PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities

Conference Paper · June 2021

DOI: 10.1109/DSN48987.2021.00031

5 authors, including:

Qiang Li
Beijing Jiaotong University
44 PUBLICATIONS **953** CITATIONS
SEE PROFILE

Song Jinke
Communication University of China
8 PUBLICATIONS **98** CITATIONS
SEE PROFILE

Haining Wang
University of Delaware
282 PUBLICATIONS **10,826** CITATIONS
SEE PROFILE

PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities

Qiang Li*, Jinke Song*, Dawei Tan*, Haining Wang †, Jiqiang Liu*
* School of Computer and Information Technology, Beijing Jiaotong University, China
† Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, USA

***Abstract*—The reuse of libraries in software development has become prevalent for improving development efficiency and software quality. However, security vulnerabilities of reused libraries propagated through software project dependency pose a severe security threat, but they have not yet been well studied. In this paper, we present the first large-scale empirical study of project dependencies with respect to security vulnerabilities. We developed *PDGraph*, an innovative approach for analyzing publicly known security vulnerabilities among numerous project dependencies, which provides a new perspective for assessing security risks in the wild. As a large-scale software collection in dependency, we find 337,415 projects and 1,385,338 dependency relations. In particular, PDGraph generates a project dependency graph, where each node is a project, and each edge indicates a dependency relationship. We conducted experiments to validate the efficacy of PDGraph and characterized its features for security analysis. We revealed that 1,014 projects have publicly disclosed vulnerabilities, and more than 67,806 projects are directly dependent on them. Among these, 42,441 projects still manifest 67,581 insecure dependency relationships, indicating that they are built on vulnerable versions of reused libraries even though their vulnerabilities are publicly known. During our eight-month observation period, only 1,266 insecure edges were fixed, and corresponding vulnerable libraries were updated to secure versions. Furthermore, we uncovered four underlying dependency risks that can significantly reduce the difficulty of compromising systems. We conducted a quantitative analysis of dependency risks on the PDGraph.**

I. Introduction

Nowadays, numerous applications have been developed using code from existing projects/libraries. Typically, software developers leverage version management tools (e.g., GitHub and Subversion) to update and extend their source code repositories. Although the code reuse saves time and resources in the software product development process, it also gives rise to underlying security concerns: different software projects may simultaneously suffer a similar flaw or threat caused by inter-dependent reused components. In other words, adversaries can exploit a security flaw in the reused component to compromise multiple projects or systems. Therefore, besides convenient development and improved quality, the reuse components also bring security vulnerabilities to the software development community [1].

In the past two decades, software vulnerabilities have been on the rise. The National Vulnerability Database (NVD) [2] reported 1,237 vulnerabilities in 2000, but that number increased to 10,594 in 2019. It does not come as a surprise that those disclosed and publicly known vulnerabilities might continue disseminating their influences through reuse components in various software projects. For instance, when the software uWSGI (version 2.0.15) is exposed to a known vulnerability CVE-2018-6758 (stack-based buffer overflow), application servers relying on APIs of uWSGI might suffer a similar underlying security risk. The benefits of the code reuse could be nullified by the cost of discovery, assessment, and mitigation of software vulnerabilities.

A deep understanding of project dependency will inform us of underlying security risks in software development and provide insights for improving project quality. More specifically, we attempt to answer the following questions: Is it common for projects to use libraries/frameworks with publicly disclosed vulnerabilities? What are the different characteristics between projects with secured libraries and those with insecure libraries? When a project uses an insecure version of another project, will it fix the insecure dependency in the future? What are the underlying risks arising due to the dependency projects of vulnerabilities? Is it possible that attackers utilize dependency relations to compromise not-yet-targeted projects? However, it is challenging to find answers, as there is no quantitative dataset for constructing dependencies of software projects on a large scale. We discover library dependencies among different projects based on a practical observation that the build files (e.g., Ant build file or pom.xml) claim dependency relations in open source-code repositories.

Constructing dependency projects of vulnerabilities faces two practical challenges. First, the build files only include the shared libraries, providing separated dependency relations rather than a full coverage of most projects. Given partial and separated dependency relations, we need to establish interconnections among those projects. Second, the vulnerability dataset and dependency project dataset come from different sources, where content and formats are varied. We need to identify a dependency project with a same vulnerability from different datasets.

To collect the quantitative dataset, we leverage three online sources: (1) the vulnerability dataset from NVD [2], (2) dependency projects from the Maven [3], and (3) dependency projects from the GitHub. Note that there are two aspects that projects on the Maven are different from that on the GitHub. First, the Maven is a typical package management tool whose repository is well maintained. Still, GitHub is a version control system whose projects are organized in an ad hoc manner. Second, the Maven provides the full dataset in its central repository, but GitHub does not. We directly parse all build files to obtain all dependency relations and construct the PDGraph for the Maven. For the GitHub, we utilize a web

crawler to collect dependent projects for the partial dataset. We further create the mapping between the vulnerability and dependency projects from the Maven and GitHub. Specifically, we employ three heuristic mapping methods, including the similarity matching, fallback URL, and third-party sources. In total, we collect 146,541 projects from the GitHub, in which 571,007 of the library dependencies exist, and 190,874 projects from the Maven, in which 814,331 dependency relations. We find that 1,014 projects involve vulnerabilities on the NVD dataset (from 2000 to 2019), and more than 67,806 projects are directly dependent on them.

In this paper, based on the datasets, we propose a new approach, called *PDGraph*, for generating a directed project dependency graph, where each vertex is a project and each edge represents the project dependency relationship. PDGraph runs common graph algorithms over the dataset, including strongly connected components (SCC) and minimum feedback arc set (MFAS). We extract the relevant information of those projects for detailed analysis, including software versions, usage, popularity, and timestamps. Further, we propose to utilize the version scheme semantic to detect insecure edges among dependency relations. There are 67,581 insecure edges among these directly linked to the projects with vulnerabilities, indicating that those projects still use outdated versions of the libraries with vulnerabilities. We further observe that only 1,266 insecure edges from GitHub are fixed while the related software is updated to secure versions over a period of eight months. Note that the existence of a vulnerable dependency is not equal to that dependent projects will certainly suffer the corresponding vulnerability.

To demonstrate PDGraph's effectiveness, we uncover four underlying dependency risks that could be leveraged by attackers for malicious purposes. (1) The number of dependency projects: our analysis shows that today's projects indeed heavily rely on libraries from others. A project indegree has an average of 4 with a standard deviation of 13, representing the number of libraries on which it depends. (2) The number of dependent projects: a project outdegree has an average of 23 with a standard deviation of 112, representing the number of other projects that use its library. (3) The length of the dependency path: the longest dependency path of a vulnerability is more than 102. The dependency risk arises because attackers can target any project in a long dependency path. (4) Circular dependencies: we find 2,691 loops in the PDGraph, indicating that they need to update/install simultaneously, which is more vulnerable to malicious attacks.

The major contributions of this paper are summarized as follows.

- We conduct the first large-scale empirical study of project dependencies with respect to security vulnerabilities by building a novel project dependency graph, PDGraph, which includes 337,415 nodes and 1,385,338 edges. We explore diverse and discrepant datasets (Maven, GitHub, and NVD) for dependency analysis of security vulnerabilities at large scale.
- We uncover 67,581 insecure edges, where outdated and vulnerable versions are still being used, even when the secure version with the patch has been released. After our eight-month observation period, only 1,266 insecure edges have been fixed due to software updates to secure versions.
- We utilize PDGraph as a new framework for revealing four underlying dependency risks. We conduct a quantitative analysis on PDGraph with respect to those risks. Our dependency dataset is available at [4].

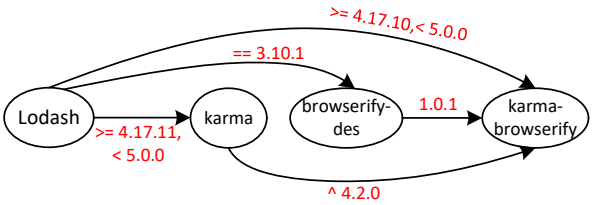


Fig. 1: An example of the interdependency of software projects with their version requirements.

The remainder of this paper is organized as follows. Section 2 provides the background of project dependency relations. Section 3 illustrates our data collection methodology. Section 4 presents the design and implementation of the PDGraph. Section 5 presents the experiments of PDGraph. Section 5 provides a detailed analysis and findings for the project dependencies. Section 6 discusses the limitations. Section 7 surveys related work, and finally, Section 8 concludes our work.

II. Background

In this section, we describe the background of software dependency. Then, we present the challenges of our work.

A. Software Dependency

Dependency. Nowadays, there are millions of projects/software packages on the Internet leveraging existing open-source libraries for resource management, communications, data access, and user interface creation. Instead of writing all of the code by themselves, software developers take advantage of existing libraries from other projects. Figure 1 shows an example of four libraries and their dependency relationships, including Lodash (a JavaScript utility library), Karma (a spectacular test utility), browserify-des (a cryptographic library), and karma-browserify (a fast Browserify integration for Karma). We found that Lodash provides its library to the other three projects, and karma-browserify relies on libraries from Lodash, Karma, and browserify-des.

A common misconception is that these reused libraries are of consistently high quality and secure. The reused components have, directly or indirectly, affected the quality and security of software development for projects. For instance, Lodash suffers a vulnerability (CVE-2019-10744 [5]) that allows an attacker to add or modify the existing property of objects. This vulnerability could affect Lodash and its dependent front-end projects (Karma, browserify-des, and karma-browserify). In a nutshell, a reused library (as shown in Figure 1) might have a security impact on its dependent software products and applications. However, there lacks a dataset for providing a quantitative analysis of numerous software projects. In this work, we created a large-scale project dependency dataset for

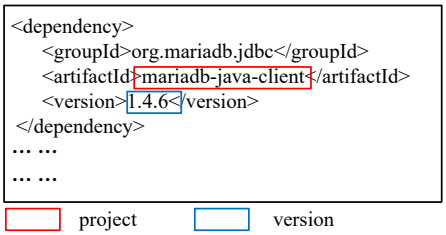


Fig. 2: An example of the build file (persistence-impl.pom) for parsing dependency relations.

understanding the underlying security risk, which is freely available to the research community.

Dependency Versions. In software development, the project dependency may involve multiple versions from a reused library. For instance, Karma relies on the Lodash from its version 4.17.11 to 5.0.0, as shown in Figure 1. For a project, we define such a set of versions from its reused library as its dependency versions. The number of dependency versions varies among different projects. In this work, we built the project dependency graph, where an edge is the dependency of two projects. Every project dependency has an attribute of the version requirement (i.e., dependency versions).

Build Files. A build file (e.g., Ant build file or pom.xml) manages required project dependencies and automates the software applications' build process. Typically, the package manager uses the build file to install prepared dependency libraries. Figure 2 depicts an instance of a Java-based project that claims its dependency libraries through the file "*.xml", including project names and associated versions. Note that software developers write a build file in the source-code repository, from which we can obtain information about the project dependency relationship. Overall, through build files, we can access those projects and identify their dependencies on a large scale.

B. Practical Challenges

Although it seems straightforward to parse build-files for dependency relations, there are two challenges in practice. First, a build file only provides separated dependency relations. For instance, the build file of Lodash (Figure 2) includes the libraries it reused, rather than its dependent projects. The construction of the graph in Figure 1 needs three other build files of dependent projects, including Karma, browserify-des, and karma-browserify. Hence, it is necessary to cover and parse most of the build files to construct a dependency graph. In practice, the information about dependency relations might be partial and separated. There are two categories of dependency relations, detailed as follows: (1) *The Full/Whole Dataset* contains all the projects and relevant dependency relations. We directly compare project names in the full dataset to find the dependency relations and construct the dependency graph. Here, the Maven provides an index of repository for the full dataset. (2) *The Partial Dataset* only contains a part of projects and relevant dependency relations. Given separated build files, we need to identify dependency among those projects. Here, the GitHub does not provide an index of repository for dependency relations. We only collect its partial dataset of

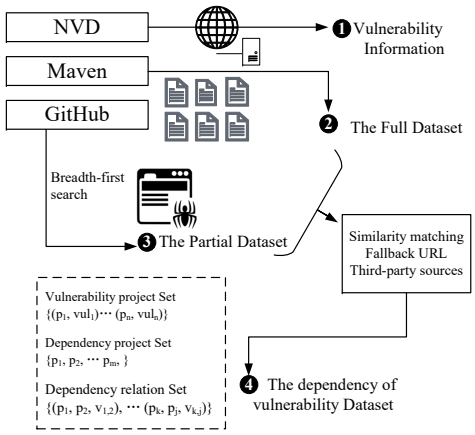


Fig. 3: An overview of our data collection methodology.

dependency relations, due to a large number of projects and access limitations of the GitHub.

Second, we need to determine whether a project involves a vulnerability; however, those datasets come from different and separated sources (NVD, Maven, and GitHub). Although those data sources are authoritative databases, the vulnerability information and project dependencies derived from them are inconsistent and discrepant. How to integrate the derived information from the three datasets is a practical challenge. A direct matching vulnerability with a dependency project leads to low precision and recall. In short, we need to quantify the consistency between vulnerability information and project dependency on a large scale.

III. Data Collection Methodology

To explore the security vulnerabilities caused by project dependency, we must collect vulnerability information and relevant project dependency. Given this goal, our data collection method centers around three different sources, including the National Vulnerability Database (NVD) [2], the GitHub website, and Maven [3]. We mined the NVD and extract vulnerabilities and their relevant information. We crawled the open-source repositories with their dependency relationships from the GitHub as the project dependency. We crawled the packages with their dependency relationships from the Maven as the project dependency. Note that projects from the Github are different from projects from the Maven. The Maven is a typical package manager that provides an automation process for installing, building, and managing projects in a consistent manner. Instead of being a package manager, the Github only provides services for software development and version control using Git. Figure 3 depicts an overview of our data collection methodology, which mainly includes four steps. (1) We extracted the vulnerability information from the NVD dataset. (2) We crawled down the full dataset of dependency projects from the Maven Index repository. (3) We utilized the breadth-first search to crawl the partial dataset of dependency projects from the GitHub. (4) We combined the vulnerability information with dependency projects through four matching techniques, including the similarity matching, fallback URL, and third-party sources.

Our data collection includes dependency project dataset

($\{p_i\}_{i=1}^{|V|}$), where project p_i relies on a library from project p_j with the dependency version $v_{i,j}$. In addition, our collected data consists of the projects involving vulnerabilities, $\{(p_i, \text{vul}_i)\}$, where vul_i represents a vulnerability.

A. Data Sources

NVD. A vulnerability is a weakness leveraged by attackers to compromise systems. For a vulnerability, professionals (software vendors, hackers, or researchers) discover it through static [6] and dynamic analysis [7]. After that, the vulnerability information may be disclosed to the public through personal blogs, public forums, and security vulnerability databases. The NVD [2] provides an official database with information to the public and disclosed software vulnerabilities. Our data collection leverages the NVD to extract vulnerability information. Specifically, we downloaded the NVD dataset from 2000 to 2019 in JSON format.

