Noname manuscript No.
(will be inserted by the editor)

Out of Sight, Out of Mind? How Vulnerable Dependencies Affect Open-Source Projects

Gede Artha Azriadi Prana · Abhishek Sharma · Lwin Khin Shar · Darius Foo · Andrew E. Santosa · Asankhaya Sharma · David Lo

Received: date / Accepted: date

Abstract

Context: Software developers often use open-source libraries in their project to improve development speed. However, such libraries may contain security vulnerabilities, and this has resulted in several high-profile incidents in recent years. As usage of open-source libraries grows, understanding of these dependency vulnerabilities becomes increasingly important.

Objective: In this work, we analyze vulnerabilities in open-source libraries used by 450 software projects written in Java, Python, and Ruby. Our goal is to examine types, distribution, severity, and persistence of the vulnerabilities, along with relationships between their prevalence and project as well as commit attributes.

Method: Our data is obtained by scanning versions of the sample projects after each commit made between November 1, 2017 and October 31, 2018 using an industry-grade software composition analysis tool, which provides information such as library names and versions, dependency types (direct or transitive), and known vulnerabilities.

Results: Among other findings, we found that project activity level, popularity, and developer experience do not translate into better or worse handling of dependency vulnerabilities. We also found "Denial of Service" and "Information Disclosure" types of vulnerabilities being common across the languages studied. Further, we found that most dependency vulnerabilities persist

Gede Artha Azriadi Prana · Abhishek Sharma · Lwin Khin Shar · David Lo
Singapore Management University
E-mail: arthaprana.2016@phdis.smu.edu.sg, abhisheksh.2014@phdis.smu.edu.sg, lkshar@smu.edu.sg, davidlo@smu.edu.sg

Darius Foo · Andrew E. Santosa · Asankhaya Sharma
Veracode
E-mail: dfoo@veracode.com, asantosa@veracode.com, asharma@veracode.com

throughout the observation period, and the resolved ones take 4-5 months to fix.
Conclusion: Our results highlight the importance of managing the number of dependencies and performing timely updates, and indicate some areas that can be prioritized to improve security in wide range of projects.

Keywords Empirical Study · Security · Software Composition Analysis

1 Introduction

Modern software is typically built using a large amount of third-party code in the form of external libraries to save development time. Such third-party components are often used as is [28], and even for trivial functions, developers often choose to use an external library instead of writing their own code [1]. Centralized repositories (such as Maven Central and PyPI) and their associated dependency management tools make it easy for software developers to download and include open-source libraries in their projects, further improving the developers' productivity.

However, such third-party libraries may contain varying amount of security vulnerabilities. While developers may get their own code reviewed by peers or checked for bugs or security issues by using static analysis tools [23], Kula et al. found that developers often do not review the security of third-party libraries, citing it as extra effort [25]. Since a software project may depend on a number of open-source libraries, which may in turn depend on many other libraries in a complex package dependency network, analysis on a software project's entire dependency tree can become very complex. Unchecked project dependencies may introduce security vulnerabilities into the resulting software, which may be hard to detect. An example of this is the buffer overread in OpenSSL library that resulted in Heartbleed vulnerability [12], which was introduced in 2012 but remained undetected until 2014. Another high-profile example is the unpatched CVE-2017-5638 vulnerability in Apache Struts that resulted in the 2017 Equifax data breach. More recently, CVE-2018-1000006 vulnerability that was discovered in the popular Electron framework in January 2018 affected a number of Windows applications built using the framework, such as Skype and Slack.

As the usage of open-source libraries grows, it becomes increasingly important to understand the risks associated with vulnerabilities in the libraries. This motivates us to investigate the prevalence of vulnerabilities in open-source libraries, the types and persistence of the vulnerabilities, along with relationships between their prevalence and project as well as commit attributes. Such an investigation can answer several open questions, for example: *What are common types of dependency vulnerabilities library users should be aware of? Is it sufficient for a library developer to fix vulnerabilities in their library and release a new version as soon as possible? How often are vulnerable dependencies left unchanged due to actual lack of newer versions? How effectively can we reduce risk due to vulnerable dependencies by adding more personnel to the* *project?* We believe that the result of such an investigation would be helpful to library users, library developers, and security researchers in a number of ways. Library developers can benefit from understanding the prevalence of persistent vulnerabilities as well as the prevalence of outdated dependencies, as they can signify the need to encourage updates and make updates easier. Library users can benefit from understanding common types of vulnerabilities since such knowledge can help them to anticipate and guard against these common vulnerabilities. This is important as vulnerabilities in libraries may not become publicly known immediately and there may also be latency before library developers provide a fix. They can also benefit from understanding whether factors such as number and experience of contributors translate into better handling of vulnerable dependencies, since this can affect personnel allocation decisions, among others. In addition, the result of such investigation can also help researchers to identify directions of research that are more likely to benefit the widest range of software projects.

There are several open-source tools such as *OWASP Dependency Check*[1], *Bundler-audit*[2], and *RetireJS*[3] that can assist development teams to check for publicly-known security vulnerabilities in their open-source dependencies. Since November 2017, GitHub has also provided a service[4] that scans dependencies of a given project in several supported languages for publicly known vulnerabilities. Beyond this, several vendors, such as *Sonatype*, *Synopsys*, *Veracode*, and *WhiteSource*, also offer software composition analysis (SCA) tools that can identify open-source libraries used in a given software project, vulnerabilities associated with those libraries (including those not yet in public vulnerability databases), associated licenses, and other metrics. Such SCA tools enable development teams to identify vulnerable dependencies and other potential issues such as outdated dependencies and license issues.

In this work, we use *SourceClear* from Veracode to perform an empirical study on a sample of projects and their associated commits on GitHub. We use SourceClear as it is available to us, includes a database of open-source libraries maintained by Veracode security researchers along with categorized list of associated vulnerabilities (as well as their severity scores), and supports the three languages investigated in this study (Java, Python, and Ruby). This enables systematic investigation and comparison of detected vulnerabilities in sampled projects' open-source dependencies. For our dataset, we sampled 450 software projects on GitHub that are written in Java, Python, and Ruby and have at least 5 commits during the 1-year period between November 1, 2017 to October 31, 2018. Being larger and more diverse than datasets of earlier works on vulnerability dependency usage [5, 27, 41]), our dataset enables better generalizability of analysis results. We subsequently checked out all commits made to the sampled projects during the 1-year period, and used *SourceClear* to scan

[1] https://www.owasp.org/index.php/OWASP_Dependency_Check
[2] https://github.com/rubysec/bundler-audit
[3] http://retirejs.github.io/retire.js/
[4] https://help.github.com/en/articles/about-security-alerts-for-vulnerable-dependencies

the complete project version after each commit. Afterwards, we analyzed the scan results, which include vulnerability details such as CVE identifier, type, and severity. We examined a variety of aspects related to characteristics of the discovered vulnerabilities in the sampled projects' open-source dependencies, including common types, frequency, persistence, as well as the relationship between the vulnerabilities with project as well as commit attributes. In summary, we intend to answer the following research questions:

- ***RQ1****: What are the common types and prevalence of dependency vulnerabilities in open-source software, and how persistent are they?*
  Understanding common types, prevalence, and persistence of dependency vulnerabilities would help us assess the severity of the problem as well as shed light on ways to resolve or mitigate the vulnerabilities. This serves as our motivation to answer this research question. Among others, we found that such vulnerabilities are persistent and take months to fix. We also found common vulnerability types across languages.
- ***RQ2****: What are the relationships between vulnerabilities in a project's open source dependencies with the attributes of the project and its commits?*
  Many open-source developers and users hold the view that more reviewers result in improved software quality. This view is phrased as "many eyes make all bugs shallow" by Eric Raymond and is known as Linus' Law [45] and has been investigated in several studies (e.g. [30, 31]). The argument is that larger size of community working on and using a particular software will make it more likely for any quality issues to be discovered and fixed. We are interested in examining whether this view holds true for dependency vulnerability count. Further, there has also been various works in vulnerability prediction (e.g. [37, 38, 61, 50, 58, 20]) that utilizes different types of metrics such as complexity, churn, and developer activity to predict vulnerability in the project's own code. Given this, we believe that it is worthwhile to examine possible correlations between some of the metrics with vulnerabilities resulting from the projects' open-source dependencies, in addition to comparing the correlation between vulnerabilities and the counts of project's direct and transitive dependencies. Direct dependencies refer to dependencies that are referenced by the project's code directly, while transitive dependencies refer to libraries that are referred to by other dependencies. We found that the vulnerability counts correlate more strongly with the project's total dependency counts compared to the project activity level, popularity, scale of commit, and experience level of the developer making a commit. This suggests, for example, that reducing the total number of dependencies (which may lack tests and have many dependencies on their own [1]) will be more effective in mitigating such vulnerabilities than recruiting additional developers into the project.

There have been several works focusing on the usage of vulnerable dependencies [5, 27, 41] as well as works that include discussion of vulnerable dependencies in context of library migration [10, 8, 25]. We expand on the earlier set of works by analyzing larger and more diverse set of software projects.

In addition, the vulnerability details in the database of the *SourceClear* tool that we use enables investigation into some aspects not covered in the above works, such as prevalence of different vulnerability types. Finally, we perform a scan on each commit made to the sampled projects within the 1-year observation period, enabling analyses related to persistence of the vulnerabilities and the correlation between vulnerabilities with commit attributes.

The paper is structured as follows: Section 2 presents an overview of *SourceClear* tool used in our study. Section 3 discusses the dataset collection method, overview of the dataset, and our methodology. Section 4 presents the results of our empirical study. Section 5 discusses the implications of our findings to library users, developers, as well as researchers. Section 6 discusses threats to validity of our study. Section 7 discusses works related to our study. Section 8 concludes our work and presents future directions.

2 Overview of *SourceClear*

SourceClear[5] is a software composition analysis (SCA) tool from Veracode. SCA tools are typically used by developers and organizations to identify open-source components used by their software projects as well as various information associated with those components, including their respective licenses, known vulnerabilities, and latest available versions. Such tools help their users to prevent or mitigate security and legal issues, in addition to providing better visibility into their software projects.

