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

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

Authors omitted for review

Received: date / Accepted: date

Abstract

In this work, we perform an empirical study on vulnerabilities in open-source libraries used by 600 software projects in GitHub written in four popular programming languages (C#, Java, Python, and Ruby). By scanning commits made on those projects between November 2017 and October 2018 using an industry-grade software composition analysis tool, we obtain information regarding open-source dependencies of the project, such as library names and versions, known associated vulnerabilities and dependency type. We analyzed common types, distributions, and persistence of vulnerabilities among the dependencies of projects in each language, relationships between vulnerabilities and project as well as commit attributes. In addition, we also studied direct and transitive dependencies of the projects. Among other findings, we found that dependency vulnerabilities are persistent and its number is unrelated to project activity level, the project's popularity, and developer experience. We also found a degree of similarity between types of most common vulnerabilities among projects in different languages. We believe the results of the study will help both open-source library users and developers in prioritizing their efforts to resolve or mitigate potential issues related to usage of open-source libraries.

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 [19], 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

Address(es) of author(s) should be given

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, and while an organization may have procedures to review code written by their own developers, or to perform static analysis on the code using various automated tools, such practices often do not extend to third-party libraries. 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 [11], 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 popular Electron framework in January 2018 affected a number of Windows applications built using the framework, such as Skype and Slack.

There are several 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 associated commits on GitHub. The SCA tool uses database of open-source libraries that is maintained by Veracode security researchers and includes detailed information on associated vulnerabilities such as severity and category labels, enabling systematic categorization of detected vulnerabilities in sampled projects' dependencies, among others. For our dataset, we sampled 600 software projects on GitHub that are written in four popular programming languages (C#, Java, Python, and Ruby) and have at least 5 commits during 1-year period between November 2017 to October 2018. Being significantly larger and more diverse than datasets of ear-

[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

lier works on vulnerability dependency usage [4, 18, 27]), our dataset enables better generalizability of analysis results. We subsequently used *SourceClear* to scan all commits made to the sampled projects during the 1-year period. Afterwards, we analyzed the scan results, which includes vulnerability details such as CVE identifier, type, and severity. We examined a variety of aspects related to characteristics of discovered vulnerabilities in the sampled projects' open-source dependencies, including common types, frequency, persistence, and associations between types, 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 characteristics of dependency vulnerabilities in open-source software?*
  Understanding characteristics 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 associations between certain dependency vulnerability types, along with some common 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 (i.e. Linus' Law [31]). We're interested to examine whether this view holds true for dependency vulnerability count. Further, there has also been various works in vulnerability prediction (e.g. [24, 25, 42, 34, 40, 15]) 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. We found that the vulnerability counts correlate more strongly with project's total dependency counts compared to the project activity level, popularity, scale of commit, and experience level of developer making a commit. This suggests, for example, that reducing total number of dependencies (e.g. by reducing use of trivial libraries, 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 usage of vulnerable dependencies [4, 18, 27] as well as works that include discussion of vulnerable dependencies in context of library migration [9, 7, 17]. We expand on the earlier set of works by analyzing significantly 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 scan on each commit made to the sampled projects within the 1-year observation period, enabling analyses related to persistence of vulnerabilities and 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 2. *SourceClear* includes static checking mechanism to aid library updates [13] as well as Security Graph Language [12], 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 2019, the SourceClear vulnerability database[7] contains 2,027,092 libraries from

[5] https://www.sourceclear.com
[6] https://cve.mitre.org/
[7] https://www.sourceclear.com/vulnerability-database

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 (C#, 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

Language	Libraries	Vuln.	Source of library information
C#	158,668	274	*www.nuget.org/packages*
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 1 provides high-level overview of the workflow. Identification of new vulnerabilities for inclusion into the database is achieved by *SourceClear* 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 [41] 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 previous state-of-the-art SVM-based classifier [28] for vulnerability detection from commit messages as well as from bug reports, achieving, for example, 54.55% higher precision with same recall rate for commit messages. These factors support our confidence in the tool's detection capability.

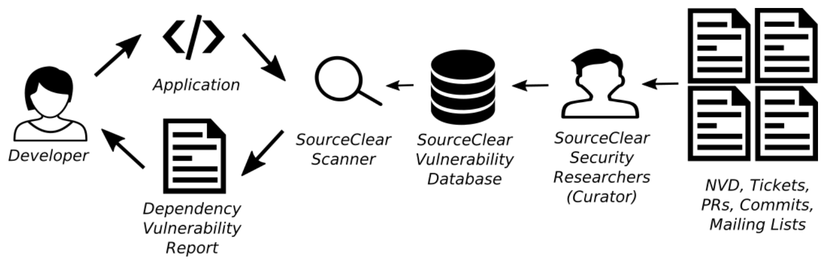
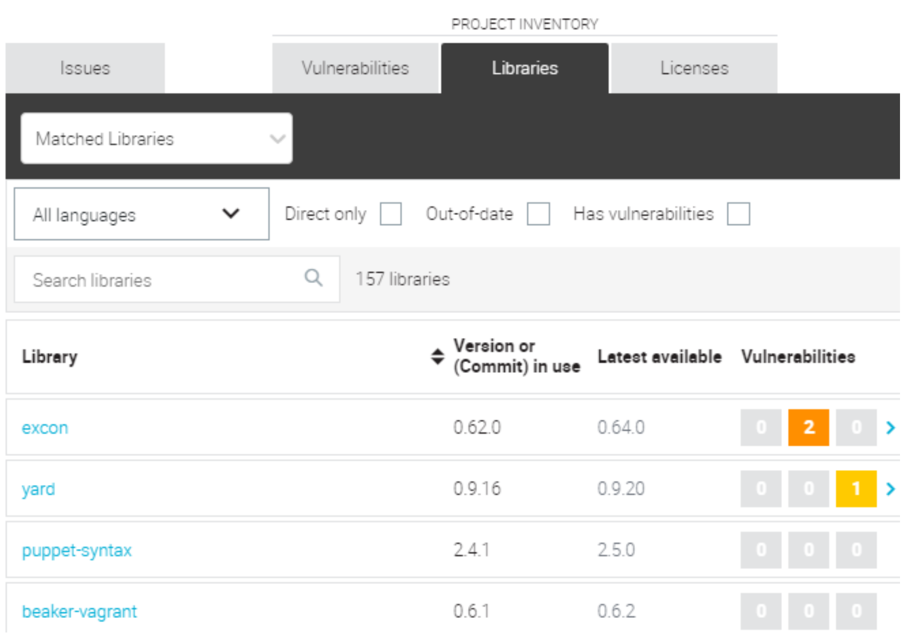

Developer
Application
Dependency Vulnerability Report
SourceClear Scanner
SourceClear Vulnerability Database
SourceClear Security Researchers (Curator)
NVD, Tickets, PRs, Commits, Mailing Lists

Fig. 1 Overview of *SourceClear* workflow

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

[8] https://nvd.nist.gov/


PROJECT INVENTORY

Issues | Vulnerabilities | Libraries | Licenses

Matched Libraries

All languages | Direct only ☐ | Out-of-date ☐ | Has vulnerabilities ☐

Search libraries | 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. 2 Part of result of a *SourceClear* scan

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

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.

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

Overall, *SourceClear* has comprehensive scan features and details in the scan results, which facilitate our analysis for characterizing the vulnerabilities in the open-source dependencies of the projects that we sampled. As our study involves multiple languages (C#, 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 and .NET but has only experimental support for Ruby and Python).

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 [16], as starting point we used the *reaper* dataset from Munaiah et al. [23] which provides a list of repositories likely to contain software projects. We set the following criteria for sampling the projects:

1. The project is written in C#, Java, Python, or Ruby, based on information from the *reaper* dataset.
2. The project repository's commit log lists at least 5 commits between beginning of November 2017 and end of October 2018.
3. The project satisfies the prerequisites to be scanned by the *SourceClear* tool, i.e. its content indicates that it uses 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.

The criteria are set to ensure that the resulting set of 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 600 software projects (150 for each programming language), and extracted the list of all commits made between November 2017 and October 2018. The statistics of the sampled projects are shown in Table 3, while the distribution of the projects' direct and transitive dependency counts are shown in Figures 3 and 4, respectively. Direct dependencies refer to dependencies that are referred by the project's code directly, while transitive dependencies refer to libraries that are referred to by other dependencies. The figures indicate 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. [8].

Table 3 Statistics of the sampled projects

Metric		C#	Java	Python	Ruby
Commits in target period	Min	5	5	5	5
	Max	4536	4579	2471	1802
	Median	16.50	23.00	20.50	16.00
	Mean	72.28	104.61	76.35	56.86
Commit authors in target period	Min	1	1	1	1
	Max	23	22	51	43
	Median	2	2	3	2
	Mean	3.14	3.91	4.69	4.11
Avg. OSS dependency	Direct	3.71	8.95	3.47	15.66
	Transitive	28.75	29.52	6.96	53.21
Projects with no OSS dependency		60	13	34	2

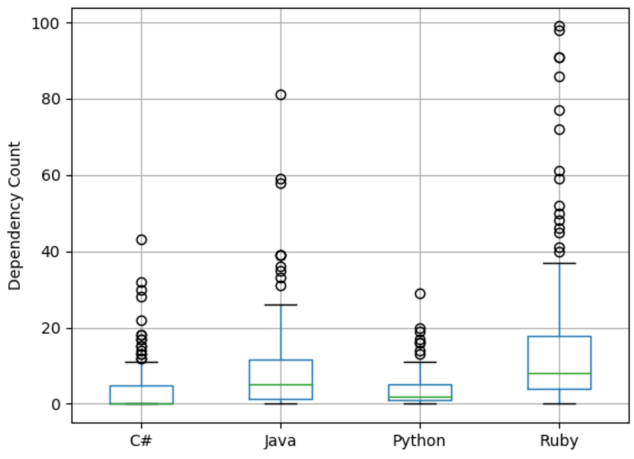


Fig. 3 Sampled projects' direct dependency counts

3.2 Methodology

After selecting the GitHub projects and downloading their commit history, we checked out each commit and perform a scan on the project repositories using *SourceClear*. The tool reports various metrics related to vulnerable dependencies and library usage that we use for subsequent analyses.

For purposes of counting distinct vulnerabilities, there are two cases to be considered: The first case is 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 next 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

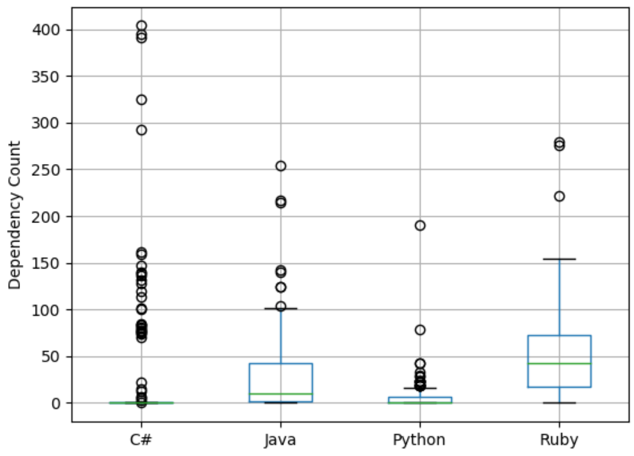


Fig. 4 Sampled projects' transitive dependency counts

CVE), we use this artifact ID instead of CVE identifier to distinguish and count individual vulnerabilities for subsequent analysis.

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

Figure 5 shows the distribution of the total counts of detected vulnerabilities in open-source dependencies of the sampled projects at the time of the latest commit in the observed period. It shows that the Java sample set has the largest overall range and variation of vulnerability counts, followed by the Ruby sample set.

In addition to the total counts, we examine the breakdown of the vulnerabilities by dependency type. Specifically, we are interested to find the average

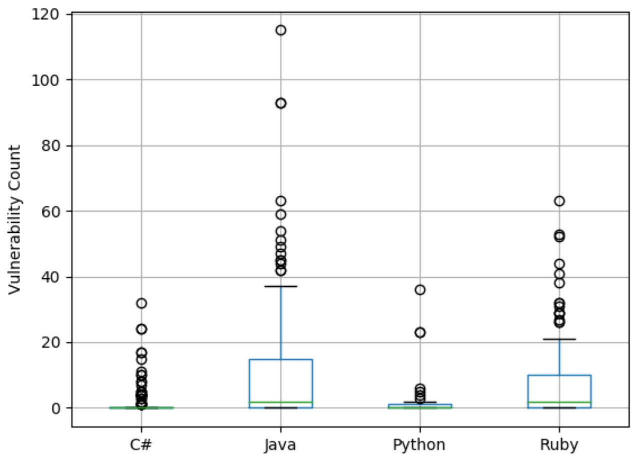


Fig. 5 Distribution of vulnerabilities in open-source libraries used by sampled projects

percentages of vulnerability 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 4. It shows that on average, for C# and 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, Java and Ruby projects have higher percentage of vulnerabilities in transitive dependencies.

Table 4 Average percentage of vulnerability count by dependency type

Language	Average percentage		
	Direct	Transitive	Both
C#	61.93	37.40	0.66
Java	32.31	58.16	9.53
Python	83.70	16.30	0.00
Ruby	14.96	77.81	7.23

Finding 1: Proportion of vulnerability counts by dependency type varies by programming language, with C# and Python projects having greater percentage of vulnerability due to direct dependency, while the opposite is true for for Java and Ruby projects.

4.1.2 Most common dependency vulnerability types

We attempt to discover common characteristics of dependency vulnerabilities in each language by identifying the vulnerability tags that occur more often at the latest commits in the period of interest. The result is shown in Table 5. 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 four languages. This suggests that improvement of practices or tools to combat those vulnerability types in open-source libraries would bring significant benefits to a wide range of software projects.

Table 5 Most common dependency vulnerability tags in each language

Language	Tag	Count
C#	Denial of Service	101
	Other	88
	Authorization	56
	Authentication	38
	Information Disclosure	33
Java	Other	769
	Denial of Service	279
	Information Disclosure	159
	Cryptography	148
	Remote Procedure Calls	136
Python	Other	70
	Information Disclosure	24
	Configuration	23
	Denial of Service	21
	Cross Site Scripting (XSS)	15
Ruby	Denial of Service	281
	Other	280
	Cross Site Scripting (XSS)	274
	Information Disclosure	175
	SQL Injection	122

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 6 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. The distribution of severity shows that most of the vulnerabilities in the sampled projects' open-source dependencies are of medium severity, but there are noticeably higher percentage of high-severity vulnerabilities in dependencies of the Java and Python projects, which suggests that Java and Python developers will benefit more from timely dependency updates.

Table 6 Distribution of Vulnerability Severity

	C#	Java	Python	Ruby
Low	2.78%	2.13%	2.10%	5.31%
Medium	91.67%	61.62%	69.23%	80.49%
High	5.56%	35.48%	25.17%	9.62%
Critical	0.00%	0.77%	3.50%	4.59%

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

We intend to see whether the vulnerabilities in the sampled projects originate from a few widely-used libraries, or many libraries that affect few projects each. 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, shown in Table 7, suggests that vulnerable libraries in Java and Ruby affect larger number of projects than the vulnerable libraries in C# and Python. This in turn indicates an opportunity to create more widespread security improvement in Java and Ruby ecosystems by patching vulnerabilities in those libraries and helping developers to upgrade to newer versions of those libraries.

Finding 4: Top vulnerable libraries in Java and Ruby affect relatively larger number of projects.

4.1.5 Association between dependency vulnerability types

In addition to occurrences of specific dependency vulnerability types, we are also interested in identifying potential associations between the different dependency vulnerability types. Similar analyses of association between types

Table 7 Top vulnerable libraries by projects affected. Project count includes projects using any version of the specified library.

Language	Library	Projects
C#	Microsoft.NETCore.App	29
	System.Net.Http	17
	System.IO.Compression.ZipFile	9
	System.Net.Security	9
	System.Security.Cryptography.Xml	6
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

have been used in prior studies to, for example, optimize testing resource allocation by predicting defects that are likely to co-occur given a discovered defect [35]. For this purpose, we applied association rule mining [2] to collection of tags associated with the set of dependency vulnerabilities detected at the time of latest commits of the sampled projects. We omit rules that contain "Other" label since the label does not convey clear information regarding vulnerability type.

Table 8 shows the top rules for each language by confidence level and support. Support indicates the percentage of time the dependency vulnerability types in the rule appears together, and confidence indicates how often the rule holds true in the dataset. For example, given the rule $\{X\} \Rightarrow \{Y\}$ with support of 0.5 and confidence of 0.75, it indicates that both X and Y appears together in 50% of the dataset, and when X appears, Y also appears 75% of the time.

To investigate the top association rules further, we performed manual inspection of a subset of the scan results. We found that in addition to associations resulting from usage of multiple libraries with different vulnerability types, there are also individual libraries that contain different vulnerability types listed in the top association rules. For example, *Microsoft.NETCore.App 2.0.0* for C# contains "Denial of Service" vulnerabilities CVE-2018-0875, CVE-2018-0764, CVE-2017-11770, as well as "Phishing attack" vulnerability CVE-2017-11879. As another example, *Django 1.8.4* for Python contains "Authentication" vulnerability CVE-2016-2513 as well as "Cross Site Scripting (XSS)" vulnerabilities CVE-2016-2512, CVE-2016-6186, CVE-2016-9014, and CVE-2017-12794. Such libraries contribute to the high confidence score of the corresponding association rules.

Table 8 Top association rules

C#	Supp.	Conf.
{Phishing attack} ⇒ {Denial of Service}	0.588	1.0
{Authorization} ⇒ {Information Disclosure}	0.471	1.0
{Authentication} ⇒ {Information Disclosure}	0.441	1.0
{Authorization, Authentication} ⇒ {Information Disclosure}	0.412	1.0
{Authentication, Denial of Service} ⇒ {Information Disclosure}	0.382	1.0
Java	**Supp.**	**Conf.**
{Remote Procedure Calls, Cryptography} ⇒ {Denial of Service}	0.389	1.0
{Remote Procedure Calls, Information Disclosure, Cryptography} ⇒ {Denial of Service}	0.358	1.0
{Authentication} ⇒ {Denial of Service}	0.347	1.0
{Cryptography, File I/O} ⇒ {Denial of Service}	0.326	1.0
{Authentication, Information Disclosure} ⇒ {Denial of Service}	0.316	1.0
Python	**Supp.**	**Conf.**
{Phishing attack} ⇒ {Information Disclosure}	0.133	1.0
{Denial of Service} ⇒ {Information Disclosure}	0.111	1.0
{Authentication} ⇒ {Cross Site Scripting (XSS)}	0.089	1.0
{Cross Site Scripting (XSS)} ⇒ {Authentication}	0.089	1.0
{Authentication} ⇒ {Information Disclosure}	0.089	1.0
Ruby	**Supp.**	**Conf.**
{Denial of Service, File I/O} ⇒ {Information Disclosure}	0.415	1.0
{Cross Site Scripting (XSS), File I/O} ⇒ {Denial of Service}	0.404	1.0
{Cross Site Scripting (XSS), File I/O} ⇒ {Information Disclosure}	0.404	1.0
{Cross Site Scripting (XSS), Information Disclosure, File I/O} ⇒ {Denial of Service}	0.404	1.0
{Cross Site Scripting (XSS), Denial of Service, File I/O} ⇒ {Information Disclosure}	0.404	1.0

We believe that understanding of such associations can help library developers and users in making more optimal mitigation and correction plan upon discovery of one of the dependency vulnerability types. For example, discovery of Remote Procedure Call and Cryptography vulnerabilities in the dependencies of a Java project may be taken as a hint to increase priority of checks and mitigation for Denial of Service vulnerability. For Ruby projects, discovery of Cross Site Scripting and File I/O types of dependency vulnerability hints at the need to guard against Denial of Service type of dependency vulnerability.

Finding 5: There are associations between certain dependency vulnerability types, which can be used to improve detection, mitigation, and correction plans.

4.1.6 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 [3]. As a result, there is often a lag between discovery of vulnerabilities by researchers until the inclusion of the vulnerability in CVE list. Due to this lag, there is a risk that developers may miss some of vulnerabilities

Table 9 Average percentage and top tags for non-CVE vulnerabilities

Language	Percentage of Vulnerability		Top non-CVE Tags
	CVE	non-CVE	
C#	97.87	2.13	Buffer Overflows Information Disclosure Cross Site Scripting (XSS)
Java	78.10	21.90	Other Denial of Service Cross Site Scripting (XSS)
Python	94.99	5.01	Denial of Service Information Disclosure Buffer Overflows
Ruby	58.48	41.52	Denial of Service Other Cross Site Scripting (XSS)

in their project dependencies even if they actively monitor and respond to CVE updates. We intend to 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. For the purpose of this analysis, we group vulnerabilities for which CVE identifiers have been reserved together with vulnerabilities that has been actually assigned CVE identifiers. Table 9 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.

Finding 6: Relying solely on public vulnerability database may cause developers to miss significant percentage of dependency vulnerabilities.

4.1.7 Overall persistence of dependency vulnerabilities

To obtain a general idea regarding the persistence of dependency vulnerabilities across the period of interest, we subsequently compute per-project percentage of dependency vulnerabilities that exist at both the time of the earliest commit and the time of the latest commit in the period. For this analysis we exclude projects in which no dependency vulnerability was detected in the earliest commit. The average percentage for each language, along with top libraries by count of persistent vulnerabilities, are shown in Table 10, and the distributions are shown in Figure 6. It demonstrates that on average, a large percentage of dependency vulnerability in the sampled projects are still unresolved throughout the 1-year period.

Finding 7: Most dependency vulnerabilities are persistent.

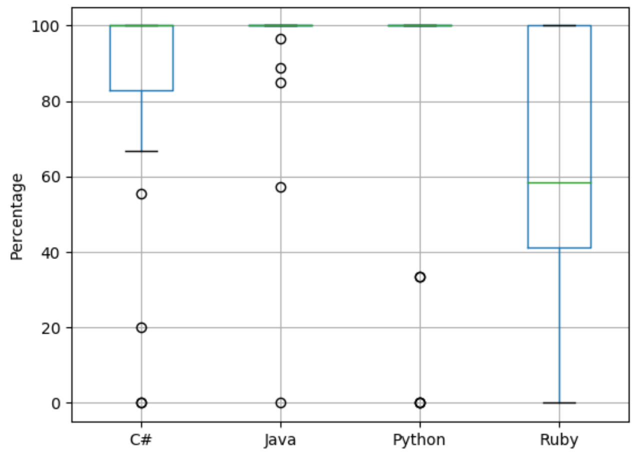


Fig. 6 Distribution of percentage of unresolved foundational dependency vulnerabilities

Table 10 Average per-project percentage of persistent foundational vulnerabilities

Language	Mean	Top libraries
C#	82.95	Microsoft.NETCore.App, System.Net.Http, System.Net.Security, System.IO.Compression.ZipFile, Microsoft.AspNetCore.Mvc.Core
Java	81.59	jackson-databind, Data Mapper for Jackson, Spring Web, Spring Web MVC, Bouncy Castle Provider
Python	87.59	Django, numpy, Pillow, PyYAML, requests
Ruby	70.06	nokogiri, activerecord, loofah, rack, actionpack

4.1.8 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 600 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 vulnerability counts at the first and 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 11 shows that except for C# sampled projects, dependency vulnerability counts tends to decrease despite increase in the number of dependencies in the 1-year period.

Table 11 Vulnerability and dependency count changes during observation period

		C#	Java	Python	Ruby
Dependency vulnerability counts	Avg. at first commit	1.29	12.04	1.04	11.64
	Avg. at last commit	1.68	11.27	0.95	7.41
	Statistic	81.0	241.5	13.0	9.0
	p-value	0.081	0.001	0.133	0.000
Dependency counts	Avg. at first commit	20.41	35.85	9.58	63.13
	Avg. at last commit	31.73	36.13	10.43	66.19
	Statistic	194.0	450.0	124.0	388.0
	p-value	0.002	0.011	0.005	0.000

Finding 8: Dependency vulnerability count tends to decrease over the 1-year study period, despite increase in number of dependencies.

4.1.9 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 author timestamp of C_3 and C_1.

We compute the figure for each repository containing the dependency vulnerability, and subsequently average the values. We find that the average time to fix dependency vulnerability is 113.92 days for C#, while for Java, Python, and Ruby, the averages are 152.62, 127.72, and 173.58, respectively, with the distribution shown in Figure 7. 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.

Finding 9: On average, resolved dependency vulnerabilities take about 4-6 months to fix.

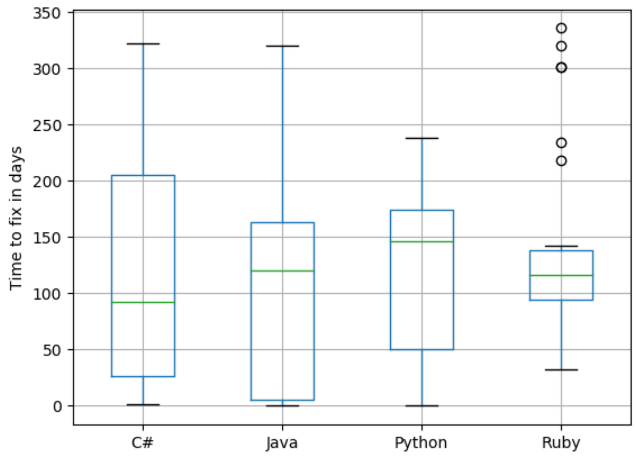


Fig. 7 Time to fix dependency vulnerability issues (in days)

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 [31]: "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 [14, 38]. Another common view regarding software projects is that higher project complexity and larger project size tend to result in the project being more prone to bugs. Given this, we are interested in examining the relationship between a project's popularity, complexity, size, and vulnerabilities in its open-source dependencies. 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 vulnerabilities resulting from the dependencies instead of 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 applied Spearman's rank correlation test [37] (ρ) between number of dependency vulnerabilities at the time of latest commit in the period of interest several of the project attributes. We chose this test as it is more suitable to handle data that is not normally distributed (compared to Pearson correlation coefficient), not sensitive to outliers, and is commonly used in software engineering domain (e.g. [5, 21, 30, 33, 42]). The list of project attributes we analyzed are as follows:

- **Age**: Project's age, measured as difference between timestamps of project's last commit in the period of interest and the first commit in the project repository
- **Commits**: Total number of commits within period of interest
- **Commit authors**: Total number of distinct commit authors in the period of interest
- **Repository size**: Current repository size
- **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

The result of the test (i.e. ρ and p-values) for all project attributes are shown in Table 12. As there is no universally-used conversion scale for correlation strengths, in this work we follow the scale used by Camilo et al. [5] to interpret the values of ρ, i.e. $\pm$ 0.00 - 0.30: Negligible, $\pm$ 0.30 - 0.50: Low correlation, $\pm$ 0.50 - 0.70: Moderate correlation, $\pm$ 0.70 - 0.90: High correlation, and $\pm$ 0.90 - 1.00: Very high correlation.

Table 12 Correlation between dependency vulnerability count and project attributes. Shaded cell indicate "moderate" to "very high" correlation with $p \leq 0.05$.

Variable	C#		Java		Python		Ruby	
	ρ	p	ρ	p	ρ	p	ρ	p
Age	0.199	0.015	0.096	0.241	-0.294	0.000	-0.189	0.020
Commits	0.241	0.003	0.114	0.164	-0.125	0.129	0.015	0.860
Commit authors	0.240	0.003	0.067	0.410	-0.090	0.275	-0.282	0.000
Repository size	-0.017	0.836	0.157	0.055	0.188	0.021	0.429	0.000
Stargazers count	0.299	0.000	-0.031	0.701	-0.180	0.028	-0.401	0.000
Direct dependencies	0.689	0.000	0.624	0.000	0.511	0.000	0.583	0.000
Transitive dependencies	0.960	0.000	0.829	0.000	0.352	0.000	0.641	0.000

The results show that there is low or negligible correlation between the dependency vulnerability counts and the project's age, number of commits, number of developers, popularity, or project size. 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 well as the presence of dependency constraints that hinder developers from updating project dependencies (despite knowledge of vulnerability in currently used versions). On the other hand, there are moderate to very high positive correlations between number of dependencies and vulnerability counts. With the exception of Python, the correlations are highest between vulnerability counts and transitive dependency counts. We found that there is relatively low percentage of common transitive dependencies between Python projects (only 18.69% of transitive dependencies are shared by two or more projects at latest commit, compared to 44.26% for C#, 32.13% for Java, and 41.35% for Ruby). This contribute to lower correlation score between transitive dependency and vulnerability counts in Python, as projects with similar transitive dependency counts may actually have very different set of dependencies (and thus associated dependency vulnerability counts).

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

Finding 10: Dependency vulnerability count can be managed more effectively by reducing the number of direct and transitive dependencies, as opposed to increasing number of contributors, activity level, or managing the project's size.

4.2.2 Commit attributes

Beyond project attributes, we are interested in investigating whether either experience of developers making the commit or the scale of change caused by a commit corresponds with number of dependency vulnerabilities detected at the time of the particular commit. More specifically, we investigate the following commit attributes:

- **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 [20] 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 [36] to obtain the metrics. As with analysis on project attributes, we also applied Spearman's rank correlation test for this part of our work and interpret the result using the scale used by Camilo et al. The results, shown in Table 13, indicate that there is only low or negligible correlation between dependency vulnerabilities with all attribute being examined. A possible explanation is that the dependency changes may be done together with a variety of other changes, such as addition of large module, small fix, or deletion of deprecated code. With more active projects, there may also be many diverse set of commits between dependency updates.

Table 13 Correlation between dependency vulnerability count and commit characteristics.

Variable	C#		Java		Python		Ruby	
	ρ	p	ρ	p	ρ	p	ρ	p
Developer experience	0.024	0.073	0.009	0.556	-0.088	0.000	0.360	0.000
Number of affected files	-0.030	0.026	0.026	0.104	-0.183	0.000	0.020	0.055
Churn	-0.003	0.811	0.078	0.000	0.119	0.000	-0.042	0.000
LOC of affected files	-0.108	0.000	0.128	0.000	0.005	0.697	0.044	0.000
Complexity of affected files	-0.056	0.000	-0.030	0.053	-0.222	0.000	-0.025	0.016

Finding 11: Commit attributes and author experience do not highly correlate with dependency vulnerability count.

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 project 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.