Maven. Maven is a project ecosystem that provides a build automation tool for the package manager, hosted by the Apache Software Foundation. We choose Maven as the data source for collecting dependency projects due to two reasons. First, the Maven provides a full indexing dataset for Java-based projects. Note that the Maven repository is likely matured and well maintained. Second, Maven provides a popular package manager for addressing how software is built and its dependencies. Software developers use a "pom" file to claim the dependencies with other reused and shared libraries/packages. In short, we can obtain the full dataset from the Maven for projects and dependency relations.

GitHub. We choose the GitHub as the data source for collecting dependency projects due to two reasons. First, GitHub provides the most popular version control system Git for hosting 100 million software projects by 2019. Second, all software projects are open-source, from which users can develop, review, manage, and build their software. In contrast to Maven, GitHub does not provide a dataset for dependency projects. Moreover, GitHub has an access limit per hour for preventing API abuse. We only use the web crawler to obtain dependency projects for the partial dataset.

B. Detecting Dependency Relations

We propose a dependency relationship among different software projects. Given any two projects $\forall(p_i, p_j)$, if the project p_j is using a library or a component from the project p_i, the $p_i \rightarrow p_j$ is denoted as the dependency relationship, where p_i is p_j's dependency provider, and p_j is p_i's dependent project. The dependency relationship ($p_i \rightarrow p_j$) is a directed edge from p_i to p_j. Every relationship $p_i \rightarrow p_j$ has the dependency version information of the reused component, denoted as $v_{i,j}$. The version $v_{i,j}$ is used to represent the version number of the project p_i that is used by the project j.

1) ***Dependency Relations on the Maven****:* As we mentioned before, the Maven provides a dataset for dependency projects. The Maven's central repository [8] provides an index for indexing, searching, and publishing the projects on the Maven. To collect dependency relations, we crawled down the index dataset from the Maven's central repository and unpacked it to obtain all build files. Every project in the dataset has a build file in the form of "pom.xml". We parsed the "pom.xml" file for obtaining the pairs of dependency projects. We used two fields in the build file to represent a project, including groupId and artifactId. The groupId stores the group information of the project, and the artifactId is the project name. Further, we extracted the version field in the build file, and assigned it to the version $v_{i,j}$ between p_i and p_j. We utilized all the relations of dependency projects as the dataset for the graph.

TABLE I: Supported languages associated with file formats.

Languages	File formats
Java	pom.xml
Ruby	Gemfile.lock,Gemfile, *.gemspec
Python	requirements.txt, pipfile.lock
.NET	.csproj, .vbproj, .nuspec
JavaScript	package-lock.json, package.json, yarn.lock

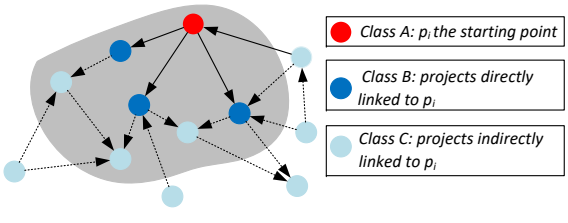
2) ***Dependency Relations on the GitHub****:* For the Github, software developers also utilize build files to claim which library dependencies their projects have, and this information is stored as the meta-data in the source-code repository. Table I lists the five programming languages that support the dependency requirement statement, including Java, Ruby, Python, JavaScript, and .NET. If a project uses one of those programming languages, project developers can write which software libraries are used in the corresponding file formats. In short, we can access those projects and identify their dependencies on a large scale from GitHub.

Aforementioned, the GitHub website hosts more than 100 million software projects and limits access for dependency projects. Therefore, we only obtain the partial dataset of dependency relation on the GitHub. Given the partial dataset, we need to detect dependent projects, denoted as $(p_i \rightarrow *)$. However, the build file only provides the libraries that the project uses, denoted as $(* \rightarrow p_i)$. Based on two directed relations ($(p_i \rightarrow *)$ and $(* \rightarrow p_i)$), we can create an association between these dependency relationships for generating our dependency graph.

Dependent Relations $(p_i \rightarrow *)$**.** Note that the Github does not provide any official API [1] to access the projects that rely on a library from the project p_i, $(p_i \rightarrow *)$. We address this problem based on a practical insight: the GitHub added new features (visualizing the dependencies and dependents of a repository) for software developers [9]. In other words, the GitHub website provides the web server interface for software developers who can observe the dependent projects that rely on their projects. The practical issue is that the web server interface only provides the owner name/repository without dependency version $v_{i,j}$.

We present how we handle this issue. First, we utilize the web crawler to scrape those projects that are relying on original projects. More specifically, we use the Scrapy framework [10] and encapsulates project p_i into an HTTP Get request and sends it to GitHub. The URL is "*https://github.com/* p_i */network/dependents*", where p_i is the name of the project. We set the timeout to 20 ms and attempt to query up to 5 times if it fails. Once receiving its response, we utilized BeautifulSoup [11] to parse HTML pages in the response packets. In the web interface, the project dependents are stored in the

[1] GitHub provides the web interface to show packages and applications that are using a targeted project.



s

Fig. 4: Three categories of projects for the partial dataset.

tag with a class name "*Boxrow dflex flex-items-center*". Second, we utilize the BeautifulSoup's parser to directly extract the list of the owner name/respostory that is reusing the shared library p_i. Third, we further use the GitHub APIv4 [12] or build files to access libraries that dependent project in the list uses ($* \rightarrow p_i$). We use the GraphQL schema [13] to query a project's dependency libraries. We set up the API key token and used the URL "*https://api.github.com/graphql*" to encapsulate a query. For dependent projects in this list, we can obtain the dependency relations between it and the shared project p_i.

Range for the partial dataset. Here, we use three categories of projects to represent the partial dataset, as shown in Figure 4, including: (1) *Class A*: a project that involves a vulnerability. (2) *Class B*: a project that directly uses the project library from Class A ($\exists p_i$, $p_j \rightarrow p_i$, $p_j \in$ Class A). (3) *Class C*: a project that indirectly uses the project library from Class A ($\exists p_i$, $p_j \rightarrow p_i$, $p_j \notin$ Class A). Note that every project in Class A has reported a publicly disclosed vulnerability. We leverage the breadth-first search (BFS) to collect dependency projects from the GitHub website. We use two parameters (N and B_{max}), where N represents the BFS-iteration number to collect relevant projects, and the parameter B_{max} represents how many dependency projects are discovered at maximum per node within one BFS-iteration.

C. Identifying Vulnerabilities with Dependency Projects

As mentioned before, those datasets are collected from three separate and independent data sources (NVD, Maven, and GitHub), and they use different formats with inconsistent content. Our objective is to identify whether a project involves a disclosed vulnerability. In this work, we propose three detection methods, including the similarity matching, fallback URL, and third-party sources.

1) Similarity Matching: A vulnerability report includes two information sources for matching dependency projects: CPE and a plaintext description. CPE provides platform information (e.g., operating systems, hardware, and software) that has been detected with a vulnerability. The plaintext is to describe the vulnerability information in the natural language, which may include software, vendor, and version information. Typically, there are two methods to identify a vulnerability with a dependency project: the regex matching and name entity recognition (NER). Regex matching finds the pre-defined and particular strings (e.g., project names) that match the regular expression. NER classifies the words in the natural language into class tags for matching with dependency projects. However, we observe that directly using those two methods has poor performance for identifying vulnerabilities with dependency projects.

Instead, we propose to use the similarity matching method. We use statistical text features to extract the most relevant words of the plaintext description and CPE. Given a word, we utilize four features, including (1) position, (2) word frequency, (3) relatedness, and (4) difference in various sentences. We calculate every word's score based on those features and rank them on the list. After that, we use the Levenshtein distance to calculate the similarity between words and dependency projects from the Maven and GitHub. We eliminate those with a low similarity score and select a suitable project for the vulnerability. If a dependency project is matched with the CPE string, we believe that it is associated with a publicly disclosed vulnerability.

2) FallBack URL: Typically, the NVD vulnerability includes several reference links to technical forums, vendor websites, and security vulnerability databases. Those reference URLs can be used to identify whether a project involves a vulnerability. We propose the fallback method to connect the information in those separate datasets.

The GitHub URL. We find that an NVD vulnerability typically has GitHub commit reference URLs to represent the vulnerability patch. Normally, a "git" link has the owner/organization name and repository/project name, which belong to the web server interface to the code repository of projects on GitHub. We used the regex "git" to extract the patch reference URL, and extracted it project names for matching with projects from the GitHub. If matched, the project involves a publicly disclosed vulnerability.

The Maven URL. In contrast to GitHub, an NVD vulnerability does not have any reference URL from the Maven. In our investigation, we check reference URLs of 3,000 NVD vulnerabilities and find that no Maven URL exists.

We observe that Maven works similarly as NVD, and both provide an index repository. An NVD vulnerability report contains the reference URLs from forums, vendor websites, and security vulnerability databases. A Maven project's build file also includes reference URLs (<url> field) from online sources. In short, we compare the reference URLs from the NVD vulnerability report with URLs from the Maven project's build file. If the reference URLs are overlapped, the Maven project involves a publicly disclosed vulnerability.

3) Third-party Sources: In the security community, mapping vulnerabilities with projects from discrepant datasets is a not-yet-solved problem. This is because the string and content are inconsistent and varied. Many professionals manually inspect, collect, and calibrate the mapping relations between vulnerabilities and projects, requiring technical background and time cost. Several calibrated datasets are open and distributed to the public. Specifically, we crawled down four third-party sources, including advisory database [14], CVE-search project [15], and manually-curated dataset [16]. Further, we manually filtered out the redundancy and overlap items. We utilized third-party sources to identify whether a dependency project involves a vulnerability.

IV. Dependency Project Graph

In this section, we first present the underlying risks caused by the vulnerabilities of dependency projects. Then we present

the design of the PDGraph, which generates a directed project dependency graph. Further, we utilize the version semantics of dependency projects to detect insecure edges.

A. *Dependency Risks*

Typically, a CVSS score is to assess vulnerability impact. Yet, CVSS does not involve any dependency factor that exists across many software ecosystems (NuGet [17], Conan [18], Maven, and Gem [19]). We reveal four underlying risks, through which adversaries can significantly reduce the difficulty and overhead of compromising systems.

Risk 1: The number of dependency projects. When a project depends on many other libraries/projects, it is more vulnerable to malicious attacks. If any reused library suffers a vulnerability, the project will likely be under a similar risk. We use each node's indegree to measure how many libraries ($x \rightarrow p_i$) a project reuses.

Risk 2: The number of dependent projects. When a library is reused by projects, the dependent number represents how many projects rely on it. When a reused library involves a vulnerability, its dependent projects will likely suffer a similar risk. Note that a dependent project might never reuse the vulnerable code block in the shared library. However, if the dependent number is large, adversaries will find a wide range of vulnerable systems, reducing the difficulty of locating a victim target. We use the outDegree to indicate the number of other projects ($p_i \rightarrow x$) that reuse its library p_i.

Risk 3: The length of dependency path. The dependency path is to measure the underlying transmission of a vulnerability. When any project p_i in the dependency chain involves a vulnerability, attackers could exploit the underlying target chain for malicious purposes. We used the longest path to measure the maximum number of dependent projects indirectly linked to the project that has suffered a vulnerability.

Risk 4: Circular dependencies. If a project p_1 depends upon a project p_2, ..., which depends on a project p_1, we call it a circular dependency. Cyclic dependencies may lead to logical contradictions that can cause deadlocks. The dependency structure between packages must be a directed acyclic graph. However, in our research, the dependency dataset contains many circular dependencies, which affect the software updates. When a project is required to update to a new version, other projects in the circle need to update simultaneously. The operation of a security patching may become complicated when a project in the cyclic contains a bug that is more vulnerable to malicious attacks.

B. Dependency Graph Generation

Project dependency graph (PDGraph) $G(V, E)$ is a collection of nodes $V = \{v_1, ..., v_n\}$ and edges $E = \{e_{i,j},\}_{i,j=1}^{n}$, where a node v_i is a project $p_i \in V$, and an edge $e_{i,j}$ is the dependency relationship between two nodes. Based on the collection dataset, we describe the methodology to generate the project dependency graph. The input is the dataset consisting of tuples of ($p_i \rightarrow p_j$, $v_{i,j}$) and the set of projects with version information. The output is a directed graph, $G = \{V, E\}$. We built the graph, whose vertices are unique projects, and all edges are assigned to zero. If there exists a dependency relationship between two projects, the edge weight is assigned to one, $w_{i,j} = 1$. If two projects p_i and p_j have no dependency, then their edge weight is zero, $w_{i,j} = 0$. Note that the version $v_{i,j}$ in the edge $e(i, j)$ is used to represent the dependency version requirement between two projects p_i and p_j.

We ran common graph algorithms over PDGraph including Strongly Connected Components (SCC) and Minimum Feedback Arc Set (MFAS). SCC is used to simplify PDGraph for improving efficiency before running any graph algorithm. We further divided the graph into SCCs and then discarded these trivial SCCs (a trivial SCC consists of a single node). This is because project developers might not write the build file in the source-code repository. In addition, we leverage MFAS to remove circular edges in every SCC, while keeping the hierarchy structure of the graph. However, MFAS is an NP-complete problem [20] without a polynomial-time approximation solution. The reason is that an edge could be a part of multiple simple cycles, and the removal of one edge might break other cycles at the same time. Enumerating all cycles of the graph is computationally expensive for large graphs. Instead of enumerating all cycles, we used a greedy algorithm to remove edges in the graph [21, 22, 23], which guarantees the hierarchy structure of the graph based on the local information. The greedy rule is simple: One should always select the node with the highest ratio d^{in}/d^{out}, and then remove all of its out-edges. Here, d^{in} is the node's indegree, and d^{out} is the node's outdegree. We sorted the nodes in SCC in descending order of the ratio d^{in}/d^{out}. For the node with the highest ratio, we removed all of its out-edges. For every SCC, we generated its directed acyclic graph (DAG) in the hierarchy structure.

C. Insecure Edge Detection

When reused components/libraries have security vulnerabilities or flaws, their dependent projects may suffer similar security risks or threats. For any project i, we divide all of its versions into two categories: secure versions V_i^s and vulnerable versions V_i^{vul}. If a project relies on vulnerable versions of libraries, the dependency is insecure or vulnerable, indicating that the project might suffer similar underlying risks or threats as its reused libraries did. The detection rule is described as follows. If the dependency version belongs to vulnerable versions, we mark the dependency as an insecure edge; otherwise, we mark it as a secure edge. We use the version semantic to detect insecure edges.

We have version information for all projects with vulnerabilities. We use the NVD to identify vulnerable versions and use the web crawler to collect all version information of target projects (detailed in Section-IV-B) for projects with vulnerabilities. For projects on vulnerable dependency, we just parse their build files (pom.xml), which exist on the newest versions of those projects. Given a project, its build file claims the dependency relationship for its latest version.

