PKVIC: Supplement Missing Software Package Information in Security Vulnerability Reports

Jinke Song, Qiang Li, Haining Wang, and Jiqiang Liu

***Abstract*—Nowadays security vulnerability reports contain commercial vendor-centric information but fail to include accurate information of open-source software packages. Open-source ecosystems use package managers, such as Maven, NuGet, NPM, and Gem, to cover hundreds of thousands of free code packages. However, we uncover that vulnerability reports frequently miss the vulnerable software package information when the software package comes from open-source ecosystems. To fill in this gap, we propose a framework called PKVIC (software package vulnerability information calibration), as the first tool to automatically associate security vulnerability reports with affected software packages from different open-source ecosystems. Specifically, PKVIC designs an ecosystem classifier to determine which ecosystem a vulnerability report belongs to. From the reports written in natural language, PKVIC extracts the entities closely related to software names in ecosystems. To efficiently and accurately locate the affected software packages from millions of packages, we propose a recursive traversal method to generate the package identifier based on the naming scheme and candidate named entities. We implemented the prototype of PKVIC and conducted comprehensive experiments to validate its efficacy. In particular, we ran PKVIC over 421,808 vulnerability reports from 20 well-known sources of security vulnerabilities and identified 11,279 unique vulnerability reports that affected 2,703 open-source software packages. PKVIC successfully found the accurate reference URLs for these 2,703 software packages across 6 open-source ecosystems, including Pypi, Gem, NPM, Packagist, Nuget, and Maven.**

***Index Terms*—Software security, vulnerability analyze.**

I. Introduction

VULNERABLE software components are becoming the major risk for cyber-security breaches. Indeed, recent years have seen a wave of vulnerabilities from open-source software packages, with prominent examples including Lodash [1], jackson-databind [2], and HtmlUnit [3]. The security community has made significant efforts to aggregate and distribute vulnerability reports to the public through various online sources (blogs, security forums, vulnerability databases, and mail lists) in a timely fashion. In general, a vulnerability report should contain vulnerability descriptions and the affected product's information (accurate and unambiguous name and URL), and then users are capable of rapidly locating affected software without any professional security background. The Common Vulnerabilities and Exposures (CVE) system provides a reference-method for publicly known vulnerabilities and exposures, which is widely used by security research and industry. In CVE, Common Platform Enumeration (CPE) [4] is used to denote the affected products of a vulnerability, and reference URLs might point to webpages of the affected products. However, it is not the case when a vulnerability report meets an open-source ecosystem's package. Many vulnerability reports lack accurate and complete information of the affected software in open-source ecosystems.

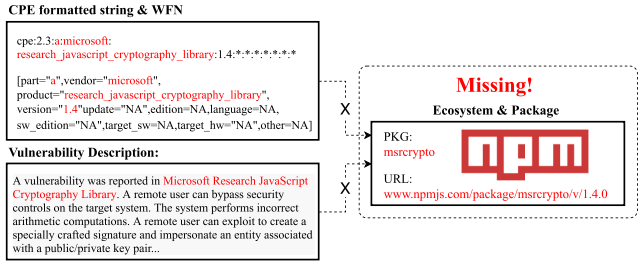

Fig. 1. Illustrations of one vulnerability example represented by three different sources.

Understanding the Gap: We identify a gap between vulnerability reports and software packages in open-source ecosystems, which is manifest in the following aspects. (1) The name convention of affected software in a security report is different from the corresponding identifiers in open-source ecosystems. A full package identifier is usually long and ecosystem-dependent, therefore, it is common that only abbreviations and a fraction of entities from the identifier appear in vulnerability reports. For example, CVE-2021-22095 only contains "Spring", "AMQP" and "Message", but the complete identifier is "org.springframework.amqp.core.Message". Our results in Table VI reveal that only 44.75% reports contain complete entities of the package identifier. (2) Many vulnerability reports do not provide reference links to affected software packages from open-source ecosystems. Our data analysis reveals that 12.3% of links are missing. In the rest 87.7% reports, there are more than 40% reports are about open-source software, however, there are only 4.84% reports providing accurate reference links to affected software packages in open-source ecosystems. For better understanding the gap, Fig. 1 presents one vulnerability

Manuscript received 5 October 2022; revised 5 September 2023; accepted 12 November 2023. Date of publication 28 November 2023; date of current version 11 July 2024. This work was supported by the National Natural Science Foundation of China under Grants 61972024 and 62272029. *(Corresponding author: Qiang Li.)*

Jinke Song, Qiang Li, and Jiqiang Liu are with the Beijing Jiaotong University, Beijing 100044, China (e-mail: jikesog@gmail.com; qiangcas@gmail.com; jqliu@bjtu.edu.cn).