SourceClear supports analysis in several languages (Java, Python, .NET, Ruby, JavaScript, PHP, Scala, Objective C, and Go), and works as follows: given a project code base, if necessary, it builds the project with the build system used by the project (e.g. Maven) and generates dependency graph from the result. It subsequently analyzes the graph to identify the open-source libraries used in the project. Afterwards, it matches the identified open-source libraries and their specific versions against a database containing information of open-source libraries obtained from variety of sources (e.g. Maven Central, Ruby Gems, public sources of vulnerability information, as well as in-house research efforts). Based on this, *SourceClear* subsequently reports open-source libraries used by the project, the specific versions of the detected libraries, their licenses, associated vulnerabilities, as well as usage of outdated libraries, as shown in Figure 1. *SourceClear* includes static checking mechanism to aid library updates [15] as well as Security Graph Language [14], a domain-specific language that is designed to describe and represent vulnerabilities. The language supports efficient queries involving relations between open-source libraries, their file contents (such as methods and classes), and vulnerabilities.

SourceClear is also able to detect publicly-known vulnerabilities in Common Vulnerabilties and Exposures (CVE) list[6] in addition to a number of vulnerabilities that have not yet been assigned CVE identifiers. As of 27 June

[5] https://www.sourceclear.com

[6] https://cve.mitre.org/

2019, the SourceClear vulnerability database[7] contains 2,027,092 libraries from all supported languages (not counting different versions), and 11,364 distinct vulnerabilities. The library information are retrieved from various open source package repositories. For the languages used in this work (Java, Python, and Ruby), the *SourceClear* vulnerability database statistics are shown in Table 1.

Table 1 *SourceClear* vulnerability database information for languages used in this work. Note: Distinct vulnerability corresponds to a CVE for publicly-known vulnerabilities, or SourceClear artifact ID for non-publicly-known vulnerabilities.

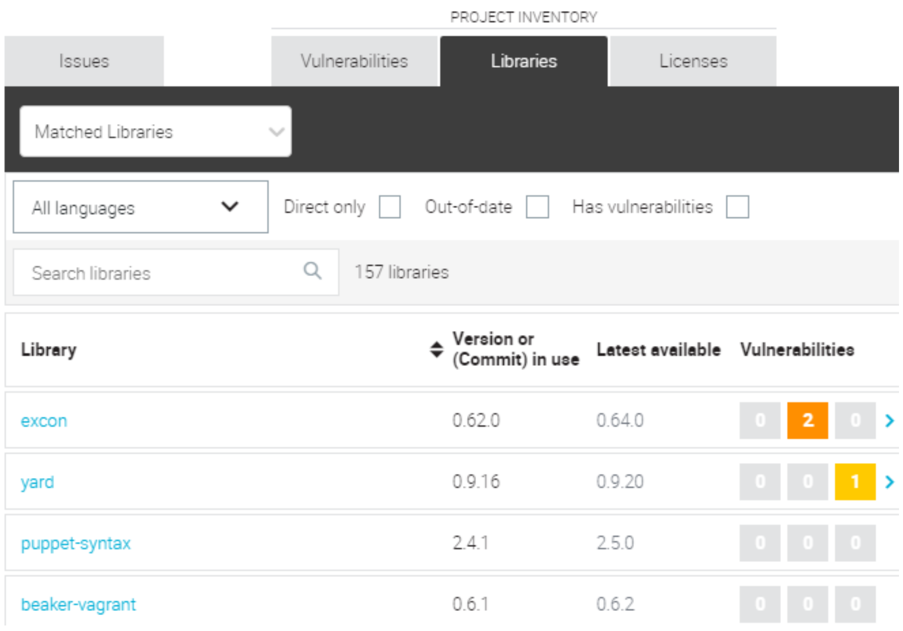
Language	Libraries	Distinct Vulnerability	Source of library information
Java	240,015	1,484	*search.maven.org*, *repo1.maven.org* (for Maven)
Python	178,633	788	*pypi.python.org*
Ruby	138,082	648	*rubygems.org*

As source of vulnerability data, *SourceClear* makes use of both publicly-known vulnerability information from National Vulnerability Database[8], as well as in-house research efforts to discover vulnerabilities that are not yet publicly known. Figure 2 provides high-level overview of the workflow. Identification of new vulnerabilities for inclusion into the database is achieved by Veracode security researchers through a variety of approaches, such as application of natural language processing and machine learning model to identify vulnerability-related commits and bug reports. The machine learning model [59] achieved precision of 0.83 and recall of 0.74 during validation at *SourceClear* production system in March 2017 - May 2017 and was able to detect more actual vulnerabilities than the number reported in CVE in the same period (349 vs 333). It outperformed the SVM-based classifier [42], one of the state-of-the-art approaches for vulnerability detection from commit messages as well as from bug reports. For instance, it achieved 54.55% higher precision with the same recall for commit messages. Beyond this, prior to publishing into the SourceClear database, each vulnerability is reviewed and researched by at least two Veracode security analysts. In addition to this, Veracode also keeps track of customer feedback regarding the vulnerability database. These factors support our confidence in the tool's detection capability. These factors support our confidence in the tool's detection capability.

By default, as part of its scan result, *SourceClear* reports Common Vulnerability Scoring System security score of vulnerabilities along with the following corresponding rating:

- 0.1 - 3.9 : Low
- 4.0 - 6.9 : Medium
- 7.0 - 8.9 : High
- 9.0 - 10.0 : Critical

[7] https://www.sourceclear.com/vulnerability-database
[8] https://nvd.nist.gov/



157 libraries

Library	Version or (Commit) in use	Latest available	Vulnerabilities		
excon	0.62.0	0.64.0	0	2	0
yard	0.9.16	0.9.20	0	0	1
puppet-syntax	2.4.1	2.5.0	0	0	0
beaker-vagrant	0.6.1	0.6.2	0	0	0

Fig. 1 Part of result of a *SourceClear* scan

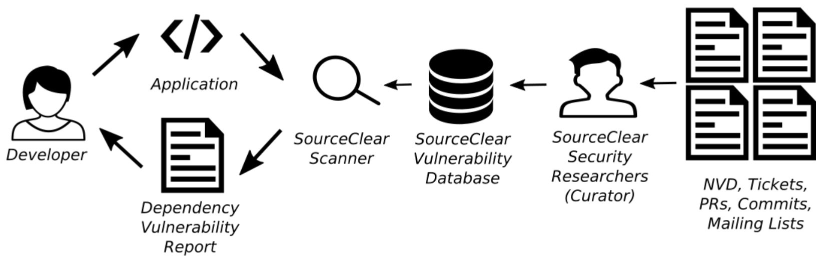

Developer
Application
Dependency Vulnerability Report
SourceClear Scanner
SourceClear Vulnerability Database
SourceClear Security Researchers (Curator)
NVD, Tickets, PRs, Commits, Mailing Lists

Fig. 2 Overview of *SourceClear* workflow

Each detected vulnerability is also associated with at least one tag (such as “Authentication” or “Cross-site Scripting (XSS)”). The complete list of tags used in *SourceClear* database is shown in Table 2.

Overall, *SourceClear*’s scan features and details in its scan results facilitate our analysis for characterizing the vulnerabilities in the open-source dependencies of the projects that we sampled. As our study involves multiple languages (Java, Python, and Ruby), *SourceClear*’s language support also put it in an advantage compared to popular open-source alternatives such as *Bundler-audit* (which only supports Ruby Gems) or *OWASP Dependency Check* (which supports Java but has only experimental support for Ruby and Python).

Table 2 List of vulnerability tags used by *SourceClear*

Authentication	Mass-assignment
Authorization	OS Command Injection
Buffer Overflows	Phishing attack
Business Logic Flaws	Race Conditions
Configuration	Remote DOS
Cross Site Scripting (XSS)	Remote Procedure Calls
Cryptography	Session Management
Data at Rest	Source code disclosure
Denial of Service	SQL Injection
EL execution	Transport Security
File I/O	Trojan Horse
Information Disclosure	XML Injection
Injection Vulnerabilities	XPath Injection
Man-in-the-middle	Other

3 Dataset & Methodology

3.1 Dataset Collection

We use GitHub as the source of software projects for this study. Since many GitHub repositories do not actually contain software projects [21], as starting point we used the *reaper* dataset from Munaiah et al. [36] which provides a list of repositories likely to contain software projects. We believe the benefit of performing sampling on this pre-filtered list of repositories outweighs the potential downside of missing newer repositories, and at the time the data collection began (December 2018) we were not aware of newer dataset of similar type. We set the following criteria for sampling the projects:

1. The project is written in Java, Python, or Ruby, based on information from the *reaper* dataset.
2. The project repository commit log lists at least 5 commits between November 1, 2017 and October 31, 2018.
3. The project satisfies the prerequisites to be scanned by the *SourceClear* tool, i.e. its content indicates that it uses a build tool supported by *SourceClear*, and it is actually buildable. For Java projects, we focused on Maven projects to reduce the potential complexity of troubleshooting build issues. For Python projects, we look for the existence of one of the following files in the project's root directory: *setup.py*, *requirements.txt*, *requirements-dev.txt*, or *dev-requirements.txt*. For Ruby projects, we look for the existence of *Gemfile* in the project root directory.

The criteria are set to ensure that the resulting set of the sample projects comprises actively-maintained software projects written in popular languages, which should subsequently improve generalizability of our analysis results. In addition, the choice of selecting projects from multiple programming languages instead of collecting a larger set from a single language is meant to enable investigation into potential differences in characteristics of vulnerabilities in different languages.

After filtering for projects that match the criteria, we randomly sampled 450 software projects. This corresponds to 150 for each programming language (out of 462,182 Java projects, 331,883 Python projects, and 363,801 Ruby projects on *reporeaper*). Afterwards, we extracted the list of all commits made between November 1, 2017 and October 31, 2018. We subsequently scan the projects using *SourceClear* to identify its open-source dependencies as well as the type of each dependency (i.e. direct, transitive, or both). The statistics of the sampled projects are shown in Table 3. Table 3 shows that the number of transitive dependencies of the sampled projects are generally much higher than that of direct dependencies, consistent with observation of Decan et al. [9].

In addition to this, Figure 3 shows the relationship between commit author count and direct dependency count of the sample projects, while Figure 4 shows the relationship between commit author counts and transitive dependency counts. Table 4 shows the correlation (computed using Spearman's rank correlation test [52]) between commit author count and commit count, as well as between commit author count and dependency counts. Following scale of interpretation of ρ used by Camilo et al. [6] (± 0.00 - 0.30: Negligible, ± 0.30 - 0.50: Low, ± 0.50 - 0.70: Moderate, ± 0.70 - 0.90: High, and ± 0.90 - 1.00: Very high), we note that there is low to moderate correlation between commit author count and commit count, but no statistically significant correlation between commit author count and dependency count.

Table 3 Statistics of the sampled projects at latest commit in the observation period.

Metric		Java	Python	Ruby
Commits in target period	Min	5	5	5
	Max	4579	2471	1802
	Median	23.0	20.5	16.0
	Mean	104.6	76.4	56.9
	Std. dev.	371.4	196.7	165.9
Commit authors in target period	Min	1	1	1
	Max	22	51	43
	Median	2	3	2
	Mean	3.9	4.7	4.1
	Std. dev.	4.4	6.3	5.7
Direct open-source software (OSS) dependency	Min	0	0	0
	Max	81	29	99
	Mean	8.8	3.5	15.7
	Median	5	2	8
	Std. dev.	12.0	4.5	20.4
Transitive OSS dependency	Min	0	0	0
	Max	254	191	280
	Mean	29.6	7.0	53.2
	Median	9.5	1	43
	Std. dev.	44.1	18.2	49.0
Projects with no detected OSS dependency		16	35	2

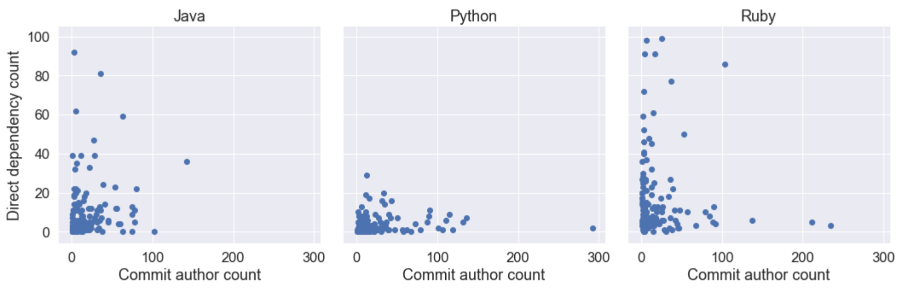


Fig. 3 Relationship between sample projects' commit author count and direct dependency count

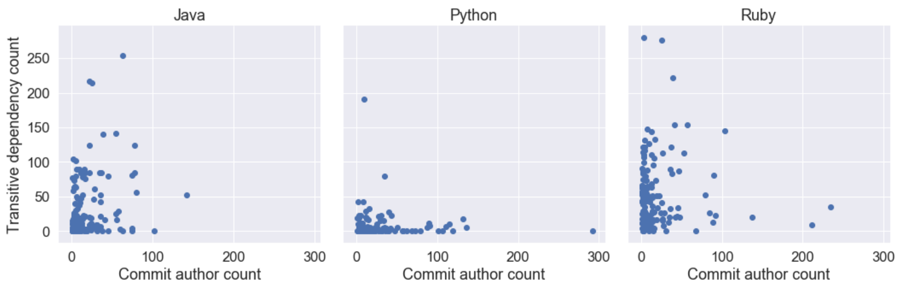


Fig. 4 Relationship between sample projects' commit author count and transitive dependency count

Table 4 Correlation between commit author count and commit count, as well as between commit author count and dependency counts

Language	Number of Commits		Dependency Count			
			Direct		Transitive	
	ρ	p	ρ	p	ρ	p
Java	0.484	0.000	0.072	0.378	0.062	0.449
Python	0.587	0.000	0.080	0.333	-0.054	0.512
Ruby	0.367	0.000	-0.075	0.364	-0.047	0.565

3.2 Methodology

After selecting the GitHub projects and downloading their commit history, we checked out each commit and performed a scan on the full project versions after each commit using *SourceClear*. The tool reports total counts of direct and transitive open-source dependencies in the specific project version scanned, list of detected vulnerabilities (including description, severity score, and specific libraries that contain them), as well as other information such as license information of the libraries. We subsequently use the information on vulnerable open-source dependencies and their associated vulnerabilities for subsequent analyses.

For the purposes of counting distinct vulnerabilities, there are two cases to be considered: The first case is the vulnerabilities that have been assigned

Common Vulnerabilities and Exposures (CVE) identifier. The CVE identifier points to a specific publicly-known vulnerability in the CVE list. The other case relates to vulnerabilities that have not yet been assigned CVE identifier after their discovery by Veracode security researchers. Since the *SourceClear* vulnerability database assigns one artifact ID for each distinct vulnerability (with or without CVE), we use this artifact ID instead of CVE identifier. For subsequent analyses, we count a combination of software project, library version, and artifact ID as individual vulnerability instance.

In this work, we use “first commit” or “earliest commit” as a shorthand for first commit in the observation period (i.e. first commit in November 2017). Similarly, “last commit” or “latest commit” refers to latest commit in the observation period (i.e. latest commit in October 2018).

4 Empirical Study Results

In this section we discuss the results of our investigation into the characteristics of vulnerabilities in the sampled projects' open-source dependencies, as well as the vulnerabilities' relationship with project and commit attributes.

4.1 RQ1: What are the characteristics of dependency vulnerabilities in open-source software?

4.1.1 Dependency vulnerability counts

Table 5 shows the distribution of the total counts of detected vulnerabilities in open-source dependencies of the sampled projects. The data shown is based on scan result at the time of the latest commit in the observed period, and is split into data on vulnerabilities with CVE (i.e. publicly-known vulnerabilities, including those for which CVE ID has been reserved at the time of scan) and vulnerabilities without CVE (i.e. non-publicly-known vulnerabilities). It shows that the Java sample set has the largest overall range and variation of vulnerability counts, followed by the Ruby sample set.

Table 5 Overview of sample projects' vulnerability counts

Vulnerabilities with CVE					
Language	Min	Max	Mean	Median	Std.dev.
Java	0	98	9.0	1	15.6
Python	0	30	0.9	0	3.5
Ruby	0	42	4.4	1	7.2
Non-CVE Vulnerabilities					
Language	Min	Max	Mean	Median	Std.dev.
Java	0	19	1.9	0	3.3
Python	0	6	0.1	0	0.5
Ruby	0	31	3.0	1	5.2

In addition to the total counts, we examine the breakdown of the vulnerabilities by dependency type. Specifically, we are interested in finding the average percentages of vulnerabilities that are associated with a project's direct dependencies, transitive dependencies, and dependencies that are used both directly and transitively. We perform this analysis at the latest commit of each project for which at least one dependency vulnerability is found. This gives us the most up-to-date information of the projects' vulnerability characteristics. The breakdown of vulnerability by dependency type is shown in Table 6. t shows that distribution of vulnerability counts by dependency type corresponds to relative proportion of dependency types. For Python projects, most of the dependency vulnerabilities are in the projects' direct dependencies, which are more visible to the project developers and more easily updated compared to transitive dependencies. On the other hand, given the higher percentage of vulnerabilities in Java and Ruby projects' transitive dependencies, developers using those languages will benefit more from careful scrutiny of their projects' transitive dependencies.

Table 6 Per-project vulnerability percentage distribution by dependency type. 'Both' denotes dependencies that are used both directly and transitively. Includes only projects with at least one vulnerability. Percentages of dependency by type ('Dep.') included for comparison.

		Java			Python			Ruby		
		CVE	Non-CVE	Dep.	CVE	Non-CVE	Dep.	CVE	Non-CVE	Dep.
Direct	Min	0.0	0.0	0.0	0.0	0.0	15.05	0.0	0.0	1.2
	Max	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	41.9
	Mean	30.3	36.7	29.2	85.0	50.0	51.4	10.0	16.3	18.2
	Median	4.0	19.5	15.7	100.0	50.0	44.4	0.0	0.0	19.6
	Std.dev	40.6	42.3	26.9	33.7	54.8	27.3	24.3	26.3	10.3
Transitive	Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	54.6
	Max	100.0	100.0	100.0	100.0	100.0	85.0	100.0	100.0	98.8
	Mean	58.6	60.3	64.4	15.0	50.0	48.6	84.5	73.4	77.9
	Median	76.3	69.7	69.2	0.0	50.0	55.6	100.0	82.6	76.5
	Std.dev	42.8	41.6	23.4	33.7	54.8	27.3	27.1	31.8	11.7
Both	Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max	100.0	75.0	100.0	0.0	0.0	0.0	100.0	100.0	23.5
	Mean	11.1	3.1	6.5	0.0	0.0	0.0	5.5	10.3	3.9
	Median	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.7
	Std.dev	27.3	11.9	14.6	0.0	0.0	0.0	15.0	20.3	3.6

Finding 1: Proportion of vulnerability counts by dependency type varies by programming language, corresponding to relative proportion of dependency types

4.1.2 Most common dependency vulnerability types

To identify common types of dependency vulnerabilities in each language, we combine scan results of latest commits of each language's sample set and count all the detected vulnerabilities. For this analysis, we count each combination of library version, vulnerability, and project separately. That is, if a particular library version with 2 vulnerabilities of "Denial of Service" type is used by 2 projects, this is counted as 4 instances of "Denial of Service". If a project uses 3 libraries containing one "Denial of Service" type of vulnerability each, this is counted as 3 instances of "Denial of Service" vulnerability. Table 7 shows the total instances.

Table 7 Summarized count of dependency vulnerability instances at latest commit. Non-CVE vulnerabilities are identified by *SourceClear* artifact ID

		Java	Python	Ruby
Total vulnerability instances	CVE	1354	131	657
	Non-CVE	282	12	455
Distinct vulnerabilities (CVE / artifact ID)	CVE	212	51	80
	Non-CVE	56	9	74

Afterwards, we identify the tags associated with each vulnerability instance, count the total for each tag, and shortlist the ones with highest counts. Table 8 shows the top 5 results for each language. The result indicates some commonalities between the kinds of dependency vulnerabilities in each language, with "Denial of Service" and "Information Disclosure" being two common top issues across the three languages. This suggests that improvement of practices or tools to combat those vulnerability types (by both open-source library developers and security researchers) would bring significant benefits to a wide range of software projects.

Finding 2: "Denial of Service" and "Information Disclosure" are common across programming languages.

4.1.3 Distribution of severity scores

In addition to number of vulnerabilities, we are also interested in the severity of the vulnerabilities detected in the sampled projects' open-source dependencies. Table 9 shows the distribution of severity according to the default rating scale, i.e. CVSS score of 0.1 - 3.9 : Low, 4.0 - 6.9 : Medium, 7.0 - 8.9 : High, 9.0 - 10.0 : Critical. Table 10 shows the distribution of the severity score for the top vulnerability types. The distribution of severity shows that most of the vulnerabilities in the sampled projects' open-source dependencies are not critical. This is also the case for the two types of vulnerability that are common across languages ("Denial of Service" and "Information Disclosure"), which is

Table 8 Most common dependency vulnerability tags in each language. "CVE" and "non-CVE" indicate publicly-known and non-publicly-known vulnerabilities, respectively. Instance count and percentage denote count and percentage across the sample set of the programming language. Note that one vulnerability may have more than one tag.

Language	Tag	Total instances	% of all instances	CVE vuln. instances	Non-CVE vuln. instances
Java	Other	749	46.0	665	84
	Denial of Service	272	17.0	205	67
	Information Disclosure	147	9.0	125	22
	Cryptography	145	9.0	130	15
	Remote Procedure Calls	133	8.0	105	28
Python	Other	70	49.0	69	1
	Information Disclosure	24	17.0	21	3
	Configuration	23	16.0	23	0
	Denial of Service	21	15.0	17	4
	Cross Site Scripting (XSS)	15	10.0	14	1
Ruby	Denial of Service	281	25.0	97	184
	Other	280	25.0	105	175
	Cross Site Scripting (XSS)	274	25.0	182	92
	Information Disclosure	175	16.0	109	66
	SQL Injection	122	11.0	122	0

reassuring. However, there are higher percentages of high-severity vulnerabilities in dependencies of the Java and Python projects. While it's possible that variance in sample projects' code quality contributes to the difference in severity distribution, Ray et al. [44] reported that the effect size of the association between programming language and code quality is small. This implies that difference in severity distributions cannot be fully explained by potential code quality difference across the three languages. Overall, the difference suggests that Java and Python developers will benefit more from timely dependency updates.

Table 9 Distribution of severity of vulnerability instances. Percentages are of all vulnerability instances in the respective programming language group (both CVE and non-CVE).

	Severity	Java		Python		Ruby	
		%	instances	%	instances	%	instances
CVE	Low	0.8	13	1.4	2	4.5	50
	Medium	46.3	757	63.0	90	46.6	518
	High	34.9	571	23.8	34	8.0	89
	Critical	0.8	13	3.5	5	0	0
Non-CVE	Low	1.3	21	0.7	1	0.8	9
	Medium	15.2	249	6.29	9	33.9	377
	High	0.7	12	1.4	1.4	1.6	18
	Critical	0	0	0	0	4.59	51

Table 10 Distribution of severity of top vulnerability types. Percentages are of all vulnerability instances in the respective programming language group (both CVE and non-CVE).

Language	Tag	Type	Percentage of all instances			
			Low	Medium	High	Critical
Java	Other	CVE	0.4	13.6	25.8	0.8
		Non-CVE	0.2	4.5	0.4	0.0
	Denial of Service	CVE	0.0	10.5	2.0	0.0
		Non-CVE	0.4	3.5	0.2	0.0
	Information Disclosure	CVE	0.1	7.2	0.3	0.0
		Non-CVE	0.6	0.7	0.0	0.0
	Cryptography	CVE	0.0	7.5	0.5	0.0
		Non-CVE	0.1	0.9	0.0	0.0
	Remote Procedure Calls	CVE	0.0	3.4	3.1	0.0
		Non-CVE	0.0	1.7	0.0	0.0
Python	Other	CVE	0.0	23.8	21.7	2.8
		Non-CVE	0.0	0.7	0.0	0.0
	Information Disclosure	CVE	0.7	13.3	0.7	0.0
		Non-CVE	0.7	1.4	0.0	0.0
	Configuration	CVE	0.0	4.2	11.9	0.0
		Non-CVE	0.0	0.0	0.0	0.0
	Denial of Service	CVE	0.0	11.2	0.7	0.0
		Non-CVE	0.0	2.8	0.0	0.0
	Cross Site Scripting (XSS)	CVE	0.0	7.7	0.0	2.1
		Non-CVE	0.0	0.7	0.0	0.0
Ruby	Denial of Service	CVE	0.0	6.2	2.5	0.0
		Non-CVE	0.5	10.2	1.3	4.6
	Other	CVE	0.3	6.7	2.4	0.0
		Non-CVE	0.0	9.7	1.4	0.0
	Information Disclosure	CVE	0.2	7.8	1.8	0.0
		Non-CVE	0.1	5.9	0.0	0.0
	Cross Site Scripting (XSS)	CVE	3.8	12.6	0.0	0.0
		Non-CVE	0.0	8.3	0.0	0.0
	SQL Injection	CVE	0.0	9.7	1.3	0.0
		Non-CVE	0.0	0.0	0.0	0.0

Finding 3: Most detected dependency vulnerabilities are of medium severity, however, there is noticeable variation in severity distribution across programming languages.

4.1.4 Vulnerable libraries affecting largest number of sampled projects

To see whether the vulnerabilities in the sampled projects originate from a few widely-used libraries, or many libraries that affect few projects each, we investigate the number of projects affected by each vulnerable library. For this analysis we list vulnerabilities discovered at the latest commit of the sampled projects, spanning both vulnerabilities with and without CVE identifier. Afterwards, we identify the library name associated with the vulnerability, and counted the number of repositories affected by each library. For the purpose of this analysis, we do not distinguish specific library version used, and we disregard the specific number of vulnerabilities associated with the library. That is, a library with 5 detected vulnerabilities and another library with 2 detected vulnerabilities will both count as 1 if they are used by one project. The top 5 result is shown in Table 11.

We note that a vulnerable library also affects security of the entire package ecosystem through other libraries that depend on it (see e.g. Zimmerman et al.'s work on *npm* [60]). Therefore, for context, we also computed the average number of libraries impacted by a library, measured as average out-degree of the transitive closure in the dependency graph of the different package ecosystems (Maven for Java, PyPI for Python, Gem for Ruby). The counts at the time data collection begins (December 2018) is 81.9 for Java, 10.3 for PyPI, and 62.2 for Gem. This indicates that the higher number of projects affected by the top vulnerable Java and Ruby libraries in the sample set is due to inherent wider impact of average library in Maven and Gem ecosystems. This in turn indicates that improving security in these ecosystems will benefit wider range of projects.

Table 11 Top vulnerable libraries by projects affected. Project count includes projects using any version of the specified library.

Language	Library	Projects
Java	Guava: Google Core Libraries for Java	45
	Apache Commons IO	33
	Spring Web	30
	jackson-databind	30
	Apache Commons Collections	28
Python	numpy	23
	PyYAML	9
	Django	7
	requests	4
	Pillow	2
Ruby	rack	59
	nokogiri	51
	loofah	42
	activejob	41
	activerecord	30

Finding 4: Top vulnerable libraries in Java and Ruby affect relatively larger number of projects, in line with wider average impact of libraries in their package ecosystems.

4.1.5 Non-CVE dependency vulnerabilities

There are differing views regarding how soon vulnerabilities should be publicly disclosed, taking into account factors such as potential vendor and attacker responses [2]. As a result, there is often a lag between the discovery of a vulnerability by researchers and the inclusion of the vulnerability in the CVE list. Due to this lag, there is a risk that developers may miss some of vulnerabilities in their project dependencies even if they actively monitor and respond to CVE updates. We investigate the extent of such risk by evaluating the average percentage of dependency vulnerabilities in the latest commits of sampled projects that have not been assigned CVE IDs at the time of scan. Table 12 shows the average percentage breakdown of CVE and non-CVE dependency vulnerabilities, along with top tags associated with the non-CVE dependency vulnerabilities. It suggests that while most dependency vulnerabilities discovered in a project are CVE vulnerabilities, developers may still miss a significant percentage of vulnerabilities in their projects' dependencies if they rely on CVE list alone.

Table 12 Percentage and top tags for non-CVE vulnerabilities

Language	Percentage of Non-CVE Vulnerability		Top non-CVE Tags
Java	Min	0.0	
	Max	100.0	Other
	Mean	21.9	Denial of Service
	Median	18.2	Cross Site Scripting (XSS)
	Std.dev	24.1	
Python	Min	0.0	
	Max	100.0	Denial of Service
	Mean	5.0	Information Disclosure
	Median	0.0	Buffer Overflows
	Std.dev	17.3	
Ruby	Min	0.0	
	Max	100.0	Denial of Service
	Mean	41.5	Other
	Median	37.5	Cross Site Scripting (XSS)
	Std.dev	28.2	

Finding 5: Relying solely on public vulnerability database may cause developers to miss significant percentage of dependency vulnerabilities.

4.1.6 Overall persistence of dependency vulnerabilities

Persistence of dependency vulnerabilities is another aspect that we are interested in, and it is affected by two factors. One factor is how long it takes for the library developers to fix the vulnerability.A number of CVEs affect more than one version of a library, making them persistent despite library updates. An example of this is CVE-2019-17267, which affects the *jackson-databind* library from version 2.0.0 to 2.9.9.4. Another factor is how fast a vulnerable dependency is updated to non-vulnerable version (or removed altogether), since there is often latency in adopting the latest version of libraries [24, 25].

To obtain a general idea regarding the persistence of vulnerabilities across the period of interest, we compute per-project percentage of distinct CVE (or SourceClear artifact ID in case of non-CVE vulnerabilities) that exist at both the time of the earliest commit and the time of the latest commit in the observation period. The percentage of persistent vulnerabilities for each language, along with top libraries by the count of persistent CVEs/artifact IDs, are shown in Table 13.

Table 13 Per-project percentages of persistent vulnerabilities

Language		CVE	Non-CVE	Top libraries
Java	Min	0.0	0.0	jackson-databind, Data Mapper for Jackson, Spring Web, Spring Web MVC, Bouncy Castle Provider
	Max	100.0	100.0	
	Mean	78.4	86.1	
	Median	100.0	100.0	
	Std.dev	34.3	29.6	
Python	Min	0.0	0.0	Django, numpy, Pillow, PyYAML, requests
	Max	100.0	100.0	
	Mean	90.7	60.0	
	Median	100.0	100.0	
	Std.dev	27.4	51.6	
Ruby	Min	0.0	0.0	nokogiri, activerecord, loofah, rack, actionpack
	Max	100.0	100.0	
	Mean	66.4	65.2	
	Median	82.4	98.4	
	Std.dev	35.7	38.7	

For context, we also investigate the percentage of dependencies that are not updated or removed since the first commit in the observation period, with the result shown in Table 14. In addition, we also conducted survival analysis using Kaplan-Meier method [22] on the library versions found at first commit. As different projects may update their dependencies at different times, for the survival analysis we treat each combination of project and library version as one instance. The result of this analysis is shown in Figure 5. Both Table 14 and Figure 5 show that in many cases, libraries that exist at the beginning of the observation period are not changed by project owners throughout the period. To investigate whether this is due to lack of a newer version of said libraries, we examine per-project percentages of unchanged dependencies for which newer versions already exist. As shown in Table 15, in most cases the unchanged dependencies are outdated, yet not replaced. In addition to this, we also investigated vulnerabilities that persist throughout the observation period despite update of associated dependencies, with the result shown in Table 16. We find that in most of the projects, such vulnerabilities form only a small part of persistent vulnerabilities. Our findings indicate that the persistence of vulnerabilities are caused more by project owners' latency in updating dependencies instead of the vulnerabilities themselves being persistent across library versions.

Finding 6: Dependencies are not frequently updated or changed by project owners despite availability of updated libraries, and therefore any vulnerabilities contained will persist.

Table 14 Per-project percentages of dependencies in first commit that remains unchanged throughout the observation period.

	Min	Max	Mean	Median	Std.dev
Java	4.8	100.0	79.6	96.9	27.8
Python	20.0	100.0	94.2	100.0	15.7
Ruby	2.7	100.0	79.5	100.0	30.8

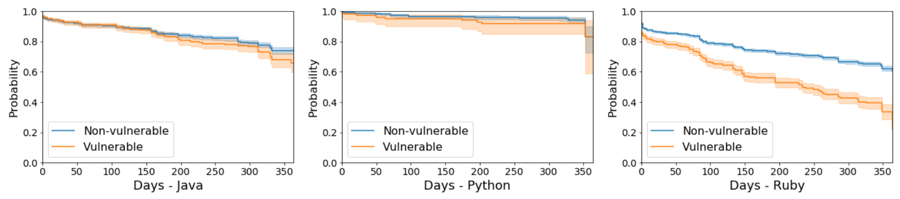


Fig. 5 Kaplan-Meier curve of vulnerable and non-vulnerable libraries detected at first commit.

Table 15 Per-project percentage of unchanged dependencies for which newer version already existed at latest commit. Percentages are of all unchanged dependencies in the same project.

	Min	Max	Mean	Median	Std.dev
Java	0.0	100.0	75.4	80.0	24.0
Python	0.0	100.0	77.6	80.0	21.0
Ruby	0.0	100.0	58.0	57.4	23.4

Table 16 Per-project percentage of vulnerabilities that persist despite update of associated dependency. Percentages shown are that of all persistent vulnerabilities in the same project.

	Min	Max	Mean	Median	Std.dev
Java	0.0	100.0	18.4	0	31.9
Python	0.0	100.0	7.4	0	25.8
Ruby	0.0	100.0	36.7	6	43.4

4.1.7 Change of number of dependency vulnerabilities over the period of observation

To examine whether the sampled projects generally become less vulnerable or more over the observation period, we computed the dependency vulnerability counts of each of the 450 projects at their first commits in the observation period (i.e. first commit in November 2017) as well as the latest commits in the period (i.e. latest commit in October 2018). We subsequently apply Wilcoxon signed-rank test on the vulnerability counts at the first and the latest commits to investigate whether they are significantly different. Since the number of dependencies of a project may also change during the same 1-year period, we also performed the same analysis on dependency counts for comparison. Table 17 shows that dependency vulnerability counts tends to decrease despite increase in the number of dependencies in the 1-year period.

Table 17 Vulnerability and dependency count changes during observation period. T denotes T statistic of Wilcoxon signed-rank test.

		Java		Python		Ruby	
		CVE	Non-CVE	CVE	Non-CVE	CVE	Non-CVE
Dependency vulnerability counts	T	162.0	25.0	12.0	0.0	0.0	36.0
	p-value	0.000	0.001	0.389	0.046	0.000	0.000
	Min at first commit	0	0	0	0	0	0
	Min at last commit	0	0	0	0	0	0
	Max at first commit	99	19	30	6	46	31
	Max at last commit	98	19	30	6	42	31
	Mean at first commit	9.8	2.1	0.9	0.1	6.6	5.1
	Mean at last commit	9.0	1.9	0.9	0.1	4.4	3.0
	Median at first commit	2.0	1.0	0.0	0.0	1.5	1.0
	Median at last commit	1.0	0.0	0.0	0.0	1.0	1.0
	Std.dev at first commit	15.8	3.5	3.6	0.6	9.9	7.7
	Std.dev at last commit	15.6	3.3	3.6	0.5	7.2	5.2
Dependency counts	T	450.0		124.0		388.0	
	p-value	0.011		0.005		0.000	
	Min at first commit	0		0		0	
	Min at last commit	0		0		0	
	Max at first commit	270		196		241	
	Max at last commit	287		196		356	
	Mean at first commit	35.8		9.6		63.2	
	Mean at last commit	36.0		10.4		66.2	
	Median at first commit	15.5		4		49.5	
	Median at last commit	17		4		50.5	
	Std.dev at first commit	47.6		19.6		54.2	
	Std.dev at last commit	47.5		20.3		61.9	

Finding 7: Dependency vulnerability counts do not increase over the 1-year study period, despite slight increase in number of dependencies.

4.1.8 Time required to resolve dependency vulnerabilities

To analyze the time to resolve dependency vulnerabilities, we listed the different dependency vulnerabilities detected in a repository during the observation period. Afterwards, we identify the commit C_1 where the dependency vulnerability is first detected in the repository during the period, as well as the commit C_2 in which the dependency vulnerability is last detected in the same repository. We subsequently identify commit C_3 which is the first commit after C_2 in the repository. For dependency vulnerabilities that already exist at first commit in the period of interest, we use the time of first commit as starting time. We exclude dependency vulnerabilities that still exist at the latest commits. We define the time to fix the dependency vulnerability (i.e. by updating the project's dependency to non-vulnerable version or removing the dependency altogether) as the difference between committer timestamp of C_3 and C_1.

We compute the figure for each repository containing the dependency vulnerability, and subsequently compute the min, max, mean, median, as well as standard deviation of the values. Table 18 show the result, broken down into vulnerabilities with and without CVE. We find that on average, the fixed vulnerabilities take 4-5 months to fix. Our finding suggests that dependency vulnerabilities not only tend to be persistent, but even the ones that are resolved take a long time to fix.

Table 18 Time taken to fix vulnerabilities in days

Vulnerabilities with CVE					
	Min	Max	Mean	Median	Std.dev
Java	0.0	361.0	145.3	126.0	125.2
Python	0.1	238.7	134.5	174.5	80.7
Ruby	0.0	364.8	98.8	81.3	99.3
Non-CVE Vulnerabilities					
	Min	Max	Mean	Median	Std.dev
Java	0.0	361.0	136.3	150.2	104.3
Python	21.6	228.2	101.3	75.2	79.0
Ruby	0.0	364.8	94.1	69.2	102.1

Finding 8: On average, resolved dependency vulnerabilities take several months to fix.

4.2 RQ2: What are the relationships between dependency vulnerabilities in a project's open-source dependencies with the attributes of the project and its commits?

4.2.1 Project attributes

A popular view regarding open-source software development is reflected in Linus' Law as formulated by Eric Raymond [45]: "Given enough eyeballs, all bugs are shallow". A larger community of developer and reviewers (official testers as well as users) is often expected to improve ability to discover bugs in a software project, including vulnerabilities. This is often used to argue that open source software is more secure [18, 56]. On the other hand, some hold the view that having too many developers can be detrimental (following the notion that "too many cooks spoil the broth"), and there have been studies by Meneely and Williams [30, 31] investigating these opposing views and the extent at which larger number of developers starts to correlate with more vulnerabilities.

Other than the two studies, a number of other works have investigated relationship between presence of vulnerabilities and various combination of metrics related to software, developer activity, and execution complexity (e.g. [61, 50, 48, 49]), typically with the overall objective of predicting location of vulnerability in a software project's source code. While the focus of our study is different, considering the view regarding Linus' Law, and another view that higher project complexity and larger project size tend to result in the project being more prone to bugs, we decide to examine whether OSS dependency vulnerability correspond to some project-level metrics: project popularity, complexity, and size. As a proxy for the project's popularity, we use number of commit authors as well as its GitHub stargazers count. As measure of project complexity, we use counts of direct and transitive dependencies of the project, since we are interested strictly in the vulnerabilities resulting from the dependencies instead of the vulnerabilities in the project's own code. Our hypothesis is that larger network of project dependencies will make it more difficult for project maintainers to track and update all dependencies to avoid vulnerable versions.

To investigate the relationship between project attributes and total count of vulnerabilities in its open-source dependencies, we constructed a negative binomial regression model [17]. We chose this regression model because it is more suitable than standard linear regression for non-negative count data [16], and it has also been used in several works in software engineering domain [3, 39, 54, 55]. For this analysis, we use the number of dependency vulnerabilities at the time of latest commit in the observation period as well as the following project attributes:

- **Age**: Project's age, measured as difference between timestamps of project's last commit in the observation period and the first commit in the project repository, in days.
- **Commits**: Total number of commits.
- **Commit authors**: Total number of distinct commit authors.
- **Repository total LOC**: Total LOC in repository excluding test code. Tests are omitted for consistency as *SourceClear* scans ignore test dependencies. Filtering is done by directory, i.e. for Java samples (which are Maven projects), we exclude *src/test/* which is the typical test location in Maven project structure. For Python and Ruby, we exclude subdirectories named *test/* and *tests/*.
- **Stargazers count**: Number of stars the repository have, as a measure of its popularity.
- **Direct dependencies**: Number of direct dependencies.
- **Transitive dependencies**: Number of transitive dependencies.

We use *statsmodels* [46] implementation of the negative binomial regression, and the results are shown in Table 19.

The results show that the project's age, number of commits, number of developers, popularity, and project size has negligible effect on the dependency vulnerability counts. This suggests that frequent commits, involvement of more developers in a project, and the project's popularity do not translate into better or worse handling of vulnerable dependencies. Possible reasons for the lack of improved handling include lack of awareness regarding the vulnerabilities as

Table 19 Negative binomial regression results on project attributes. Shaded cells indicate attributes with statistically significant contribution to dependency vulnerability count.

Variable	Java		Python		Ruby	
	coef	$P > \|z\|$	coef	$P > \|z\|$	coef	$P > \|z\|$
Age	0.000	0.218	0.000	0.596	-0.001	0.007
Commits	0.000	0.922	0.001	0.868	-0.001	0.220
Commit authors	-0.015	0.668	0.072	0.191	-0.030	0.179
LOC	0.000	0.488	0.000	0.254	0.000	0.520
Stargazers	0.000	0.962	0.000	0.079	0.000	0.170
Direct dependencies	0.013	0.109	0.145	0.002	0.031	0.000
Transitive dependencies	0.016	0.000	-0.046	0.006	-0.005	0.103

well as the presence of dependency constraints that hinder developers from updating project dependencies or switching to a different library (even if there's known vulnerability in the currently-used versions). On the other hand, most direct dependency counts have more statistically significant effects.

Overall, the results suggest that dependency vulnerabilities can likely be managed more effectively through reduction of number of direct dependencies than through recruitment of additional personnel. This reduction can for example be achieved by replacing multiple small libraries with single library that is known to have good security track record.

Finding 9: Dependency vulnerability count can be managed more effectively by reducing direct dependencies, as opposed to increasing number of contributors, activity level, or managing the project's size.

4.2.2 Commit attributes

Beside works that focus on predicting vulnerability location using various metrics, a number of works in the field of vulnerability prediction focus on identifying vulnerability-contributing commits [32, 4, 42]. Among other findings, larger changes and developer inexperience have been found to be associated with higher likelihood of a commit introducing vulnerability. Given this, we are also interested in investigating whether experience of developer making the commit or the scale of change caused by a commit correspond with number of dependency vulnerabilities detected after the particular commit. For this analysis, we consider three types of commits:

1. Commits that increase vulnerability count (e.g. due to introduction of vulnerable dependency)
2. Commits that decrease vulnerability count (e.g. due to removal or update of vulnerable dependency)
3. Commits that do not change vulnerability count

The breakdown of the three types of commits for the three programming languages is shown in Table 20. As for the commit attributes, we examine the following attributes in this analysis:

– **Developer experience**: We use the number of prior commits in the project as a proxy, since it is not possible to objectively measure and compare actual experience of commit authors directly.
– **Number of affected files**: Total count of files affected by the commit, regardless of operation type (line addition, line deletion etc.).
– **Churn**: Total number of added and deleted lines in the commit.
– **Total LOC of affected files**: Sum of number of lines of code of files affected by the commit, as a measure to distinguish commits affecting small files versus commits affecting large ones.
– **Total Complexity of affected files**: Sum of cyclomatic complexity [29] of files affected by the commit, as a measure to distinguish commits affecting simple files versus commits affecting complex ones.

With the exception of developer experience calculation, we use PyDriller [51] to obtain the metrics. We constructed a logistic regression model using *statsmodels* [46] implementation, and examined the resulting coefficients for the different attributes. The result, shown in Table 21, indicates that there is no clear relationship between dependency vulnerabilities with all attributes being examined. A possible explanation is that the dependency changes often occur together with a variety of other changes, such as addition of a large module, a small fix, or deletion of deprecated code. These changes that involve addition or removal of dependencies are diverse in size, for example, a library update from vulnerable versions to non-vulnerable version may only involve changing one line. In addition, as vulnerabilities in dependencies are less visible to developers than issues in their own code, developer experience do not necessarily translate to better handling of security risk from dependencies. In view of this, it seems discouraging large changes on each commit or assigning more experienced developers will not be an effective way to manage security risk from dependencies. It may be better, for example, for the development team to instead maintain a list of "known-good" libraries that each developer can use as they see fit.

Table 20 Counts of the three categories of commits for Java, Python, and Ruby samples

	Java	Python	Ruby
Commits that decreases vulnerability count	69	12	231
Commits that increases vulnerability count	28	8	94
Commits that causes no count change	3936	6125	8901

Finding 10: There is no clear relationship between dependency vulnerability count with attributes of the commit including author experience.

Table 21 Logistic regression results on commit attributes. Shaded cells indicate attributes with statistically significant contribution to dependency vulnerability count.

Attribute	Java		Python		Ruby	
	coef.	$P > \|z\|$	coef.	$P > \|z\|$	coef.	$P > \|z\|$
Vulnerability-increasing commits						
Affected files	0.0069	0.448	0.0291	0.590	-0.0040	0.643
Churn	0.0001	0.026	-0.0010	0.875	-3.004e-06	0.885
LOC	-8.63e-05	0.694	-0.0038	0.177	1.803e-05	0.639
Complexity	-0.0002	0.847	0.0010	0.487	0.0002	0.660
Author experience	-0.0017	0.105	-0.0309	0.123	-0.0002	0.039
Vulnerability-decreasing commits						
Affected files	-0.0051	0.723	0.0237	0.353	0.0012	0.178
Churn	0.0002	0.087	-0.0019	0.644	-2.682e-05	0.042
LOC	-2.865e-06	0.941	-0.0009	0.330	3.784e-05	0.040
Complexity	-0.0010	0.242	0.0005	0.650	-4.729e-05	0.279
Author experience	-0.0010	0.050	-0.0023	0.144	-9.916e-05	0.058

5 Discussion and Implications

Our results indicate that dependency vulnerability issue affects a wide range of projects, and that such vulnerabilities tend to be persistent, despite overall tendency for the count to decrease.

5.1 Implications for library users

Examination of the dataset shows that most libraries used by sampled projects are used transitively, and the number of transitive dependencies is typically much larger than those of direct dependencies. Further, Finding 1 highlights the importance for development teams to perform checks beyond their own code and direct dependencies, and Finding 6 reinforces the need for developers to be vigilant of potential dependency vulnerability beyond those in public database. Finding 7 suggests importance of monitoring and applying updates to project dependencies. Overhead of such effort can be reduced by integrating vulnerability scanning tools or more comprehensive software composition analysis tools into the development team's Continuous Integration workflow. In view of typical latency before vulnerabilities become publicly known and additional latency before a fix is available, understanding of common vulnerability types (Finding 2) enables library users to anticipate security risks from such vulnerabilities when designing their software or production environment, for example by applying relevant recommendations from organizations such as OWASP[9].

Finding 9 suggests that there is value in attempting to simplify a project's dependency set to reduce vulnerabilities. Relating our finding to the findings of Abdalkareem et al. [1] regarding prevalent usage of trivial libraries (that often lack tests and have many dependencies of their own), one practical step library

[9] https://cheatsheetseries.owasp.org/index.html

users can try is to reduce their projects' dependency on trivial libraries, for example, by replacing a group of such libraries with single library that covers the same set of functions and has a good security track record. Beyond this, library users will likely also benefit by selecting a set of libraries that share a common set of dependencies (including the specific version numbers) for their projects.

5.2 Implications for library developers

The update latency which contributes to dependency vulnerability persistence (as per Finding 6) and long resolution time (as per Finding 8) suggest that it is important for library developers to make library updates easier, as well as to encourage library users to perform timely update of their projects' dependencies. Given that library users' beliefs regarding potential risks of updating will be strongly affected by personal experience [11], it will be useful to allay library users' concerns about potential risks of update by providing comprehensive tests and documentation, in addition to maintaining good communication with library users.

5.3 Implications for researchers

The update latency related to vulnerable dependencies, which result in high persistence of dependency vulnerabilities and long resolution time (Findings 6 and 8), suggests the need for better dependency monitoring and update approaches. One line of work that needs to be explored further is automatic program transformation to allow client code to catch up with the latest updates [13, 26, 53]. Such technique will facilitate smoother dependency update, however, accuracy of existing works are not perfect, and they tend to be limited to particular set of API (e.g. Android APIs). Related to this, our work also demonstrates the value of research into automated techniques to detect breaking changes in library updates, particularly those that are generalizable, as existing works [19, 34, 35] focus on specific language and package ecosystem (*SourceClear* itself supports detection of whether an update is likely to break a build, but supported languages are currently limited to Java, Python, and Ruby).

The findings also demonstrate value in researching approaches to recommend libraries known to be secure to developers starting new projects, as developers may not readily update or change their project's dependency set afterwards, even after the discovery of vulnerabilities. In addition, over lifetime of a project, some of its dependencies may cease to be actively maintained, and those dependencies may subsequently become less secure compared to contemporary alternatives. Detection of such situation and recommendation of alternatives may help project developers keep their work secure. Some tools such as WhiteSource (which is used by GitHub) and SourceClear are able to detect outdated libraries and automatically generate pull request for updates to newer version of the same libraries[10,11]. However, to our knowledge, currently SCA tools do not provide alternative library recommendations based on security track record and update frequency.

Lastly, the prevalence of certain types of dependency vulnerabilities across different languages (e.g. "Denial of Service" and "Information Disclosure") as per Finding 2 indicates potential widespread benefit from research into the resolution or mitigation of such vulnerabilities.

6 Threats to Validity

6.1 Threats to internal validity

Threat to internal validity stems from limitations related to data and analysis capability of the *SourceClear* tool and its associated platform database. It makes no claim of complete identification of libraries and associated information, and is affected by information in the files it analyzes. We attempt to mitigate this threat by focusing on software projects developed using popular programming languages. Another threat to validity, which affects analyses related to correlation between vulnerability and project attributes, originates from time difference between latest commit analyzed and extraction time of the project metadata from GitHub, during which there may be change in attribute's values. The next threat to internal validity, which affects computation of average time needed to fix dependency vulnerabilities, originate from vulnerabilities that have already existed in sample projects since before beginning of the observation period, as well as vulnerabilities that are not yet fixed by the end of the observation period.

6.2 Threats to external validity

Generalizability of our findings may be affected by two factors. First, different software projects may use different open-source libraries, which may in turn have different kinds of vulnerabilities and licenses. We attempt to mitigate this threat by performing random selection from *reaper* dataset without regard to project type. Another external threat to validity comes from the fact that the sampled repositories contain projects that have existed for a few years and are still actively developed. While our results indicate no strong correlation between the number of commits in the period of interest and the number of vulnerabilities, there may still be differences between characteristics of the sampled projects with, for example, those of recently started projects that are more likely to use the latest library versions from the beginning.

[10] https://help.veracode.com/reader/hHHR3gv0wYc2WbCclECf_A/EDLOi6PYdFYDvenrK_0vCQ
[11] https://help.github.com/en/github/managing-security-vulnerabilities/about-security-alerts-for-vulnerable-dependencies

7 Related Work

7.1 Characteristics of Vulnerabilities

Security vulnerabilities of software projects have been a subject of a number of empirical studies. For example, Shahzad et al. [47] performed analysis on a data set of software vulnerabilities from 1988 to 2011, focusing on seven aspects related to their life cycle. Among other findings, they noted that Denial of Service, Buffer Overflow, and remote code execution are the three most exploited forms of vulnerabilities, but SQL injection, cross-site scripting (XSS), and PHP-specific vulnerabilities were also on the rise. Our findings indicate that at the time of the writing, SQL injection, XSS, and Denial of Service also rank highly among common vulnerability types, although with the exception of Denial of Service, this is not universal across languages. Camilo et al. [6] performed statistical analyses on bugs and vulnerabilities mined over five releases of Chromium project to examine the relationship between the two groups, and discovered that bugs and vulnerabilities are empirically dissimilar. Ozment and Schechter [40] performed a study on code base of OpenBSD operating system and compiled a database of vulnerabilities identified within a 7.5 year period, and discovered, among others, that 62% of vulnerabilities identified during the period are *foundational*, i.e. the vulnerabilities are already present in the source code at the beginning of the study. Our analysis regarding persistence of vulnerabilities in the sample projects' OSS dependencies found similar vulnerability persistence issues across languages. More recently, Zahedi et al. [57] performed a study on security-related issues from a sample of 200 repositories on GitHub, and discovered that most security issues reported are related to identity management and cryptography, and that security issues comprise only about 3% of all reported issues. We found that in case of vulnerability in OSS dependencies, there is variation across languages. For example, while cryptography-related vulnerabilities ranks among top five in Java, it is not so in other languages. In contrast to the above-mentioned works however, our work focus on vulnerabilities in the sample projects' OSS dependencies instead of vulnerabilities in the sample projects' source code.

Beyond this, there have also been studies that focus on code and programming practice descriptions in StackOverflow posts. For example, Meng et al. [33] conducted an empirical study on 497 StackOverflow posts related to Java security to understand challenges faced by Java developers in attempting to write secure code. They discovered issues that hinder secure coding practices such as complexity of cryptography APIs and Spring security configuration methods, as well as vulnerabilities in code blocks within accepted answers. Rahman et al. [43] studied code blocks contained in 44,966 Python-related answers on StackOverflow, and found 7.1% of them to contain one or more insecure coding practice, with code injection being the most frequent type of issue. They also found no relation between user reputation and presence of insecure coding practice in the answer provided by the user. While the scope of our work does not include StackOverflow post, we believe all these factors contribute to spread and persistence of vulnerabilities observed in our study. If a language's security features are difficult to use, and example code in that language commonly contain vulnerabilities, library developers may not be aware of the proper way to write secure code in that language.

7.2 Relationship between Software Metrics and Vulnerabilities

A number of work investigate relationship between presence of vulnerabilities and various software metrics. Many of those works also propose vulnerability prediction models based on software metrics. For example, Zimmerman et al. [61] investigated whether software metrics that are commonly used in defect prediction (such as code churn and complexity) are useful for predicting vulnerabilities in Windows Vista. They found that using such metrics achieves high precision but low recall. Meneely and Williams [30, 31] examined correlations between vulnerabilities in several open-source software projects and various developer activity metrics, such as number of commits made to a file and number of developers who had changed a file. Among other findings, they found that files that have been changed by six or more developers were four times more likely to contain vulnerability compared to files changed by five or fewer developers. Shin and Williams [48] examined potential usage of execution complexity metrics (such as frequency of function calls) collected from common usage pattern, for predicting software components that may contain vulnerability. They compared the performance between static complexity metrics and the combination of both sets of metrics. They found that the effectiveness of prediction of vulnerable code location using execution complexity metrics vary between the software projects analyzed, with good result for one of the projects (Firefox) but no significant discriminative ability and low recall in the other (Wireshark).

Other than works focusing on relationship between software metrics and vulnerable part of source code, a number of works use software metrics to identify vulnerability-contributing commits. For example, Bosu et al. [4] analyzed code review requests from 10 open source projects to identify characteristics of changes that are more likely to contain vulnerabilities. They found, among others, that larger number of changed lines correspond to higher likelihood of vulnerability, and that new files are less likely to contain vulnerabilities compared to modified files. Another example is Perl et al.'s work [42] in which they performed mapping between CVEs and GitHub commits of 66 open-source projects and subsequently experimented with using combination of software repositories metadata and software metrics to train a classifier for vulnerability-contributing commit identification. They found that the combination enabled significant reduction of false positives by 99% compared to the then state-of-the-art approach, while maintaining level of recall.

Our work differs from the above-mentioned works since our focus is on vulnerabilities in OSS dependencies of software projects, instead of vulnerabilities in the project code itself. In consequence, our metrics differ. We use project-level metrics and metrics related to software dependencies, instead of file-level metrics. Furthermore, our objective is not vulnerability prediction, but rather investigation into characteristics of projects known to contain OSS dependency vulnerabilities.

7.3 Vulnerable Dependencies

There has been several works that discuss vulnerable dependencies in context of library updatability or migrations. Derr et al. [10] conducted a large-scale library updatability analysis on Android applications along with a survey with developers from Google Play, and reported that among the actively-used libraries with known security vulnerability, 97.8% can actually be updated without changing application code. They found that reasons for not updating dependencies include lack of incentive to update (since existing versions work as intended), concern regarding possible incompatibility and high integration effort, as well as lack of awareness regarding available updates. Zimmerman et al. [60] conducted a study on security risks in the *npm* ecosystem, and found that single points of failure exists within the ecosystem due to the dependency network structure. The main issues include possibility of vulnerability in single libraries to impact large parts of *npm* ecosystem, and possibility of very small number of package maintainers to introduce vulnerability to large part of the ecosystem. Decan et al. [8] studied the evolution of vulnerabilities in *npm* dependency network using 400 security reports from a 6-year period. Among their findings, they reported that dependency constraints prevented more than 40% of package releases with vulnerable dependencies from being fixed automatically by switching to newer version of the dependencies. Kula et al. [25] conducted a study on impact of security advisories on library migration on 4,600 software projects on GitHub and discovered, among others, that many developers of studied systems do not update vulnerable dependencies and are not likely to respond to a security advisory. Our finding related to persistence of vulnerabilities (Findings 7 and 9) confirms their findings regarding prevalence of significant delay in updating vulnerable dependencies.

Related to usage of vulnerable dependencies, Cadariu et al. [5] investigated the prevalence of usage of dependencies with known security vulnerabilities in 75 proprietary Java projects built with Maven. They found that 54 of the projects use at least 1 (and up to 7) vulnerable libraries. Lauinger et al. [27] analyzed the usage of Javascript libraries by websites in top Alexa domains as well as random sample of .com websites, and found that around 37% of them include at least one library known to contain vulnerability. Paschenko et al. [41] performed a study on instances of 200 Java libraries that are most often used in SAP software, and found that about 20% of affected dependencies are not actually deployed, and 81% of vulnerable dependencies can be fixed by simply updating the library version. Dashevskyi et. al. [7] identified three different cost models to estimate the amount of security maintenance effort (e.g. vulnerability fixes) required when using open-source components in proprietary software products. They analyzed usage of 166 open-source components in SAP products and found that open-source component size (measured as lines of code) and age are the main factors influencing security maintenance effort.

