Large Scale Study of Orphan Vulnerabilities in the Software Supply Chain

David Reid
University of Tennessee
Knoxville, TN, USA
dreid6@vols.utk.edu

Kristiina Rahkema
University of Tartu
Tartu , Estonia
kristiina.rahkema@ut.ee

James Walden
Northern Kentucky University
Highland Heights, KY , USA
waldenj@nku.edu

ABSTRACT

The security of the software supply chain has become a critical issue in an era where the majority of software projects use open source software dependencies, exposing them to vulnerabilities in those dependencies. Awareness of this issue has led to the creation of dependency tracking tools that can identify and remediate such vulnerabilities. These tools rely on package manager metadata to identify dependencies, but open source developers often copy dependencies into their repositories manually without the use of a package manager.

In order to understand the size and impact of this problem, we designed a large scale empirical study to investigate vulnerabilities propagated through copying of dependencies. Such vulnerabilities are called orphan vulnerabilities. We created a tool, VCAnalyzer, to find orphan vulnerabilities copied from an initial set of vulnerable files. Starting from an initial set of 3,615 vulnerable files from the CVEfixes dataset, we constructed a dataset of more than three million orphan vulnerabilities found in over seven hundred thousand open source projects.

We found that 83.4% of the vulnerable files from the CVEfixes dataset were copied at least once. A majority (59.3%) of copied vulnerable files contained C source code. Only 1.3% of orphan vulnerabilities were ever remediated. Remediation took 469 days on average, with half of vulnerabilities in active projects requiring more than three years to fix. Our findings demonstrate that the number of orphan vulnerabilities not trackable by dependency managers is large and point to a need for improving how software supply chain tools identify dependencies. We make our VCAnalyzer tool and our dataset publicly available.

CCS CONCEPTS

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

KEYWORDS

Orphan Vulnerabilities, Copy-based Code Reuse, Software Supply Chain

PROMISE '23, December 8, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0375-1/23/12…$15.00
https://doi.org/10.1145/3617555.3617872

ACM Reference Format:
David Reid, Kristiina Rahkema, and James Walden. 2023. Large Scale Study of Orphan Vulnerabilities in the Software Supply Chain. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '23), December 8, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3617555.3617872

1 INTRODUCTION

Open source software is an essential component of the software supply chain, found in 97% of commercial codebases [39]. Open source is not only found in most code bases; it also makes up the large majority (78%) of code in commercial software projects [39]. Using open source libraries improves developer productivity by eliminating time spent on duplicating functionality already present in open source code. Such use of open source code has been shown to improve productivity at the enterprise level [29].

While widespread use of open source libraries improves productivity, it also exposes software to vulnerabilities discovered in those libraries. A vulnerability in a single package, log4j, a popular Java logging library, exposed more than 17,000 packages in the Maven Central repository to the log4shell remote code execution vulnerability [41]. A software project that used any of those thousands of packages was exposed to the vulnerability.

The spread of vulnerabilities in open source software dependency networks and how fast they are fixed has been studied for multiple open source software ecosystems [7, 11, 21, 45]. Although developers may be reluctant to update their library dependencies [21], dependency tools exist that can make the task of fixing vulnerable dependencies easier.

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 is 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 which vulnerabilities an application is exposed to. Tools like GitHub Dependabot[1] 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.

[1] https://github.com/dependabot

Reid et al. [36] use the term "orphan vulnerabilities" to describe vulnerabilities in software dependencies that have been copied into repositories without the use of a package management tool. Their study used the World of Code[2] to track orphan vulnerabilities copied from an initial set of four vulnerabilities. They discovered thousands of projects that included copies of the vulnerable code, with many of the projects still containing the vulnerable code in their most recent version. They used an exploratory case study approach to investigate 4 vulnerabilities in depth since the current understanding of the problem is limited. As is typical with case studies, they primarily looked at qualitative data. Based on their results, it is clear that a larger scale empirical study addressing additional questions is warranted. We expand on their work by starting our investigation with a significantly larger set of initial vulnerabilities, considering the project characteristics that may affect the likelihood of a project being fixed, and investigating how long it took to apply a fix. Our tool, VCAanlyzer, collects additional quantitative data as described in Section 5. We use this data to better understand which projects are more or less likely to include orphan vulnerabilities and to be fixed. We also look at the time it takes to apply a fix. The purpose is to identify ways to mitigate these types of security risks. To secure the software supply chain, developers need to identify and remediate orphan vulnerabilities in addition to vulnerabilities detected by dependency tracking tools.