Haining Wang is with the Virginia Tech, Blacksburg, VA 24061 USA (e-mail: hnw@vt.edu).

Digital Object Identifier 10.1109/TDSC.2023.3334762

example represented by two different sources: the CPE formatted string from NVD [5] and the textual description from SecurityTracker [6], as well as the missing package identifier and URL in NPM [7]. Though a complete package identifier and an accurate reference link are supposed to appear in the vulnerability report, the aforementioned gap still exists due to different background knowledge and focus between the security community and the open-source software community. Since vulnerability reports are scattering across the Internet, including archives, forums, security advisory, and software packages come from various open-source ecosystems, it would require tremendous efforts to link reports and affected open-source software manually. Filling in the information gap between security reports and open-source ecosystems is by no means trivial in practice.

Motivation: To fill the gap, we aim to automatically match vulnerability reports to their affected software packages in different open-source ecosystems. Such calibrated reports can bring significant benefits to both vulnerability management and dependency checking. (1) Vulnerability management, which aims to identify, assess and remediate potential threats, heavily depends on vulnerability database. Many prior works [8], [9], [10], [11], [12] have investigated data quality concerns in vulnerability databases. The vulnerability database and reports have several different audiences, such as software maintainers, developers and end-users. When a gap exists between reports and affected packages, it is difficult for them to be aware of the vulnerabilities in time, leading to delayed fixations and high risks. (2) Today's software reuses numerous open-source packages and there is an increasing trend of supply chain attacks [13], [14]. Dependency checking becomes a prerequisite for ensuring the quality and security of reusable packages, and heavily relies on vulnerability reports from online sources. A dependency checking tool, e.g., OWASP [15], conducts vulnerability scan by matching the product information from CPE with package identifiers in the Manifest file. Current CPE lacks complete identifier and accurate link of affected packages, leading a high false negative rate for dependency-checking tools, resulting high potential threats. Besides, automatic mapping between vulnerabilities and source codes can further enable deeper security analysis, such as analyzing features of source codes corresponding to different vulnerabilities.

Technical Challenges: Accurate and efficient matching between vulnerability reports and software packages in open-source ecosystems faces several technical challenges. *First*, a report is written in natural language and unstructured text, lacking well-formatted information of the affected software. Besides, the software name in the report could be inconsistent with its identifier in the ecosystem, which could be only abbreviations or a fraction of entities. Leaving aside the fact that named entity recognition (NER) is a difficult problem in the NLP area, it is even more difficult to extract entities that are relevant to the software name in a target ecosystem, because different ecosystems use different naming schemes, as examples in Table III, and there is no high-quality corpus for such an NER task. *Second*, even if all relevant entities have been extracted, they often cannot constitute the complete software identifier, since there may be only a fraction of entities from the identifier. *Third*, an active open-source software ecosystem generates millions of new packages each year (Fig. 2), and many packages may have very similar identifiers (as shown in Fig. 15). Meanwhile, the number of reports also grows rapidly (Fig. 3). To link extracted entities from reports to the accurate packages, simple text similarity calculation will lead to a high false positive rate and an expensive computational cost. How to accurately and quickly locate the target package from millions of packages is also challenging.

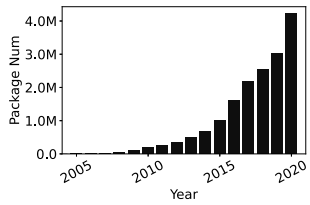

Fig. 2. Increasing number of packages on Maven per year.

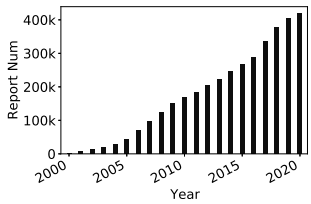

Fig. 3. Increasing number of security reports on different sources per year.

