Towards Safer Code Reuse: Investigating and Mitigating Security Vulnerabilities and License Violations in Copy-Based Reuse Scenarios

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

David Reid

December 2023

© by David Reid, 2023

All Rights Reserved.

Acknowledgments

I am grateful to the individuals whose support and guidance have been instrumental in the completion of this dissertation. Their encouragement and contributions have shaped the quality and direction of this research.

Foremost, I extend my gratitude to my advisor, Dr. Audris Mockus, for his mentorship, expert insights, and encouragement. Dr. Mockus, has been a constant source of guidance and inspiration, shaping the research's scope and quality.

I would also like to express my sincere appreciation to the members of my dissertation committee: Dr. Scott Ruoti, Dr. Jian Huang, and Dr. Jeffery Case.

I want to extend a special thank you to my wife, Crystal Reid, for her unwavering support, patience, and understanding throughout this challenging journey. Her encouragement and belief in me have been a constant source of strength.

I am also grateful to my esteemed co-authors of academic papers: Mahmoud Jahanshahi, Kristiina Rahkema, James Walden, Kalvin Eng, Chris Bogart, and Adam Tutko. Our collaborative efforts have significantly contributed to my growth as a researcher and have shaped the perspectives presented in this dissertation.

Abstract

Background: A key benefit of open source software is the ability to copy code to reuse in other projects. Code reuse provides benefits such as faster development time, lower cost, and improved quality. There are several ways to reuse open source software in new projects including copy-based reuse, library reuse, and the use of package managers. This work specifically looks at copy-based code reuse.

Motivation: Code reuse has many benefits, but also has inherent risks, including security and legal risks. The reused code may contain security vulnerabilities, license violations, or other issues. Security vulnerabilities may persist in projects that copy vulnerable code, even if fixed in the project from where the code was appropriated. License terms may not be propagated with the copied code, potentially causing license violations unknown to users of the project.

The extent of the spread of risks through copy-based code reuse, the potential impact of such spread, or avenues for mitigating those risks have not been studied in the context of a nearly complete collection of open source code.

Aim: We aim to find ways to detect security, legal, and other risks induced by copy-based code reuse, determine how prevalent they are, and explore how they may be addressed in order to help developers safely and effectively reuse code from other projects.

Method: We rely on World of Code infrastructure that provides a curated and cross-referenced collection of nearly all open source software to conduct a case study

of a few known vulnerabilities, conduct an empirical study of a large number of known vulnerabilities, and to produce a tool to help mitigate security, legal, and other risks.

Results: We find numerous instances of security vulnerabilities and license violations caused by copy-based code reuse in currently active and in highly popular projects. The often long delay in fixing orphan vulnerabilities even in highly popular projects increases the chances of it spreading to new projects. We provided patches to a number of project maintainers and found that only a small percentage accepted and applied the patch. We present an approach to produce a universal version history which links files across multiple repositories and multiple repository hosting platforms to construct a single history by tracing the version of a single file across all repositories and revision histories where either parents or descendants of that file reside. We then show how this approach can reduce the risks of copy-based code reuse.

Table of Contents

1 Introduction **1**

1.1 Overview . . . 1

1.2 Research Goals . . . 3

1.2.1 Case Study about Orphan Vulnerabilities . . . 4

1.2.2 Large Scale Empirical Study . . . 7

1.2.3 Universal Version History . . . 9

1.3 Background . . . 12

1.3.1 Software Reuse . . . 12

1.3.2 World of Code . . . 13

1.4 Application Scenarios . . . 15

1.4.1 Security Vulnerabilities . . . 15

1.4.2 License Compliance . . . 17

1.4.3 AI-generated code from Large Language Models . . . 18

1.4.4 Additional Scenarios . . . 19

1.5 Universal Version History Concept . . . 21

1.6 Summary . . . 23

2 Literature Review **24**

2.1 Large Scale Software Archives . . . 24

2.2 Code Reuse . . . 26

2.3 Software Provenance . . . 27

2.4 Package Managers . . . 29
2.5 Security and License Compliance Issues . . . 31
2.6 Universal History . . . 32
2.7 Commercial Tools . . . 33

3 Methods and Tools **35**
3.1 Case Study Research Methodology . . . 35
3.2 The VDiOS Tool . . . 38
3.2.1 Architecture . . . 40
3.2.2 Algorithm . . . 41
3.3 Large Scale Empirical Study . . . 42
3.3.1 CVEfixes Dataset . . . 45
3.3.2 The VCAnalyzer Tool . . . 45
3.3.3 Empirical Study Method . . . 47
3.4 The UVHistory Tool . . . 49
3.4.1 Infrastructure: World of Code . . . 49
3.4.2 UVHistory . . . 50
3.4.3 Algorithm . . . 51
3.4.4 Output . . . 52
3.5 Universal Version History . . . 53

4 Results **56**
4.1 Case Study Results . . . 56
4.1.1 Case 1: CVE-2021-3449 in OpenSSL . . . 57
4.1.2 Case 2: CVE-2014-0160 in OpenSSL . . . 59
4.1.3 Case 3: CVE-2021-29482 in Package xz . . . 62
4.1.4 Case 4: CVE-2017-12652 in libpng . . . 63
4.2 Empirical Study Results . . . 67
4.2.1 RQ1: Prevalence of Orphan Vulnerabilities . . . 68
4.2.2 RQ2: Characteristics of Projects that Copy Vulnerabilities . . . 69

4.2.3 RQ3: Orphan Vulnerabilities that are Fixed 70
4.2.4 RQ4: Projects that Fix Orphan Vulnerabilities 71
4.2.5 RQ5: Survival of orphan vulnerabilities 73
4.3 UVHistory Evaluation 75
4.3.1 RQ6: Can the declared license be trusted? 75
4.3.2 RQ7: Can UVHistory help with license compliance issues? . . 78
4.3.3 RQ8: Can UVHistory help identify projects with security vulnerabilities? 80
4.3.4 RQ9: Is the UVHistory prototype feasible? 81
4.3.5 Evaluation of Existing Tools 82
4.4 Limitations 84
4.4.1 Infrastructure 84
4.4.2 Methods 86
4.4.3 Tools 86

5 Conclusion **88**
5.1 Discussion 88
5.2 Future Work 93
5.2.1 Improvements to detection methods 93
5.2.2 Additional studies 94
5.2.3 VDiOS 94
5.2.4 UVHistory 95
5.2.5 Artificial Intelligence 96
5.3 Conclusion 97

References **100**

A List of Publications **114**

Vita **115**

List of Tables

4.1 Percentage of copied vulnerabilities with status fixed, not fixed, and unknown for each project language 72

4.2 Percentage of copied vulnerabilities with status fixed, not fixed, and unknown for each file ending for active projects. 74

List of Figures

3.1 VDiOS Architecture Diagram 39

Chapter 1

Introduction

1.1 Overview

The rapid growth of high quality open source software has significantly increased the different kinds of software that can be built upon, thus potentially enhancing developer productivity [1], increasing code quality [2], and improving software security [3] [4]. A key feature of open source code is that it may be copied into new projects*, but such copying may bring vulnerabilities, license problems, or other issues [3]. It is important for developers to carefully consider the risks and benefits of reusing open source code and to take appropriate measures to mitigate potential issues.

There are multiple ways to copy code into a new project, including:

- Copy-based code reuse: Existing source code is copied and committed into a new repository, and possibly modified.
- Library reuse: External libraries are linked (either statically or dynamically) into the new project.

*Subject to licensing terms of the original and target projects.

- Package Managers: A centralized repository of software packages, along with tools to manage installing, upgrading, and removing of the software, allows external software to be installed and used in a new project.

This paper examines risks of copy-based code reuse, where source code is copied and committed into the source code repository of the new project. The copied code may be modified over time in the new repository.

If developers link to external libraries or use package managers to manage their library dependencies, vulnerable dependencies can be fixed by simply upgrading software libraries once a fix is available. Tools such as OWASP Dependency-Check[†] and GitHub Dependabot[‡] exist that monitor public vulnerability databases and notify developers if their library dependencies are vulnerable. Unfortunately, despite the existing solutions, developers are reluctant to update their library dependencies [5]. How vulnerabilities spread in library dependency networks and how fast they are fixed in dependent projects has been studied for different ecosystems [6, 7, 8].

If developers copy code from other projects and commit that code into their own repositories, the copied code cannot be found by package dependency tools. Such copying for reuse in other projects is widespread [9, 10, 11, 12]. Much of the work on software supply chain issues, such as security and license management, focuses on software dependencies. A software dependency is generally considered an external component (such as a library or package) that is used within a project. When the external component is copied and committed into a project's repository, it is no longer an external component but rather is now part of the project. This copy-based reuse approach is sometimes called clone-and-own [13] [14] [15] or vendoring [16] [17]. The cloned component is clearly part of the supply chain, but it is often overlooked because it may be considered part of the core project rather than a dependency once it is committed into the project's repository. This clone-and-own method can cause problems with code maintenance because of the lack of information about the

[†] `https://owasp.org/www-project-dependency-check/`
[‡] `https://github.com/dependabot`

connection between the clone and the original. In fact, even the originating projects sometimes do not contain public security vulnerability fixes implemented in projects that copied the code.

Much of this work focuses on projects in languages like C and C++ because they do not have a standard package manager system. When using a package manager, it is easier to find the origin of the code and any known vulnerabilities or license issues. However, many projects using languages with good package managers don't take advantage of the package manager. Therefore, we also look at languages like Java, JavaScript, Python, etc, that do have standard package manager systems.

1.2 Research Goals

Existing research has identified a number of risks associated with code reuse in open source software, including security vulnerabilities and license violations. Much of the research considers the use of package managers or linking to external libraries (black-box reuse). This work examines copy-based code reuse (white-box reuse). It builds on the World of Code infrastructure to study risks in copy-based code reuse at a scale that has traditionally been computationally infeasible. The work is divided into three main areas:

1. We first conducted a case study of a few known vulnerabilities. To conduct our case study, we developed a tool, VDiOS, to help identify and fix white-box reuse induced vulnerabilities that have been already patched in the original projects (orphan vulnerabilities). We sent patches to project maintainers to see if they would accept and apply the fix.

2. Next, we built on the previous work by conducting a large scale empirical study to investigate vulnerabilities propagated through copy-based code reuse. We produced a dataset containing over 3 million files with known vulnerabilities that have been copied into over 700,000 unique open source projects. The data

set also contains metadata about each project. We examined the characteristics of projects that fixed and projects that did not fix the vulnerability once the fix in the original project was published.

3. Finally, we proposed a method and a tool to help mitigate the risks of copy-based code reuse by constructing a universal version history of a source code file across all repositories and repository hosting platforms contained in World of Code. We show that this tool can help identify copied vulnerabilities that other tools miss.

The following three sections describe these three areas in more detail.

1.2.1 Case Study about Orphan Vulnerabilities

We define "orphan vulnerabilities" as vulnerabilities in copied code that still exist in a project after they are discovered and fixed in another project. In some cases, the copying is a result of forking, and the link to the original code is readily available. In other cases, especially when the copying is a result of many iterations, the link to the original code may not exist. Either way, the vulnerable code is publicly exposed until the orphan vulnerability is fixed or the vulnerable code is removed. The aim of this study is to determine if the ability to copy open source code software actually results in widespread orphan vulnerabilities. Orphan vulnerabilities present significant risk for several reasons. First, an exploit for such vulnerabilities may be widely known, making it easier to attack software with known vulnerabilities [18]. Second, the code in such repositories may be copied to other projects that may not be aware of the vulnerability. Third, code in such repositories may be built into applications and run by unsuspecting users. Fourth, if a substantial number of open source projects contain known and unfixed vulnerabilities, open source may suffer reputational damage as a dump of low quality code where it may be hard to find high quality projects.

To better understand and address the problem of copied and unpatched code, we first would like to create a tool that, given a vulnerability fix in one project, identifies

all other projects that contain either still vulnerable or fixed code. Such a tool, if widely deployed, would have at least two positive impacts: inform maintainers and users of still vulnerable projects about the risks of the vulnerability in their code and warn users that contemplate reusing such code about the unpatched vulnerabilities.

Second, we want to determine if and how the still vulnerable projects may differ from the patched ones. For example, we expect that the more active projects are more likely to fix known vulnerabilities than the less active projects. This would suggest that the risks posed by unpatched projects may be attenuated by, presumably, more narrow deployment. Linus's Law states that "given enough eyeballs, all bugs are shallow" [19]. This would suggest that projects with more developers are less likely to contain vulnerabilities. But little empirical evidence exists to support this [20]. We want to see if our results support Linus's Law.

Third, we would like to understand how quickly patches to known vulnerabilities propagate to unpatched projects. We expect that older vulnerabilities are more likely to be fixed in a project than the more recent ones, as it takes time and effort for project maintainers to patch their project. Presence of such a trend would suggest that convenient tools supporting such patching may speed up the deployment of patches.

Fourth, we want to determine if the tool we introduced detects vulnerabilities of a different kind than one of the most widely known tools, Dependabot [21], to determine if the approach used in our tool is practically relevant or if developers may safely rely on Dependabot.

Fifth, we would like to identify how many of the projects that contain orphan vulnerabilities are not just forked from the original project where the vulnerability was fixed. Since many forks are done simply to contribute a patch, not to start a new development, it would not be surprising if such forks are not updated and do not patch their code. For any developer, it would be easy to look up the origin of the fork to get the most authoritative code. However, it may be harder to do with cloned

projects. If, on the other hand, many of the projects are not forks, it would be much more difficult for potential users to identify such authoritative versions.

Sixth, we would like to understand to what extent the still vulnerable projects are willing to accept patches of the vulnerability offered to them. For example, while Dependabot creates warnings and provides patches, not all projects are willing to accept them, as the patches may break functionality.

To produce the tool, VDiOS, we build on top of World of Code (WoC) [22] infrastructure that attempts to approximate the source code in public git version control systems and provides cross-references among versions of the code, projects, and changes to the code.

To answer our research questions, we employ a mixed methods approach where we analyze large volumes of data to select candidates for a case study. Such an approach is suitable for our investigation because on one hand we have a very large and complex datasource representing almost all open source code, and we need computational approaches to select meaningful examples for our case study. The case study approach is needed because we have limited understanding of the problems, and a case study approach provides "an in-depth, multi-faceted exploration of complex issues in their real-life settings" [23]. We carefully pick the subjects (vulnerabilities) to shed light on all of the above research questions.

It is important to note that here we are exclusively focused on the so-called white-box reuse where the source code is copied into a new repository. Furthermore, we only consider matching any exact version of the vulnerable code, though the approach can be straightforwardly extended to cases where the copied code has been modified and does not match exactly any of the known fixed or vulnerable versions.

We succeeded in building VDiOS, a tool that identifies projects with orphan vulnerabilities, and applied it in four cases investigating four vulnerabilities in PNG library, OpenSSL, and xz compression (written in Go language). None of the vulnerabilities were reported by Dependabot in thousands of vulnerable projects that are not forks of the original projects. Only a fairly small fraction of projects accepted

the pull request fixing the known vulnerability. On the positive side, we found older vulnerabilities to be more likely to be fixed, and the still-vulnerable projects tended to be less active than the patched ones.

In summary, our work in this area makes the following contributions:

- We provide a working approach to find file-level exact code reuse in any language across all open source repositories.
- We provide a tool to implement our approach.
- We conduct a case study with four cases to answer our research questions regarding vulnerabilities that are spread via file-level code reuse.

Our primary objective is to reduce security vulnerabilities in software by identifying cases where a known vulnerability has been fixed, but copies of versions that are still vulnerable are still in use in other projects. This is a well-known security risk in the software supply chain. The Open Web Application Security Project (OWASP) lists "Using Components with Known Vulnerabilities" in its Top 10 Web Application Security Risks (OWASP Top 10) [24]. The software supply chain is a significant source of data breaches [25], with one estimate suggesting 80% of such breaches come from supply chain vulnerabilities [26]. Finding file-level duplication and locating where a file originated helps identify vulnerable or buggy code.

1.2.2 Large Scale Empirical Study

We next build on the orphan vulnerability work described in the previous section to identify and study orphan vulnerabilities on a large scale. We use the CVEFixes dataset[§] to identify the original vulnerable files and their fixed versions. We then find copies of both vulnerable and fixed versions of these files in World of Code, along with metadata about the files and the projects to which they belong.

[§] `https://github.com/secureIT-project/CVEfixes`

If developers use package management software to install and update their open source library dependencies, then software dependencies can be identified by analyzing metadata stored in the repository by the package manager. There are a wide variety of dependency checking and software bill of materials (SBOM) tools that use this metadata to identify dependencies, then cross-reference software dependency data with vulnerability databases to identify to which vulnerabilities an application is exposed. Tools like GitHub Dependabot[¶] even submit pull requests to update vulnerable software components.

However, there are vulnerabilities that are hidden from dependency tracking tools because these vulnerabilities exist in dependencies that are not documented in package management metadata. Some developers use open source software libraries by copying the software into the code repository of a project that uses the library instead of using a package manager. Gharehyazie et al. [10] showed that cross-project code copying is prevalent on GitHub. Ossher et al. [9] studied Java projects and found that over 10% of all files in the studied projects were copied from other projects.

We conduct a large scale study of projects which copy files with known security vulnerabilities. We find characteristics of those projects that may affect the likelihood of vulnerabilities being fixed. Our work in this area makes the following contributions:

- We conduct a large scale empirical study to analyze security vulnerabilities in source code in multiple languages that are propagated through copy-based code reuse. Using the World of Code infrastructure, we are able to analyze the extent of cloning vulnerable files at a scale that has traditionally been infeasible.

- We present the design and implementation of a tool, VCAnalyzer, which finds source code files in any language with known security vulnerabilities that have been copied and committed to open source repositories. We also make the tool publicly available.

[¶] `https://github.com/dependabot`

- Using VCAnalyzer, we produce a dataset, which we make publicly available, containing over 3 million files with known vulnerabilities that have been copied into over 700,000 unique open source projects. The data set also contains metadata about each project.

1.2.3 Universal Version History

Version control systems were a major advance in software engineering by automating storing and making accessible a complete history of the source code within a repository. The rapid growth in open source software and its widespread use made obvious the need to create Universal Version History (UVH) (a full version history of a file across all repositories and revision histories where either parents or descendants of that file reside). More recently, the concept of a universal version history appears to be articulated in “Improving the Nation’s Cybersecurity” [27] presidential order with the key component of “Enhancing Software Supply Chain Security.” Specifically, “maintaining accurate provenance (i.e., origin) of software, providing a Software Bill of Materials, and ensuring and attesting to the integrity and provenance of open source software used within any portion of a product.” This recent presidential order highlights an issue, software supply chain security, that has long been known to be important for various reasons but which is often overlooked. While the Executive Order applies to all U.S. federal information systems, the same standards will provide security benefits to anyone and will certainly be required in many sectors, not just the U.S. government. In addition to the security implications explicit in the presidential order, knowing the software provenance and providing a software bill of materials is also vital for ensuring compliance with license requirements, finding additional useful features that may be available in different versions of the software, improving code quality problems, and addressing aspects of developer reputation. The realization of this idea, however, depends on the ability to collect, clean, curate, and integrate

version control system data from over a hundred million open source repositories, and remained out of reach for many years.