In this study, we examine the prevalence and characteristics of orphan vulnerabilities at scale. We use the CVEfixes dataset [3] as our initial set of vulnerabilities. This dataset consists of all vulnerability fixing commits that the authors could automatically identify from the National Vulnerability Database. We identify orphan vulnerabilities by searching for versions of files identified in CVEfixes that existed before vulnerability fixing commits were applied in the World of Code [24], a dataset of 173 million open source code repositories and 3.1 billion commits from multiple hosting platforms, including GitHub, GitLab, and Bitbucket. We created a custom tool to find files with orphan vulnerabilities in the billions of files in the World of Code by matching file hashes. This approach allows us to scale to the size of the World of Code - nearly all of open source.

Our work makes the following contributions:

- We conduct the first large scale empirical study of orphan vulnerabilities in open source software, describing the frequency of their creation and mitigation, identifying the impact of programming languages on these features, and analyzing the properties of projects that create and remediate orphan vulnerabilities. Using the World of Code infrastructure, we are able to analyze the extent of cloning vulnerable files at a scale that has traditionally been very difficult
- We present the design and implementation of a tool, VCAnalyzer, which finds source code files in any language that contain orphan vulnerabilities. We make the tool publicly available.
- Using VCAnalyzer, we produce a dataset, which we make publicly available, containing over 3 million files with orphan vulnerabilities that have been copied into over 700,000 unique open source projects. The dataset also contains metadata about each project.

[2] https://worldofcode.org

The rest of the paper is organized as follows: we start with our research questions in Section 2, discuss related work in Section 3, present our data sources in Section 4, and discuss our VCAnalyzer tool in Section 5, We describe our research methods in Section 6, present results in Section 7, discuss the impacts of our results in Section 8, address threats to validity in Section 9, and summarize our findings in Section 10.

2 RESEARCH QUESTIONS

Motivated by previous research showing vulnerabilities propagated by copy-based code reuse in a small number of vulnerable projects [20, 22, 36], 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.

As our first step, we wanted to measure the prevalence of orphan vulnerabilities, including determining how many original vulnerabilities are copied and how frequently they are copied. We investigated the impact of programming language on the frequency of orphan vulnerabilities, and examined the 20 most copied orphan vulnerabilities in detail.

Research Question 1: How prevalent are orphan vulnerabilities? What are the characteristics of orphan vulnerabilities?

For our second question, we investigated characteristics of open source projects that contained orphan vulnerabilities. We measured how many vulnerabilities were copied in each project. We also examined project activity, including duration of activity, numbers of commits, authors, stars, and additional metadata to find commonalities among these projects.

Research Question 2: What are the characteristics of projects that have orphan vulnerabilities?

We investigated the fixing of orphan vulnerabilities. While all of the vulnerabilities in our dataset have been fixed in the original project, we wanted to see how many of the orphan vulnerabilities were fixed. Some of our copied vulnerable files could have been created using package managers that install packages locally under the development directory, but many other copies were created through manually copying the code. While dependency tracking tools can help developers remediate packages installed through package managers, there is no means of informing users of manually copied code when security patches become available.

Research Question 3: How many orphan vulnerabilities are fixed?

We know that projects behave differently when updating vulnerabilities using package management tools. While some projects may not fix any orphan vulnerabilities, others may fix some or all of the orphan vulnerabilities. We studied which projects fixed vulnerabilities, and what percentage of vulnerabilities were fixed in projects that fixed any vulnerabilities. We also studied how project and vulnerability characteristics, e.g. the programming language used or project activity, affect whether vulnerabilities are fixed.

Research Question 4: How do different characteristics of projects affect how many vulnerabilities are fixed?

Lastly, we investigated the time required to fix orphan vulnerabilities. To secure systems, it is important to fix vulnerabilities before they begin to be widely exploited. A late fix may be no improvement over no fix if the software was exploited before the fix was deployed. We also examined the impact of project activity on the time required to remediate vulnerabilities.

Research Question 5: How long does it take for an orphan vulnerability to be fixed?

