z., Liu, Y., and Ji, H. (2023). Lidetector: License incompatibility detection for open source software. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–28. 94, 115

Xu, W., Gao, K., He, H., and Zhou, M. (2024). A first look at license compliance capability of llms in code generation. *arXiv preprint arXiv:2408.02487*. 107

Yan, D., Niu, Y., Liu, K., Liu, Z., Liu, Z., and Bissyandé, T. F. (2021). Estimating the attack surface from residual vulnerabilities in open source software supply chain. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 493–502. IEEE. 4, 110

Yin, R. K. (2015). *Qualitative research from start to finish*. Guilford publications. 81

Zacchiroli, S. (2022). A large-scale dataset of (open source) license text variants. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 757–761. 94

Zhao, S. (2023). Github copilot now has a better ai model and new capabilities. *The GitHub Blog*. 140

Zhao, Y., Liang, R., Chen, X., and Zou, J. (2021). Evaluation indicators for open-source software: a review. *Cybersecurity*, 4:1–24. 4, 5

Zhuge, H. (2002). Knowledge flow management for distributed team software development. *Knowledge-Based Systems*, 15(8):465–471. 14

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., and Aftandilian, E. (2022). Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29. 140

Appendix A

License Types

List of SPDX license identifiers aggregated by their respective license types:

Permissive: 0BSD, AFL-3.0, Apache-2.0, BSD-2, BSD-2-Clause, BSD-3-Clause, BSL-1.0, ISC, Libpng, MIT, MIT-0, MITNFA, MIT-Wu, MS-PL, OpenSSL, PHP-3.01, Pixar, PSF-2.0, Ruby, SGI-B-2.0, TCL, WTFPL, Zlib

Copyleft: deprecated_AGPL-3.0, deprecated_GPL-3.0+, GPL-2.0, GPL-3.0+, GPL-CC-1.0, OSL-3.0

Weak Copyleft: Artistic-1.0-Perl, Artistic-2.0, CDDL-1.0, deprecated_LGPL-2.1, eprecated_LGPL-3.0, EPL-1.0, EPL-2.0, LGPL-2.0+, LGPL-3.0, MPL-1.1, MPL-2.0-no-copyleft-exception

Conditional Open: CC-BY-3.0, CC-BY-4.0, CC-BY-SA-3.0, CC-BY-SA-4.0, ODC-By-1.0, OFL-1.0, OFL-1.1

Public Domain: CC0-1.0, libtiff, Unlicense

Vita

Education

- **Ph.D. in Computer Science**
  University of Tennessee, Knoxville — May 2021 - May 2025
- **M.Sc. in Industrial Engineering**
  Sharif University of Technology — Sep 2011 - Sep 2013
- **B.Sc. in Industrial Engineering**
  Mazandaran Institute of Technology — Jan 2007 - Jul 2011

Professional Experience

- **Graduate Research Assistant**
  University of Tennessee Knoxville — May 2021 - Present
- **Senior Data Scientist**
  Mobile Communications Company of Iran — May 2019 - Apr 2020
- **Strategic Investments Lead**
  Mobile Communications Company of Iran — Feb 2018 - May 2019
- **International Investment Analyst**
  Mobile Communications Company of Iran — Feb 2016 - Feb 2018

Publications

- **Jahanshahi, M.**, Reid, D., & Mockus, A. "Beyond Dependencies: The Role of Copy-Based Reuse in Open Source Software Development". Accepted in ACM Transactions on Software Engineering and Methodology (TOSEM). 2025.
- **Jahanshahi, M.**, Reid, D., McDaniel, A., & Mockus, A. "OSS License Identification at Scale: A Comprehensive Dataset Using World of Code". Accepted in 2025 IEEE/ACM 22st International Conference on Mining Software Repositories (MSR). IEEE, 2025.
- **Jahanshahi M.**, Mockus A. "Cracks in The Stack: Hidden Vulnerabilities and Licensing Risks in LLM Pre-Training Datasets". Accepted in the Second International Workshop on Large Language Models for Code (LLM4Code). 2025.
- Miller, C., **Jahanshahi, M.**, Mockus, A., Vasilescu, B., & Kästner, C. "Understanding the Response to Open-Source Dependency Abandonment in the npm Ecosystem". Accepted in the 47th International Conference on Software Engineering (ICSE). 2025.
- **Jahanshahi, M.** & Mockus, A. "Dataset: Copy-based Reuse in Open Source Software". 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR) (pp. 42-47). IEEE, 2024.
- Reid, D., **Jahanshahi, M.**, & Mockus, A. "The extent of orphan vulnerabilities from code reuse in open source software". Proceedings of the 44th International Conference on Software Engineering (ICSE) (pp. 2104-2115). 2022. Nominated for ACM SIGSOFT Distinguished Paper Award.
- Lyulina, E., & **Jahanshahi, M.** "Building the collaboration graph of open-source software ecosystem". 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (pp. 618-620). IEEE, 2021.