In this research, we present an approach to produce a universal version history which links files across multiple repositories and multiple repository hosting platforms to construct a single history by tracing the version of a single file across all repositories and revision histories where either parents or descendants of that file reside. We then show how this approach can reduce the risks of copy-based code reuse. Unknown provenance of cloned code can have a number of negative consequences. Our proposed method and tool can improve a number of areas including security vulnerabilities, license compliance issues, code quality problems, potential enhancements, and aspects of developer reputation.

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

To establish the feasibility of producing the universal version history in general and in being able to address the issues related to code copying, we introduce a prototype

tool, UVHistory, which automates the process of finding the universal version history of a file across all open source repositories. We build on the World of Code [22] infrastructure to discover and report the complete history of code in any language from a nearly complete collection of open source software. The results can be used to look for new features or functionality available in other revisions, look for security vulnerabilities reported in other revisions, find which developers have worked on the code throughout its evolution, look for license requirements that may have been lost as the code propagated, and more. Also, different revisions can be compared with public sources, such as the National Vulnerability Database, for known vulnerabilities or other bugs. This version history tracks changes to a specific file even when the file is copied to a new and possibly unrelated repository, allowing a developer to trace changes over time and across different repositories and different repository hosting platforms.

Our work in this area makes the following contributions:

- We propose a computationally feasible approach to produce a universal version history which links source code by content and its modification history across multiple repositories and hosting platforms.

- We present a prototype tool, UVHistory, that implements our proposed approach efficiently over the nearly complete collection of open source software in World of Code. Our tool is the first application of the universal version history concept using such a large collection of open source software.

- We evaluate the application of the universal version history concept to address two specific software engineering problems identified in this paper. We show that producing a universal version history can help mitigate problems with potential license violations and security vulnerabilities caused by copy-based code reuse.

- We show that the declared license of a project cannot always be trusted, and we show how our tool can help identify those projects with incorrect copyright and license information.

1.3 Background

1.3.1 Software Reuse

Software reuse is the practice of using existing software components when building new software systems [28]. There are two types of software reuse, often referred to as black-box and white-box reuse. Black-box reuse refers to external code that is used by a project but generally not committed into the project's repository. This may include, for example, linkable libraries. Black-box reuse is code that is not modified by the developer. White-box reuse refers to the case where source code is reused by copying the original code and committing the duplicate code into a new repository. White-box reuse code may be modified by the developer. White-box reuse results in multiple copies of the source code across multiple repositories. These copies may be changed, and therefore, there may be multiple different versions of the code. This paper specifically looks at white-box reuse. We look at code reuse on the individual file level, not at the function or method level.

White-box reuse presents several challenges. Vulnerabilities and other bugs may be found and fixed in a copy of the code that exists in one project, but the fixes may not get propagated to all projects that use the file. Similarly, useful enhancements may have been added to different versions of the code. The result is that fixes to known vulnerabilities as well as other bug fixes and enhancements may exist in one project but not in other projects. Also, license terms may not be properly propagated from the original code, causing license violations for developers who do not know the origins of the code. For quality and security reasons, it is important to understand where the reused code came from, who has worked on it, and if better versions of it

exist in other repositories. Knowing where else the code exists can help identify if there are known vulnerabilities in the code by seeing known vulnerabilities in other projects where the same code exists.

1.3.2 World of Code

World of Code (WoC) [22] is a large collection of open source project repository data collected from many different source code repository hosting platforms, including GitHub, GitLab, Bitbucket, SourceForge, etc. WoC contains detailed version control data, including commits, authors, and file blobs of more than 173 million repositories, encompassing a nearly complete collection of open source software. We used WoC version U, which includes data collected in October and November of 2021.

In WoC, commits are linked to files changed in that commit. Files are linked to metadata, such as timestamps and authorship, as well as file contents, which are called blobs. As a file changes over time, it is associated with different blobs, representing the contents of the file after each change. Blobs can belong to multiple commits and even multiple repositories. If a blob is connected to two repositories, this indicates that both repositories contain a file with identical contents. Therefore, it is possible to compare blobs to quickly find exact copies of any file in the WoC.

Due to the vast quantity of open source software available from many different public source code repository hosting platforms, it has traditionally been too computationally intensive to find origins of a duplicated piece of code and all revisions of that code across all open source projects. Therefore, previous research on code reuse has typically looked at a relatively small subset of open source software. However, World of Code (WoC) [29] opens up new research possibilities in this area. WoC provides an infrastructure that makes it possible to efficiently find all versions of reused source code files across all of the major source code repository hosting platforms. We build on the WoC infrastructure to find file-level code duplication in any language from a WoC's expansive collection of open source software. Additional

tools that build on WoC, such as Developer Reputation Estimator (DRE) [30], can be used to help identify the best of several versions of a file.

The tools described in this paper use the World of Code infrastructure to find duplicate code across many public source code hosting platforms. WoC is a nearly exhaustive and continually updated collection of open source software, along with tools to efficiently extract and analyze the extremely large set of code. Without the infrastructure provided by WoC, it would not be possible to find such a complete collection of code copied (and possibly modified) across such a large collection of code in many repositories across many hosting platforms.

Since most open source software today is stored in git repositories, WoC uses similar constructs to store the data. For example, blobs, trees, and commits in WoC are identical to the same objects in git and are referenced with a sha1 hash just like git.

Black-box reuse can be detected with static analysis techniques that look for dependencies. These dependencies can be checked against public sources like libraries.io. But white-box reuse, which is the subject of this paper, requires access to the source code for all projects from which code may be reused. WoC provides not only the near complete collection of open source software, but also organizes its databases for efficient searching.

WoC provides a number of mappings that allow us to efficiently extract the information that we need. WoC maintains a database of several objects including blobs, files, commits, projects, and authors, allowing for efficient mappings. For example, given the contents of a file, we compute the SHA-1 hash (using the same mechanism that git uses) that identifies the blob. We then use WoC's blob to commit mapping to get the SHA-1 hash of the commit. The commit to project and commit to time author mappings give us the project name (from which we can identify the git repo from which it came) and the author and time of commit (which helps us identify where the file originally came from). We also use the blob to old blob mapping to find old versions of the source code of a particular file.

In order to better understand the entirety of open source in our context, we need to get a handle on the set of distinct projects. Easy creation of clones in version control systems result in numerous repositories that are almost entirely based on some parent repository. There have been various attempts to detect communities in this ecosystem to address this issue, some using visual models [31] and some community detection algorithms [32]. We use the latter, utilizing WoC mappings that maps each repository to a central repository in a detected set of repositories which presumably represent the same project. This mapping is called project to deforked project (p2P) in WoC.

1.4 Application Scenarios

This section discusses scenarios of high-risk copy-based code reuse. The findings presented in this paper offer valuable insights for mitigating these risks.

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

1.4.1 Security Vulnerabilities

When a security vulnerability is discovered in open source software, it is typically documented in the Common Vulnerabilities and Exposures (CVE) system [33] maintained by The Mitre Corporation. Developers know to look at the CVE system for possible security vulnerabilities. However, when a vulnerable file has been copied to other projects, those other projects may not be listed in the CVE entry. When code with security vulnerabilities is cloned, the target project may inherit the vulnerability. When the vulnerability is found and fixed in the original project, the fix may not be propagated to the clone, especially when the target project does not maintain a link

to the parent. We coined the term "Orphan Vulnerability" to refer to these kinds of vulnerabilities that exist in copied code even after they are fixed in another project. We found many instances where vulnerable code in one project was copied into many other projects, the vulnerability was later identified and fixed in the original project, but still persisted in many of the copied projects, even many years later. Our proposed tool would aid developers in knowing the origin and history of the code, which would allow them to learn about the reported vulnerabilities in the original project.

It is also possible that a vulnerability is found and fixed in a file copied from an original project, but the fix is not back patched into the original project. Woo et al. [34] reported an example of this in the jpeg-compressor project [35]. CVE-2017-0700 [36] describes a vulnerability in the Android System UI that allows remote code execution. The file jpgd.cpp, which is the source of the vulnerability, was copied from the jpeg-compressor project. The vulnerability was discovered and reported in the Android source code [37], and a CVE was created. However, the vulnerability was not reported and not fixed in the jpeg-compressor project, which is the original source of the vulnerable file in Android. Therefore, developers who copy and reuse the Android code can easily find the vulnerability and the patch. However, developers who copy the jpeg-compressor project are not easily able to find out about that vulnerability, which was found and fixed in a derivative work. Clearly, it is not safe to assume that the original source is the best or most secure version. Our research aims to aid developers in finding not only the origin, but all revisions of a file across all open source repositories.

In order to secure the software supply chain, developers must know exactly what components are used in their product. Many organizations are now using Software Composition Analysis (SCA) [38] tools to find vulnerabilities in code in their dependencies. Many SCA tools match a project's dependencies with a database of vulnerable libraries. A key challenge in building this feature in SCA tools is figuring out what code is related to a reported vulnerability. That challenge is even more difficult if a project has copied code without maintaining a link to the original source

of that code. Our research aids in one small part of that challenge: finding cloned code in cases where the clone has no clear link to the origin of the code.

1.4.2 License Compliance

Another concern about copied code is license requirements. If a developer wishes to reuse software, it is important to understand the license terms of the original code. Trusting the declared license terms is not always safe. In some cases, the complete license information is not copied with the code. As an example, the jpeg-compressor project mentioned above is available under the Apache license version 2.0 or in the public domain. The Android version is only available as public domain. In most cases, public domain is a good option. However, in some jurisdictions, a specific license is better than public domain. Knowing the origin of the file would allow a developer to see the original license information, and they may choose to use the code from the original project in order to use a different license as allowed by the original project. We found many cases where the license terms were not propagated with the copied code. As just one example, libofa [39] contains license information in a file named COPYING in the top level directory. But the license information is not included in every source code file. Our UVHistory tool found several cases where projects copied the source files from libofa without copying the file that contains the license information. The result is that a developer who copies the copied code without knowing the origin will not have the license information, resulting in a potential license violation. It is important to understand the origin of the code being copied in order to comply with the license terms.

With the significant amount of code reuse in today's projects, it can be difficult to keep track of all of the license requirements. The license terms must be understood and met by the project maintainers. Also, the various licenses used within a project must be compatible with each other. Software with different licences can be combined, but combining such software increases the chances of license incompatibilities.

The 2021 Open Source Security and Risk Analysis report (OSSRA) [40] indicates that 65% of the codebases audited in 2020 have license conflicts with open source software. Additionally, it reported that 26% of codebases use open source software that has no license or a customized license. Knowing the license requirements can be complicated when a developer reuses an open source project which itself reuses components from other projects. In some cases, the original license information is not propagated with the code, which means that the necessary knowledge required to answer the license question is not available to the developer, making it practically impossible to comply with the license requirements. Therefore, finding the history of open source code is critical to understanding and complying with license requirements of reused code. The tool we present later in this paper, UVHistory, traces a file to its origins which helps identify missing license information.

Tools in the area of license compliance typically consider explicitly declared package licenses included via a package manager, a build system, or an exact clone of lines of code in file. We consider code copied (and modified) among projects multiple times without copying the license. This history allows us to find the original license (or vulnerability). We demonstrate that cases like this are common, and the universal version history we propose will allow developers to find all license information associated with a specific source code file.

1.4.3 AI-generated code from Large Language Models

The growing use of AI-generated code from Large Language Models (LLMs) raises concerns about potential risks, such as the presence of security vulnerabilities and license violations. These risks can be present in large language models for code (Code LLMs). LLM-generated code might inadvertently introduce security vulnerabilities or inadvertently incorporate code without proper attribution or adherence to open-source licenses, potentially causing security, legal and ethical issues. If open source code with security vulnerabilities is present in the training data of AI, including

Large Language Models (LLMs), it can inadvertently lead to vulnerabilities in the generated code. The Common Vulnerabilities and Exposures (CVE) database [33] and other sources identify known vulnerabilities, allowing the known vulnerable code to be removed from LLMs. However, our research shows that vulnerable code may be copied to other open source repositories with no clear link to the original code, and those copies may not be fixed when the original code is fixed. Additionally, these "orphan vulnerabilities" are not connected to the CVE entries, and thus there is no way to know that they should be removed from the LLMs. The result is that known security vulnerabilities exist in popular LLMs.

License issues with LLM-generated code arise from the potential incorporation of external code into the code generated by LLMs. We show that code is sometimes copied to new repositories without the proper license information being copied. Since code LLMs are trained on a vast corpus open source code, there's a possibility that the generated code might inadvertently include code that is subject to specific licensing terms that are unknown to the user. Legal issues surrounding LLM-generated code are not entirely clear because AI technology is so new and advancing so fast. In many cases, there is no legal precedent, and existing copyright provisions haven't been tested against AI.

The universal version history that we propose will help identify copied code with security vulnerabilities that might not be documented in sources like NVD, and with license terms that might not be otherwise readily identifiable.

1.4.4 Additional Scenarios

Our paper focuses on addressing security and licensing concerns. The universal version history concept can be valuable in other areas as well, including identifying other code flaws, such as defects, incompatibilities, or even missing functionality. It may even support better author attribution, considered an important motivation for

open source developers [30]. We briefly describe those scenarios here, but leave in-depth study of these areas for future work. Finding the complete file history, including all ancestor and descendant code, can be valuable for finding additional useful features available for a piece of software. Source code is often copied into different projects and then improved for use in that project. These useful enhancements or fixed bugs would often be valuable to other projects, but maintainers of other projects are often unaware of them. An old clone may be missing the latest enhancements/fixes that could add improved quality, functionality, performance, or other benefits.

Bug fixes beyond just security updates may be added to the original project or another clone of the project, but missing from the target project. Knowing the origin and history of the code can help developers find bugs that have been fixed in other revisions of that code and also to have an idea about the robustness of the code by seeing how many bugs have been fixed in other revisions. It is also important to know the likelihood that the original code will continue to be maintained in the future and to have bugs fixed.

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

Including arbitrary source code from an open source repository has trust and security issues. How can the developer trust that the cloned code is of high quality and does not contain security issues. Knowing which developers have worked on the specific code in question can help build trust in that code. Once our tool has identified all authors that have contributed to a specific source code file, systems such

as Developer Reputation Estimator (DRE) [30] can help identify the reputation of those authors, which can help build trust in the safety of using that file.

1.5 Universal Version History Concept

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

It is worth defining UVH more precisely and comparing to a common version control system such as git. The essential entities are versions of the source code (blobs). In git, each blob can be associated with all versions (commits) of the repository where it is present, and each version may be associated with one or more filenames (including the full path from the root of the repository). The blobs associated with such file with a pathname can be used to determine a version history of a file (and git provides several heuristics methods on how to do that: the lack of determinism of file history arises with merges). A particular commit "creates" a blob if there was no such file in the previous version (parent commit(s)) or if the previous blob was different. We can thus use the time of the first commit creating a blob (the same blob can be created multiple times in different folders or even in the same folder) to obtain the time when the blob was introduced to a repository. Furthermore, each time a blob is created by a commit, we link it to an old blob: a blob (if any) that exists in the parent commit for the same filename. Hence within a repository is simply a graph linking each blob to the "old blob" and to commits that created it (including all commit attributes such as time, author, commit message, and the pathname of blob-associated file). Notice that it is a bit different from version control systems

such as CVS or SVN, where versions of individual files are tracked. Notice that the resulting graph has several distinct kinds of nodes (e.g., blobs and commits), and multiples types of links (e.g., blob to old blob, commit to parent commit, and blob to creating commit). In UVH, we simply add one more type of node: a repository. Each repository is linked to all commits within that repository and, transitively, to all blobs contained therein. Blobs, on the other hand, are linked to all commits in all repositories. The first creation time for a blob is defined the same way. We thus can identify the original commit and original author for every version of the source code in World of Code. In addition to the time of the commit, a partial temporal order based on the old-new blob relation is available. Notably, as we expand the scope of UVH across repositories we lose some aspects of a sequence. For example, let blob a be first created in repository A, then in B, and lastly in C. In such case, without additional information (who did the commit and what other blobs were created), it would be impossible to determine if repository C got the blob from repository A or from repository B. For some applications, determining if the blob came from A or B is not as important as identifying licenses, vulnerabilities, or other attributes of the blob that may vary among these various repositories. In cases where knowing the true origin is critical, the UVH reduces the search space required so that manual inspection is feasible. The tool cannot know the origin for sure, but can point to the earliest commit into a public repository.

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

1.6 Summary

This paper investigates the area of code reuse in open source software, focusing on the copy-based approach. While acknowledging the benefits such as faster development and cost reduction, it highlights the accompanying security and legal risks posed by reused code, including vulnerabilities and licensing issues. Leveraging the World of Code infrastructure, the study conducts a multi-faceted examination, encompassing case studies of known vulnerabilities, large-scale empirical analyses, and the development of a pioneering tool to mitigate these risks. The research uncovers widespread security vulnerabilities and licensing violations in active and popular projects. It also introduces a novel method for creating a universal version history, spanning repositories and repository hosting platforms, to enhance the safety of code reuse in open source development. This study identifies the risks and also provides solutions to foster safer code reuse practices.

Chapter 2

Literature Review

In this literature review chapter, we examine the existing research concerning risks from code reuse in open source software, and we identify gaps in the research that need further investigation.

Significant amount of research in the area of code reuse is dominated by studies of projects using package managers or linking external libraries, so-called black-box reuse. Research on white-box code reuse, where code is reused by copying the original code, possibly modifying it, and committing the duplicate code into a new repository, is limited due to the difficulty of searching the entirety of open source software looking for duplicates. Using the World of Code [22] infrastructure opens new research possibilities in the area of white-box code reuse. We use World of Code to find cases of code reuse across open source projects.

2.1 Large Scale Software Archives

Most prior research in the area of copy-based code reuse is limited to a small number of repositories relative to the totality of open source projects and is limited to a small number of languages. Newer technologies such as World of Code [29] and Software Heritage [41] provide an infrastructure to find copied files across a much larger set of software repositories. World of Code and Software Heritage provide large scale code

archives. We use World of Code in our work. Related work using Software Heritage is also relevant.

World of Code [22] is a very large collection of version control data for open source projects that are hosted on many different source code repository hosting platforms, including GitHub, GitLab, BitBucket, SourceForge, and more. World of Code contains detailed version control data, including commits, authors, and file blobs of more than 173 million repositories, encompassing a nearly complete collection of open source software. Our tools are layered on top of the World of Code infrastructure to leverage this huge collection of open source repositories, allowing us to study copy-based code reuse on a very large scale. The World of Code's periodically updated and curated data allows our tools to efficiently search for code duplication in any language across many different source code repository hosting platforms.

The Software Heritage project aims to collect and preserve all open source software source code. They collect as complete a collection of software as possible and encourage others to create curated archives on top of Software Heritage to collect useful software from the large amount of noise in the full collection. Some work using Software Heritage is related to our work. Software Heritage Graph Dataset [42] links together source code file contents, which allows duplicate code to be found across projects, much like what is provided by the World of Code data maps that we use. What they do not include is the linkage of the history of each file within a project to all other projects containing any version of the file. This linkage is what our UVHistory tool provides. Provenance work by Rousseau et al. [43] using Software Heritage looks at occurrences of the "exact same file content." They specifically state that they do not look at "predecessors or successors in a given development history" and that that is "outside the scope of the present work." The strength of our work, and much of the effort to produce it, comes from tracing the full history by following the predecessors and successors, thus giving us a complete history that follows the evolution of a file as it changes over time, not just instances of exact copies.

2.2 Code Reuse

There is significant research in the area of code reuse through code cloning [44, 8, 45, 46]. This kind of reuse is sometimes called vendoring [16, 17] or clone-and-own [13, 14, 15]. Since our work looks at copy-based code reuse, the use of package managers is not as relevant. However, some of the work is similar, therefore, we include some related work in that area as well.

Schwarz et al. [47] studied cloned methods in the Squeaksource ecosystem and found that 15% to 18% of methods were cloned. Ossher et al. [9] studied 13,000 Java projects from the Sourcerer Repository. They found that over 10% of all files are clones and that 15% of projects contain at least one cloned file. They found that most commonly cloned files were Java extension classes and popular third-party libraries.

Gharehyazie et al. [10] looked at the prevalence of cross-project code reuse and report large amounts of code cloned across multiple projects. They find that most cloned code comes from projects in a similar domain. GitHub was the only repository hosting platform that they looked at, and Java was the only language. In our work, we look at code in many different languages and from many different repository hosting platforms including GitHub, Bitbucket, SourceForge, GitLab, and more.

Xia et al. [46] performed an empirical study to find the proportion of out-of-date third-party code reused by C language open source projects. Using OpenCCFinder [48], which used external code search engines Google code search and SPARS [49], they found 123 projects that reused outdated code copied from three original projects. Similar to our findings, they determined that a significant number of open source projects reused out-of-date code that contain security vulnerabilities. They report that OpenCCFinder only returns "a very small subset" of open source projects. By using our VDiOS tool layered on top of World of Code's nearly complete collection of open source software in any language, we are able to find a significantly larger number of projects that reuse vulnerable code. They also found that 18.7% of

the projects studied copied only the source file but no companion files like readme or changelog files; therefore, the version information and links back to the original project are lost. This is particularly relevant to our study of license terms, as the license and copyright information is often only in the companion files.

Kawamitsu et al. [50] studied code reuse across repositories but only looked at reuse between pairs of repositories rather than across the full spectrum of open source repositories. They introduce a method to detect code reuse across 2 repositories.

Decan et al. [8], through empirical study using Java projects that use Maven [51], show that it is common practice to use third-party software components that have known security vulnerabilities, suggesting that what we found for C and Go languages in white-box reuse also applies to black-box reuse in Java. Alqahtani et al. [52] link the NVD* with Maven to identify known vulnerabilities in Maven projects. We expand on that by including white-box reuse and by looking at projects in any language that may not use or have management tools like Maven.

2.3 Software Provenance

Software provenance refers to the history of source code, including the origin and chain of custody. This section considers work about provenance and how that relates to our work finding the origin of a file and how it evolves over time and across different repositories. Our solution, implemented in our UVHistory tool, traces the provenance of a file to identify risks in reused code.

Ishio et al. [12] proposed a method to find the original version of cloned source code files. Their method finds files that are similar, not just files that are exact copies. We only look for exact copies of any revision of the file. Their method may find additional matches that our method would miss due to minor changes in a cloned file before it is committed the first time. Our method may find matches that theirs miss because we run it over a much larger dataset of code repositories.

*National Vulnerability Database: https://nvd.nist.gov

Inoue et al. [53] designed and implemented a tool that used source code search engines to take source code fragments and find sets of cloned code fragments in order to track the history of the code. Limitations of those search engines, such as only allowing keywords and/or code attributes as their inputs or not allowing automated queries, posed challenges to the tool. The source code search engines they used (Koders, Google Codesearch, and SPARS/R) are no longer available. We use World of Code, which is currently actively maintained.

Davies et al. [54] introduced a method to reduce the search space when looking for the origin of a piece of code in cases where a direct link to the origin is not clearly available. Once the search space is reduced, manual inspection or other expensive methods can be used to identify the origin from the reduced set. They demonstrated their method on a collection of Java files.

Godfrey et al. [55] pointed out that it is becoming increasingly important to determine the origin of software in cases where code is cloned into a new project with no clear link to the origin, but that effective techniques for finding such code provenance do not yet exist. We aim to help fill the gap that they identified.

Woo et al. [34] proposed an approach to find the original software where a vulnerability originated, and they created a tool V0Finder to find the origin of a vulnerability. They noted that many CVE [33] reports do not give the correct origin of the vulnerability. Finding the true origin can help mitigate further propagation of the security risk. Their method uses function-level clone detection methods, which can be more precise but not as efficient at large scale as the file-level clone detection we use. They only used about 10,000 projects, and only from GitHub, for their evaluation.

Kawamitsu et al. [11] proposed a technique to find which file revision a copied file comes from in another project for the purpose of keeping copies up to date. They aimed to identify which revision of a file was reused and how that file was modified over time. Their method only looked at project pairs to find files that were copied from one project to the other, but it cannot handle a large number of projects. Ishio

et al. [12] expanded on the idea of tracking code changes by taking a set of source files in C/C++ and Java and finding files that are likely to include the original version of the file. They look at a relatively small subset of projects compared to what is available in World of Code. They note that tracking file changes across repositories is tedious. We further expand code change tracking by using World of Code's massive collection of projects to track modifications to files in any language across a nearly complete collection of open source software.

SZZ Unleashed [56] finds information about when bugs were introduced. Currently, VDiOS relies on the user to specify the commit that introduced a vulnerability, but if it is not available, all previous revisions of a file are considered vulnerable. Using SZZ might reduce that set. SZZ Unleashed could help us identify when the vulnerability was introduced so that we can rule out revisions prior to the vulnerability being introduced.

2.4 Package Managers

Part of the motivation for our work arises from a lack of research and tools dealing with copy-based reuse induced vulnerabilities. GitHub's Dependabot [21] creates pull requests for projects that rely on vulnerable libraries but only works for GitHub projects and only when dependencies are defined in a supported package ecosystem. Our VDiOS tool, on the other hand, looks for file level code duplication and does not rely on supported package ecosystems. VDiOS also works with projects across all repository hosting platforms, not just GitHub. Other popular dependency checking tools such as GitHub Dependency Graph [57], Google Open Source Insights [58], and OWASP Dependency Check [59] depend on package management metadata and thus miss copy-based code reuse. Much prior research on vulnerabilities arising from code reuse looks at reuse through package managers [60, 7, 6, 52].

Kula et al. [61] studied how developers update library dependencies in over 4600 GitHub projects. They found that 81.5% of the analyzed projects contain outdated

dependencies, and that 69% of the interviewees claimed to be unaware of their vulnerable library dependencies. The developers also cited extra workload as reason not to update library dependencies.

Alfadel et al. [60] studied 550 vulnerability reports in the Python ecosystem (PyPi) and found that the number of vulnerabilities in Python packages is increasing over time, and that vulnerabilities take more than three years to be discovered on average. The majority of studied vulnerabilities (50.55%) are only fixed after being publicly announced.

Zimmermann et al. [7] studied dependencies between package maintainers, as well as the packages themselves. They examined 609 vulnerabilities in 5,386,237 package versions with 199,327 maintainers. They found the mean number of dependencies for an npm package to be 79 packages and 39 maintainers. Packages in the npm ecosystem have a higher number of dependencies than Java packages and include micropackages with only a few lines of source code. They found that up to 40% of packages have dependencies with at least one known vulnerability.

Düsing et al. [6] analysed the direct and transitive impact of vulnerabilities in Maven, NuGet, and npm. They found that over 25% of vulnerabilities are still unpatched and that almost 75% of patches are released before the vulnerability is disclosed publicly. On average, a patch for a vulnerability is released 184 days before public vulnerability disclosure. They found that 16.3%, 15.5% and 0.5% of all libraries for npm, Maven, and NuGet, respectively, are released with publicly known vulnerable dependencies. Some of the vulnerable dependencies were updated immediately after public vulnerability disclosure, suggesting that monitoring of vulnerability databases and automatic dependency updates might be configured for some projects.

Decan et al. [62] studied 399 vulnerability reports affecting 269 npm packages and 6,752 releases of those packages. They found that 72,470 other packages are affected by those vulnerable releases through dependencies. Of the 269 vulnerabilities, 1/3 were fixed by discovery date, 1/2 were fixed after discovery but before publication date, and 15% were fixed after discovery date or not at all. They found that more

than 40% of packages cannot be fixed by upgrading the vulnerable package due to dependency constraints that do not allow the fixed package to be installed.

Pashchenko et al. [63] studied dependency managements and its security implications by interviewing developers. They found that developers focus on functionality over security when choosing dependencies. While developers perceive security fixes easy to adopt, they avoided updated dependencies, if possible, to avoid breaking changes and preferred security fixes that did not include improvements in functionality. If no fix is available, developers preferred to disable affected functionality or to do nothing, although a few developers had created their own fixes.

Our tool specifically looks at cases where code is copied from one project and committed into the repository of another project. Most commercial SCA tools also reply on package manager information, although some include limited support for vendoring.

2.5 Security and License Compliance Issues

Davies et al. [64] performed manual license and security audits in real-world applications and found potential legal and security issues in some of the studied applications.

Kula et al. [5] looked at Java projects that use a dependency management tool and found that 81.5% of projects in their study still have outdated dependencies, many with security vulnerabilities. They also, through surveys, found that 69% of the developers were not aware of the vulnerability. We hypothesize that the number of outdated cloned copies of files that have no link back to the origin would be even higher.

Chen et al. [65] designed and implemented a machine learning system to help identify which libraries in open source dependencies contain vulnerabilities listed in the National Vulnerability Database (NVD) [66]. It relies on package management systems including Maven Central, npmjs.com, and PyPi.

German et al. [67], through an empirical study of license issues in open source projects, show instances of incompatible licenses when open source code is reused in different projects. They found that there are often mismatches between the declared license of a package and the license of the source code within the package, and also incompatibilities between packages contained within one project. They note that auditing of license issues is “quite complex” and suggest that improving automation in this area would be beneficial. This kind of automation improvement is exactly the aim of our work.

Wu et al. [68] looked at license inconsistencies within large projects. In their conclusion and future work section, they say “These problems highlight the need for a method to find and maintain provenance between applications.” Our work, using World of Code, looks for inconsistencies across all open source projects as they suggest.

Wolter et al. [69] found that the license declared at the top level of the repository does not always match the license found in source code files. With this information, we know that we need to look deeper to know the correct license terms for a particular source code file.

Baltes et al. [70] show, through a large-scale empirical study of Java code snippets from Stack Overflow, that many code snippets are used without complying with license terms.

Qiu et al. [71] looked at dependency-related license violation and report a relatively small number of dependency-related violation in npm. The small number is in part because permissive licenses are more common in npm. Our work looks at code clones rather than dependencies.

2.6 Universal History

Early work on finding a complete version history was conducted by Chang and Mockus [72]. They looked for cases where directories of source code contain many

files with the same names and then compared those files to find clones. The matching files and their version histories were used to construct the file history. In follow-up work [73], they proposed a large-scale copy detection and validation process and improved reuse detection. Mockus [74], using the same algorithm, found significant large scale code reuse where many files were copied. At the time of their work, there were no complete collections of open source code like World of Code, which limited their work to a small number of repositories and only worked when multiple files in a directory were duplicated and the filenames did not change. They concluded that there was still a challenge to scale the work to very large numbers of open source repositories [73]. World of Code provides the infrastructure to meet that challenge, which is the goal of this work.

2.7 Commercial Tools

There are many commercial Software Composition Analysis (SCA) tools which help find security, license, and code quality issues. Because these tools are proprietary, they are harder to evaluate. Our primary goal is to expand on the publicly available research. But we also look at these commercial tools. Some tools claim to find clones from a large collection of open source software, but we do not have access to that collection and cannot evaluate its completeness. We tested some of those tools and point out where our work is different.

Current open source SCA tools that detect license compliance issues typically look at licenses that are explicitly declared in a project being reused through code clones or through a package manager. They trust the declared license in a project or source code file. What they fail to find are cases where code is copied from project to project multiple times, and sometimes modified, without the license information also being copied. The history is lost, making it impossible to find the original license.

Similarly, with vulnerabilities, current tools fail to trace the history as a file is modified and copied across repositories and, therefore, often miss vulnerable code

that has been copied from a known vulnerable project to a different project. Our research shows that cases like this are common and that our tool can help identify these cases.

We tested several commercial tools by creating a small test case example project. We find that, in some specific cases, our tool can help a developer find an issue that commercial SCA tools miss.

Chapter 3

Methods and Tools

In this chapter, we present the methodologies and tools driving our research, aimed at highlighting the risks of copy-based code reuse and enhancing the security and reliability of open source development. Our approach encompasses a case study, including the development of VDiOS, a specialized tool to identify white-box reuse-induced vulnerabilities. We reveal the findings of a large-scale empirical study spanning millions of files and thousands of open source projects. Additionally, we introduce a universal version history tool that transcends repository and platform boundaries, enabling the identification of copied vulnerabilities.

3.1 Case Study Research Methodology

We conducted an exploratory case study to better understand issues surrounding the spread of software security vulnerabilities caused by copying open source software. We chose to use an exploratory case study because we are in the early stages of understanding the problem and possible solutions. We hope to generate ideas to mitigate these types of security vulnerabilities and spur additional academic research. The case study approach allowed us to look at a small number of widely-reused projects in depth and within their real-life context. This in depth examination allowed

us to increase our understanding and gain insights that would otherwise be difficult to obtain.

Consistent with best practices conducting case studies, we investigated a small number of cases in depth and in their context using multiple data sources and emphasizing qualitative data and analysis while also collecting significant quantitative data. The subject of each case is a known vulnerability (as described by the Common Vulnerabilities and Exposures (CVE) database [33] hosted at MITRE) and the open source project containing the vulnerable code as described by the CVE entry.

We examined in detail four specific cases of known software security vulnerabilities that have been fixed in their original project repository. We used multiple cases to increase the confidence of the results and increase generalization of the results. We avoid making broad generalization claims based on just four cases, although we believe that our results provide insights that are applicable to a broader range than just our four specific cases. We carefully selected these four cases by searching for vulnerabilities in popular open source projects that have been widely copied. We used VDiOS to screen out cases of known vulnerabilities in code that is not widely copied. We specifically selected common cases, not unique or edge cases. We selected a vulnerability in libpng* that was in the code for a long time, allowing for many copies over that time. We selected a new and an old vulnerability in OpenSSL† to highlight differences in the age of the vulnerability. OpenSSL was chosen in part because it is critical to Internet security. We selected the xz package‡ written in the Go language to contrast with the other projects that were all C language projects. We selected cases that we believe are representative of the broader group of known vulnerabilities in open source software. Our cases represent both literal replication (because they are representative of the broader group of known vulnerabilities) and theoretical replication (because we compare an old vs new vulnerability in the same project and projects in different programming languages).

*http://www.libpng.org/
†https://www.openssl.org/
‡https://github.com/ulikunitz/xz

We examined these four cases in context by looking at specific open source projects that reuse vulnerable code and that are hosted on public hosting platforms including GitHub, Bitbucket, SourceForge, and others. By looking at the cases in context, we see a realistic picture of vulnerable code reuse in the real world rather than contrived results we might get in a traditional lab-based study.

Multiple data collection techniques provided corroborating evidence. We used artifacts, observations, and direct contact with project maintainers (through pull requests and issues), allowing us to gather more insights than using just one method. Looking at artifacts (the code actually committed in real world repositories) provides a concrete view of actual practice without any biases. Our observations allow for some qualitative analysis. Contacting project maintainers provides more insight into their willingness to address issues once they are aware of the vulnerabilities.

Case studies tend to focus more on qualitative data than quantitative data. Our study contains both. VDiOS produces significant quantitative data, which we report in detail. We also attempt to describe behavior based on our observations and interactions with project maintainers. Our results are also being used to craft survey questions for future research to collect more qualitative data to explain the behavior of project maintainers.

Our primary data source is World of Code. We also collect some data directly from the source code hosting platforms like GitHub, Bitbucket, SourceForge, and others. Data collection is accomplished using the VDiOS tool that we developed specifically for this research project. VDiOS is described in detail in section 3.2.

We started by selecting a sample of known vulnerabilities, identifying all affected (and fixed) versions of the source code files in the primary repositories and using WoC to identify all other OSS projects that have versions of the code that either precede (in version history) the affected version or is modified past it without applying the patch. We codified this algorithm as a tool that can be used for any vulnerability or any other type of defect. We then obtained and analyzed the numbers, activity states, and properties of the affected, patched, and potentially patched projects. Furthermore,

we manually investigated many instances of cases where the code is still vulnerable to identify if the project is still active, if the defect has been fixed, and if not, whether the maintainers are willing to accept the patch.

3.2 The VDiOS Tool

In this section we describe VDiOS (Vulnerability Detection in Open Source), our tool for finding file level code reuse across all open source repositories and tracing the version of a single file across all repositories and version history. We build on the WoC infrastructure to find duplicate files at a scale that has traditionally been computationally infeasible.

VDiOS takes the contents of a file and finds all duplicate versions of that file or any revision of that file across all of the open source software available in WoC. These vulnerable files are then traced back to the open source project in which they are contained. These projects may be hosted on many different source code hosting platforms such as GitHub, Bitbucket, SourceForge, etc. VDiOS displays a URL link to the project and the affected file or files within the given project.

Our approach looks for file-level reuse, that is, exact copies of entire files. We include all files in the version control history when looking for duplicate files. This allows us to find files that were duplicated and then modified.

When looking for Security Vulnerabilities, VDiOS has the ability to separate revisions of files into two lists: revisions that contain the vulnerability and revisions that do not contain the vulnerability. This allows VDiOS to identify projects that are still vulnerable, projects that used to be vulnerable but have now been fixed, and projects that used to be vulnerable and have changed but we do not know if the change fixed the vulnerability.

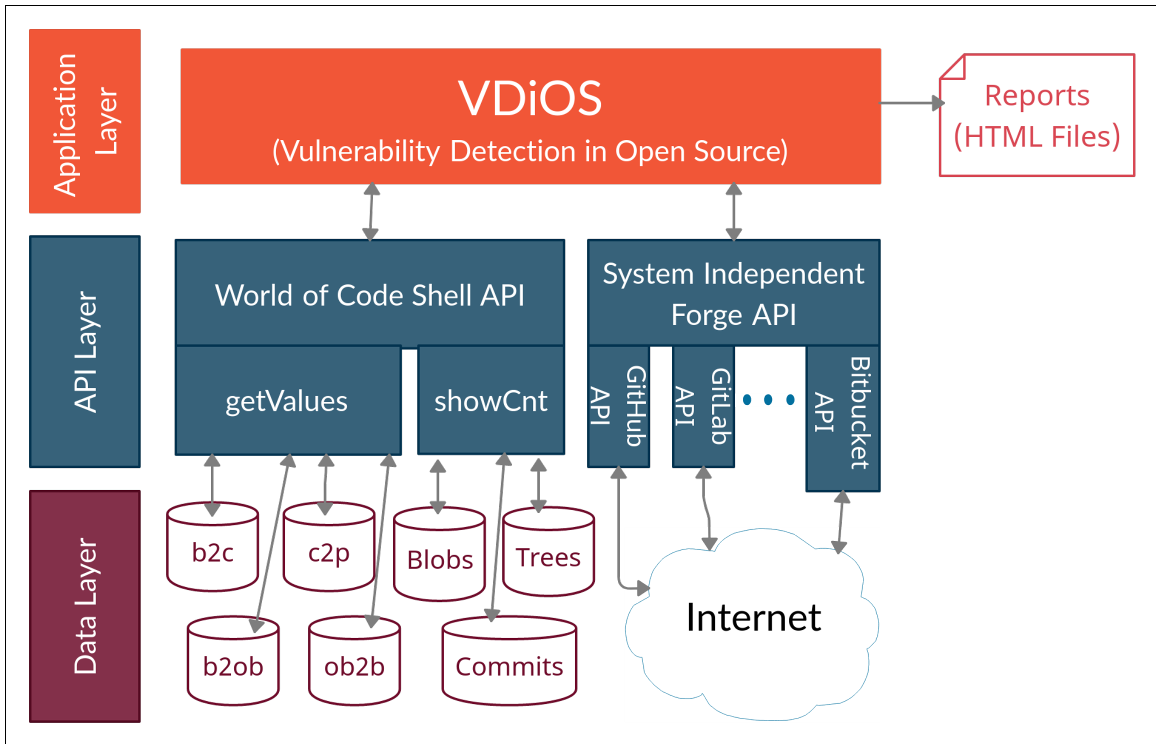

Figure 3.1: VDiOS Architecture Diagram

3.2.1 Architecture

VDiOS is implemented as a layer on top of the World of Code (WoC) shell APIs [29] as shown in figure 3.1. The WoC shell APIs provide a convenient way to access the WoC data. Specifically, VDiOS needs access to WoC's data maps as well as information about the objects. WoC stores data maps in a way that allows VDiOS to efficiently look up information. The specific information we need pertains to blobs, commits, projects, files, authors, and times.

There are two primary WoC shell APIs used by VDiOS: getValues and showCnt. showCnt is used to show the content of the basic git objects blob, tree and commit. getValues is used to access the WoC data maps to get the following information:

- blob to commit (b2c) finds all commits of the specified blob.
- commit to project (c2p) finds all projects containing the specified commit.
- commit to Project (c2P) is like c2p except it finds deforked [32] projects.
- commit to parent commit (c2pc) and commit to child commit (c2cc) finds the parent and child commit respectively from a given commit.
- commit to time author (c2ta) finds the time of the commit and the author of the commit.
- blob to old blob (b2ob) finds the predecessor of the given blob. old blob to blob (ob2b) is the inverse of b2ob.

VDiOS also retrieves a small amount of data directly from the source code hosting platform (GitHub, Bitbucket, GitLab, etc). A system independent interface allows VDiOS to use a single call to get data, hiding the platform specific details. A system dependent layer, which calls the appropriate API (for example, the GitHub API), provides a "glue layer" to connect to the popular hosting platforms to retrieve the data. The system dependent layer can be extended to support additional hosting platforms as needed.

The VDiOS output is a set of reports generated in HTML format for viewing in a web browser.

3.2.2 Algorithm

VDiOS is divided into four phases, each of which is described in this section.

The first phase identifies all of the blobs that contain the vulnerability and all the blobs that contain the fix. Starting with a commit that fixes a vulnerability, VDiOS finds the relevant blob or blobs in that commit. When looking for a security vulnerability, it is likely that not only is the revision before the fix vulnerable, but the predecessors of that revision are also likely to be vulnerable. VDiOS uses WoC's blob to old blob mapping or commit to parent commit mapping recursively to find all predecessor blobs. If we know the commit that introduced the vulnerability, VDiOS looks at only blobs between the breaking commit and the fixing commit. It is highly likely that all of those blobs will contain the vulnerability. VDiOS next finds the descendent blobs using WoC's old blob to blob mapping or commit to child commit mapping. These blobs are highly likely to contain the fix. Manual inspection of these lists can be done at this point to confirm that the blobs in the first list are vulnerable, and the blobs in the second list are fixed. At the end of phase one, we have two lists of blobs. The first list contains one or more blobs that contain the vulnerability. The second list contains zero or more blobs that are fixed.

The second phase searches for all projects in WoC that contain a duplicate of any of the vulnerable blobs identified in phase one by using WoC's blob to commit mapping and commit to project mapping. Note that VDiOS looks for duplicates in any revision within a project. That is, it will find all projects that have ever contained the vulnerable blob even if it has been fixed or removed in the most current version. At the end of phase two, we have a list of all projects that have ever contained one of the potentially vulnerable versions of the file.

The third phase checks if the blob(s) in question are in the most current revision of the project. In this phase, VDiOS looks through the projects found in the second phase. Those are projects that have at some point in time contained a known vulnerable blob. We now want to find out if the project still contains a vulnerable blob. Using the hosting platform's API, we find the most current revision of the file. Now we look to see if that revision matches any of the vulnerable blobs. If so, we know the project still contains the vulnerable code. Next, we look to see if that revision matches any of the known good blobs. If so, we know that the vulnerable file has been fixed. If we do not find a potentially vulnerable or known good file, then we know that the project has contained a vulnerable file, that file has been changed, but we do not know if the change fixed the vulnerability.

The final phase generates the reports in HTML format for viewing in a browser. The first page of the report shows the commit that fixed the vulnerability (if applicable). Next, it has a link to a list of blobs and filenames where the vulnerability was fixed, a link to a list of ancestors of those blobs (which presumably contain the vulnerability), and a link to the descendants of those blobs (which presumably all contain the fix). Finally, it has links to lists of vulnerable projects, not vulnerable projects, and projects where the vulnerable file has been changed but we do not know if it is fixed or if it is still vulnerable. For each of the three categories (vulnerable, not vulnerable, and unknown), a report provides more detailed information.

3.3 Large Scale Empirical Study

Motivated by previous research showing vulnerabilities propagated by copy-based code reuse in a small number of vulnerable projects [75, 76, 77], we planned a large scale empirical study of orphan vulnerabilities. We focused on cases where files containing vulnerable code are copied from one project and committed into another project, as these could be detected in a scalable manner using the World of Code infrastructure.

Our primary goal is to better understand the problem of security vulnerabilities propagated through copy-based code reuse. We want to understand the extent of the problem, characteristics of projects affected, and characteristics of projects that remediate orphan vulnerabilities. Based on these goals, we constructed five research questions to guide our study.

RQ1: How prevalent are orphan vulnerabilities? What are the characteristics of orphan vulnerabilities? Our initial research question delves into the prevalence of orphan vulnerabilities, aiming to ascertain both the extent of duplication of original vulnerabilities and the frequency of such occurrences. Additionally, we explore how programming languages influence the frequency of these orphan vulnerabilities, conducting a thorough analysis of the top 20 most duplicated cases to gain deeper insights.

RQ2: What are the characteristics of projects that have orphan vulnerabilities? Our second research question revolves around examining the attributes of open source projects housing orphan vulnerabilities. We quantified the number of vulnerabilities duplicated within each project and scrutinized various facets of project activity, encompassing its lifespan, commit frequency, authorship, popularity (indicated by stars), and additional metadata. This comprehensive analysis seeks to uncover commonalities and patterns among these projects, shedding light on their distinctive characteristics.

RQ3: How many orphan vulnerabilities are fixed? Our third question centers on the remediation of orphan vulnerabilities. Although all vulnerabilities within our dataset have been successfully addressed in their original projects, our focus shifts to understanding the extent to which orphan vulnerabilities are fixed. While some of the copied vulnerable files may have been generated through package managers which offer mechanisms for localized package installations under the development directory, many others have been created through manual code copying. Unlike dependency tracking tools that aid developers in resolving issues arising from package manager installations, there exists no mechanism to alert users when security

patches become available for manually copied code. This investigation aims to shed light on the challenges and solutions related to the remediation of these orphan vulnerabilities.

RQ4: How do different characteristics of projects affect how many vulnerabilities are fixed? Our research acknowledges the variance in project behavior concerning the updating of vulnerabilities through package management tools. Indeed, while certain projects may opt not to rectify any orphan vulnerabilities, others exhibit a commitment to addressing some or even all of these vulnerabilities. Our study looks at the factors influencing which projects decide to fix vulnerabilities and the proportion of vulnerabilities they successfully address within their codebase. We also consider the interplay between project characteristics (such as programming language and project activity) and vulnerability attributes, exploring how these elements influence the likelihood of vulnerabilities being fixed.

RQ5: How long does it take for an orphan vulnerability to be fixed? In the final phase of our investigation, we turned our attention to the time aspects of orphan vulnerability remediation. Recognizing the critical importance of addressing vulnerabilities promptly to safeguard systems, our analysis focused on determining the duration required to fix these vulnerabilities. Our aim was to assess whether vulnerabilities were resolved before they became susceptible to widespread exploitation, as a belated remedy might offer little improvement over the absence of a fix, particularly if the software had already been exploited. Additionally, we looked at the influence of project activity levels on the timeframe for vulnerability mitigation, seeking to uncover how the activity level of projects impacted the speed of remediation efforts.

3.3.1 CVEfixes Dataset

Vulnerability databases such as the National Vulnerability Database (NVD)[§] contain information on publicly reported vulnerabilities. Each NVD vulnerability has been assigned a Common Vulnerabilities and Exposures (CVE) identifier before inclusion in the database. The NVD provides a description, severity metrics, affected software configurations, and links to references about the vulnerability.

Bhandari et al. [78] extracted CVEs with fixing commits, analyzed the files changed by these commits, and created a dataset containing vulnerabilities and their fixes - the CVEfixes dataset.[¶] This dataset contains information on 5,495 vulnerability fixing commits in 1,754 projects covering 5,365 CVEs. They also provided the code used to generate their dataset, so that future researchers could generate updated versions of the dataset.

We ran the CVEfixes code to generate a current database of CVEs with fixing commits as of November 2022. We removed vulnerabilities whose fixes were identified as being in non-executable files like `README`s from the dataset. We also eliminated vulnerabilities where more than one file was modified in the fixing commit, as our data collection was designed to handle one file per vulnerability. The resulting dataset contained 3,615 CVE entries.

3.3.2 The VCAnalyzer Tool

To study orphan vulnerabilities at a very large scale, we created the VCAnalyzer (Vulnerable Clones Analyzer) tool. VCAnalyzer leverages the World of Code infrastructure to find vulnerabilities that are propagated through copy-based code reuse in open source projects at a scale that has traditionally been infeasible. The tool starts with an initial set of vulnerabilities with fixing commits. For each vulnerability, it searches for projects which have copied a vulnerable file and collects statistics about

§ `https://nvd.nist.gov`
¶ `https://github.com/secureIT-project/CVEfixes`

those projects. VCAnalyzer uses the World of Code to find duplicated files. It collects data about files and projects from both World of Code and APIs provided by code hosting platforms such as GitHub, GitLab, and Bitbucket.

The input to VCAnalyzer is the CVEfixes dataset CSV file, which describes one vulnerability per line. Each line identifies the vulnerability by its CVE number and includes the URL of the repository containing the vulnerable code, the path of the original vulnerable file, the identifier for the git commit that fixed the vulnerability, the date of the fix, and the date on which the CVE record was created. VCAnalyzer uses hash-based matching of files to quickly identify copies of vulnerable files in World of Code. VCAnalyzer examines the entire history of a file, starting by retrieving the entire commit history of the original vulnerable file. It then finds all revisions of the vulnerable file before the fixing commit and all revisions after the fixing commit. The commit history is retrieved using the API of the hosting platforms. File revisions that predate the fixing commit are potentially vulnerable files. We refer to these as bad blobs. A blob refers to the contents of a file at a specific commit. We refer to the blob created by the fixing commit and blobs that postdate the fixing commit as good blobs, as they do not contain the vulnerability. VCAnalyzer also identifies blobs that are found in both lists, which indicates that a fixed version of the file has been replaced by a vulnerable version of the file, possibly because the fix introduced bugs or incompatibilities. If the fixing commit is not found in the default repository branch, the tool skips that CVE.

VCAnalyzer searches World of Code for projects that have ever contained blobs from the bad blob list to identify projects that have contained orphan vulnerabilities. These projects are found by first using the World of Code's blob to commit (b2c) mapping to find all commits containing each bad blob, and then using the commit to project (c2P) mapping to find all deforked projects containing those commits. These are the projects that have copied a known vulnerable file and thus contain an orphan vulnerability. The c2P mapping uses a community detection algorithm [32] to find unrelated projects. The mapping excludes forks and exact copies, unless a fork is

developed into an independent project. For each such project, VCAnalyzer determines if the project has been fixed by finding projects that contain a blob from the good blobs list. The tool identifies the date on which the project copied a vulnerable file as the date on which a bad blob was first committed to the project. It identifies the vulnerability fixing date as the first date on which a good blob was committed. Many vulnerabilities are never fixed, so the fixing date may be NULL.

If a project only contains vulnerable versions of the file (from the bad blobs list), then the project is considered still vulnerable. If the vulnerable file has been replaced with a fixed version of a file from the good blobs list, then the project is considered not vulnerable. If the vulnerable file has been replaced by a file that is in neither the good blobs nor the bad blobs list, then it is categorized as unknown, as we know the vulnerable file has been changed, but we do not know if the change fixed the vulnerability.

Finally, VCAnalyzer collects statistics on each project copying a vulnerable file. Most statistics are available from World of Code. For some statistics, the tool uses the API of the repository hosting platform to retrieve the information directly. The information collected includes the number of authors, date of earliest commit, date of latest commit, number of months the project was active, root fork, number of stars, number of core developers (who commit more that 80% of the code), community size, total number of commits, number of forks, and the most used language in this project.

3.3.3 Empirical Study Method

We conducted a large scale empirical study by mining open source software using the VCAnalyzer tool described above. We studied copy-based code reuse of files containing publicly disclosed vulnerabilities and used those results to answer our research questions.

We cleaned the CVEfixes data by removing vulnerable files whose names indicate that the file is not part of the source code that would be executed then the program is

run. We removed files with the names `CHANGES`, `KConfig`, and `README`, as well as files with the following suffixes: `.md`, `.old`, `.txt`, and `.svn-base`. Files with those suffixes will not be treated as source code. Files with the `.svn-base` extension are part of subversion repository structure and might have been included in a git repository when a subversion repository was converted into a git repository. We can ignore these files, as any changes to the actual copy of the vulnerable file would not be reflected in the subversion repository structure. We also removed copies of vulnerable files identified by VCAnalyzer, where VCAnalyzer reported missing data in critical fields like the identification of the first vulnerable version, the pathname, or project activity.

We only consider files that are currently publicly available. World of Code maintains copies of all files in all projects, even if they are removed or made private. If the potentially vulnerable file is removed from a project or if the project is no longer publicly available, we exclude that project from our results.

To compute the number of occurrences of orphan vulnerabilities, we count the number of vulnerable files identified by VCAnalyzer outside the original project for each CVE identifier. For each original and copied vulnerability, our dataset contains the pathname of the file containing the vulnerability. We identify the programming language used in vulnerable source files by the file suffix found in pathnames. For example, we identify files as C source code by the presence of either `.c` or `.h` file extensions.

Each vulnerability in the CVEfixes dataset includes the URL for the git repository in which the vulnerability was found. Multiple vulnerabilities can share the same URL if they were found in the same repository. VCAnalyzer creates project names for each vulnerability using the hosting platform name and the last two path components of the git repository. Project names are case sensitive. We do not merge projects with similar names. The same process is used when creating project names for copied vulnerabilities found in the World of Code. The collection of project metadata is explained in Section 3.3.2 above.

We create a subset of projects with high levels of activity by selecting projects with at least 100 GitHub stars. We use GitHub stars as a metric, since the numbers of commits and authors are copied to the new project when a project is forked, while the number of GitHub stars is not. The threshold of 100 stars is often used in prior work [79, 80].

We compute multiple project metrics. The first metric we examine is the primary programming language of the project. This metric is provided by the World of Code. As projects often use multiple programming languages, we also identify the programming language used in the vulnerable file using the file suffix, as described above. If a filename does not contain a ".", the file suffix is blank. For each project metric, we filter out values over the 99th percentile to exclude possible outliers.

We classify orphan vulnerabilities as fixed, unfixed, or unknown using the approach described in Section 3.3.2. For each orphan vulnerability that was fixed, we calculate the survival time by computing the time difference between the first fixing commit and either the first vulnerability introduction commit or the original vulnerability fix time, depending on which came later. We then report characteristics of the survival time.

3.4 The UVHistory Tool

UVHistory is our tool for automating the process of tracking changes across all open source repositories and their version histories by operationalizing the universal version history concept introduced in section 1.5. It supports the study of issues concerning source code reuse in real-world open source projects.

3.4.1 Infrastructure: World of Code

Before describing our tool, we need to introduce the infrastructure, World of Code(WoC), on which our tool is built. World of Code [29] is a nearly complete

collection of all publicly available open source software. Software source code is periodically collected from many sites including GitHub, Bitbucket, SourceForge, and many others. The software is then curated and stored using methods that allow for efficient searching of the very large amount of source code collected. World of Code, which aims to support research in software engineering, currently contains over 20 billion Git objects with over 100 million unique public repositories (not including forks or empty repositories) [81]. Since most open source software today is stored in Git repositories, World of Code stores data as Git objects. When we use the terms blob, commit, and tree, we are using the terms as they are used in Git. For example, a blob refers to the contents of a file and is named by a SHA-1 hash.

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

3.4.2 UVHistory

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

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

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

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

3.4.3 Algorithm

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

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

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

When identifying projects in the universal version history, we want to find distinct projects. GitHub projects often have many forks, sometimes tens of thousands. Most of those forks are not independent projects; many were only created for the purpose of issuing pull requests to the original project. Showing tens of thousands of related forks makes it far more difficult to find the useful information. Using the community detection algorithm described by Mockus et al. [32], WoC maps each Git repository to a central repository which is expected to represent the same project. This mapping in WoC allows us to create a list of deforked projects. UVHistory displays the deforked projects on the main page and then includes a link to a secondary page that contains a list of all projects, including (possibly irrelevant) forks.

3.4.4 Output

Our final output contains:

- A list of all blobs in the universal version history. The list is sorted by date in reverse chronological order. The blobs are named by the sha1 hash of the blob as computed by Git.

- For each blob, a list of all commits of which this blob is a part. The commits are named by the sha1 hash of the commit as computed by Git. The commit information includes the author, time of commit, and the commit log message.

- For each commit, a list of all of the projects where this commit was applied, the full pathname of the file, and the URL linking to the file on the source code hosting platform (note that the link may be dead if the project is no longer publicly accessible).

3.5 Universal Version History

In this section, we present four research questions and the research methods used to address each of the questions related to the construction of the universal version history. Our study is designed to show the relevance of the problem presented and also to evaluate the effectiveness of our proposed solution.

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

RQ6: Can the declared license in an open source software be trusted?

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

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

Our research method to answer RQ6 was an exploratory study designed to see the extent of the problem. We examined open source projects which have copied code from popular open source projects to see if the correct copyright and license information was propagated along with the copied code. We developed some tools to select a set of projects that are likely to contain license problems. We then manually inspected that subset.

RQ7: Can our UVHistory tool, by constructing a universal version history, help identify projects with missing copyright and license information and help find the correct information for the given code?

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

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

We used a case study to answer RQ7. We studied two cases where license information was not properly propagated. The two case were selected from results of

the study for RQ6. The case study method allowed us to look in depth at two specific projects.

RQ8: Can our UVHistory tool, by constructing a universal version history, help identify projects which are subject to security vulnerabilities that have been found and fixed in another project but which still persist unknowingly in other projects?

Previous research [82] [34] [64] has identified this as a real problem in popular real-world projects. Due to the seriousness of this problem, a tool that could help mitigate this problem would have value.

We answered RQ8 with another case study. This case study examined a case we introduced section 1.4.1 as one of the motivating examples. Again, the case study method allowed us to have an in-depth look at a project. This time, the project studied contained a known security vulnerability propagated through code reuse.

RQ9: Is the performance of UVHistory such that it can run in reasonable time on commodity hardware for source code files in typical open source projects?

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

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

Chapter 4

Results

This chapter presents the results of the case study, the empirical study, and the evaluation of the universal version history method and tool. The results highlight the risks of copy-based code reuse and a tool to help mitigate those risks.

4.1 Case Study Results

In this section, we present the results of our case study involving four cases that demonstrate some of the security problems caused by orphan vulnerabilities. The four cases are four known security vulnerabilities that have now been fixed in popular open source projects. Our case study looks at projects that copied vulnerable files before the files were fixed in the original project from where they were copied. We look at two vulnerabilities within the widely used cryptography library OpenSSL. The first vulnerability is very recent and the second, heartbleed, is relatively old. We look at one recent vulnerability in a Go language package supporting xz compression. We look at one vulnerability that was fixed more than three years ago in the mature and proven open source PNG [83] graphics library libpng, which is very widely copied. Our case study looks at code written in different languages to show that our approach is language agnostic. It works the same regardless of the language.

We find tens of thousands of open source projects that contain files with known vulnerabilities even though the vulnerabilities have been fixed in the original project from where the vulnerable file was copied. Many of the vulnerable projects appear to be inactive, but some are clearly still active. In some cases the fix is recent, and project maintainers have not had much time to apply patches. In other cases the fix is several years old, and yet many projects still contain the vulnerable code. Patches we provided were only accepted by a small percentage of project maintainers.

4.1.1 Case 1: CVE-2021-3449 in OpenSSL

OpenSSL is a very widely used open source cryptography library implementing Secure Socket Layer (SSL) and Transport Layer Security (TLS) [84]. Projects that incorporate OpenSSL play a vital role in Internet security. This was made clearly evident with the discovery in 2014 of the security vulnerability in OpenSSL known as heartbleed [85]. OpenSSL is the leading cryptography library used for email and website encryption and for software security in many other open source software packages. In this case study, we look at two vulnerabilities in OpenSSL. First, we look at the most recent (as of this writing) known vulnerability in OpenSSL. This vulnerability, described in CVE-2021-3449 [86], allows a maliciously crafted renegotiation ClientHello message to crash a TLS server. OpenSSL considers this a high severity vulnerability [87]. It was fixed in March 2021. Since it was only recently discovered and fixed, we might expect to find a number of projects that still contain the vulnerable code. The second OpenSSL vulnerability we look at, heartbleed, is discussed in the next section, 4.1.2.

The first OpenSSL vulnerability we look at, CVE-2021-3449, was introduced in the file ssl/statem/extensions.c in commit c589c34e61 in January 2018 and fixed in commit 02b1636fe3 in March 2021. According to the OpenSSL vulnerabilities list* "All OpenSSL 1.1.1 versions are affected by this issue. Users of these versions should

*https://www.openssl.org/news/vulnerabilities.html

upgrade to OpenSSL 1.1.1k." Since the vulnerability only existed in a few versions of OpenSSL, we expect to find a relatively small number of projects that use one of the vulnerable versions.

Following the algorithm described in section 3.2.2, VDiOS finds 56 revisions of extensions.c that contain the vulnerability. That is, there are 56 revisions between the commit that introduced the vulnerability and the commit that fixed the vulnerability. VDiOS finds three revisions of the file that contain the fix and are thus known to be not vulnerable to this specific issue. Additionally, VDiOS finds the following:

- 1,614 projects contain one of the known vulnerable revisions of ssl/statem/extensions.c in the most current revision of the project.

- 11 projects contain one of the known fixed revisions of ssl/statem/extensions.c in the most current revision, meaning it used to be vulnerable, but now it is fixed.

- 1,079 projects contain a revision of ssl/statem/extensions.c that is not in either the list of vulnerable blobs or the list of fixed blobs, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability.

- 253 projects used to contain a vulnerable version of the file, but the file has since been removed.

For further investigation of these projects, we used WoC p2P mappings [32] to see how many of these projects are forked. Deforking 1614 vulnerable projects resulted into 132 projects. To see if they are active projects or not, we looked to see if they have any commit in the past 6 and 18 months. We found that 23 of them have at least one commit in the past 6 months, and 64 have at least one commit in the past 18 months. To have an idea about their impact in the OSS community, we looked

at the number of stars [88] each of these projects have. We observed that four of these projects have more than 10,000 stars and 10 have more than a thousand stars, implying their wide impact in the OSS community.

4.1.2 Case 2: CVE-2014-0160 in OpenSSL

We next look at the OpenSSL heartbleed vulnerability. Heartbleed, described in CVE-2014-0160 [89], is a very serious vulnerability [90] that was fixed in 2014. Due to a bounds check error in the TLS heartbeat extension, the bug allows disclosure of information that should be protected. Since this was a high profile serious vulnerability that was fixed seven years ago, we expect not to find many, if any, active projects still using code vulnerable to heartbleed. We use VDiOS to test this hypothesis and then investigate the projects we find that still contain the heartbleed vulnerability.

Heartbleed was introduced by commit 4817504d06 on December 31, 2011, in the files ssl/t1_lib.c and ssl/dl_both.c. The first release of OpenSSL with this vulnerability was release 1.0.1 on March 14, 2012. The vulnerability was fixed two years later by commit 731f431497f made on April 7, 2014, and released in release 1.0.1g on April 7, 2014. VDiOS first finds all revisions of the file ssl/t1_lib.c between the December 2011 commit that introduced the vulnerability and the commit in April 2014 that fixed the vulnerability. It finds 90 vulnerable revisions of ssl/t1_lib.c. Following the procedure described in section 3.2.2 above to find projects containing the vulnerability, we discover the following results:

- 121 projects contain one of the known vulnerable revisions of ssl/t1_lib.c in the most current revision of the project.
- 3,156 projects contain one of the known fixed revisions of ssl/t1_lib.c in the most current revision, meaning it used to be vulnerable but now it is fixed.

- 211 projects contain revisions of ssl/t1_lib.c that is not in either the list of vulnerable blobs or the list of fixed blobs, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability.

Because of the very serious nature of heartbleed [91], we believe it is important to investigate all 121 projects that contain a known vulnerable version of ssl/t1_lib.c. We find the following information about these 121 projects:

- 110 of the projects are forks that were all forked between when the vulnerability was released in 2012 and when it was fixed in 2014 and that have had no activity on the project since before the vulnerability was fixed in 2014.
- Three of the projects are clones that were all cloned between when the vulnerability was released in 2012 and when it was fixed in 2014 and that have had no activity on the project since before the vulnerability was fixed in 2014.
- The remaining eight projects have had some activity (commits or issues) dated 2017 or later, well after the vulnerability was fixed. These projects are a potential concern; therefore, we investigated these eight in more depth.

The 113 projects with no activity later than 2014 appear to be inactive projects. Of course any publicly available project containing heartbleed has the potential to be copied and reused, even if the project is not active. We find the remaining eight projects, the ones with activity dated 2017 or later, to be more concerning since they have been active since the vulnerability was fixed, yet they do not contain the fix. We looked into those eight projects in more detail and found the following information:

- One project has several commits the year of our study (2021). This clearly indicates that it is an active project and potentially concerning since it contains the heartbleed code. Upon further investigation, we find that this project contains tools for the purpose of an empirical study of bugs in real world C

software. OpenSSL is included as one of the subjects of the study rather than being linked into this project's software. Thus, this project is not vulnerable to heartbleed.

- Two related projects on GitHub have commits in 2018 and 2019, which would seem to indicate that they are still active. One of them added a WhiteSource Bolt[†] configuration file in 2019. The other is a fork of that project and updated its Configure script[‡] and Travis CI[§] files in 2018. No other changes have been made to either project since 2013, well before heartbleed was discovered and fixed.

- Two related projects on GitHub have activity more recent than the 2014 fix. One of the projects has two open issues from November 2018 where someone asks questions that indicate they are actively using the project. The questions were never answered, and there are no recent commits, which indicates that the project is not active. But this does show that people could be using projects that have been inactive for many years. Another project is a fork of a fork of this project and has one commit in 2018. The commit is only a change in whitespace in a README. There are no other commits since 2013.

- Two related projects on GitLab have changes in 2018 that only affect whitespace in a README. One is a fork of the other. No substantive changes have been made to either project since the 2014 fix of heartbleed.

- One project, which is not a fork, has a number of commits in 2017, indicating that it has been active much more recently than the heartbleed fix. This GitHub project has zero stars, zero forks, and only 27 commits.

Based on the above information, we conclude that heartbleed is virtually eliminated, although not completely eliminated, from active open source software

[†]https://www.whitesourcesoftware.com/free-developer-tools/bolt/
[‡]https://www.gnu.org/software/autoconf/autoconf.html
[§]https://travis-ci.com/

projects. However, a number of inactive projects that are still vulnerable to heartbleed are still readily available online and thus could still be reused.

4.1.3 Case 3: CVE-2021-29482 in Package xz

Our next case looks at a vulnerability in a popular Go language package. Most of our work to date has studied C language projects. VDiOS is completely independent of the language. We wanted to look at a Go project to demonstrate the language independence of VDiOS and WoC. The project at github.com/ulikunitz/xz is a Go language package supporting xz compression. The project, which is still under development, is subject to the vulnerability described by CVE-2021-29482 [92], which is identified as high severity by the National Vulnerability Database. The vulnerability was fixed in the file bits.go by commit 69c6093c7b on August 19, 2020, and released in release v0.5.8.

VDiOS found 11 versions of the file that are potentially vulnerable and two versions that are fixed. Using these two lists, VDiOS found 7,105 projects that are known to contain a vulnerable version of bits.go in the most current revision and 185 projects that are known to contain a fixed version in the most current revision. Since this is a very new (at the time of our study) vulnerability, it is not surprising that there are only 185 projects containing a fixed version. Only one project was found that contained the vulnerable file in the past but does not currently contain any of the known vulnerable or known fixed versions. We looked into this one case and found the only difference was that it used the DOS/Windows format with carriage return and line feed ("\r\n") at the end of each line instead of the Unix format with only line feed ("\n"). There were 2,037 projects found that used to contain the vulnerable file but that no longer contain the file at all.

To examine projects further, we used WoC project to deforked project (p2P) mappings [32] to find out how many of the projects are not forked. Out of 7,105 vulnerable projects, this resulted in 758 unique projects that are not forked and

contain this vulnerability. The numbers for not vulnerable projects are 185 and 82 respectively. To see how many of these deforked projects are actively maintained, we looked for those that have at least one commit in the past 6 and past 18 months (at the time of our study). We found that in vulnerable projects, 271 have at least one commit in the past 6 months, and 472 have a commit in the past 18 months. In the case of not vulnerable projects, these numbers are 68 for 6 months and 82 for 18 months. As we can see, the percentage of active projects in vulnerable projects (36% & 62%) are significantly lower than in not vulnerable projects (83% & 100%), which was intuitively expected. Nevertheless, not having a commit in a certain period of time does not mean that the other projects are not being used, and so it is still important to address the vulnerability issue. This already shows the significance of the vulnerability being widely spread.

To investigate the impact of this vulnerability from a different standpoint, we looked at the number of stars each of these projects has been given as a measure of their popularity in OSS [88]. The results show that in vulnerable projects, at least 443 projects have more than one star, 273 more than 10, 101 more than 100, 31 more than 1,000, and 10 projects have more than 10,000 stars. In not vulnerable projects, the numbers are 71, 44, 23, 10 and 4 respectively.

The number of stars in projects that fixed the vulnerability is relatively higher than vulnerable projects, which again is what we would intuitively expect. We have also looked at the number of contributing authors in each project using WoC project to author mappings (P2A), which maps the deforked projects to aliased author IDs [93]. Looking at the percentages, it seems that vulnerable projects have relatively fewer developers involved.

4.1.4 Case 4: CVE-2017-12652 in libpng

Libpng [94] is a very popular open source graphics library for manipulating PNG (Portable Network Graphics) image files. It is an old library, dating back to 1995,

and is still actively maintained. Because of its popularity and its very long history, we expect to find many copies in other open source projects, making it a strong case for our study. The libpng source code [95] is hosted on SourceForge [96] and mirrored on GitHub [97].

Libpng was the first case we studied. Lessons learned from this case were applied to our study of the other cases. Improvements to VDiOS, as described later in this section, were applied based on those lessons learned.

This case specifically looks at the libpng file pngpread.c. That file is the subject of the vulnerability described by CVE-2017-12652 [98], which is labeled as a critical vulnerability in the National Vulnerability Database (NVD) [66]. The vulnerability was fixed in August of 2017 in release 1.6.32. This fix is in commit 347538e, and the blob for pngpread.c at that revision is 45b23a7.

Using WoC's blob to old blob (b2ob) mapping recursively, VDiOS finds 951 old versions of the file pngpread.c. The old versions are the potentially vulnerable versions. Using WoC's old blob to blob (ob2b) mapping, VDiOS finds 964 new versions of that file. The new versions presumably all contain the fix. Next, VDiOS looks at each potentially vulnerable blob and uses WoC's blob to commit mapping to find the commits. Once it has the commits, it uses WoC's commit to project mapping to find all of the projects containing the discovered commits. This gives us a list of all projects that have ever contained one of the potentially vulnerable versions of the file pngpread.c. Finally, VDiOS looks at the head commit of each project to see if it contains a version of the file from the potentially vulnerable list, the presumably fixed list, or neither.

The results are as follows:

- 63,441 projects contain one of the potentially vulnerable blobs in the most current revision, even though it was fixed in the original file more than three years ago.

- 458 projects contain one of the presumably fixed blobs in the most current revision, meaning it used to be vulnerable but now it is no longer vulnerable.

- 20,274 projects do not contain blobs from either of the two previous lists, meaning that the project contained a potentially vulnerable blob in the past, the blob has been modified in the most current version, but we do not know if the modification fixed the vulnerability. We manually inspected the first 10 of those projects and found that two out of the 10 projects still contain the vulnerability. In those two cases, the file was modified, but the specific vulnerability was not fixed. In the remaining eight cases, the vulnerability was fixed.

- 28,376 projects used to contain a vulnerable version of the file, but the file has since been removed.

We see that over sixty thousand projects contain a vulnerable version of the file. We selected a subset of those projects to analyze in more detail. To select the subset, we first selected projects that have a commit within the last 18 months to eliminate long dormant projects. Next, we selected non-forked projects to get a list of independent projects. Finally, when one commit went to multiple projects, we selected the first one that VDiOS found and eliminated the remaining duplicates. This process of elimination leaves us with 1,457 projects. From those 1,457 projects, we randomly selected 88 projects to analyze in more detail. In looking at these projects, we find that they copy the entire contents of libpng, not just selected files.

Our first step is to verify that the 88 projects do indeed contain the vulnerable code. We manually inspected all of the projects and found six false positives. There were four projects that had deleted the vulnerable file and two projects that had fixed the vulnerable file. We removed those six projects from further analysis, leaving 82 projects. We investigated these six cases to understand why VDiOS produced false positives. In all six cases the reason was timing. We ran VDiOS to produce the results in early February 2021 and analyzed the results over the next two months. WoC is continuously updated but will always be a little bit behind what is live on

the source code repository hosting platforms. We ran VDiOS on version S of WoC which was updated in August 2020. We found in those six cases that the vulnerable files had been fixed or removed after the WoC version that VDiOS used to produce the reports and before we verified the results in April 2021. We conclude that VDiOS produced the correct output, but the continuously changing open source projects will be different from our reports to the extent that changes are made after the most recent WoC update. As a result of this discovery, we modified VDiOS to use the APIs of the hosting platforms to get the most current revision of the file. The results presented in this case are based on the original version of VDiOS; this new enhancement to VDiOS is used for the rest of the cases.

For projects hosted on GitHub, we also verified that GitHub's Dependabot [21] did not find the vulnerability. While dependabot has similar goals to VDiOS in finding vulnerable dependencies in the software supply chain, it uses a very different mechanism. Dependabot requires that a repository define dependencies in a supported package ecosystem while VDiOS looks for file level code duplication. As expected, none of the projects we found with a vulnerable version of libpng were identified by Dependabot. Several of the projects had other issues identified by Dependabot, but not the libpng issue we are investigating. This shows that Dependabot is enabled for these projects. Clearly VDiOS finds different supply chain dependency vulnerabilities than GitHub's Dependabot.

Finally, we wanted to find out if the maintainers of the projects that contain known vulnerable files are willing to accept a patch to fix the vulnerability. For the 82 vulnerable projects, we produced a patch and sent a message to the maintainers through a pull request, an issue, or an email. We waited up to two weeks for responses. Seven project maintainers accepted our pull request with the patch. One project maintainer updated to a newer version of libpng because of our contact. Two project maintainers responded and said they would continue using the existing (vulnerable) code. We received no responses about the remaining projects.

Aside from what we found through these 82 projects, we wanted to have some overall statistics on activity and popularity measures of vulnerable and not vulnerable projects as we had in previous cases. Following the same procedures, we found that the 63,441 vulnerable projects reduce to 9,680 deforked projects from which 660 have at least one commit in the past 6 months and 2095 in the past 18 months. Other than that, there are at least 25 projects with more than 10,000 stars and 131 projects with more than 1,000 stars which attest the importance of such vulnerabilities.

4.2 Empirical Study Results

The widespread copying of open source software in software projects has exposed them to vulnerabilities. Dependency tracking tools have been developed to address this issue, but they primarily rely on package manager metadata. However, a significant problem arises when open source developers manually copy code to other repositories without a clear link to the origin, leading to what we term "orphan vulnerabilities."

To assess the scope of this problem, we conducted a large-scale study. We created the VCAnalyzer tool to investigate vulnerabilities introduced through the manual copying. Analyzing 3,615 vulnerable files from the CVEfixes dataset, we identified over three million orphan vulnerabilities in over seven hundred thousand open source projects.

Our findings are striking: 83.4% of vulnerable files were copied at least once, with 59.3% containing C source code. Only 1.3% of these vulnerabilities were remediated, taking an average of 469 days. For active projects, more than half required over three years to address.

These results highlight the challenge of tracking orphan vulnerabilities, necessitating improvements in dependency identification within software supply chain tools. To foster collaborative efforts in addressing this issue, we have made VCAnalyzer and our dataset publicly accessible.

The rest of this section addresses our four research questions concerning orphan vulnerabilities in open source software.

4.2.1 RQ1: Prevalence of Orphan Vulnerabilities

Searching for the prevalence of orphan vulnerabilities in open source software, we observed that 83.3% of vulnerable files in our CVEfixes dataset were replicated in a staggering 3,044,644 instances. Interestingly, 601 initial vulnerabilities remained unreplicated, leaving more than a thousand isolated vulnerabilities on average for each copied one. A noteworthy finding was that 95.4% of copied vulnerable files shared directory paths with the original ones, indicating that orphan vulnerabilities typically stem from the wholesale import of dependencies into a project's repository.

The majority of copied original vulnerable files were written in C (59.3%), with C++ accounting for 10.2%. This can be attributed to the prevalent use of code copying in C and C++ projects, as opposed to package managers like Conan [99]. Vulnerable C++ files were less frequently copied, making up only 1.3% of copied files. Similarly, JavaScript vulnerabilities represented 5.7% of the originals but surged to 24.3% among orphans due to the npm package manager's influence.

We also found instances where dependencies and their metadata were committed to repositories, rather than following the recommended package manager approach. This practice accounted for 17% of C++ files, 36% of Go files, 67% of JavaScript files, 66% of JSON files, 47% of PHP files, 9% of Ruby files, and 53% of Swift files. Most other languages had less than 5% of files introduced through package managers.

C, PHP, and C++ files comprised the majority of both original vulnerable files and copied files. However, PHP and C++ files were underrepresented among copied files at 6.4% and 1.3%, respectively.

While the average number of times a vulnerability was copied was 1,010, there was significant variance, with some vulnerabilities copied only once (10.1%) and others more than a thousand times (12.6%).

The most frequently copied vulnerabilities were linked to npm package manager usage, with CVE-2021-32640 copied a staggering 112,297 times. Excluding npm, the top 20 most commonly copied vulnerabilities included cross-site request forgery issues in Fat Free CRM, PHP software vulnerabilities, C vulnerabilities in nanopb, and JavaScript vulnerabilities in Moment.js.

It is worth noting that many of the top 20 copied orphan vulnerabilities were not found in libraries, making them challenging to detect with supply chain tools focusing solely on third-party library dependencies.

4.2.2 RQ2: Characteristics of Projects that Copy Vulnerabilities

In examining the characteristics of projects that copy vulnerabilities, we aim to uncover patterns and trends that shed light on how vulnerabilities propagate within software development ecosystems. By analyzing these distinctive traits, we can gain valuable insights into the dynamics of code reuse and potential strategies for enhancing software security.

In the CVEfixes dataset, consisting of 1,114 open-source projects, 800 of them (71.8%) had orphan vulnerabilities, resulting in the dissemination of vulnerable files to 719,131 distinct projects within World of Code. While 58.3% of these projects contained a solitary orphan vulnerability, and a staggering 97.5% had 10 or fewer such vulnerabilities, a subset of 9,428 projects (1.3%) harbored 100 or more copied vulnerable files, with a maximum of 806. Notably, seven of the top ten projects with the highest counts of vulnerable copies were closely associated with "linux" or "kernel" in their project names, suggesting their roots in the Linux kernel, which had the most vulnerable files in our CVEfixes dataset.

To assess project activity, we considered multiple metrics, including active project lifespan, commit counts, authorship, and GitHub stars. Commit activity exhibited significant variability, ranging from a mere one commit to an impressive 36 million

commits. Predominantly, projects featured low commit activity levels, with 94.3% of them registering 100 or fewer commits, and 61.3% having 10 or fewer. Furthermore, 99% of projects had 10 or fewer commit authors, with 71% having only one. The duration of active months spanned a broad spectrum, ranging from one month to an impressive 428 months (equivalent to 35 years). Some well-known projects like Emacs, FreeBSD, gcc, Kerberos, and Python boasted more than 30 years of historical data.

GitHub repositories were prevalent among our projects, with 98.5% of them having a presence on GitHub. However, GitHub stars were less ubiquitous, as 83.3% of GitHub-hosted projects had none, while 10.4% had garnered a solitary star, with a handful amassing more.

Notably, most projects housing orphan vulnerabilities lacked documented GitHub security policies (as indicated by the absence of a `SECURITY.md` file). Only 1.7% of these projects included such policies.

We further curated a subset of 2,021 projects with high levels of activity, defined by having at least 100 GitHub stars. These projects exhibited an average of 6.28 copied vulnerabilities, compared to 4.23 for the entire dataset. Despite having nearly 50% more copied vulnerable files on average, active projects were more likely to have published a security policy, with 11.6% of them featuring a `SECURITY.md` file, compared to 1.7% across all projects. Only 0.013% of active projects had fewer than 100 commits.

4.2.3 RQ3: Orphan Vulnerabilities that are Fixed

Looking into orphan vulnerabilities that are fixes, our analysis reveals that out of the three million copied files within the World of Code dataset, a mere 100,889 files (3.3%) witnessed the substitution of the vulnerable file with a corrected version from the original project at a subsequent point in time. Furthermore, 68,760 copied files (2.3%) underwent modifications from their original vulnerable state, yet the nature of these alterations remains uncertain, as they may have been intended for vulnerability

remediation or other purposes. Notably, the bulk of the copied files, totaling 2,875,018 (94.4%), remained identical to the original vulnerable file throughout the history of the projects that adopted them.

Turning our focus to fixed vulnerabilities, we identified them in 26,801 (3.7%) out of the 719,204 projects in the dataset. Interestingly, more than half of the projects that addressed one vulnerability also resolved all their vulnerabilities. However, it is essential to note that among the 14,276 projects that rectified all their vulnerabilities, a significant majority (79%) had just one vulnerability to address. Conversely, only a mere 1.6% of projects within the World of Code that featured a single copied vulnerable file succeeded in mitigating that specific vulnerability.

4.2.4 RQ4: Projects that Fix Orphan Vulnerabilities

Our research delved into the correlation between various project characteristics and the percentage of fixed vulnerabilities. We examined several project attributes, including project language, vulnerable file language, number of commits, contributors, community size, core contributors, active months, and the number of stars.

Regarding the project language, we analyzed the primary language identified by World of Code using heuristic methods. Vulnerabilities stemmed from projects written in 14 different programming languages, with not all projects having a designated primary language. A detailed breakdown of the percentages of fixed, not fixed, and unknown status vulnerabilities for each project language can be found in Table 4.1. Notably, the majority of copied vulnerabilities across most languages remained unfixed, with exceptions like Rust, Go, and SQL, where 36.3 17.1%, and 10.9% of copied vulnerabilities were rectified, respectively.

We further scrutinized how various project metrics influenced the percentage of fixed and unfixed orphan vulnerabilities. We considered how these percentages changed concerning the growth of metrics such as the number of commits, active months, community size, core members, forks, and GitHub stars. Generally, there

Table 4.1: Percentage of copied vulnerabilities with status fixed, not fixed, and unknown for each project language

Project Language	Not Fixed	Fixed	Unknown
	98.9	0.6	0.5
C/C++	92.9	4.1	3.0
Fortran	96.4	2.9	0.7
Go	79.4	17.1	3.5
Java	95.8	2.8	1.4
JavaScript	97.9	1.6	0.5
Lua	83.5	9.7	6.8
PHP	95.4	2.5	2.1
Perl	95.4	3.3	1.3
Python	90.1	9.1	0.8
Ruby	97.4	2.1	0.5
Rust	62.3	35.3	2.5
Sql	88.1	10.9	1.0
Swift	97.4	1.7	0.9
TypeScript	94.4	0.6	5.0

was a trend of reduced unfixed vulnerabilities as these project metrics increased, indicating that larger and more actively maintained projects were more inclined to address copied vulnerabilities. Nonetheless, it is essential to note that even in larger projects, a significant number of copied vulnerabilities remained unresolved.

The most pronounced trends were observed in the number of active months and GitHub stars, suggesting that projects with extended lifespans and substantial popularity were more likely to fix copied vulnerabilities.

Additionally, we examined whether vulnerabilities copied through package managers exhibited a higher likelihood of being fixed. Surprisingly, we found no consistent and clear trend across different programming languages.

Lastly, we investigated the percentage of fixed orphan vulnerabilities in large projects, focusing on a subset with at least 100 GitHub stars. This metric was chosen to mitigate potential biases from other metrics, such as commit count, being influenced by forking. Table 4.2 provides insights into the percentage of fixed, not fixed, and unknown status vulnerabilities for the top 10 file suffixes in large projects. Although a higher percentage of orphan vulnerabilities were fixed in these projects, a notable proportion remained unresolved, highlighting the persistence of security challenges even in large and popular open-source endeavors.

4.2.5 RQ5: Survival of orphan vulnerabilities

We conducted an analysis to determine the timeframe within which orphan vulnerabilities were addressed. To calculate this, we computed the time difference between the first fixing commit and either the initial vulnerability introduction commit or the original vulnerability's resolution time, depending on which event occurred later.

Across all projects, our findings revealed that 15.6% of copied vulnerabilities exhibited a negative time delta, indicating that the orphan vulnerability was rectified before its resolution in the original project. In contrast, 84.3% of the copied vulnerabilities showed a positive time delta, reflecting fixes that occurred later.

Table 4.2: Percentage of copied vulnerabilities with status fixed, not fixed, and unknown for each file ending for active projects.

File Suffix	Not Fixed	Fixed	Unknown
.c	30.3	61.6	8.1
.cc	75.2	2.2	22.6
.cpp	42.5	41.5	16.0
.h	36.9	56.0	7.1
.js	40.0	53.1	6.9
.json	31.2	6.2	31.2
.php	29.4	49.2	21.4
.py	10.2	79.5	10.2
.rb	67.5	27.3	5.2

The mean duration required to remediate an orphan vulnerability was 459 days. Notably, in 15% of cases, orphan vulnerabilities were fixed within 0 to 1 days, suggesting potential automated update interventions. While half of the orphan vulnerabilities were addressed in less than 80 days, 25% of them lingered unresolved for over 560 days. Surprisingly, we observed that copied vulnerabilities likely introduced through package managers did not exhibit a trend toward quicker fixes (i.e., less than a day).

Furthermore, we investigated the duration for which copied vulnerabilities persisted in larger projects, with the expectation that more prominent and active projects would resolve vulnerabilities more swiftly. In projects boasting at least 100 GitHub stars, 75% of orphan vulnerabilities endured within the project for more than 426 days. Half of these orphan vulnerabilities remained unresolved for over three years.

4.3 UVHistory Evaluation

The free exchange of code among open source projects poses challenges for code provenance, crucial for cybersecurity and license compliance. To address these challenges, we developed UVHistory, a tool linking code pieces to all their project homes and version histories, forming a "universal version history." In this section, we explore how UVHistory aids developers in tracking code origins, evolution, authors, and modifications, and evaluate its effectiveness in identifying license non-compliance and unfixed vulnerabilities in active and popular projects. We evaluate our method and tool by answering each of the four research questions.

4.3.1 RQ6: Can the declared license be trusted?

To answer our first research question, we searched for cases where code was copied from one project to another, but the copyright and license information was not copied.

We were not looking for projects that used package managers or linked to external libraries, but only cases of code cloned from one project and committed into the repository of another project.

We selected a small subset of projects to investigate in more detail. We randomly selected 100 source code projects from GitHub which include a top-level file named LICENSE.txt containing a license that requires copies to retain the copyright notice, had no reference to copyright or license information in the individual source code files, and had more than 1,000 stars. We chose projects with a top-level LICENSE.txt file because it is common for projects to put the license information in a single file in the top-level directory of a repository and not duplicate the license in every source code file. LICENSE.txt is one common filename used for the license file. When the license information is not included in every source code file, it is easy for a developer to copy a copyrighted source code file without copying the relevant license information. We limited our selection to repositories with more than 1,000 stars so that we would find popular projects [88] that are likely to have files that are copied into other projects. The selected projects were composed of projects in a variety of languages and using a variety of licenses.

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

Our procedure for finding out if the projects containing cloned code also contained the proper license was as follows: First, we used UVHistory to find projects containing copies of the code in question. Next, we used a tool we developed (also layered on top of World of Code) to find all licenses used in a project. The tool used the winnowing

algorithm [100] to find the most similar license from among 1862 licenses provided by spdx[¶] for each blob in a project. The winnowing algorithm relies on extracting a collection of signatures from text and matching them among documents (blobs in the project and blobs representing licenses). We compared the known license in the original project to the licenses found by our tool to see if there was a match. If there was an exact match, then clearly the license information was correctly propagated with the code. If there was not a match, we manually inspected the project to make sure that, in fact, the correct license was not included. We also checked to see if the project was still publicly available, as World of Code will still have information about removed projects, but we only care about projects that are currently publicly available. If neither our tool nor our manual inspection found a match, then we conclude that the correct license information is not properly included.

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

The answer to RQ6 is clearly no, the declared license cannot always be trusted. License violations caused by license omission are common in real world projects since we found a high percentage of popular projects where code is copied but the license and copyright information are not retained (as required by the license), and there is no link from the copied project back to the original project where the license can be found. These non-compliant projects are publicly available, which means that someone might very well copy and use the code without being aware that they are violating the license terms.

[¶] https://spdx.org/licenses

The answer to RQ6 suggests that future work involving a large scale empirical study concerning license omissions in cloned code would be valuable.

4.3.2 RQ7: Can UVHistory help with license compliance issues?

Based on the answer to RQ7, someone who wants to reuse code from one of these non-compliant projects would have no easy way to know the license terms that must be followed unless they obtained some additional information. Our second research question considers whether our UVHistory tool can provide the additional information necessary to ensure compliance. To answer this question, we conduct a case study with two cases looking in detail at specific projects chosen from the ones identified in the section above.

The first project chosen was AIOHTTP[‖], an asynchronous HTTP client/server framework. We chose AIOHTTP because it has over 12,000 stars on GitHub, indicating it is a very popular project likely to be copied, and because it has the Apache license which is a very common license which requires the copyright notice to be retained in all copies. We have already discovered, in answering RQ7 above, that there are projects which reuse AIOHTTP without following the license terms that require attribution. To answer RQ7, we want to find out if UVHistory can confirm that proper license terms are followed or, if not followed, find the correct license for the project. We start with projects that we know, from our study of RQ6, do not comply with the license requirement to include the copyright notice and do not provide a clear link to the original project that contains the correct license terms. There are many cases of projects that use AIOHTTP without retaining the copyright notice as required. We pick just one, Hackathon-Torrent[**], to show that UVHistory is able to identify the correct license and copyright notice that should be included with

[‖] github.com/aio-libs/aiohttp
[**] github.com/AdoenLunnae/Hackathon-Torrent/

any reuse of this project. Running UVHistory on a source code file in the Hackathon-Torrent project, we find the AIOHTTP project as the project in the universal version history with the earliest date. Following the link produced by the tool takes us to the AIOHTTP project on GitHub where the license and copyright information is very clearly available. This means that someone wishing to reuse the Hackathon-Torrent project could, by looking at the universal version history produced by UVHistory, find the correct license and copyright information that is missing from Hackathon-Torrent.

The second project chosen was VirtualXposed[††]. We chose this project because it is widely copied and because it has a commercial license. The commercial license is particularly problematic when copied into open source projects, especially when the license information is not propagated with the copy. We traced the code from the VirtualXposed project to its origin, which is VirtualApp[‡‡]. VirtualApp's README is very clear that in order to use this software you must purchase a license. However, VirtualXposed includes the GPL license in its LICENSE.txt file, which would make it appear that it is available under GPL, but that is not completely correct since it also includes commercially licensed code. VirtualXposed has more than 15,000 stars on GitHub and more than 2,000 forks, indicating that it is a widely used and copied project. Following the history of the project produced by UVHistory, we find several copies of this code that include an open source license such as Apache or no license at all. Without a tool like UVHistory, there would be no way to know that these copies of VirtualApp are restricted by a commercial license. Some examples of projects that do not propagate the commercial license follow. We only list a few examples; there are many more than what we have listed here. VirtualDump (github.com/LiveSalton/VirtualDump) contains copies of some of the code that originated in VirtualApp, but it does not include any license information or any link back to the original VirtualApp project. YCVaHelpTool (github.com/yangchong211/YCVaHelpTool), which also uses code from VirtualApp,

[††]github.com/android-hacker/VirtualXposed

[‡‡]github.com/asLody/VirtualApp

includes the Apache license in a file named LICENSE. There is no mention of the VirtalApp or the commercial license, leaving a developer wishing to reuse the code assuming that it is available through the Apache license. Following the universal version history to the origin again leads to the correct license and copyright information.

The answer to RQ7 is yes, the universal version history can help identify missing license information and find the original project containing the correct license information. We demonstrated that our UVHistory tool can effectively find the original project, allowing developers wishing to reuse code to be able to find the correct license information.

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

4.3.3 RQ8: Can UVHistory help identify projects with security vulnerabilities?

To answer RQ8, we follow a similar procedure as for RQ7, except our case study for RQ8 looks at projects with known security vulnerabilities rather than potential license violations.

In section 1.4.1, we used a security vulnerability in a jpeg compression library as a motivating example for this work. That case was particularly challenging because the fix for the vulnerability was not in the original project from where the code came, but rather in a project that had reused the vulnerable code and then fixed it. Thus finding the origin is not enough, we also need to look at other projects in the history. The specific example we look at, Entropia Engine++()github.com/Spartan/eepp), is a cross-platform game and application development framework. With recent commits and a number of stars, it appears to be a reasonably active and popular project. It reuses the vulnerable file jpgd.cpp. The header comment in that file references the jpeg-compressor project from where the code was copied. A developer wishing to reuse Entropia Engine++ could easily know from where it

was copied. However, the CVE identifying the vulnerability lists the Android System UI (source.android.com/security/bulletin/2017-07-01#system-ui) where the vulnerability was fixed. Therefore, there is no clear way for the developer reusing it to know that it, in fact, contains the vulnerable version of jpgd.cpp. This is where UVHistory proves its value. By finding the universal version history using UVHistory, we are able to see not only the original jpeg-compressor project, but also other projects which reuse it, including Android. Searching the universal version history for common strings like "CVE" or "vulnerability" finds hints about potential problems. In this example, we find two hits when searching for "vulnerabil": "38889eb Fix series of JPEG vulnerabilities by xxxxx" and "890381c Fix security vulnerability by xxxxx" (author names redacted for privacy), both from the Android project. This allows a developer wishing to reuse Entropia Engine++ to find the potential vulnerability CVE-2017-0700 by searching the universal version history. Commit 38889eb fixes this vulnerability.

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

4.3.4 RQ9: Is the UVHistory prototype feasible?

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

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

By leveraging the World of Code infrastructure, which has already curated the data and stored useful information in a database which can be efficiently searched,

we are able to produce results relatively quickly. We looked at the timing on the 100 cases selected for RQ6. Our timing results varied greatly based on how many different projects contain a cloned version of the file in question. Most of the projects in our 100 cases had less than 500 clones. The elapsed time for a case with 187 cloned projects was nine minutes. The worst case, which found clones in well over 10,000 projects, took just under three hours. Thus we conclude for RQ9 that, yes, UVHistory is able to finish in practical time.

4.3.5 Evaluation of Existing Tools

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

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

Similarly, with vulnerabilities, current open source tools fail to trace the history as a file is modified and copied across repositories, and therefore often miss vulnerable

code that has been copied from a known vulnerable project to a different project. Our research shows that cases like this are common and that our tool can help identify these cases. Again, commercial tools are harder to evaluate. Most appear to have the same limitations. We tested several using our example project containing a cloned vulnerability.

We created a small test case example project where we built a very simple HTTP client and server using code cloned from a vulnerable version of aiohttp (github.com/aio-libs/aiohttp) (a project which we identified when we collected data for RQ6). We cloned only the directory that contains the source code, but we did not clone the top-level directory, which contains the License.txt file. We added an MIT license for our example project. Our project cloned v3.7.3 of aiohttp, which is subject to the vulnerability described in CVE-2021-21330. Anyone wanting to reuse our project would assume everything in the repository is available under the MIT license. It is not immediately clear that parts of the project are actually subject to a different license. Additionally, the project contains a known vulnerability, but our project is not listed in any CVE entry. This example project mimics real-world cases that we found in many open source repositories.

Popular free dependency checker tools such as GitHub Dependency Graph [57], Dependabot [21], Google Open Source Insights [58], and OWASP Dependency-Check [59] rely on supported package ecosystems that use a supported file format because they rely on the packaging information to find the dependencies. This means that languages like C and C++, which don't have a standard package management system, are not well supported by these kinds of tools. Even projects using languages that have popular package management systems sometimes copy and commit the code into their own repositories rather than using the package management system. In our tests using our example project, none of these four tools detected the license or security issue. This is as expected since our example uses cloned code rather than a package manager.

We next tested two commercial SCA tools, which we refer to as Tool A and Tool B. We chose those two because they were listed in "The Forrester Wave" (forrester.com/policies/forrester-wave-methodology/) 2021 Q3 Report as having strong market presence, and they have free downloadable trials available. We did not look at commercial tools that do not provide a free download of a trial version. While it is harder to know the exact capabilities of closed source tools, the public documentation and trial versions give us good insight.

Tool A traditionally relied on package manager information to find license compliance issues. They recently announced support for "vendored code" (what we in our introduction call "clone-and-own or vendoring"). We tried out their free version (which supports license compliance but not vulnerability management) on our example project. Tool A did not detect the missing license information from the cloned file we inserted.

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

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

4.4 Limitations

4.4.1 Infrastructure

We rely of the World of Code infrastructure, which contains a relatively complete collection of open source software. But the collection is not complete, with some

projects missing and a several month delay between the versions of World of Code when new projects with vulnerabilities may be created. Our tools will miss any code that is not included in World of Code. Only increasing open source coverage for World of Code would address this limitation.

It is important to note that some vulnerabilities are never discovered or fixed, or not reported in public vulnerability databases. Our study looks at known vulnerabilities, and thus does not consider these vulnerabilities that have not yet been discovered.

Our tools trust the timestamp and author information in the Git commit. There are occasional cases where that information is not correct. Flint et al. [101] demonstrated that while timestamps are usually accurate, there are unusual cases where the timestamp is not correct. We used the reported suggestions and additional techniques, like identifying unreasonable (empty, too old, or too new) and inconsistent (parent commit occurring after the child commit) time stamps to weed out some of the problematic commits.

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

We are specifically interested in cases where files are copied from one project to be reused in another unrelated project. We do not want to include forks that are only created to submit pull requests, or cases of re-appropriation of entire projects as described by Lopes et al. [102]. Our tools use World of Code's commit to deforked project (c2P) mapping. This mapping uses the community detection algorithm described by Mockus et al. [32] to find unrelated repositories, and it excludes most forks and complete copies of projects. If the c2P mapping returns related projects, we will over-count duplicates. We believe this to be a small number of projects relative to the 173 million projects contained in World of Code.

4.4.2 Methods

Our findings about the scope and age of orphan vulnerabilities is limited by the relatively small sample of vulnerabilities explored. We hope that by highlighting the scope and seriousness of the problem with our study, we will spur improvements and wider studies of vulnerabilities in the future.

If a source code file has been identified as containing a security vulnerability, the project using that file might be subject to the vulnerability. However, the project might not be using the vulnerable file in a vulnerable way. Our tools can help identify if a vulnerable file is included in a project, but cannot identify whether it is used in a vulnerable way. It is important for project maintainers to understand if a project contains a vulnerable file, even if it does not use the code in a vulnerable way. A developer may later make a change that uses the code in a vulnerable way, thus unknowingly making the project vulnerable. Also, an important part of our study is to help developers who wish to copy code. It is valuable to know that a vulnerable file exists in a project one wants to copy, even if the project does not currently use the file in a vulnerable way.

We rely on heuristics to determine if a vulnerable file was installed by a package manager or by manual copying of files. Errors in these heuristics may result in an undercount or overcount of orphan vulnerabilities.

We use GitHub stars as one metric to try to find active projects and eliminate many useless projects. GitHub stars is not a perfect measure but is useful in many cases [103].

4.4.3 Tools

Our tools look for exact matches at the file level for the set of code versions between the versions that introduced and fixed the reference code. Using hash-based matching of files allows us to scale to the entire World of Code. Code fragments copied from within a file may not be detected. It will also not detected if a developer

copies a file and modifies it before committing to the new repository. First, this provides only a conservative estimate of vulnerable files, as minor modifications to vulnerable (or fixed) files may not be detected. Second, it is fairly straightforward to enhance the tools to look for snippets, patches, or more abstract representations of the vulnerability.

If we know when a vulnerability was introduced, the tools have the ability to look at only revisions between the introduction and fix of the vulnerability. If we don't know when the vulnerability was introduced, we consider all prior revisions of the file. The tools take the revision of a file that fixes a vulnerability and then use WoC's blob to old blob (b2ob) and old blob to blob (ob2b) mappings recursively to find older and newer revisions of the file. Alternatively, they can use WoC's commit to parent commit (c2pc) and commit to child commit (c2cc) to find older revisions (up to the revision that introduced the vulnerability, if we know that) and newer revisions. The older revisions are likely to contain the vulnerability, and the newer revisions are likely to contain the fix. However, that is not guaranteed to be the case. In extremely rare cases projects revert back to vulnerable code even after fixing it. The tools allow manual inspection and modification of the lists of old and new blobs to see if they are actually vulnerable and fixed respectively before moving on to the next phase. This manual intervention solves the problem, but to scale the solution the tools will need to be enhanced.

If a developer copies a file and makes a small change before committing for the first time, the tools will not find the match. Adding a new copyright notice, making formatting changes to match a style guide, or changing the CR/LF format at the end of lines are examples of inconsequential changes that would affect the ability to find a match. It will only find the match if the initial commit is identical to the copied file or a previous revision of the copied file. The tools can be enhanced to catch these and other modifications, and it is the subject of future work.

Chapter 5

Conclusion

5.1 Discussion

Our research is motivated by the orphan security vulnerabilities and license violations caused by code reuse in open source software. Recent high profile data breaches underscore the seriousness of software security. Our primary goal is to better understand these issues and how they may be mitigated. In this work, we explore the scope of the problem on a small sample of vulnerabilities and the willingness of project maintainers to fix issues.

The vast quantity of open source projects distributed over different hosting platforms complicates our task. By exploiting the World of Code infrastructure, we build tools that collect code reuse data with the coverage and scale that had previously been impractical.

First, using VDiOS, we find a very large number of projects with orphan vulnerabilities based on the four vulnerabilities in our case study. As hypothesized, the probability of an orphan vulnerability is lower for more active projects. Also, supporting Linus's Law [19], the probability of an orphan vulnerability is lower for projects with more developers. Orphan vulnerabilities appear to concentrate in inactive or no longer maintained projects, but they are also present in very popular

(over 10K stars) and very active projects as well. Orphan vulnerabilities, even if they are in unmaintained or inactive projects, still pose risks. First, a developer might copy code from such projects as, for example, they may have a unique feature that fixed projects lack. Second, code from inactive projects may still be running in existing systems, for example in embedded devices. We, in fact, found a case where someone asked a question about a project that appeared to be inactive, indicating that they were using it. By looking both at relatively old orphan vulnerabilities and very new orphan vulnerabilities, we observe relatively fewer old orphan vulnerabilities, suggesting that often orphan vulnerabilities are eventually fixed or removed. The time to fix appears to be substantial, providing opportunity for the orphan vulnerability to propagate further. Even very well-known and very old vulnerabilities still persist in the orphan form.

Our attempts to gauge willingness of the project maintainers to fix orphan vulnerabilities yielded mixed results, with only a small fraction applying the patch.

Our case study suggests that orphan vulnerabilities are widespread, they take a very long time to be fixed, or they persist. They exist not only in forks or abandoned projects but also in highly active and popular projects as well. Even once an orphan vulnerability is identified and the fix provided to a maintainer, only a small fraction act upon the suggested fix. We conclude that orphan vulnerabilities pose an ongoing problem that needs to be addressed not just by identifying and providing fixes to the projects, but also by providing screening tools to projects reusing source code and by educating the open source development community.

We found that orphan vulnerabilities are widespread in open source software. Out of the 3,615 vulnerable files in our CVEfixes dataset, 3,014 (83.3%) were copied, resulting in more than three million orphan vulnerabilities. The orphan vulnerabilities came from 800 (71.8%) of the projects in the CVEfixes dataset and were distributed across 719,131 projects found in the World of Code. The majority of the original vulnerable files (59.3%) and their copies (63.7%) are written in the C programming language, which predates the use of package managers.

While most projects containing orphan vulnerabilities displayed low levels of commit activity and had small numbers of contributors, we examined a subset of 2021 projects that had at least 100 GitHub stars. These active projects had an average of over six years of activity. While the number of copied vulnerable files was about 50% higher in active projects than the entire dataset, active projects were much more likely (11.6% compared to 1.7%) to have published a security policy.

We found that only 100,889 (1.3%) out of over three million orphan vulnerabilities were fixed by replacing the file's contents with the fixed version of the file from the CVE fixes database. Another 68,760 (2.3%) copied files had their contents modified, but we do not know if these modifications remediated the vulnerability or not. Fixed vulnerabilities were only found in 26,801 (3.7%) of the more than seven hundred thousand projects that contained orphan vulnerabilities.

We found that larger, more active, and longer-lived projects are more likely to fix copied vulnerabilities, but even for the largest projects, the large majority of vulnerabilities are not remediated. When dividing projects by primary programming language, we found that 90% of vulnerabilities are not fixed for most languages. However, projects using a few languages, like Rust, Go, and SQL fixed more than 10% of their orphan vulnerabilities.

Orphan vulnerabilities that were fixed required an average of 459 days to be remediated. However, 15% of projects fixed orphan vulnerabilities in less than one day, indicating constant watching of security updates or automated update tools. Orphan vulnerabilities survived a long time even in active projects, where half of orphan vulnerabilities required more than three years to remediate.

For all popular projects with orphan vulnerabilities we searched for repositories that contained a `SECURITY.md` file. For each of these repositories, we checked if the copied vulnerable file was still present in the repository and if the repository corresponded to an actual project (not a collection of samples or a collection of vulnerabilities). We contacted the e-mail address listed in the `SECURITY.md` file to disclose the vulnerability. We received responses from two-thirds of the projects

with promises to either look into the potential security issue or to update the vulnerable file. One month after the disclosure, half of the projects with disclosed orphan vulnerabilities had fixed the vulnerabilities by either upgrading the library dependency or by removing the vulnerable file.

The case where files from a package manager are copied into and committed to the project's repository posed a dilemma for our research. On one hand, we are specifically looking for files that are copied from one repository and committed into another, and not cases where files are included via a package manager. This suggests that we should exclude these files. On the other hand, those files might have been copied from another repository that used a package manager. And since they are committed, they may be copied into other projects. Since we are studying copy-based code reuse, any file committed into a public repository is of interest. In either case, the vulnerable files are committed to a publicly available repository, thus able to be copied. Since our motivation is to mitigate vulnerabilities caused by copy-based code reuse, we chose to include these files. Section 4.2.1 addresses the prevalence of package manager files that are committed to repositories. In order for the analysis to be done without the package manager files, we filtered the output to remove the files that were originally added through package managers, and we make that filtered output available with the other data and source code from this research so that others can redo our analysis with the subset that does not include these files.

For developers, we recommend identifying and documenting copied code, so that it can be updated when vulnerabilities are reported. We also recommend using package managers instead of copying source code directly, so that vulnerabilities are easier to find with existing tools. Software security teams need to be aware that most software supply chain tools do not detect orphan vulnerabilities and that orphan vulnerabilities are common in C/C++ code. New tools that can identify orphan vulnerabilities are needed.

Additionally, our findings highlight the need for better tools to detect copied vulnerabilities. Current tools work well with package managers, but do not adequately

detect all copy-based reuse induced vulnerabilities. Tool builders have an opportunity to develop tools for orphan vulnerabilities that are similar to tools for other types of copied vulnerabilities. Better tools could improve the accuracy of copied code detection, be easily integrated into developer's workflows, track code provenance at the scale of all open source code, work with any programming language, and integrate with vulnerability databases. Our VCAnalyzer tool provides foundational work in that area, but it is only a beginning.

Researchers also need to be aware of the limitations of software supply chain tools and the high prevalence of orphan vulnerabilities. Studies are needed to advance our understanding of the risks associated with copy-based code reuse and identify best practices for minimizing these risks. Researchers can also help identify and analyze the specific types of vulnerabilities that are most commonly introduced through copy-based code reuse, as well as the factors that contribute to the prevalence of this practice.

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

UVHistory in two contexts: to identify license non-compliance and to find instances of unfixed vulnerabilities. We find that in active and popular projects both problems are common, and anyone can easily identify them using our approach.

Both open source and commercial tools fail to find many security vulnerabilities and license violations because they fail to trace the history of a file as it is copied and modified over time. We show that finding the universal version history of a file can help find issues that these tools miss.

5.2 Future Work

5.2.1 Improvements to detection methods

We would like to improve our file-level duplication detection while still maintaining the large-scale efficient method provided by World of Code. Minor modifications made to a copied file before it is committed for the first time causes the file not to match the original file. Removing comments and white space or using ctags and then storing the resulting hash in World of Code would allow those copied files with inconsequential changes to be discovered.

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

to the same bucket (unlike traditional hash) could be employed. Additions to the underlying World of Code infrastructure would be required to support these kinds of enhancements.

Additionally, we could learn techniques from research in signature analysis, speech recognition, pattern analysis, and image recognition, which typically involves specialized algorithms, machine learning, and data analysis techniques.

5.2.2 Additional studies

In order to better understand why code is copied without maintaining links to the code origin, we would like to conduct more broad studies including surveys and/or interviews with project maintainers.

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

5.2.3 VDiOS

In future work, we would like to make improvements to VDiOS that increase its precision, provide more useful reporting, and conduct more broad studies of orphan vulnerabilities and other orphan flaws to better understand why and under what circumstances the fixes are more likely to be (or not to be) applied.

We believe that improvements can be made in the algorithm used for identifying the vulnerable and not vulnerable blobs. There is probably no perfect method without manual inspection, but improvements are possible.

VDiOS checks to see if a project contains a known vulnerable file, but it has no way to know if the project uses the code in a vulnerable way. In other words, we know the project contains a vulnerable file, but that does not prove that the project is in fact vulnerable. It would be valuable, though time-consuming, to determine if the project is in fact vulnerable.

If we do not know when a vulnerability was introduced, VDiOS looks at a vulnerable revision of a file and then looks at all prior revisions since the prior revisions are likely to also contain the vulnerability. It would be valuable to know when the vulnerability was introduced so that we can look at all the revisions between when the vulnerability was introduced and when it was fixed. A tool like SZZ Unleashed [56] might be helpful in this area.

VDiOS could easily be expanded for other purposes. In addition to checking for security vulnerabilities, VDiOS could also be used to look for cases where one project has added enhancements that may be useful to other projects. VDiOS could also be enhanced to search for cases where code is reused but the license requirements are not propagated or have changed, which could help developers ensure that they follow the license terms correctly.

The primary focus of VDiOS is to collect the relevant information. The user interface is very rudimentary. In a future version, we would like to provide a better user interface.

5.2.4 UVHistory

The UVHistory tool presented here, as described earlier, is a prototype. Based on the success of our tests, we would like to further develop the tool into a production quality open source project. We would like to make improvements that increase its capabilities and provide more useful reporting. In this section, we describe some of the valuable improvements that could benefit the UVHistory tool.

In order to find the file history across the very large collection of open source software, performance is critical. World of Code provides the efficient search methods required for searching such a large collection. The focus of the initial work was to get the content correct. Performance issues were secondary. A future version would benefit from performance enhancements.

The current tool has a basic command line interface for running the tool. The output is produced in a simple HTML format for viewing in a browser. The primary focus of UVHistory is to collect the relevant information. The user interface is very rudimentary. In a future version, we would like to add an easy-to-use graphical user interface with a more readable output display.

This file history can be rather long and convoluted in some cases. In these cases, it is hard to see the complete path of the file history. A useful addition in a GUI version of the tool would be to display the history as a directed acyclic graph (DAG) across repositories in a manner similar to how a Git tool might display the Git history as a graphical representation of the Git DAG.

5.2.5 Artificial Intelligence

While we looked at the prevalence and risks associated with code reuse in open source software, it is evident that the landscape of software development is continually evolving, with emerging trends in AI and machine learning playing a pivotal role. The growing use of AI-generated code, driven by technologies such as large language models, opens opportunities to investigate the security vulnerabilities and licensing issues within this automatically generated code. We looked at the case where orphan vulnerabilities were included in the BigCode[*] open source large language model. Future research can look into the identification and assessment of vulnerabilities and compliance challenges unique to AI-generated code.

[*] www.bigcode-project.org

5.3 Conclusion

Code reuse through code duplication (white-box reuse) is a common practice in software development. While it has benefits, such as faster development time, lower cost, and improved quality, it also has inherent risks as the reused code may contain security vulnerabilities, license violations, or other problems. In some cases, those issues may be orphan (known and fixed in other repositories).

In this paper, we first described a case study with four different cases that show the extent of security vulnerabilities in open source software caused by code reuse. We also presented a tool, VDiOS, to find file-level code reuse in any language across the entirety of open source software by leveraging the World of Code infrastructure. Using VDiOS, we found very extensive white-box reuse of vulnerable code with a large number of projects that do not appear to fix the upstream vulnerability. These are cases where reused code contains known vulnerabilities or other bugs that persist in open source projects even though they have been fixed in other projects.

We next described a large scale empirical study of orphan vulnerabilities, which are vulnerabilities directly copied into open source repositories. We investigated the scale of the problem, along with characteristics of vulnerable projects and fixed projects. We developed a tool to find copied files and their project's characteristics across the expansive software collection in World of Code and created a dataset of vulnerable copied files and their fixes.

Finally, we articulated the concept of universal version history and argued for its usefulness in the context of the entirety of open source software. We introduced a prototype tool, UVHistory, that leverages the World of Code infrastructure to collect information about the source code and other artifacts to help better understand and manage widespread copying of source code. We demonstrated the value of the universal version history concept by finding evidence of negative effects of reuse, including reuse of outdated code that contains known vulnerabilities or other bugs, is missing useful features, or has different license restrictions. Our UVHistory tool helps

automate the production of the universal version history of source code by tracing code among repositories and enables finding the origins and version history for any source code file. We have shown the potential of our approach by demonstrating a solution in two different contexts which have practical relevance: license compliance and security vulnerabilities.

Overall, we may conclude that extensive code copying in OSS results in an extensive spread of vulnerable code that may take years to fix and that affects not only inactive, but also highly active and popular projects. We also found that many of the projects may not be willing to patch the vulnerabilities even after being provided a fix.

We found that 83.4% of the 3,615 vulnerabilities in our CVEfixes dataset were copied into more than three million files found in over seven hundred thousand open source projects in the World of Code. The majority (63.7%) of vulnerable copied files were C source or header files. We discovered that orphan vulnerabilities are rarely fixed. Only 100,889 (1.3%) of the three million vulnerable copied files were ever replaced with the fixed version of those files. Fixed vulnerabilities were only found in 26,801 (3.7%) of projects that contained orphan vulnerabilities. While large, active projects were more likely to remediate some vulnerabilities, the large majority of vulnerabilities were not remediated in such projects.

These findings suggest that addressing unfixed vulnerabilities in OSS requires at least three types of support. On one hand, if a patch is provided, some of the projects are willing to apply it. On the other hand, for projects that do not fix vulnerable code, we need to provide information to potential users of the code that their application still contains unfixed vulnerability. Finally, developers who are contemplating reusing the code in a project that contains unfixed vulnerabilities need to be informed about the risks and provided with suggestions on how to patch or with patches fixing the existing vulnerabilities.

In this comprehensive research endeavor, we investigated security vulnerabilities and license violations stemming from copy-based code reuse in open source software.

This work underscores the pervasive issue of orphan vulnerabilities, which are vulnerabilities persisting in copied code even after being fixed in their original projects. This work also considers license violations stemming from copy-based code reuse. We show how finding the universal version history of a file across open source repositories can help mitigate these issues. The findings emphasize the critical need for better tools, developer awareness, and practices to mitigate these risks in the open source community, ultimately contributing to the enhancement of software supply chain security, code quality, and license adherence.

References

[1] F. Nagle, "Open source software and firm productivity," *Management Science*, vol. 65, no. 3, pp. 1191–1215, 2019. [Online]. Available: https://doi.org/10.1287/mnsc.2017.2977 1

[2] W. Frakes and K. Kang, "Software reuse research: status and future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005. 1

[3] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110653, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301199 1

[4] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *Commun. ACM*, vol. 50, no. 1, p. 79–83, Jan. 2007. [Online]. Available: https://doi.org/10.1145/1188913.1188921 1

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5 2, 31

[6] J. Düsing and B. Hermann, "Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories," *Digital Threats: Research and Practice*, 2021. 2, 29, 30

[7] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010. 2, 29, 30

[8] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 181–191. [Online]. Available: https://doi.org/10.1145/3196398.3196401 2, 26, 27

[9] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 283–292. 2, 8, 26

[10] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301. 2, 8, 26

[11] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "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*, 2014, pp. 305–314. 2, 28

[12] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, "Source file set search for clone-and-own reuse analysis," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 257–268. 2, 27, 29

[13] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 391–400. 2, 26

[14] ——, “The ecco tool: Extraction and composition for clone-and-own,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 665–668. 2, 26

[15] F. Pérez, M. Ballarín, R. Lapeña, and C. Cetina, “Locating clone-and-own relationships in model-based industrial families of software products to encourage reuse,” *IEEE Access*, vol. 6, pp. 56 815–56 827, 2018. 2, 26

[16] T. Zimmermann, *A First Look at an Emerging Model of Community Organizations for the Long-Term Maintenance of Ecosystems' Packages*. New York, NY, USA: Association for Computing Machinery, 2020, p. 711–718. [Online]. Available: https://doi.org/10.1145/3387940.3392209 2, 26

[17] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “When and how to make breaking changes: Policies and practices in 18 open source software ecosystems,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3447245 2, 26

[18] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, “Tracking known security vulnerabilities in proprietary software systems,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 516–519. 4

[19] E. Raymond, “The cathedral and the bazaar,” *Knowledge, Technology and Policy*, vol. 12, pp. 23–49, 1999. [Online]. Available: https://doi.org/10.1007/s12130-999-1026-0 5, 88

[20] D. Favato, D. Ishitani, J. Oliveira, and E. Figueiredo, “Linus's law: More eyes fewer flaws in open source projects,” in *Proceedings of the XVIII Brazilian Symposium on Software Quality*, ser. SBQS'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 69–78. [Online]. Available: https://doi.org/10.1145/3364641.3364650 5

[21] Dependabot. (2021) Github Dependabot. [Online]. Available: https://github.com/dependabot 5, 29, 66, 83

[22] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "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)*, 2019, pp. 143–154. 6, 11, 13, 24, 25

[23] C. S, C. K, R. A, H. G, A. A, and S. A, "The case study approach," *BMC Medical Research Methodology*, 2011. 6

[24] OWASP. (2017) The Open Web Application Security Project OWASP Top 10. [Online]. Available: https://owasp.org/www-project-top-ten 7

[25] N. Kshetri and J. Voas, "Supply chain trust," *IT Professional*, vol. 21, no. 2, pp. 6–10, 2019. 7

[26] P. Myerson. (2017) Can't turn back time: Cybersecurity must be dealt with. [Online]. Available: https://www.industryweek.com/supply-chain/article/22006116/cant-turn-back-time-cybersecurity-must-be-dealt-with 7

[27] The White House. (2021) Executive order 14028 on improving the nation's cybersecurity. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity 9

[28] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, p. 131–183, Jun. 1992. [Online]. Available: https://doi-org.proxy.lib.utk.edu/10.1145/130844.130856 12

[29] Y. Ma, T. Dey, C. Bogart, S. Amreen, M. Valiev, A. Tutko, D. Kennard, R. Zaretzki, and A. Mockus, "World of code: enabling a research workflow for mining and analyzing the universe of open source vcs data," *Empirical Software Engineering*, vol. 26, 2021. 13, 24, 40, 49

[30] S. Amreen, A. Karnauch, and A. Mockus, "Developer reputation estimator (dre)," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1082–1085. 14, 20, 21

[31] E. Lyulina and M. Jahanshahi, "Building the collaboration graph of open-source software ecosystem," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 618–620. 15

[32] A. Mockus, D. Spinellis, Z. Kotti, and G. J. Dusing, "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*, 2020, pp. 513–517. 15, 40, 46, 52, 58, 62, 85

[33] The MITRE Corporation. (2021) Common vulnerabilities and exposures (cve). [Online]. Available: https://cve.mitre.org/ 15, 19, 28, 36

[34] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0finder: Discovering the correct origin of publicly reported software vulnerabilities," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3041–3058. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/woo 16, 28, 55

[35] R. Geldreich. (2020) richgel999/jpeg-compressor . [Online]. Available: https://github.com/richgel999/jpeg-compressor 16

[36] The MITRE Corporation. (2017) Cve-2017-0700. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0700 16

[37] Android. (2017) Android Security Bulletin—July 2017 . [Online]. Available: https://source.android.com/security/bulletin/2017-07-01 16

[38] P. Ombredanne, "Free and open source software license compliance: Tools for software composition analysis," *Computer*, vol. 53, no. 10, pp. 105–109, 2020. 16

[39] MusicIP. (2007) musicip-libofa . [Online]. Available: https://code.google.com/archive/p/musicip-libofa/ 17

[40] Synopsys Technology. (2021) 2021 open source security and risk analysis. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html?intcmp=sig-blog-ossra1 18

[41] R. Di Cosmo and S. Zacchiroli, "Software heritage: Why and how to preserve software source code," in *iPRES 2017-14th International Conference on Digital Preservation*, 2017, pp. 1–10. 24

[42] A. Pietri, D. Spinellis, and S. Zacchiroli, "The software heritage graph dataset: Public software development under one roof," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 138–142. 25

[43] G. Rousseau, R. Di Cosmo, and S. Zacchiroli, "Software provenance tracking at the scale of public source code," in *Empirical Software Engineering*, 2020. 25

[44] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "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*, 2014, pp. 305–314. 26

[45] A. Mockus, "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)*, 2007, pp. 7–7. 26

[46] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue, "Studying reuse of out-dated third-party code in open source projects," *Information and Media Technologies*, vol. 9, no. 2, pp. 155–161, 2014. 26

[47] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1289–1292. 26

[48] P. Xia, Y. Manabe, N. Yoshida, and K. Inoue, "Development of a code clone search tool for open source repositories," *Information and Media Technologies*, vol. 7, no. 4, pp. 1370–1376, 2012. 26

[49] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005. 26

[50] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "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*, 2014, pp. 305–314. 27

[51] The Apache Software Foundation. (2021) Apache maven project. [Online]. Available: https://maven.apache.org/ 27

[52] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Sv-af — a security vulnerability analysis framework," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 219–229. 27, 29

[53] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? - integrated code history tracker for open source systems," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 331–341. 28

[54] J. Davies, D. German, M. Godfrey, and A. Hindle, "Software bertillonage," in *Empirical Software Engineering*, 2013. [Online]. Available: https://doi.org/10.1007/s10664-012-9199-7 28

[55] M. W. Godfrey, D. M. German, J. Davies, and A. Hindle, “Determining the provenance of software artifacts,” in *Proceedings of the 5th International Workshop on Software Clones*, ser. IWSC ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 65–66. [Online]. Available: https://doi.org/10.1145/1985404.1985418 28

[56] M. Borg, O. Svensson, K. Berg, and D. Hansson, “Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 7–12. [Online]. Available: https://doi.org/10.1145/3340482.3342742 29, 95

[57] Github. (2021) About the dependency graph. [Online]. Available: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph 29, 83

[58] Google. (2021) Open Source Insights. [Online]. Available: https://deps.dev/ 29, 83

[59] OWASP. (2022) OWASP Dependency-Check. [Online]. Available: https://owasp.org/www-project-dependency-check/ 29, 83

[60] M. Alfadel, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in python packages,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 446–457. 29, 30

[61] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018. 29

[62] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191. 30

[63] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1513–1531. 31

[64] J. Davies, "Measuring subversions: Security and legal risk in reused software artifacts," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1149–1151. [Online]. Available: https://doi.org/10.1145/1985793.1986025 31, 55

[65] Y. Chen, A. E. Santosa, A. Sharma, and D. Lo, "Automated identification of libraries from vulnerability data," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 90–99. [Online]. Available: https://doi.org/10.1145/3377813.3381360 31

[66] National Institute of Standards and Technology. (2021) National Vulnerability Database. [Online]. Available: http://nvd.nist.gov 31, 64

[67] D. M. German, M. Di Penta, and J. Davies, "Understanding and auditing the licensing of open source software distributions," in *2010 IEEE 18th International Conference on Program Comprehension*, 2010, pp. 84–93. 32

[68] Y. Wu, Y. Manabe, T. Kanda, D. M. German, and K. Inoue, "A method to detect license inconsistencies in large-scale open source projects," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 324–333. 32

[69] T. Wolter, A. Barcomb, D. Riehle, and N. Harutyunyan, "Open source license inconsistencies on github," *ACM Trans. Softw. Eng. Methodol.*, dec 2022. [Online]. Available: https://doi.org/10.1145/3571852 32

[70] S. Baltes and S. Diehl, "Usage and attribution of stack overflow code snippets in github projects," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1259–1295, Jun 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9650-5 32

[71] S. Qiu, D. M. German, and K. Inoue, "Empirical study on dependency-related license violation in the javascript package ecosystem," *Journal of Information Processing*, vol. 29, pp. 296–304, 2021. 32

[72] H.-F. Chang and A. Mockus, "Constructing universal version history," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 76–79. [Online]. Available: https://doi.org/10.1145/1137983.1138002 32

[73] ——, "Evaluation of source code copy detection methods on freebsd," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 61–66. [Online]. Available: https://doi.org/10.1145/1370750.1370766 33

[74] A. Mockus, "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)*, 2007, pp. 7–7. 33

[75] D. Reid, M. Jahanshahi, and A. Mockus, "The extent of orphan vulnerabilities from code reuse in open source software," *International Conference on Software Engineering*, p. 2104–2115, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510216 42

[76] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614. 42

[77] Z. Liu, Q. Wei, and Y. Cao, "Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint," in *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, 2017, pp. 548–553. 42

[78] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39. 45

[79] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 2169–2185. 49

[80] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in c/c++ ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12. 49

[81] World of Code. (2022) World of Code. [Online]. Available: https://worldofcode.org/ 50

[82] D. Reid, M. Jahanshahi, and A. Mockus, "The extent of orphan vulnerabilities from code reuse in open source software," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022. 55

[83] T. Boutell, "PNG (Portable Network Graphics) Specification Version 1.0," RFC 2083, Mar. 1997. [Online]. Available: https://rfc-editor.org/rfc/rfc2083.txt 56

[84] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8446.txt 57

[85] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Security Privacy*, vol. 12, no. 4, pp. 63–67, 2014. 57

[86] The MITRE Corporation. (2021) Cve-2021-3449. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3449 57

[87] OpenSSL. (2021) News/Vulnerabilities. [Online]. Available: https://www.openssl.org/news/vulnerabilities.html 57

[88] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018. 59, 63, 76

[89] The MITRE Corporation. (2014) Cve-2014-0160. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 59

[90] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu, "Risk assessment of buffer "heartbleed" over-read vulnerabilities," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 555–562. 59

[91] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488. 60

[92] The MITRE Corporation. (2021) Cve-2021-29482. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29482 62

[93] T. Fry, T. Dey, A. Karnauch, and A. Mockus, "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*, 2020, pp. 518–522. 63

[94] G. Roelofs. (2006) libpng.org. [Online]. Available: http://www.libpng.org 63

[95] C. Truta and G. Randers-Pehrson. (2020) LIBPNG: PNG reference library. [Online]. Available: https://sourceforge.net/projects/libpng 64

[96] Slashdot Media. (2020) SourceForge. [Online]. Available: https://sourceforge.net 64

[97] Glenn Randers-Pehrson. (2020) glennrp/libpng . [Online]. Available: https://github.com/glennrp/libpng 64

[98] The MITRE Corporation. (2017) Cve-2017-12652. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12652 64

[99] A. Miranda and J. Pimentel, "On the use of package managers by the c++ open-source community," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1483–1491. 68

[100] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85. 77

[101] S. W. Flint, J. Chauhan, and R. Dyer, "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)*, 2021. 85

[102] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017. 85

[103] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018. 86

Appendix A

List of Publications

This dissertation is largely based on the following published works:

- David Reid, Mahmoud Jahanshahi, and Audris Mockus. "The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software." Proceedings of the 44th International Conference on Software Engineering. 2022

- David Reid, Kristiina Rahkema, and James Walden. "Large Scale Study of Orphan Vulnerabilities in the Software Supply Chain." Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering. 2023

- David Reid and Audris Mockus. "Applying the Universal Version History Concept to Help De-Risk Copy-Based Code Reuse." IEEE 22nd International Working Conference on Source Code Analysis and Manipulation. 2023

- David Reid, Kalvin Eng, Chris Bogart, and Adam Tutko. "Tracing Vulnerable Code Lineage." IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). 2021

Vita

David Reid grew up in Knoxville, Tennessee and attended Doyle High School. After high school, he attended Milligan University and earned a Bachelor of Science degree in Math and Computer Science. Continuing his education at the University of Tennessee Knoxville, he earned a Master of Science degree in Computer Science and served as a graduate teaching assistant. He also taught evening classes at Knoxville Business College (now South College) while in graduate school. David has been a software engineer and engineering manager in the Knoxville area for over 30 years. With an interest in teaching computer science at the college level, he returned to the University of Tennessee to pursue a Ph.D. in computer science. David is currently teaching COSC 340 Software Engineering at the University of Tennessee while finishing his degree.