3 RELATED WORK

3.1 Copy-based Code Reuse

Our work looks at vulnerabilities propagated through copy-based code reuse. There is significant research in the area of copy-based code reuse [8, 19, 27, 35, 44], sometimes called vendoring [4, 46] or clone-and-own [12, 13, 34]. Gharehyazie et al. [14] analyzed copy-based code reuse by looking at 5,753 Java projects on GitHub, and found that cross-project code reuse is prevalent. Schwarz et al. [38] studied cloned methods in 2,705 projects in the Squeak-source ecosystem and found that 15% to 18% of methods were cloned. Ossher et al. [31] studied 13,000 Java projects from the Sourcerer Repository, and found that over 10% of all files are clones and that 15% of projects contain at least one cloned file. Xia et al. [44], using OpenCCFinder [43], looked at C language reuse of out-of-date code. OpenCCFinder only returns small subset of open source projects. Tang et. al. [40] introduce CCScanner, a tool to find dependencies in C/C++ projects, and they evaluate the tool with a dataset of 24,000 projects. Kim et. al. [20] propose VUDDY, an approach for the scaleable detection of vulnerable code clones, and test it with a pool of 25,253 C/C++ projects. All of the previously mentioned works are limited to a small number of repositories relative to the totality of open source projects, and are limited to a small number of languages. Newer technologies such as World of Code [25] and Software Heritage [9] provide an infrastructure to find copied files across a much larger set of software repositories and across all languages. Our work utilizes the World of Code infrastructure containing over 173 million projects, allowing us to study vulnerabilities propagated through copy-based code reuse on a scale much larger than previous work.

3.2 Software Provenance

Software provenance refers to the history and chain of custody of software. Godfrey et al. [16] noted the importance of finding software provenance and the lack of current tools and techniques in that area. We aim to fill part of the gap they identified with our VCAnalyzer tool, which traces the evolution of source code files that are copied and modified over time and across different source code repositories.

Woo et al. [42] created a tool, V0Finder, to find the origin of a vulnerability. They use function-level clone detection, which allows fine-grained detection of vulnerabilities, but cannot scale to the number of repositories that we are able to search using file-level hash matching. Inoue et al. [18] created a tool to trace code origin and evolution. They used source code search engines Koders, Google Codesearch, and SPARS/R, all of which had limitations and are no longer available. We use World of Code, which contains a larger set of projects and is currently actively maintained.

Rousseau et al [37] used Software Heritage to look at copies of file content over time and across repositories, similar to what VCAnalyzer does using World of Code. One key difference is that they do not look at "predecessors or successors in a given development history." VCAnalyzer specifically looks at all parents and descendants of a file when tracing file evolution over time and across repositories, allowing the tool to find files that are copied and then modified.

3.3 Package Managers

Part of the motivation for our work arises from a lack of research and tools dealing with copy-based reuse induced vulnerabilities. Popular dependency checking tools such as GitHub Dependency Graph [15], Google Open Source Insights [17], and OWASP Dependency Check [32] 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 [1, 2, 11, 45]. Kula et al. [21] 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. Zimmermann et al. [45] 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. Decan et al. [7] 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. Pashchenko et al. [33] studied dependency managements and its security implications by interviewing developers. They found that developers focus on functionality over security when choosing dependencies.

4 DATA

In this section, we describe the two primary data sources that we use to conduct our analysis: CVEfixes and the World of Code.

4.1 CVEfixes Dataset

Vulnerability databases such as the National Vulnerability Database (NVD)[3] 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. [3] extracted CVEs with fixing commits, analyzed the files changed by these commits, and created a dataset containing vulnerabilities and their fixes - the CVEfixes dataset.[4] 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.

[3] https://nvd.nist.gov
[4] https://github.com/secureIT-project/CVEfixes

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.

4.2 World of Code

The World of Code (WoC) [24] 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 the 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.

Our VCAnalyzer tool is layered on top of the WoC 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 tool to efficiently search for code duplication in any programming language across multiple source code repository hosting platforms.

Using VCAnalyzer, we constructed a dataset of more than three million copied files containing orphan vulnerabilities from the CVEfixes dataset. The dataset includes CVEs, as well as file and project metadata, including pathnames, timestamps, project activity, etc.

5 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 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 [28] 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.

6 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 with 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 5 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 [10, 40].

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 5. 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.

7 RESULTS

7.1 RQ1: Prevalence of Orphan Vulnerabilities

We found that 3,014 (83.3%) of the 3,615 vulnerable files from our CVEfixes dataset were copied into 3,044,644 files. While 601 original vulnerabilities were not copied into other projects, there are more than a thousand orphan vulnerabilities on average for each original vulnerability that was copied. We found that 95.4% of copied vulnerable files shared directory paths with the original vulnerable files, which indicates that orphan vulnerabilities are typically introduced by copying an entire dependency into the project's repository.

The majority of original vulnerable files that were copied are written in the C programming language (59.3%), while another 10.2% are written in C++. An even larger percentage of orphan vulnerabilities are found in copied C source code files (63.7%). These facts can be explained in part by the fact that most developers do not use package management tools like Conan for C and C++ projects [26], and prefer to either use system libraries or to copy code into their repositories [40].