Our Work: We propose a framework called *PKVIC* (software **P**ac**k**age **V**ulnerability **I**nformation **C**alibration) to establish the information connection between vulnerability reports and software packages of ecosystems. PKVIC consists of four major components to address above challenges. (1) To deal with diverse naming schemes of different ecosystems, we propose a novel ecosystem classifier. Given a vulnerability report, the classifier determines which ecosystem the report belongs to. (2) To accurately extract package-related entities from textual descriptions, for each ecosystem, PKVIC automatically generates an NER corpus and then builds an NER model. (3) To address the problem that those entities cannot constitute the package's full-name, we propose to extend those entities via today's search engines and generate a candidate set for the full coverage of package names. (4) For efficient and accurate matching, PKVIC uses a recursive traversal method to generate a package identifier based on the naming scheme and entity candidate set. Every ecosystem has its naming scheme and package identifier index dataset. In particular, we compare the generated identifiers with those in the ecosystem package dataset. If matched, PKVIC outputs a software package identifier for the vulnerability report.

The main contributions of this work are summarized as follows:

- *New problem:* This paper reveals an information gap between vulnerability reports and software packages of open-source ecosystems, including the inconsistent naming conventions and missing URL links to affected packages.

- *New approach:* We propose a framework PKVIC to automatically fill in the missing information between vulnerability reports and software packages of open-source ecosystems. We implement a prototype of PKVIC and validate its effectiveness.
- *New findings:* We collected over 421,808 reports covering 20 popular online sources and 2,673,313 software packages across 6 popular open-source ecosystems, including NuGet [16], PyPi [17], Maven [18], Gem [19], NPM [20], and Packagist [21]. First, we revealed and quantified the missing connection between vulnerability reports and packages from open-source ecosystems. Only 44.75% reports contain complete entities of the package identifier. Reference links to affected software packages are missing in 12.3% reports, and only 4.84% reports in open-source ecosystems provide accurate links. Then, we ran PKVIC over the 421,808 reports and found 11,279 unique reports related open-source software. PKVIC automatically found 2,703 packages affected by the vulnerabilities in the 11,279 unique reports and their accurate links to the packages in open-source ecosystems, as well as their CPE strings. We released this calibrated dataset, which is new to the public.[1]
- *Extensive evaluation:* In the experiments of classification evaluation, our ecosystem classifier achieves 97% average accuracy and 95% average F1 score. For the entity extraction module, PKVIC achieves the average F1 score at the token level of 91.64%, compared with the F1 score of 24.23% for directly extracting software full names from vulnerability reports. For software package identifier generation, the top-1 rate of PKVIC is around 85%, and the top-5 rate is about 90%. Our results demonstrate that PKVIC can automatically pinpoint affected open-source software in the vulnerability reports, which enables a quick notification to software developers and users, provides a guideline for patching, and facilitates the dependency checking tool to locate vulnerable packages.

We have made our tool, prototype, and dataset openly available on GitHub at the following repository: https://github.com/inksong/pkvic. The data set is also available at https://ieee-dataport.org/documents/calibdb.

II. Data-Driven Analysis and Motivation

To gain deeper insights into the missing connections between vulnerability reports and their corresponding software packages across diverse open-source ecosystems, this section presents a large-scale data analysis. Specifically, we have gathered 421,808 vulnerability reports from 20 well-known sources of security vulnerabilities and 2,673,313 software package names and associated metadata from 6 primary open-source ecosystems. Leveraging these extensive datasets, we present a quantitative analysis that sheds light on the missing information linkage between vulnerability reports and software packages. This analysis serves as the driving force behind our proposal for a novel framework that automatically bridge the gap.

[1]https://drive.google.com/file/d/11HOUsUMQxlddJ9UTclFV9Wwy1xD_qrvU/view?usp=drive_link

A. Data Collection

We utilize a web crawler to collect security reports and software packages. Specifically, we utilize the Scrapy library [22] to crawl websites of online sources. BeautifulSoup [23] is used to parse the HTML files and extract various vulnerability reports.

Dataset for Security Reports: We run our crawler across the online security website sources listed in Table I, including vulnerability databases (e.g., NVD and Exploit Database), security advisories (e.g., Ubuntu Security Advisory), and Vendor Alerts (e.g., IBM X-Force Exchange). The vulnerability databases are archives that collect security vulnerability information of interesting software and PoC (proof of concept) of vulnerabilities. The security advisories either come from a software vendor or a third-party organization for providing information security services. In total, we collect 421,808 vulnerability reports from those 20 well-known online sources.

Note that a vulnerability report may not always represent a unique vulnerability. Different security reports from various sources can point to the same vulnerability. In this study, we employ a heuristic rule to establish a mapping relationship between these different sources. If a security report contains a CVE-ID (Common Vulnerabilities and Exposures Identifier), we establish a mapping between the report's ID and the corresponding CVE-ID. CVE-ID is widely used by cybersecurity vendors and researchers as a standardized method for identifying vulnerabilities. