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

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.

I. 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 [1], and even for trivial functions, developers often choose to use an external library instead of writing their own code [2]. 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, 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, even in case of open-source dependencies. 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 [3], 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 go beyond identifying vulnerable dependencies, but also 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. Given the number of repositories on GitHub on GitHub (100 million as of November 2018[5] including non-software projects and inactive projects), the sample size allows generalization of our findings

[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

[5]https://github.com/about

with confidence interval of 4 at confidence level of 95%. 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 RQ. Among others, we found that such vulnerabilities are persistent and takes months to fix. We also found associations between certain 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?*
  Static code metrics (e.g. size) as well as commit information has been used for various works in defect and vulnerability prediction (e.g. [4], [5]). Given this, we are interested to examine possible relationship between such information with vulnerabilities in a project's dependencies. We found that project activity level, popularity, scale of commit, and experience level of developer making a commit do not correlate with vulnerability counts.

There has been several works focusing on usage of vulnerable dependencies [6]–[8] as well as works that include discussion of vulnerable dependencies in context of library migration [9]–[11]. 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 II presents an overview of *SourceClear* tool used in our study. Section VII discusses works related to our study. Section III discusses the dataset collection method, overview of the dataset, and our methodology. Section IV presents the results of our empirical study. Section V discusses the implications of our findings to library users, developers, as well as researchers. Section VI discusses threats to validity of our study. Section VIII concludes our work.

II. Overview of *SourceClear*

SourceClear[6] from Veracode is a software composition analysis tool which examines usage of open-source libraries in a given software project. It supports a number of popular programming languages such as C# and Java. It works as follows: given a project code base, if necessary, it builds the project with the build system (e.g. Maven) used by the project, generates dependency graph from the result, and subsequently analyzes the graph to identify the libraries used in the project. By matching the identified libraries 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), it is able to report vulnerabilities associated with the open-source libraries used by the project, the libraries' respective licenses, as well as usage of outdated libraries. It includes static checking mechanism to aid library updates [12] as well as Security Graph Language [13], a domain-specific language that is also designed to describe and represent vulnerabilities. The language is intended to enable efficient queries involving relations between open-source libraries, their file contents (such as methods and classes), and vulnerabilities.

SourceClear is able to detect publicly-known vulnerabilities in Common Vulnerabilties and Exposures (CVE) list[7] in addition to a number of vulnerabilities that have not yet been assigned CVE identifiers. Identification of new vulnerabilities for inclusion into *SourceClear* database is achieved through a variety of techniques, such as application of natural language processing and machine learning model to identify vulnerabilities from commits and bug reports [14]. The machine learning model in particular is reported to be able to detect 349 unique vulnerabilities in the period of March 2017 - May 2017, compared to 333 CVEs reported in the same period.

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

Each detected vulnerability is also associated 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 I.

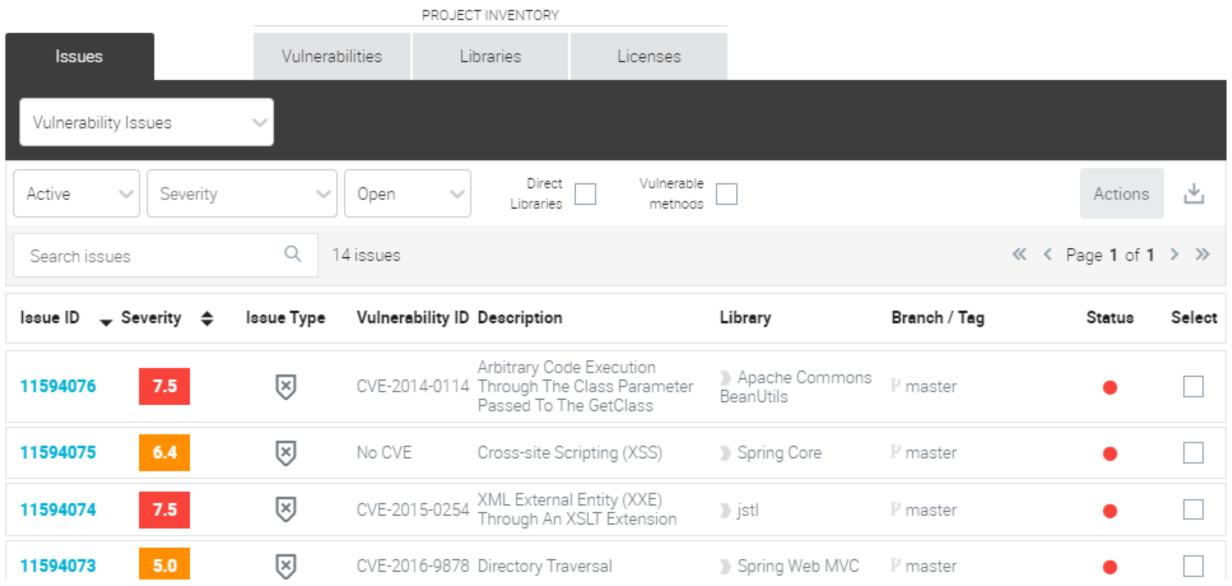
Overall, *SourceClear* has enough scan features and details in the scan results to facilitate our analysis into the characteristics of the vulnerabilities in the open-source dependencies of the projects that we sampled.

III. Dataset & Methodology

A. Dataset Collection

We use GitHub as the source of software projects for this study. Since many GitHub repositories do not actually contain

[6]https://www.sourceclear.com
[7]https://cve.mitre.org/



Issue ID	Severity	Issue Type	Vulnerability ID	Description	Library	Branch / Tag	Status	Select
11594076	7.5	🛡	CVE-2014-0114	Arbitrary Code Execution Through The Class Parameter Passed To The GetClass	Apache Commons BeanUtils	master	●	☐
11594075	6.4	🛡	No CVE	Cross-site Scripting (XSS)	Spring Core	master	●	☐
11594074	7.5	🛡	CVE-2015-0254	XML External Entity (XXE) Through An XSLT Extension	jstl	master	●	☐
11594073	5.0	🛡	CVE-2016-9878	Directory Traversal	Spring Web MVC	master	●	☐

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

TABLE I
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

software projects [15], as starting point we used the *reaper* dataset from Munaiah et al. [16] 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 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 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 II, while the distribution of the projects' direct and transitive dependency counts are shown in Figures 2 and 3, 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. [17].

B. 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 CVE), we use this artifact ID instead of CVE

TABLE II
STATISTICS OF THE SAMPLED PROJECTS

Language	Commits in target period				Commit authors in target period				Avg. OSS Dependency		Project with no OSS dependency
	Min	Max	Median	Mean	Min	Max	Median	Mean	Direct	Transitive	
C#	5	4536	16.5	72.28	1	23	2	3.09	3.69	28.75	62
Java	5	4579	23.0	104.61	1	22	2	4.01	8.85	29.21	13
Python	5	2471	20.5	76.28	1	51	3	4.72	3.50	7.04	33
Ruby	5	1802	16.0	56.86	1	43	2	4.24	15.39	52.97	2

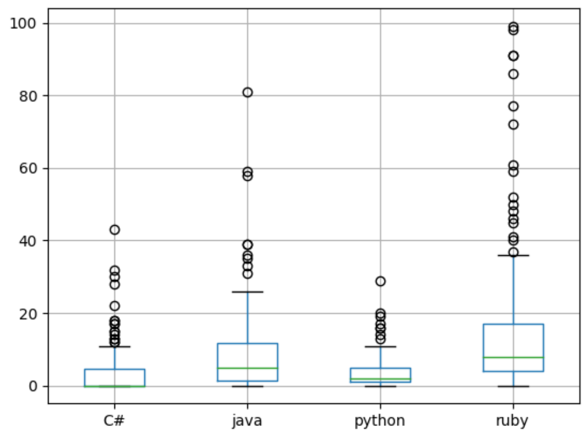


Fig. 2. Sampled projects' direct dependency counts

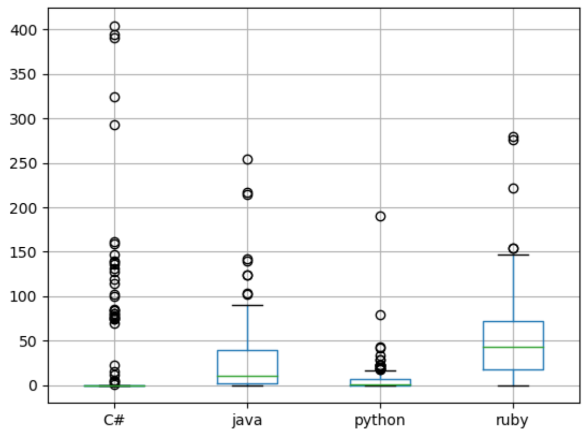


Fig. 3. Sampled projects' transitive dependency counts

identifier to distinguish and count individual vulnerabilities for subsequent analysis.

In this work, we use "first commit" or "earliest commit" as 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).

IV. Empirical Study Results

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

1) Vulnerability counts: Figure 4 shows the distribution of total counts of detected vulnerabilities in open-source dependencies of the sampled projects at the time of latest commit in the observed period. Overall, the numbers of such vulnerabilities appear to be higher in Java and Ruby projects.

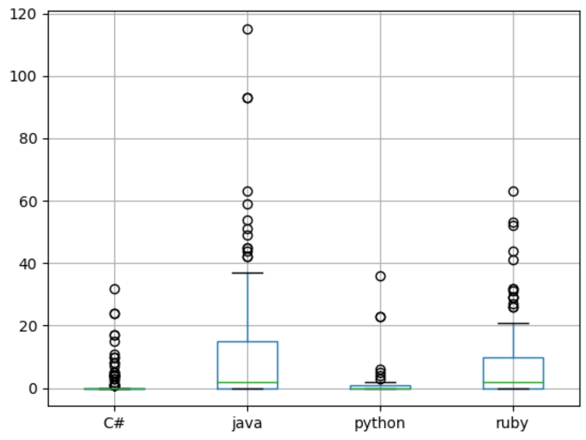


Fig. 4. Distribution of vulnerabilities in open-source libraries used by sampled projects

In addition to total counts, we examine the breakdown of the vulnerabilities by dependency type. Specifically, we are interested to find the average 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. As shown by Table III, the breakdown of vulnerability by dependency type varies, with Java and Ruby projects having more vulnerabilities in transitive dependencies and dependencies that are used both directly and transitively on average.

Finding 1: Proportion of vulnerability counts by dependency type varies by language.

TABLE III
AVERAGE PERCENTAGE OF VULNERABILITY COUNT BY DEPENDENCY TYPE

Language	Average percentage		
	Direct	Transitive	Both
C#	61.93	37.40	0.66
Java	33.34	57.4	9.26
Python	83.70	16.30	0.00
Ruby	14.53	78.27	7.2

2) Most common vulnerability types: We attempt to discover common characteristics of 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 IV. The result indicates some commonalities between the kinds of 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.

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

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 V shows the distribution of severity according to the default rating scale. 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.

Finding 3: Most detected vulnerabilities are of medium severity.

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 consider the number of vulnerabilities resulting from the use of 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 VI, 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.

5) Association between vulnerability types: In addition to occurrences of specific vulnerability types, we are also interested in identifying potential associations between vulnerability types. For this purpose, we applied association rule mining [18] to collection of tags associated with the set of vulnerabilities in the latest commits of the sampled projects. Such analyses 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 [19]. We omit rules that contain "Other" label since the label does not convey clear information regarding vulnerability type.

Table VII shows the top rules for each language by confidence level and support. Support indicates the percentage of time the vulnerability types in the rule appears together, and confidence indicates how often the rule holds true in the dataset. For example, given the rule $\{\text{X}\} \Rightarrow \{\text{Y}\}$ 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. We believe understanding of such associations can help library developers and users in making more optimal mitigation and correction plan upon discovery of one of the vulnerability types. For example, discovery of Remote Procedure Call and Cryptography vulnerabilities in 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 vulnerability type hints at the need to guard against Denial of Service vulnerability type.

Finding 5: There are associations between certain vulnerability types, such as Denial of Service, File I/O, and Information Disclosure in Ruby projects.

6) Non-CVE vulnerabilities: Since there is often a lag between discovery of vulnerabilities by researchers until the inclusion of the vulnerability in CVE list, 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 intend to investigate the the extent of such risk by evaluating the average percentage of 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 VIII shows the average percentage breakdown of CVE and non-CVE vulnerabilities, along with top tags associated with the non-CVE vulnerabilities. It suggests that while most vulnerabilities discovered in a project are CVE vulnerabilities, developers

TABLE IV
MOST COMMON VULNERABILITY TAGS IN EACH LANGUAGE

C#		Java		Python		Ruby	
Tag	Count	Tag	Count	Tag	Count	Tag	Count
Denial of Service	101	Other	758	Other	70	Denial of Service	275
Other	88	Denial of Service	274	Information Disclosure	24	Cross Site Scripting (XSS)	268
Authorization	56	Information Disclosure	154	Configuration	23	Other	267
Authentication	38	Cryptography	149	Denial of Service	21	Information Disclosure	169
Information Disclosure	33	Remote Procedure Calls	137	Cross Site Scripting (XSS)	15	SQL Injection	113

TABLE V
DISTRIBUTION OF VULNERABILITY SEVERITY

	C#	Java	Python	Ruby
Low	2.78%	2.11%	2.1%	5.51%
Medium	91.67%	61.46%	69.23%	80.56%
High	5.56%	35.71%	25.17%	9.35%
Critical	0%	0.72%	3.5%	4.58%

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 vulnerabilities.

7) Overall persistence of vulnerabilities: To obtain a general idea regarding the persistence of vulnerabilities across the period of interest, we subsequently compute per-project percentage of vulnerabilities that exist at both the time of earliest commit and the time of latest commit in the period. For this analysis we exclude projects in which no 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 IX, and the distributions are shown in Figure 5. It demonstrates that on average, a large percentage of vulnerability in the sampled projects are still unresolved throughout the 1-year period.

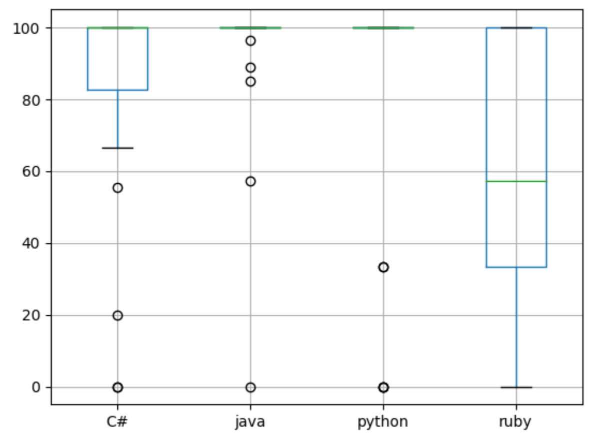


Fig. 5. Distribution of percentage of unresolved foundational vulnerabilities

Finding 7: Most dependency vulnerabilities are persistent.

8) Change of number of 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 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 X shows that except for C# sampled projects, dependency vulnerability counts tends to decrease despite increase in number of dependencies in the 1-year period.

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

9) Time required to resolve vulnerabilities: To analyze the time to resolve vulnerabilities, we listed the different dependency vulnerabilities detected in a repository during the observation period. Afterwards, we identify the commit C_1 where the vulnerability is first detected in the repository during the period, as well as the commit C_2 in which the 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 vulnerabilities that already exist at first commit in the period of interest, we use the time of first commit as starting time. We exclude vulnerabilities that still exist at the latest commits. We define the time to fix the vulnerability as the difference between author timestamp of C_3 and C_1.

We compute the figure for each repository containing the 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 164.74, 127.72, and 170.78, respectively, with the distribution shown in Figure 6. 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 VI
Top libraries by projects affected

C#		Java		Python		Ruby	
Library	Projects	Library	Projects	Library	Projects	Library	Projects
Microsoft.NETCore.App	29	Guava: Google Core Libraries for Java	47	numpy	23	rack	58
System.Net.Http	17	Apache Commons IO	33	PyYAML	9	nokogiri	49
System.IO.Compression.ZipFile	9	jackson-databind	30	Django	7	loofah	41
System.Net.Security	9	Apache Commons Collections	27	requests	4	activejob	40
System.Security.Cryptography.Xml	6	Spring Web	29	Pillow	2	activerecord	28

TABLE VII
Top association rules

C#
{Phishing attack} ⇒ {Denial of Service}
{Authorization} ⇒ {Information Disclosure}
{Authentication} ⇒ {Information Disclosure}
{Authorization, Authentication} ⇒ {Information Disclosure}
{Denial of Service, Authentication} ⇒ {Information Disclosure}
Java
{Remote Procedure Calls, Cryptography} ⇒ {Denial of Service}
{Remote Procedure Calls, Information Disclosure, Cryptography} ⇒ {Denial of Service}
{Authentication} ⇒ {Denial of Service}
{File I/O, Cryptography} ⇒ {Denial of Service}
{File I/O, Information Disclosure, Cryptography} ⇒ {Denial of Service}
Python
{Phishing attack} ⇒ {Information Disclosure}
{Denial of Service} ⇒ {Information Disclosure}
{Authentication} ⇒ {Cross Site Scripting (XSS)}
{Cross Site Scripting (XSS)} ⇒ {Authentication}
{Authentication} ⇒ {Information Disclosure}
Ruby
{Denial of Service, File I/O} ⇒ {Information Disclosure}
{Cross Site Scripting (XSS), File I/O} ⇒ {Denial of Service}
{Cross Site Scripting (XSS), File I/O} ⇒ {Information Disclosure}
{Cross Site Scripting (XSS), Information Disclosure, File I/O} ⇒ {Denial of Service}
{Cross Site Scripting (XSS), Denial of Service, File I/O} ⇒ {Information Disclosure}

TABLE VIII
Average percentage and top tags for non-CVE vulnerabilities

Language	Percentage of Vuln.		Top non-CVE Tags
	CVE	non-CVE	
C#	97.87	2.13	Buffer Overflows Information Disclosure Cross Site Scripting (XSS)
Java	77.72	22.28	Other Denial of Service Cross Site Scripting (XSS)
Python	94.99	5.01	Denial of Service Information Disclosure Buffer Overflows
Ruby	57.90	42.10	Denial of Service Other Cross Site Scripting (XSS)

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

TABLE IX
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	81.83	jackson-databind, Data Mapper for Jackson, Spring Web, Spring Web MVC, Bouncy Castle Provider
Python	87.59	Django, numpy, Pillow, PyYAML, requests
Ruby	69.59	nokogiri, activerecord, loofah, rack, actionpack

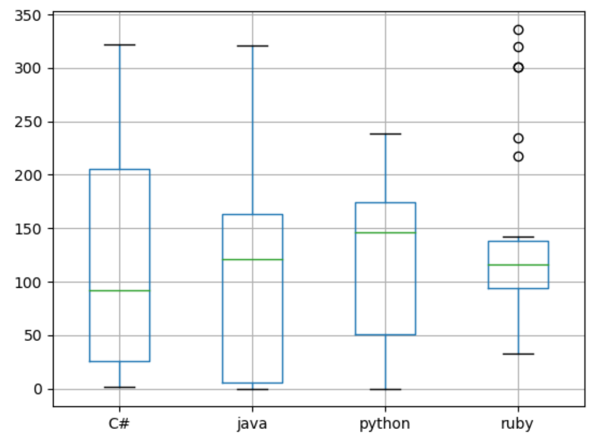


Fig. 6. Time to fix dependency vulnerability issues (in days)

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

1) Project attributes: To investigate the relationship between project attributes and total count of vulnerabilities in its open-source dependencies, we computed pairwise Pearson correlation between number of vulnerabilities at the time of latest commit in the period of interest and each of the following project attributes:

- Commits: Total number of commits within period of interest
- Commit authors: Total number of distinct commit authors in the period of interest

TABLE X
VULNERABILITY AND DEPENDENCY COUNT CHANGES DURING OBSERVATION PERIOD

Language	Dependency vulnerability counts				Dependency counts			
	Avg. at first commit	Avg. at last commit	Statistic	p-value	Avg. at first commit	Avg. at last commit	Statistic	p-value
C#	1.29	1.68	81.0	0.081	20.39	31.71	194.0	0.002
Java	11.79	11.05	255.5	0.001	35.66	35.88	445.0	0.016
Python	1.04	0.95	13.0	0.133	9.69	10.54	124.0	0.005
Ruby	11.35	7.13	9.0	0.000	62.55	65.68	383.0	0.000

- 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

TABLE XI
PEARSON CORRELATION BETWEEN DISTINCT VULNERABILITIES AND PROJECT ATTRIBUTES

Variable	C#	Java	Python	Ruby
Commits	0.125	0.079	-0.037	-0.01
Commit authors	0.332	0.067	-0.017	-0.145
Repository size	-0.037	-0.015	0.083	0.051
Stargazers count	-0.030	0.028	-0.047	-0.091
Direct dependencies	0.754	0.454	0.289	0.425
Transitive dependencies	0.949	0.843	-0.017	0.328

The result, shown in Table XI, indicates that there is no strong correlation between the first four project attributes and number of vulnerabilities detected in the projects' open-source dependencies. 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). The vulnerability count correlates more strongly with number of dependencies, especially transitive dependencies in case of C# and Java.

Finding 10: Project activity level, popularity, size, and number of contributors do not correlate with vulnerability count.

2) Commit attributes: We investigate whether either experience of developers making the commit or the scale of change caused by a commit corresponds with number of vulnerabilities detected at the time of the particular commit. Since it is not possible to objectively measure and compare actual experience of commit authors directly, we use the number of prior commits in the project as a proxy. For the scale of change caused by the commit, we use the count of affected files as a measurement. As shown in Table XII, there appears to be no strong correlation between vulnerabilities with either attribute.

TABLE XII
CORRELATION BETWEEN VULNERABILITY COUNT AND COMMIT CHARACTERISTICS

	C#	Java	Python	Ruby
With number of affected files	-0.009	-0.013	-0.015	0.015
With developer experience	0.003	0.085	-0.112	0.271

Finding 11: Scale of commit and author experience do not correlate with vulnerability count.

V. DISCUSSION AND IMPLICATIONS

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

A. 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 vulnerability types may be used to aid planning. For example, discovery of "Remote Procedure Calls" and "Cryptography" type of vulnerability in a Java project hints at the need to anticipate existence of "Denial of Service" vulnerability.

B. 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 [20], 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.

C. 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 vulnerability types (Finding 5) may also be worth further investigation, since they may point to a common cause between groups of vulnerability types.

VI. Threats to Validity

A. 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.

B. 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 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.

VII. Related Work

A. Characteristics of Vulnerabilities

Security vulnerabilities of software projects have been a subject of a number of empirical studies. For example, Camillo et al. [21] 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 [22] 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. [23] 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. [24] performed an 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.

B. 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. [10] studied 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. [11] 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. [6] 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. [7] analyzed 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. [8] 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.

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 [6]–[8]. 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 identification of general changes in the one-year observation period as well as relationship between vulnerabilities and commit attributes.

VIII. 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 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 correlates more strongly with the number of direct and transitive dependencies. This highlights the importance for library users to perform comprehensive analysis on their project's dependencies to avoid security issues. In addition, it also demonstrates the importance for library developers to facilitate users of their work to perform timely updates.

In addition to increasing the scale of the study to cover larger set of projects and additional programming languages, 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] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio, "Development with off-the-shelf components: 10 facts," *IEEE software*, vol. 26, no. 2, pp. 80–87, 2009.

[2] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "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, 2017, pp. 385–395.

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

[4] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.

[5] T. Zimmermann, N. Nagappan, and L. Williams, "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, 2010, pp. 421–428.

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

[7] T. Lauinger, A. Chaabane, and C. B. Wilson, "Thou shalt not depend on me," *Commun. ACM*, vol. 61, no. 6, pp. 41–47, May 2018. [Online]. Available: http://doi.acm.org.libproxy.smu.edu.sg/10.1145/3190562

[8] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018, p. 42.

[9] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "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, 2017, pp. 2187–2200.

[10] A. Decan, T. Mens, and E. Constantinou, "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, 2018, pp. 181–191.

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

[12] D. Foo, H. Chua, J. Yeo, M. Y. Ang, and A. Sharma, "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, 2018, pp. 791–796.

[13] D. Foo, M. Y. Ang, J. Yeo, and A. Sharma, "Sgl: A domain-specific language for large-scale analysis of open-source code," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 61–68.

[14] Y. Zhou and A. Sharma, "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, 2017, pp. 914–919.

[15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[16] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[17] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[18] R. Agarwal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. of the 20th VLDB Conference*, 1994, pp. 487–499.

[19] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69–82, 2006.

[20] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 108–119.

[21] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities?: a study of the chromium project," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 269–279.

[22] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *USENIX Security Symposium*, 2006, pp. 93–104.

[23] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 771–781.

[24] M. Zahedi, M. Ali Babar, and C. Treude, "An empirical study of security issues posted in open source projects," in *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.