We find that vulnerable C++ files are much less likely to be copied than vulnerable C files. Although C++ files represent 10.2% of original vulnerable files, only 38,533 (1.3%) copied vulnerable source files are written in C++. While a relatively small quantity (5.7%) of original vulnerable files are written in JavaScript, 24.3% of orphan vulnerabilities are found in JavaScript files. We found that overrepresentation of JavaScript files among the copies is due to use of the `npm` package manager.

While typical use of a package manager calls for committing metadata files that indicate which packages are used rather than committing the packages themselves, we found that some projects committed dependencies as well as metadata into their repositories. As a result, some of the copied vulnerabilities we discovered were copied using package managers instead of through direct copying of the code. To check if a package manager was used, we checked the most common file path prefixes used by package managers for each programming language. We discovered some file path prefixes that indicated the use of package managers, such as `node_modules`, `Pods`, and `Carthage`. The percentage of vulnerable files included through package managers rather than manual copying is 17% for C++ files, 36% for Go files, 67% for JavaScript files, 66% for JSON files, 47% for PHP files, 9% for Ruby files, and 53% for Swift files. For all other languages, the percentage of files copied by package managers was less than 5%.

While the majority of both original vulnerable files and copied files are written in C, PHP, and C++ are underrepresented among copied files, making up 6.4% (compared to 15.1% of original files) and 1.3% (compared to 10.1% of original files) respectively. Table 2 shows the top 10 programming languages found in copied files.

While the mean number of times a particular vulnerability is copied is 1,010, the standard deviation is 4,581, indicating a wide variance in the number of times vulnerabilities are copied. At the low end, 305 (10.1%) vulnerabilities are copied only once, while at the high end 380 (12.6%) vulnerabilities are copied more than a thousand times.

Table 1: Original Vulnerable Files by Language

Language	Vulnerable Files	Percent of Files
C	1789	59.3%
PHP	455	15.1%
C++	307	10.2%
JavaScript	157	5.7%
Python	59	2.0%
Java	44	1.5%
Ruby	29	1.0%
Go	22	0.7%
Perl	18	0.6%
TypeScript	10	0.3%

Table 2: Copied Vulnerable Files by Language

Language	copied files	Percent of Copies
C	1,939,190	63.7%
JavaScript	741,433	24.3%
PHP	194,991	6.4%
Ruby	102,986	3.4%
C++	38,533	1.3%
Python	17,478	0.6%
Go	1,758	0.06%
Perl	1,282	0.04%
Zsh	1,253	0.04%
Java	963	0.03%

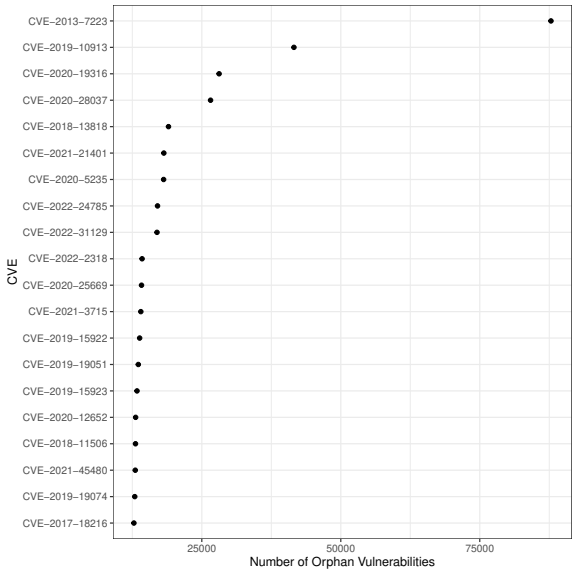


Figure 1: Top 20 CVEs by Number of Times Copied

The most copied vulnerabilities are dominated by files copied via the `npm` package manager, with CVE-2021-32640 being copied 112,297 times. After eliminating files copied by `npm`, the twenty most commonly copied vulnerabilities are shown in Figure 1. The most commonly copied vulnerability, CVE-2013-7223, is a cross-site request forgery vulnerability in Fat Free CRM, a customer relationship management platform written in Ruby. It was copied 87,808 times. The second through fifth most commonly copied vulnerabilities are from PHP software, including the Symfony framework for web projects, Laravel Framework, WordPress, and the Twig templating system. Numbers of orphan vulnerabilities ranged from 19,005 for CVE-2018-13818 in Twig to 41,564 for CVE-2019-10913 in Symfony.