We first utilized the NVD vulnerability to collect vulnerable versions for software projects. The vulnerability information is typically disclosed and exposed to the public through a central data repository (e.g., NVD [2]). We extracted public vulnerabilities to find corresponding software projects and then extracted their vulnerable versions, denoted as V_i^{vul}. For other projects, we just parsed their build files (detailed

TABLE II: Comparison operator for version semantic of software products.

Operator	Description
$\sim=$, $\hat{}$ $\sim$, $\sim>$	return true if candidate version is compatible
$==$, $!=$	equality comparison; exclusion operator
$<=$, $>=$ $<$, $>$	return true if the comparison matches any version number

in Section III-B), which exist on the latest versions of those projects.

Then, we need to determine whether a dependency version belongs to the vulnerable version set. In software engineering, associated vulnerable versions may involve multiple versions in the version semantic scheme [24][25]. Dependency versions might involve multiple software versions, rather than a single version. Therefore, we utilized the version operator to compare the dependency version and the vulnerable version to detect insecure edges. Table II lists version operators for version semantics of software products. The compatible operators include $\sim=$, $\hat{}$, $\sim$, and $\sim>$, which return versions compatible with the specified version. The version matching operators include $==$ and $!=$, which return true if the versions are exactly equal or exclusive. The ordered comparison operators include $<=$, $>=$, $<$, and $>$, which return versions when the comparison is satisfied. We used the semantic version regex to compare requirement versions in dependency edges with vulnerable version sets, and then we detected insecure edges in PDGraph.

V. Experimental Evaluation

In this section, we measure the overhead of our PDGraph. Then, we validate the effectiveness of our data collection methodology.

A. Implementation and Overhead

We implemented a prototype of PDGraph as a self-contained piece of software based on open-source libraries. (1) For the data collection, we utilize the Scrapy framework [10] to crawl web interface and BeautifulSoup [11] to parse the HTML files for dependent projects. (2) We use the GraphQL schema [13] to collect relevant metadata of dependency projects on GitHub. (3) For identifying vulnerability with a project, we use the YAKE library [26] to calculate statistical features and Levenshtein distance. (4) We implemented the graph algorithms based on the NetworkX [27] package, which is used to create and manipulate the structure and functions of a graph. (5) We leveraged the Python package [28] to detect insecure edges in PDGraph, which is implemented to specify the versions of dependency based on the standard semantic version scheme [24, 25]. Our module supports the version form ("X.Y.Z") and the comparison operators ("<", "=", "~", and "^"). Specifically, we wrote a custom Python script to pipeline all modules for our PDGraph, including the data collection, graph algorithms, and edge detection.

In the experiment, we ran our prototype on Amazon Elastic Container Service (ECS) with CentOS Linux, Intel Xeon(R) 2.5GHz, 2G memories. We chose 1,000 nodes and 17,000 edges to measure the overheads in our PDGraph. Table III lists the overheads of our PDGraph, including the data collection and graph algorithms. The major time cost lies in that PDGraph needs to use a web crawler to access the projects and their dependency information. This overhead provides evidence that PDGraph can scale to a large amount of project dependency analysis in the wild.

TABLE III: Component overheads given 1,000 nodes and 17,000 edges.

	Memory	Time Latency (s) / Percentage
Data collection	57.6MB	1,966 / 87.4%
Graph algorithm	574MB	284 / 12.6%

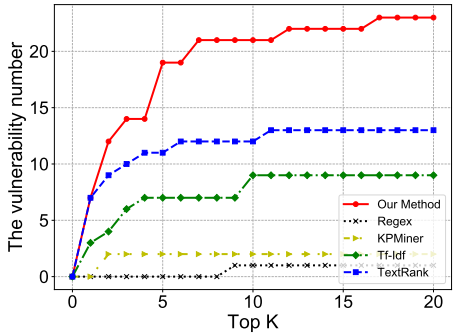


Fig. 5: Performance for identifying NVD vulnerabilities with Maven projects.

B. Validation

Full Data Collection. We crawled down 4,465,344 "pom.xml" files from the Maven central repository [8]. Not all build files provide dependency relationship. We found that only 893,571 files provide 5,778,318 dependency relations. In Maven, a project in a build file is represented in the form of "groupdId:artifactId:version." When the "groupdId:artifactId" of projects is the same, we merge those projects and store version information in the edge. Finally, the full dataset contains 190,874 maven projects (i.e., nodes) and 814,331 dependency relations (i.e., edges).

Partial Data Collection. We utilize the breadth-first search to collect partial data, which has two parameters, N and $B_{\max}$, indicating the iteration number and the maximum number of dependency projects per node within one iteration. In our research, we adopted the parameter setting of ($N = 10$ and $B_{max} = 400$) to discover projects and their dependency relationships for generating PDGraph. In total, the partial dataset from GitHub collects 146,541 projects and 571,007 dependency relations.

Validating Project Dependency. The labeled data is that we chose sample sizes (typically of 100) of vulnerabilities with projects. We used four baseline approaches to compare with our similarity matching method as follows: (1) the simple regex, (2) KP-Miner [29], (3) TF-IDF, and (4) TextRank. Figure 5 shows the performance of identifying NVD vulnerabilities with Maven projects. Compared with baseline approaches, our similarity matching would identify 27 vulnerabilities with dependency projects. Further, we use the

TABLE IV: The overview of the data collection for PDGraph.

	Node	Edge	Dependency Versions per edge
Maven	190,874	814,331	2.41
GitHub	146,541	571,007	2.32
Total	337,415	1,385,338	

TABLE V: The overview of dependency relations with projects vulnerabilities.

	Vulnerability	Project in Class A	Project in Class B	A→ B
Maven	503	174	32,381	56,993
GitHub	3,326	840	35,425	46,084
Total	3,829	1,014	67,806	103,077

fallback URL to find vulnerabilities with projects. Our fallback URL archives 100% accuracy for identifying vulnerabilities with GitHub projects because an NVD vulnerability would involve a GitHub commit reference URL. For Maven projects, our fallback URL identifies 30 vulnerabilities with dependency projects.

Combining vulnerability reports and projects from different databases is still a not-yet-solved problem. As best we know, manual inspection is the only reliable manner in the security community. We propose three detection methods (the similarity matching, fallback URL, and third-party sources), which would reduce the time cost and technique background to identify vulnerabilities with dependency projects.

VI. Quantitative Security Analysis

In this section, we quantitatively analyze the vulnerability impacts of PDGraph, including insecure edges and dependency risks.

A. Measurement

Landscape. We first provide the landscape of the PDGraph. Overall, we discovered 146,541 projects with 571,007 dependency relationships from the GitHub website, and 190,874 projects with 814,331 dependency relationships from the Maven. Table IV lists the number of nodes and edges associated with dependency versions. Note that, in our analysis, we extract all released versions of nodes and dependency versions of edges. For the dependency edge $e_{i,j}$, the $v_{i,j}$ is the p_i version used by project p_j. We observe that the average number of required versions is nearly 2.41 (Maven) project, and the average number of Github is 2.32. When a project reuses a library, software developers might claim the range of its required version in the build file.

Project dependency and vulnerabilities. Table V lists across edges between two nodes, where a node involves a vulnerability. We found 503 CVE vulnerabilities associated with 174 Maven projects and 3,326 CVE vulnerabilities related to 840 GitHub projects. Those projects are classified into Class A. When a project uses a library in Class A, we classify it into Class B. There are 32,381 Maven projects and 35,425 GitHub projects directly dependent on the projects with publicly disclosed CVE vulnerabilities. The across-edge (A → B) represents the dependency relationship between a project of Class A and one of Class B. Those across edges indicate the possibility that the projects of Class B might be affected by the vulnerabilities reported in the projects of Class A. We found that Maven has 56,993 across edges and GitHub has 46,084 across edges.

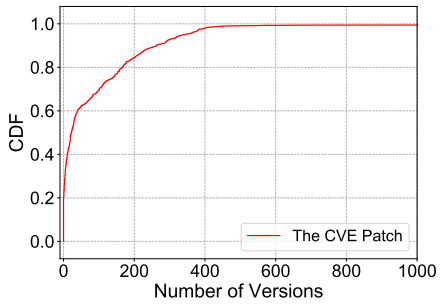


Fig. 6: The CDF of the version number per vulnerability patch.

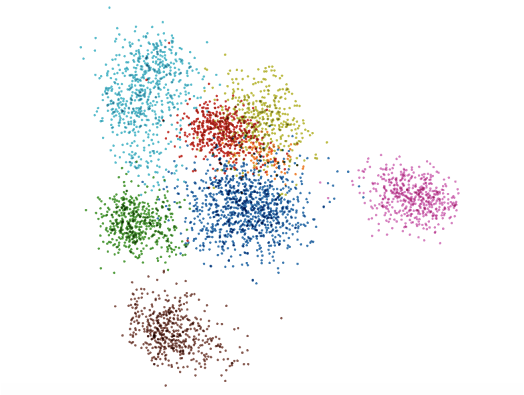


Fig. 7: The visualization of the learned embeddings of a set of randomly chosen projects.

The version number of a CVE vulnerability. When a vulnerability is found, the software vendor should provide a patch to fix it, which leads to the change (remove or add) in source code of the project. The vulnerability patch causes the change of software version in a formal specification ("X.Y.Z"). Figure 6 depicts the distribution of the number of versions for those patches. Obviously, a patch is only related to the "Z" number, which represents the modification for fixing a bug/vulnerability in software development. There is a long tail effect for the distribution of the version number involved by a patch. We observe that 60% of CVE patches involve fewer than 45 software versions, and 80% involve fewer than 170 versions. For a project with a vulnerability, its vulnerable versions also involve dozens of versions, rather a single version.

Visualization. Furthermore, we plotted PDGraph by employing an information visualization tool. We used BigGraph [30] (network embedding [31, 32]) to characterize the graph structure with respect to every node. Intuitively, two nodes with a similar/identical structure should have a very close distance, while nodes with different structures are far apart. After that, we projected those nodes into a 3-D plane via principal component analysis (PCA). The TensorBoard-visualization [33] tool is used to visualize embedding nodes.

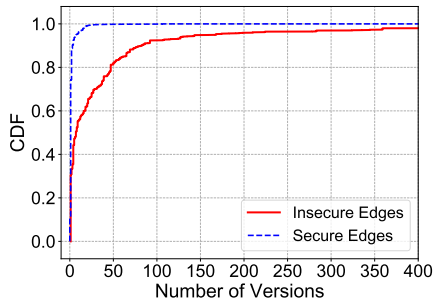


Fig. 8: The CDF of version numbers per dependency edge.

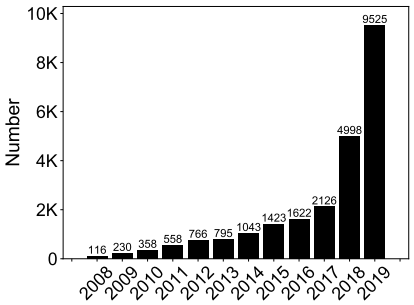


Fig. 9: The number of insecure edges along time.

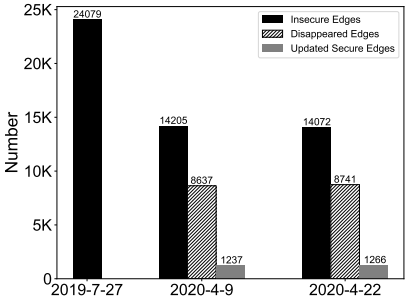


Fig. 10: The changing of insecure edges after eight months on the GitHub.

TABLE VI: The insecure and secure edges for across edges A→ B.

	Secure Edge	Insecure Edge	Relevant Projects from Class B
Maven	13,491(23.7%)	43,502(76.3%)	26,986
GitHub	22,005(47.7%)	24,079(52.3%)	15,455

We used TensorBoardX supported by PyTorch to perform the project dependency visualization. The results are shown in Figure 7 (10,000 nodes), where one color represents one project group.

B. Insecure Dependency Relations

As we mentioned before, when a project uses existing code from libraries and frameworks of other projects, it could suffer a similar security risk of vulnerable reused components. We detected an insecure edge if the dependency version belongs to the vulnerable versions. Table VI lists the insecure and secure edges among A→ B for the Maven and GitHub. In total, we found 24,079 insecure edges from 15,455 Github projects within Class B, and 43,502 insecure edges from 26,986 Maven projects. In other words, nearly 76.3% of across edges are insecure for the Maven dataset, and 52.3% of across edges are insecure for the GitHub dataset. Note that those insecure dependencies imply that the related projects are still using the outdated and vulnerable versions of reused libraries. Although updating to the latest version is the preferred solution, many software developers are not aware of the security issues and fail to do so.

What is the difference between the version number of secure and insecure edges? As mentioned before, developers might claim the range of the required versions in the build file. Figure 8 depicts the CDF of the number of versions per dependency edge. The red curve represents the case of insecure edges, and the blue curve represents the case of secure edges. Nearly 88% of secure edges have fewer than 3 versions, while more than 60% of insecure edges have more than 20 versions. It is obvious that insecure edges have many more dependency versions than secure edges. The reason is that a secure version is usually much closer to the latest version of the software.

What is the creation time for a dependency edge? Figure 9 depicts the approximate timeline of the dependency edges. The X-axis is the timeline grouped by month, and the Y-axis is the number of edges being created. Since we could not obtain an accurate creation time of an edge that is established between two projects, we used the creation time of the dependent project to represent the lower bound of the creation time of the edge. We found that the number of insecure edges has been increasing along time. There were 116 insecure dependencies for projects in 2008, but the number increased to 9,525 in 2019. There are two probable causes: (1) the number of vulnerabilities and the number of projects have both been increasing, and (2) software developers lack security awareness.

Will developers update the insecure dependency version to the secure version in the future? When a vulnerability is reported, developers should provide a patch to fix it. After patching, the version number increases, and its dependent projects should update it to fix the insecure dependency. Figure 10 plots the dynamics of vulnerable dependency edges on the GitHub from July 2019 to April 2020. For those 24,079 insecure edges at the beginning, only 1,266 insecure edges were fixed and updated to secure edges. Meanwhile, there are 8,741 insecure edges disappeared during this period. The disappearance of edges (8,741) is caused by two factors. First, 2,681 insecure edges disappeared because corresponding nodes became unavailable and emptied. Second, 6,060 insecure edges disappeared because their dependency relationships no longer existed. However, the majority of insecure edges (14,072) still remain insecure.

Although updating to the latest version is the recommended solution, there are three probable reasons preventing project developers from updating in a project's development. First, developers might lack the security awareness of the software product. Second, the patch-version release time is always later than when its vulnerability or bug is detected. When bugs and vulnerabilities of projects are detected, developers must spend time fixing them and release the patch version. Third, product stability and compatibility are the top priorities for developers who might have concerns about potential instability and incompatibility caused by updates.

C. Dependency Risks

We conducted a quantitative analysis of dependency risks on the PDGraph.

(1) The number of dependency projects. Given a project p_i, its InDegree represents how many libraries ($x \rightarrow p_i$) it reuses. The higher the InDegree of p_i, the more that other

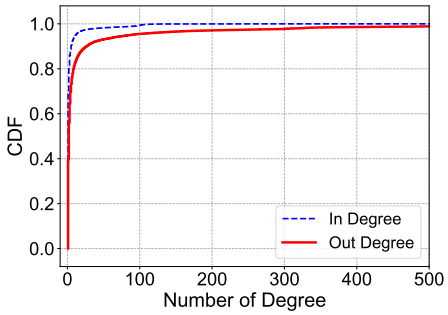


Fig. 11: The cumulative distribution function (CDF) of node degree in the graph.

projects' libraries are reused by p_i. Figure 11 depicts a blue dot line to represent the distribution of the node InDegree. The InDegree has an average of 4 with a 13.34 standard deviation. The distribution is a long tail effect, where the InDegree of 80% of nodes is less than 3. Meanwhile, there are 929 projects whose InDegree is larger than 100, implying that they heavily reuse the libraries from others. If any reused component involves a security problem, the project may also suffer similar issues, leading to a more substantial security risk.

(2) The number of dependent projects. Given a project p_i, its OutDegree indicates the number of other projects $(p_i \rightarrow x)$ that reuse its library. The higher the OutDegree of p_i, the more that dependent projects reuse the p_i's library. The red line in Figure 11 depicts the distribution of the node OutDegree, where the OutDegree of 85% of nodes is less than 15. The OutDegree has an average of 23.41 with a 111.91 standard deviation. There are 1,095 projects whose OutDegree is larger than 100, indicating that they are heavily relied by other projects. If those projects suffer a vulnerability or bug, their dependent projects might suffer the same problem.

We further identified the correlation between associated vulnerabilities and a node's OutDegree. Given a vulnerability in a project (p_i), all of the related projects under the edge $(p_i \rightarrow x)$ are in the range of directly affected projects. If a vulnerability is associated with a higher outdegree of a node, it has more dependent projects. We further lists the top 5 OutDegree projects with their aggregation CVE number, including Brace Expansion, Rails, YARD, IPython, and jQuery.

(3) The length of dependency path. We use the longest dependency path (LDP) to represent the dependency chain in the PDGraph. Given a vulnerability, the LDP represents the maximum number of underlying indirectly affected projects in PDGraph. Further, we list the top 5 longest paths of projects with CVEs in Table VIII. The longest path is 102 for Brace Expansion with CVE-2017-18077, and the 2nd longest path is 83 for Rack with 4 different CVE IDs.

(4) Circular dependencies. There are two types of circles, self-loop and two-way loop. Table VII lists these two types of special edges. A self-loop edge (relationship) is a link that connects a project to itself, $\exists\ p_i \rightarrow p_i$. We found 1,618 self-loop edges, e.g., Sinatra, Netty, and Raven-Ruby. A two-way loop is that the two edges connect two projects in reverse directions, $\exists p_i \rightarrow p_j$ and $\exists p_j \rightarrow p_i$. There are 2,691 two-way loops. For instance, IPython and Matplotlib both rely on the other's libraries. We further investigated those projects with two-way loops, and found that most of them are popular libraries being widely used.

TABLE VII: Circle dependencies: self-loop and two-way loops.

	Self-loop	Two-way loop
Number	1,618	2,691

TABLE VIII: The top 5 longest paths for CVE IDs in the PDGraph.

CVE ID	Length	Project Name	Base Score
CVE-2017-18077	102	Brace Expansion	5.0
CVE-2013-0183	83	Rack	5.0
CVE-2013-0262			4.3
CVE-2013-0263			5.1
CVE-2012-6109			4.3
CVE-2017-14064	83	JSON for Ruby	7.5
CVE-2017-17042	82	YARD	5.0
CVE-2013-1800	75	Crack	7.5

VII. Discussion

While PDGraph is the first large-scale project dependency graph built for security analysis, there are several limitations when we use it.

Manual Factors. PDGraph gathers project dependency relationships by reading the build files in the source-code repository of a software project. However, the build file (e.g., pom.xml and Gemfile.lock) is manually written by software developers, which might have inconsistencies and errors. The only way to eliminate such a manual error is to understand the source-code of a software project and identify the erroneous dependence, which is time-consuming and requires professional background knowledge. It is impossible to inspect all build files on a large scale for identifying dependency relationships. In addition, developers may forget to write a build file, leading to the missing of dependency data.

Coverage Limitation. We rely on the Maven and GitHub to collect projects and their dependency relationships. So far, our PDGraph does not support C and C++. However, there are many vulnerabilities exposed to projects written in C and C++. In our future work, we will investigate other project management tools for vulnerability dependencies, such as Conan [18] for C/C++, NuGet [17] for C#, and Gem [19] for Ruby.

Vulnerable Dependency. The existence of vulnerable dependency is an indicator to help professionals detect new vulnerabilities in their software projects. However, even if a project has a vulnerable dependency, it does not imply that the project will certainly suffer the corresponding vulnerability. Some vulnerabilities are transitive, while other vulnerabilities are not. So far, our DPGraph has no automatic mechanism to identify whether a project indeed suffers a vulnerability when it has a vulnerable dependency. In the future, we will leverage the code property graph [34] to determine whether

a vulnerability is transitive and could be exploited among the vulnerable dependency.

Vulnerability Impact. Not all dependencies are the same for a vulnerability. The vulnerable code may never be used or trigged by dependent projects. For instance, a python library to pretty-print output could have a vulnerability that allows code execution (e.g., through an eval statement) in a specific parameter of one function. Then, a developer writes a command-line utility that uses this library to print messages with colors. Some dependency relations may cause vulnerability propagation on dependent projects, while some may not.

VIII. Related Work

Security Audit. In current defensive programming paradigms, a security audit might be considered as the best practice to reduce software flaws/vulnerabilities before code is released. Several prior works leverage the project version information for assisting the security audit. Zimmermann et al. [35] mined the version archives of the software for identifying whether a code change might introduce a flaw or bug. Ozment and Schechter [36] examined the version history of the OpenBSD operating system and reported that its foundational vulnerabilities have a median lifetime of at least 2.6 years. There are several prior works [37, 38] that utilize machine learning algorithms to identify vulnerable components from open-source projects. Scandariato et al. [37] studied and explored features of a bug or flaw (e.g., code churn size and exposure time) to audit source code repositories. Perl et al. [38] proposed VCCfinder, which utilizes metadata features of a project on GitHub, including programming language, author credit, keyword, code size, and star and fork counts. By contrast, we used the version scheme to detect vulnerable dependencies between different projects and found 67,581 edges are still under versions with publicly disclosed CVEs. Our PDGraph can assist developers to audit their source code when their projects' components/libraries have underlying security risks.

Code Reuse. The reused code may bring underlying security risks when the reused components have security flaws or vulnerabilitiess. Nappa et al. [39] demonstrated security threats caused by shared libraries among 10 desktop applications on Windows. Göktas et al. [40] introduced code-reuse attacks and developed exploits against Asterisk and Firefox applications. Agadakos et al. [41] identified and removed the unused code within shared libraries if source code was available. Duan et al. [42] utilized the patches from open-source software to protect vulnerable mobile applications. By contrast, our PDGraph is complementary to these previous works of defending against code-reuse attacks.

Vulnerability Information. Once a vulnerability is detected and disclosed, the professionals and vendor developers should release a security patch and update a new version to remediate the vulnerability. While the outdated versions of the project still suffer the vulnerability, the latest version is more secure. Several prior works leverage project versions for vulnerability detection and analysis. Neuhaus et al. [43] proposed combining version archives and the vulnerability database to detect vulnerable components in the Mozilla project. Edwards and Chen [44] explored the version archive of Sendmail, Postfix, Apache HTTPd, and OpenSSL, and utilized the CVE information to identify exploitable bugs/vulnerable components. Meneely et al. [45] used the version control tool (Git, SVN, or CVS) to trace vulnerabilities in the Apache HTTP server. Feng et al. [46] utilize the security reports to analyze vulnerabilities from Internet-of-Things (IoT) systems. Gkortzis et al. [47] provided a dataset that integrates the vulnerabilities from NVD and project versions to the security community. Instead of focusing on a small number of software projects, PDGraph is the first work to investigate project versions and corresponding dependencies for vulnerability analysis on a large scale.

Vulnerability Assessment. CVSS comprises more than a dozen key characteristics, e.g., attack complexity, privilege required, user interaction, and confidentiality. Without the professional background or proper guidance, it is easy to miscalculate a CVSS score for a vulnerability. Prior works [48, 49] found that it is common to measure false positives of scoring CVSS. Allodi et al. [48] conducted a survey, in which participants responded to a set of CVSS questions, but only 57% of questions were accurately answered. In another survey study, Spring et al. [49] found that nearly half of the participants had four points of deviation between their answers and the accurate CVSS scores. Note that CVSS has a score range of zero to 10. Sabottke et al. [50] proposed using the textual information on Twitter for assessing vulnerability impacts. They used the linear support vector machine (SVM) classifiers to detect tweets related to vulnerabilities and real-world exploits. Differing from prior works, our work provides an alternative approach to assessing the vulnerability impact in the wild. We propose to score a vulnerability of a project based on the degree of its interdependency.

IX. Conclusion

In this paper, we conducted a large-scale empirical study on project dependency regarding security vulnerabilities. We developed a project dependence graph called PDGraph for analyzing 337,415 software projects and 1,385,338 dependency relationships collected from the Maven and Github. Our analysis is focused on the security impact of publicly known security vulnerabilities propagated by project dependencies, providing a new perspective for assessing security risks in the wild. We found that today's projects indeed use code from existing open-source libraries or frameworks. On average, each project has 23 dependent projects, and there is a positive correlation between project dependency and associated publicly disclosed vulnerabilities. In addition, we have monitored the dynamics of 67,581 insecure edges caused by the vulnerable versions of re-used libraries for an eight-month observation period. We found that only 1,266 insecure edges from GitHub were fixed and updated to secure versions. Further, we conducted a quantitative analysis of four dependency risks on the PDGraph.

X. Acknowledgments

We are grateful to our shepherd Marco Vieira and anonymous reviewers for their insightful feedback. The work was partially supported by National Key R&D Program of China (No. 2020YFB2103802 and No. 2018YFB0803402), and National Natural Science Foundation of China (No. 61972024). Jiqiang Liu is the corresponding author.

REFERENCES

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

[2] (2007) NVD, U.S. National Institute of Standards and Technology National Vulnerability Database. https://nvd.nist.gov/home.cfm.

[3] A. S. Foundation. (2004) Apache Maven is a software project management and comprehension tool. . https://maven.apache.org/.

[4] S. Wang. (2021) The dataset for PDGraph. . http://www.infolab.top/research/Vul-graph/.

[5] Lodash. (2019) Prototype pollution attack (lodash). . https://hackerone.com/reports/310443.

[6] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 430–441.

[7] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[8] A. Lucene. (2010) Central repository provides an index that is built using Maven Indexer. https://maven.apache.org/repository/central-index.html.

[9] GitHub, "Github universe is a conference for the builders, planners, and leaders defining the future of software." https://githubuniverse.com/2018/, 2018.

[10] Scrapy. (2008) A Fast and Powerful Scraping and Web Crawling Framework. https://scrapy.org.

[11] B. Soup. (2012) package for parsing HTML and XML documents. https://www.crummy.com/software/BeautifulSoup/.

[12] G. Developer. (2018) GitHub REST API v3. . https://developer.github.com/v3/.

[13] GraphQL. (2018) An open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. . https://graphql.org/.

[14] GitHub. (2018) Advisory Database. . https://github.com/advisories.

[15] A. Dulaunoy, P.-J. Moreels, and R. Vinot. (2017) Cve-Search Project. . https://www.cve-search.org/about/.

[16] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," 2019. [Online]. Available: https://arxiv.org/pdf/1902.02595.pdf

[17] Microsoft, "Nuget is the package manager for .net." https://www.nuget.org/, 2020.

[18] Conan.io, "Conan, the c/c++ package manager." https://conan.io/, 2020.

[19] R. community, "Rubygems.org is the ruby community's gem hosting service." https://rubygems.org/, 2020.

[20] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[21] P. Eades, X. Lin, and W. F. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, vol. 47, no. 6, pp. 319–323, 1993.

[22] C. Demetrescu and I. Finocchi, "Combinatorial algorithms for feedback problems in directed graphs," *Information Processing Letters*, vol. 86, no. 3, pp. 129–136, 2003.

[23] A. Baharev, H. Schichl, A. Neumaier, and T. Achterberg, "An exact method for the minimum feedback arc set problem," *University of Vienna*, vol. 10, pp. 35–60, 2015.

[24] P. 440, "Version identification and dependency specification," https://www.python.org/dev/peps/pep-0440/, 2013.

[25] T. Preston-Werner. (2014) Semantic versioning 2.0.0. [Online]. Available: https://semver.org/

[26] R. Campos, V. Mangaravite, A. Pasquali, A. Jorge, C. Nunes, and A. Jatowt, "Yake! keyword extraction from single documents using multiple local features," *Information Sciences*, vol. 509, pp. 257–289, 2020.

[27] NetworkX. (2002) A Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. https://networkx.github.io/documentation/stable/index.html.

[28] P. P. Authority. (2015) A core requirement of dealing with dependency is the ability to specify what versions of a dependency. https://packaging.pypa.io/en/latest/specifiers/.

[29] S. R. El-Beltagy and A. Rafea, "Kp-miner: Participation in semeval-2," in *Proceedings of the 5th International Workshop on Semantic Evaluation*, ser. SemEval '10. USA: Association for Computational Linguistics, 2010, p. 190–193.

[30] A. Lerer, "Pytorch-biggraph (pbg) is a distributed system for learning graph embeddings for large graphs." https://github.com/facebookresearch/PyTorch-BigGraph, 2019.

[31] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[32] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo, "struc2vec: Learning node representations from structural identity," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 385–394.

[33] Tensorflow. (2017) TensorBoard: TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard/.

[34] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[35] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999460

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

[37] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[38] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi,

K. Rieck, S. Fahl, and Y. Acar, "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*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813604

[39] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 692–708.

[40] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 227–242.

[41] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[42] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries." in *NDSS*, 2019.

[43] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315311

[44] N. Edwards and L. Chen, "An historical examination of open source releases and their vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 183–194.

[45] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013, pp. 65–74.

[46] X. Feng, X. Liao, X. Wang, H. Wang, Q. Li, K. Yang, H. Zhu, and L. Sun, "Understanding and securing device vulnerabilities through automated bug report analysis," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 887–903. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/feng

[47] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "Vulinoss: A dataset of security vulnerabilities in open-source systems," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 18–21. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196454

[48] L. Allodi, S. Banescu, H. Femmer, and K. Beckers, "Identifying relevant information cues for vulnerability assessment using cvss," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 2018, pp. 119–126.

[49] J. Spring, E. Hatleback, A. D. Householder, A. Manion, and D. Shick, "Towards Improving CVSS," in *Carnegie Mellon University, WHITE PAPER*, 2018.

[50] C. Sabottke, O. Suciu, and T. Dumitraş, "Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-world Exploits," in *the 24th USENIX Security Symposium, Berkeley, CA, USA, 2015*, 2015, pp. 1041–1056.