When resource constraints limit possible scope of analysis and mitigation of dependency vulnerabilities, Finding 5 suggests that associations between dependency vulnerability types may be used to aid planning. For example, discovery of "Remote Procedure Calls" and "Cryptography" type of vulnerability in dependencies of a Java project hints at the need to anticipate existence of "Denial of Service" vulnerability. Finding 10 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 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

Finding 5 suggests that library developers can exploit discovered associations between vulnerabilities to check their code more efficiently, for example, by analyzing potential cross site scripting issue when a potential authentication issue is discovered. Prevalence of persistent dependency vulnerability (as per Finding 7) and long resolution time of dependency vulnerability (as per Finding 9) suggest that improvement is still needed in encouraging 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 [10], 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 overall high persistence of dependency vulnerabilities and long resolution time across different languages (Findings 7 and 9) suggests the need for better dependency monitoring and update approaches. It demonstrates 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 discovery of vulnerabilities. In addition, it also demonstrates the value of research into automated techniques to update vulnerable dependencies while avoiding breakage. Further, 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 resolution or mitigation of such vulnerabilities. The association between dependency vulnerability types (Finding 5) may also be worth further investigation, since they may point to a common cause between groups of vulnerability types.

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 number of commits in the period of interest and 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 latest library versions from the beginning.

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, Camilo et al. [5] 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 [26] 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. Shahzad et al. [32] performed analysis on a data set of software vulnerabilities from 1988 to 2011, focusing on seven aspects related to their life cycle. More recently, Zahedi et al. [39] 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. Beyond this, there has also been studies that focuses on code and programming practice descriptions in StackOverflow posts. For example, Meng et al. [22] 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. [29] 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.

7.2 Vulnerable Dependencies

There has been several works that discuss vulnerable dependencies in context of library updatability or migrations. Derr et al. [9] 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. Decan et al. [7] 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. [17] 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. [4] 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. [18] 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. [27] 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. [6] identified three different cost models to estimate the amount of security effort required when using open-source components in proprietary software products, analyzed usage of 166 open-source components in SAP products, and found that the open-source component size (measured as lines of code) and age are the main factors influencing security maintenance effort.

Our study uses significantly 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 as well as associations between them, which has not been analyzed in [4, 18, 27]. Further, the database also includes a number of non-CVE vulnerabilities in addition to publicly-known vulnerabilities in CVE list, which should improve 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 600 GitHub projects written in four popular programming languages. We scanned the commits made to those projects between November 2017 and October 2018, and identified common vulnerability types, association between 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 further 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 identification of characteristics of projects with track record of resolving dependency vulnerabilities quickly and how the characteristics can be emulated in other projects. Beyond those, we are also interested in investigating 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. Agarwal R, Srikant R, et al (1994) Fast algorithms for mining association rules. In: Proc. of the 20th VLDB Conference, pp 487–499
3. Arora A, Telang R (2005) Economics of software vulnerability disclosure. IEEE security & privacy 3(1):20–25
4. 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
5. 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
6. 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
7. 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

8. 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
9. 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: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, pp 2187–2200
10. 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
11. 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
12. 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
13. 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
14. Hoepman JH, Jacobs B (2007) Increased security through open source. Communications of the ACM 50(1):79–83
15. 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
16. 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
17. Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empirical Software Engineering 23(1):384–417
18. 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
19. 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
20. McCabe TJ (1976) A complexity measure. IEEE Transactions on software Engineering (4):308–320
21. McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, pp 192–201

22. 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
23. Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empirical Software Engineering 22(6):3219–3253
24. 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
25. 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
26. Ozment A, Schechter SE (2006) Milk or wine: does software security improve with age? In: USENIX Security Symposium, pp 93–104
27. 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
28. 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, ACM, pp 426–437
29. 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
30. Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 432–441
31. Raymond E (1999) The cathedral and the bazaar. Knowledge, Technology & Policy 12(3):23–49
32. 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
33. Shin Y, Williams L (2008) Is complexity really the enemy of software security? In: Proceedings of the 4th ACM workshop on Quality of protection, ACM, pp 47–50
34. 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
35. Song Q, Shepperd M, Cartwright M, Mair C (2006) Software defect association mining and defect correction effort prediction. IEEE Transactions on Software Engineering 32(2):69–82
36. 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

37. Spearman C (1904) The proof and measurement of association between two things. American journal of Psychology 15(1):72–101
38. Witten B, Landwehr C, Caloyannides M (2001) Does open source improve system security? IEEE Software 18(5):57–61
39. 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
40. 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
41. 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
42. 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