The sixth and seventh most commonly copied vulnerabilities are both from `nanopb`, an implementation of the protocol buffers serialization mechanism written in C. Both vulnerabilities were copied more than 18,000 times, with CVE-2021-24401 being copied 38 times more than CVE-2021-5235. The eighth and ninth most commonly copied vulnerabilities are from `Moment.js`, a JavaScript date library, with both being copied more than 17,000 times. The tenth through twentieth most commonly copied vulnerabilities are all found in C source code files originating in the Linux kernel, with numbers of copies ranging from 12,785 to 14,253. It is worth noting that the majority of the top 20 most commonly copied orphan vulnerabilities are not found in libraries, and thus would not be detected by software supply chain tools that focus on third-party library dependencies.

7.2 RQ2: Characteristics of Projects that Copy Vulnerabilities

There are 1,114 open source projects in the CVEfixes dataset. Of these projects, 800 (71.8%) had orphan vulnerabilities. Vulnerable files from these 800 projects were copied into 719,131 different projects found in the World of Code. While a majority (58.3%) of the 719,131 projects contain only a single orphan vulnerability and 97.5% of projects have 10 or fewer such vulnerabilities, 9428 (1.3%) projects include 100 or more copied vulnerable files up to a maximum of 806. Seven of the ten projects with the highest numbers of vulnerable copies have the words "linux" or "kernel" in their project names, suggesting that they are copies of the Linux kernel, which is the project with the highest number of vulnerable files in our CVEfixes dataset.

We analyzed project activity through several metrics, including active project lifetime and counts of commits, authors, and GitHub stars. With a dataset of so many projects, it is unsurprising that commit activity varied widely, ranging from a low of one commit to a high of over 36 million commits. Most projects have low levels of commit activity, with 94.3% of projects having 100 or fewer commits and 61.3% having 10 or fewer commits. Additionally, 99% of projects have 10 or fewer commit authors, with 71% of projects having only a single commit author. The number of active months varies widely between projects, from 1 to 428 (35 years). Projects with more than 30 years of history include some well known projects like Emacs, FreeBSD, gcc, Kerberos, and Python.

The vast majority (98.5%) of our projects have repositories on GitHub, so we also examined the number of GitHub stars for those

projects. The majority of projects had no stars on GitHub (83.3% of GitHub projects). A substantial number of projects (10.4%) had received a single star, with a small number of outliers having received more than one.

Most projects that contain orphan vulnerabilities do not have GitHub security policies as documented in the `SECURITY.md` file. These files describe how to report security vulnerabilities to project maintainers. Only 1.7% of projects with orphan vulnerabilities include a security policy file. Summary statistics of project metrics, authors, commits, stars, and vulnerabilities can be found in Table 3.

Table 3: Project Activity Metrics

Metric	Min	Median	Mean	StdDev	Max
Active Months	1	1	3.18	8.6	428
Authors	1	1	5.15	263.5	109,725
Commits	1	7	353	53,916	36,468,369
GitHub Stars	0	0	11.2	757.5	357,516
Vulnerabilities	1	1	4.23	18.5	806

We created a subset of 2,021 projects that had high levels of activity by selecting projects with at least 100 GitHub stars. These projects have means of over 6 years (82 months) of activity, 58 authors, 2,413 commits, and 544 stars. They contain an average of 6.28 copied vulnerabilities compared to 4.23 for all projects. While active projects have almost 50% more copied vulnerable files on average, they are more likely to have published a security policy, with 11.6% of active projects having a `SECURITY.md` file compared to 1.7% of all projects. A summary of statistics for active projects is available in Table 4. Although the smallest number of commits for active projects was 3, only 0.013% of the active projects had less than 100 commits.

Table 4: Active Subset Project Metrics

Metric	Min	Median	Mean	StdDev	Max
Active Months	1	64	82.0	69.6	428
Authors	1	15	58.0	2864	93,072
Commits	3	410	2413	499,701	17,719,890
GitHub Stars	100	204	544	13,617	357,516
Vulnerabilities	1	1	6.28	31.4	677

7.3 RQ3: Orphan Vulnerabilities that are Fixed

We find that there are only 100,889 (3.3%) files out of over three million copied files in the World of Code dataset, where the vulnerable file was replaced at a later time with the fixed version of the file from the original project. Another 68,760 copied files (2.3%) were modified from the original vulnerable version, but we do not know whether the modifications were to remediate the vulnerability or for another purpose. The remaining 2,875,018 (94.4%) copied files remained identical to the original vulnerable file throughout the history of the project that copied them.