Our study uses a larger and more diverse dataset compared to the existing works on vulnerable dependency usage. Our dataset comprises software projects with different characteristics (type, language, authors, organization, etc.), which improves generalizability of our findings. In addition, the software composition analysis tool we use includes a database which includes details on vulnerabilities such as type labels, enabling more systematic grouping and analyses of vulnerability by their characteristics. This allows us to derive insights related to popularity of different vulnerability types, which has not been analyzed in [5, 27, 41]. Further, the database also includes a number of non-CVE vulnerabilities in addition to publicly-known vulnerabilities in CVE list, which improves comprehensiveness of the scan results. In addition, we scan the dependency graphs of the projects' code bases directly to obtain information on its open-source dependencies and associated vulnerabilities. This approach enables higher accuracy compared to reliance on proxies such as text content of reported issues. Finally, the commit-level granularity of our analysis enables the identification of general changes in the one-year observation period as well as the relationship between vulnerabilities and commit attributes.

8 Conclusions and Future Work

In this work we conducted an empirical study on open-source dependencies of 450 GitHub projects written in three popular programming languages. We scanned the commits made to those projects between November 1, 2017 and October 31, 2018, and identified common vulnerability types, as well as vulnerable libraries that affect the most projects. We also found evidence that number of vulnerabilities associated with open-source dependencies tend to be higher in Java and Ruby projects, indicating opportunity to improve software security by improving open-source libraries, notification of vulnerability discovery, and ease of library update in those languages. Our results indicate that significant percentage of vulnerable dependency issues are persistent, and among the issues that are fixed, the average time taken is about 4-6 months. Related to project and commit attributes, we found that number and experience of contributors, project activity level, and size do not appear to correlate with better handling of vulnerable dependencies. Rather, vulnerability counts correlate more strongly with the number of direct and transitive dependencies. This highlights to library users the importance of managing the number of their projects' dependencies carefully, in addition to performing timely updates.

A potential direction of future work is expansion of the scale of the study to cover projects written in other programming languages supported by *SourceClear*, as well as investigation of commits from longer time period. Beyond this, our future work lies in investigation into associations between dependency vulnerability types as well as the factors that promote or mitigate them. Another element of potential future work related to our study is the identification of characteristics of projects with track record of resolving dependency vulnerabilities quickly and how the characteristics can be emulated in other projects. In addition, we are also interested in investigating the techniques to automatically identify and update vulnerable dependencies in project codebase.

References

1. Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 385–395
2. Arora A, Telang R (2005) Economics of software vulnerability disclosure. IEEE security & privacy 3(1):20–25
3. Bell RM, Ostrand TJ, Weyuker EJ (2013) The limited impact of individual developer data on software defect prediction. Empirical Software Engineering 18(3):478505, DOI 10.1007/s10664-011-9178-4
4. Bosu A, Carver JC, Hafiz M, Hilley P, Janni D (2014) Identifying the characteristics of vulnerable code changes: An empirical study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 257–268
5. Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, pp 516–519
6. Camilo F, Meneely A, Nagappan M (2015) Do bugs foreshadow vulnerabilities?: a study of the chromium project. In: Proceedings of the 12th Working Conference on Mining Software Repositories, IEEE Press, pp 269–279
7. Dashevskyi S, Brucker AD, Massacci F (2016) On the security cost of using a free and open source component in a proprietary product. In: International Symposium on Engineering Secure Software and Systems, Springer, pp 190–206
8. Decan A, Mens T, Constantinou E (2018) On the impact of security vulnerabilities in the npm package dependency network. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, pp 181–191
9. Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering 24(1):381–416
10. Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep me updated: An empirical study of third-party library updatability on android. In: Pro-

ceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, pp 2187–2200
11. Devanbu P, Zimmermann T, Bird C (2016) Belief & evidence in empirical software engineering. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, pp 108–119
12. Durumeric Z, Li F, Kasten J, Amann J, Beekman J, Payer M, Weaver N, Adrian D, Paxson V, Bailey M, et al (2014) The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference, ACM, pp 475–488
13. Fazzini M, Xin Q, Orso A (2019) Automated api-usage update for android apps. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 204–215
14. Foo D, Ang MY, Yeo J, Sharma A (2018) Sgl: A domain-specific language for large-scale analysis of open-source code. In: 2018 IEEE Cybersecurity Development (SecDev), IEEE, pp 61–68
15. Foo D, Chua H, Yeo J, Ang MY, Sharma A (2018) Efficient static checking of library updates. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, pp 791–796
16. Gardner W, Mulvey EP, Shaw EC (1995) Regression analyses of counts and rates: Poisson, overdispersed poisson, and negative binomial models. Psychological bulletin 118(3):392
17. Hilbe JM (2011) Negative binomial regression. Cambridge University Press
18. Hoepman JH, Jacobs B (2007) Increased security through open source. Communications of the ACM 50(1):79–83
19. Jezek K, Dietrich J (2017) Api evolution and compatibility: A data corpus and tool evaluation. Journal of Object Technology 16(4):2–1
20. Jimenez M, Papadakis M, Le Traon Y (2016) Vulnerability prediction models: A case study on the linux kernel. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 1–10
21. Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 92–101
22. Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. Journal of the American statistical association 53(282):457–481
23. Kononenko O, Baysal O, Godfrey MW (2016) Code review quality: how developers see it. In: Proceedings of the 38th International Conference on Software Engineering, pp 1028–1038
24. Kula RG, German DM, Ishio T, Inoue K (2015) Trusting a library: A study of the latency to adopt the latest maven release. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, p 520524, DOI 10.1109/SANER.2015.7081869, URL http://ieeexplore.ieee.org/document/7081869/

25. Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empirical Software Engineering 23(1):384–417
26. Lamothe M, Shang W, Chen TH (2018) A4: Automatically assisting android api migrations using code examples. arXiv preprint arXiv:181204894
27. Lauinger T, Chaabane A, Wilson CB (2018) Thou shalt not depend on me. Commun ACM 61(6):41–47, DOI 10.1145/3190562, URL http://doi.acm.org.libproxy.smu.edu.sg/10.1145/3190562
28. Li J, Conradi R, Bunse C, Torchiano M, Slyngstad OPN, Morisio M (2009) Development with off-the-shelf components: 10 facts. IEEE software 26(2):80–87
29. McCabe TJ (1976) A complexity measure. IEEE Transactions on software Engineering (4):308–320
30. Meneely A, Williams L (2009) Secure open source collaboration: an empirical study of linus' law. In: Proceedings of the 16th ACM conference on Computer and communications security, pp 453–462
31. Meneely A, Williams L (2010) Strengthening the empirical analysis of the relationship between linus' law and software security. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp 1–10
32. Meneely A, Srinivasan H, Musa A, Tejeda AR, Mokary M, Spates B (2013) When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, pp 65–74
33. Meng N, Nagy S, Yao D, Zhuang W, Arango-Argoty G (2018) Secure coding practices in java: Challenges and vulnerabilities. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, pp 372–383
34. Mezzetti G, Møller A, Torp MT (2018) Type regression testing to detect breaking changes in node. js libraries. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
35. Møller A, Torp MT (2019) Model-based testing of breaking changes in node. js libraries. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 409–419
36. Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empirical Software Engineering 22(6):3219–3253
37. Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering, ACM, pp 284–292
38. Neuhaus S, Zimmermann T, Holler C, Zeller A (2007) Predicting vulnerable software components. In: ACM Conference on computer and communications security, Citeseer, pp 529–540

39. Ostrand TJ, Weyuker EJ, Bell RM (2010) Programmer-based fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE 10, ACM Press, p 1, DOI 10.1145/1868328.1868357, URL http://portal.acm.org/citation.cfm?doid=1868328.1868357
40. Ozment A, Schechter SE (2006) Milk or wine: does software security improve with age? In: USENIX Security Symposium, pp 93–104
41. Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2018) Vulnerable open source dependencies: counting those that matter. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, p 42
42. Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp 426–437
43. Rahman A, Farhana E, Imtiaz N (2019) Snakes in paradise?: insecure python-related coding practices in stack overflow. In: Proceedings of the 16th International Conference on Mining Software Repositories, IEEE Press, pp 200–204
44. Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 155–165
45. Raymond E (1999) The cathedral and the bazaar. Knowledge, Technology & Policy 12(3):23–49
46. Seabold S, Perktold J (2010) statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference
47. Shahzad M, Shafiq MZ, Liu AX (2012) A large scale exploratory analysis of software vulnerability life cycles. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE, pp 771–781
48. Shin Y, Williams L (2011) An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: Proceedings of the 7th International Workshop on Software Engineering for Secure Systems, pp 1–7
49. Shin Y, Williams L (2013) Can traditional fault prediction models be used for vulnerability prediction? Empirical Software Engineering 18(1):25–59
50. Shin Y, Meneely A, Williams L, Osborne JA (2010) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE transactions on software engineering 37(6):772–787
51. Spadini D, Aniche M, Bacchelli A (2018) Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, pp 908–911
52. Spearman C (1904) The proof and measurement of association between two things. American journal of Psychology 15(1):72–101

53. Thung F, Haryono SA, Serrano L, Muller G, Lawall J, Lo D, Jiang L (2020) Automated deprecated-api usage update for android apps: How far are we? In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 602–611
54. Trockman A (2018) Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, p 524526, DOI 10.1145/3183440.3190335, URL https://dl.acm.org/doi/10.1145/3183440.3190335
55. Weyuker EJ, Ostrand TJ, Bell RM (2007) Using developer information as a factor for fault prediction. In: Third International Workshop on Predictor Models in Software Engineering (PROMISE07: ICSE Workshops 2007), IEEE, p 88, DOI 10.1109/PROMISE.2007.14, URL http://ieeexplore.ieee.org/document/4273264/
56. Witten B, Landwehr C, Caloyannides M (2001) Does open source improve system security? IEEE Software 18(5):57–61
57. Zahedi M, Ali Babar M, Treude C (2018) An empirical study of security issues posted in open source projects. In: Proceedings of the 51st Hawaii International Conference on System Sciences
58. Zhang Y, Lo D, Xia X, Xu B, Sun J, Li S (2015) Combining software metrics and text features for vulnerable file prediction. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, pp 40–49
59. Zhou Y, Sharma A (2017) Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 914–919
60. Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Small world with high risks: A study of security threats in the npm ecosystem. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp 995–1010
61. Zimmermann T, Nagappan N, Williams L (2010) Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: 2010 Third International Conference on Software Testing, Verification and Validation, IEEE, pp 421–428