Fixed vulnerabilities are found in 26,801 (3.7%) of the 719,204 projects. We found that more than half of projects that fixed one vulnerability have fixed all of their vulnerabilities, though it is worth noting that of the 14,276 projects have fixed all of their vulnerabilities, 11,627 (79%) had only one vulnerability to fix. However, only 1.6% of World of Code projects with a single copied vulnerable file have fixed that vulnerability.

7.4 RQ4: Projects that Fix Orphan Vulnerabilities

We analyzed the relationship between different characteristics of projects and the percentage of fixed vulnerabilities. We looked at the following project characteristics: project language, vulnerable file language, number of commits, number of contributors, community size, number of core contributors, number of active months, and number of stars.

For project language, we analyzed the primary language of the project provided by the World of Code, which uses heuristics to determine the primary language of the project. In total, vulnerabilities originated from projects written in 14 different programming languages. Not all projects had a primary programming language indicated in the World of Code. Table 5 shows percentages of copied vulnerabilities with status fixed, not fixed, and unknown for each project language. For most languages, over 90% of copied vulnerabilities are not fixed. Clear outliers are Rust, Go, SQL where 36.3%, 17.1%, and 10.9% of copied vulnerabilities were fixed, respectively.

Table 5: 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

Next, we looked at the percentage of fixed and not fixed copied files based on the language in which the vulnerable file was written. To simplify determining the project language, we used the suffix of the vulnerable file to identify the programming language. The following analysis is performed on the 17 most often occurring file endings in the dataset. Percentages of fixed, not fixed, and unknown copied vulnerabilities are listed in Table 6. For most file endings, as with programming languages, the percentage of copied

vulnerabilities not being fixed was over 90%. Outliers include the file suffixes '.S', '.go', '.h', '.py' and '.swift'. Interestingly, vulnerabilities in '.S' (assembly language) files are fixed in 73.1% of cases.

Table 6: Percentage of copied vulnerabilities with status fixed, not fixed, and unknown for each file ending

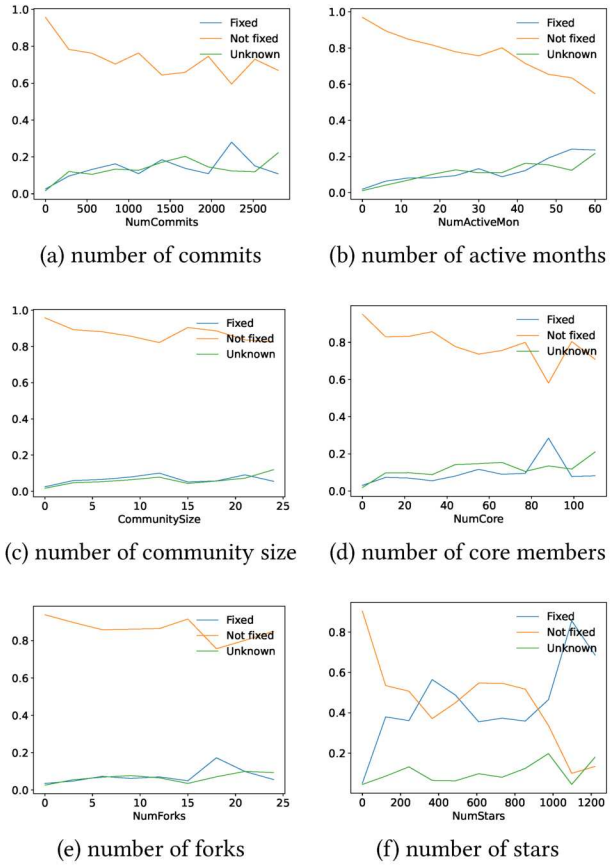
File Suffix	Not Fixed	Fixed	Unknown
.c	92.9	4.1	3.0
.cc	99.0	0.2	0.8
.cpp	95.1	3.2	1.7
.cs	94.4	3.5	2.1
.go	76.9	19.1	4.0
.h	88.6	9.6	1.7
.htm	94.8	4.3	0.9
.java	93.3	2.8	3.9
.js	98.2	1.5	0.3
.json	90.9	0.0	9.1
.php	95.2	2.6	2.2
.pm	98.5	0.7	0.8
.py	87.1	12.1	0.8
.rb	97.4	0.4	2.2
.S	26.8	73.1	0.1
.swift	88.9	9.1	2.0
.zsh	98.2	0.0	1.8

We analyzed how different project metrics affected the percentage of fixed and unfixed orphan vulnerabilities. Figure 2 shows how the percentage of fixed and unfixed vulnerabilities changes with the growth of each of the following metrics: number of commits, number of active months, number of community size, number of core members, number of forks, and number of GitHub stars. There is an overall trend in the percentage of unfixed vulnerabilities decreasing when any of the project metrics is growing, which might indicate that larger and more frequently updated projects are more likely to fix a copied vulnerability. It is important to note that for large projects there is still a significant number of copied vulnerabilities that are not fixed.

The strongest trends are found with number of active months and number of stars, indicating that long lived projects and very popular projects are more likely to fix copied vulnerabilities.

We checked if vulnerable copies that were copied through a package manager were more likely to be fixed. Surprisingly, we found no consistent clear trend over different programming languages.

Lastly, we analyzed how the percentage of fixed orphan vulnerabilities changes for large projects. We analyzed a subset of projects having at least 100 GitHub stars. The GitHub star rating was chosen to mitigate issues arising from other metrics (such as commit count) being transferred to a project through forking. The number of GitHub stars is not a perfect metric, but is beneficial for our purposes [6]. Table 7 shows the percentage of copied vulnerabilities that are fixed, not fixed, and with status unknown for the top 10 file suffixes. For some file suffixes, the majority of orphan vulnerabilities are fixed in large projects. We found, however, that a large percentage of orphan vulnerabilities are not fixed even in large and popular projects.



(a) number of commits (b) number of active months

(c) number of community size (d) number of core members

(e) number of forks (f) number of stars

Figure 2: Percentage copied vulnerabilities with status fixed, not fixed, and unknown for growing (a) number of commits, (b) number of active months, (c) community size, (d) number of core members, (e) number of forks, and (f) number of stars.

Table 7: 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

7.5 RQ5: Survival of orphan vulnerabilities

We analyzed how long it takes for an orphan vulnerability to be fixed. We calculated 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.

Looking at all projects, we found that 15.6% of copied vulnerabilities had a negative time delta, meaning that the orphan vulnerability was fixed before it was fixed in the original project. 84.3% of the copied vulnerabilities had a positive time delta.

The mean number of days required to fix an orphan vulnerability was 459. Orphan vulnerabilities were fixed in 0 to 1 days in 15% of the cases, indicating that the fix might have been introduced through automated updates. While half of the orphan vulnerabilities were fixed in less than 80 days, 25% of orphan vulnerabilities remained in the repository over 560 days. We checked if copied vulnerabilities that were likely included through package managers were more likely to be fixed very quickly (in less than a day) aorthern Kentuckynd found the opposite to be true.

Additionally, we analyzed how long copied vulnerabilities remained in larger projects, expecting vulnerabilities to be fixed sooner in more popular and active projects. In projects with at least 100 GitHub stars, 75% of the orphan vulnerabilities remained in the project for longer than 426 days. For half of the orphan vulnerabilities that we found, the orphan vulnerability remained in the project over three years.

8 DISCUSSION

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 6 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 file 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 7.1 addresses the prevalence of package manager files that are committed to repositories.

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.

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.

9 THREATS TO VALIDITY

We rely on data from the CVEfixes[5] dataset being correct. As this dataset is extracted from the NVD, we indirectly rely on NVD data about fixing commits being correct. Nguyen et al. [30] demonstrated some errors in vulnerability reporting in NVD.

Since we are looking to see if a project fixes a vulnerability and how long it takes, we only look at the small fraction of vulnerabilities where the fixing commit can be automatically identified. Furthermore, we only consider vulnerabilities where the patch commit only changed a single file. As a result, our count of orphan vulnerabilities is an undercount, based on 3,615 initial vulnerabilities out of the more than 200,000 vulnerabilities found in NVD.

World of Code provides a nearly complete collection of publicly available open source software. We rely on World of Code to find open source repositories. We will miss any projects that are not available in World of Code. This will also lead to undercounting orphan vulnerabilities.

Our tool looks at file-level copy-based code reuse of vulnerable files using hash-based matching of files in order to scale to the entire World of Code, which includes nearly all open-source repositories. We are not aware of any algorithms or tools that could detect code clones at the method or line level that would scale to that level. It is important to note that our tool does not only look at one version of the vulnerable file, but also finds copies for previous versions of the vulnerable file. While our current tool will only find file-level copies, we still find an alarming number of projects that have copied public vulnerabilities in a way that will not be detected by dependency tracking tools. We understand that this number of projects is a minimum bound on the number of projects that have copied vulnerable code.

We created a subset of projects with over 100 GitHub stars so that we could find active projects and eliminate many useless projects. GitHub stars is not a perfect measure, but is useful in many cases [6]. We found that excluding projects with fewer than 100 commits would only reduce the number of projects by 0.013% from what we get when only excluding projects with less than 100 stars.

We only check if the project contains a vulnerable file, not if the project is vulnerable. 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.

If we know when a vulnerability was introduced, VCAnalyzer has the ability to look at only revisions between the introduction and fix of the vulnerability. Since CVEfixes does not give us the introduction date, we look at all prior revisions since the prior revisions are likely to also contain the vulnerability. This gives us all revisions before the fix. Revisions before the fix may include revisions before the vulnerability was introduced. In these cases, we count projects that are not vulnerable (although they contain a revision before the fix), causing us to overestimate vulnerable projects. Tools like SZZ unleashed [5] can be used to find some but not all vulnerability introduction dates, but do not scale to the size of the WoC.

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. [23]. Our VCAnalyzer tool uses World of Code's commit to deforked project (c2P) mapping. This mapping uses the community detection algorithm described by Mockus et al. [28] 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.

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

10 CONCLUSION

In this paper, we 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.

Copy-based reuse of vulnerable code is widespread in open source software. 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.

The time from introduction to remediation of orphan vulnerabilities was long, averaging 459 days. orphan vulnerabilities survived a long time even in active projects, with half of orphan vulnerabilities requiring more than three years to be fixed. However, it is worth noting that a substantial minority (15%) of orphan vulnerabilities were repaired in under a day.

ACKNOWLEDGMENTS

This work was partially supported by NSF awards 1633437, 1901102, 1925615, and 2120429, the Austrian ministries BMVIT and BMDW, the Province of Upper Austria in frame of the Software Competence Center Hagenberg (SCCH), and grant PRG1226 of the Estonian Research Council.

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 446–457.

[2] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. 2016. SV-AF — A Security Vulnerability Analysis Framework. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 219–229. https://doi.org/10.1109/ISSRE.2016.12

[3] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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*. 30–39.

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

[5] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. 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* (Tallinn, Estonia) *(MaLTeSQuE 2019)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3340482.3342742

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

[7] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[9] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software heritage: Why and how to preserve software source code. In *iPRES 2017-14th International Conference on Digital Preservation*. 1–10.

[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. 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*. 2169–2185.

[11] Johannes Düsing and Ben Hermann. 2021. Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories. *Digital Threats: Research and Practice* (2021).

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

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

[14] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 291–301.

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

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

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

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

[19] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue. 2014. Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 305–314.

[20] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[21] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[22] Zhen Liu, Qiang Wei, and Yan Cao. 2017. VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint. In *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*. 548–553. https://doi.org/10.1109/ITOEC.2017.8122356

[23] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[24] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*. IEEE, 143–154.

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

[26] André Miranda and João Pimentel. 2018. On the use of package managers by the C++ open-source community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1483–1491.

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

[28] Audris Mockus, Diomidis Spinellis, Zoe Kotti, and Gabriel John Dusing. 2020. A complete set of related git repositories identified via community detection approaches based on shared commits. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 513–517.

[29] Frank Nagle. 2019. Open source software and firm productivity. *Management Science* 65, 3 (2019), 1191–1215.

[30] Viet Hung Nguyen and Fabio Massacci. 2013. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 493–498.

[31] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File cloning in open source java projects: The good, the bad, and the ugly. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 283–292.

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

[33] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1513–1531.

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

[35] David Reid, Kalvin Eng, Chris Bogart, and Adam Tutko. 2021. Tracing Vulnerable Code Lineage. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 621–623.

[36] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. *International Conference on Software Engineering* (2022), 2104–2115. https://doi.org/10.1145/3510003.3510216

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

[38] Niko Schwarz, Mircea Lungu, and Romain Robbes. 2012. On how often code is cloned across repositories. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1289–1292.

[39] Sonatype. 2022. Open Source Security and Risk Analysis Report. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf

[40] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards Understanding Third-party Library Dependency in C/C++ Ecosystem. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[41] James Wetter and Nicky Ringland. 2021. Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html

[42] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. {V0Finder}: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *30th USENIX Security Symposium*. 3041–3058.

[43] Pei Xia, Yuki Manabe, Norihiro Yoshida, and Katsuro Inoue. 2012. Development of a Code Clone Search Tool for Open Source Repositories. *Information and Media Technologies* 7, 4 (2012), 1370–1376. https://doi.org/10.11185/imt.7.1370

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

[45] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.

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

Received 2023-07-07; accepted 2023-07-28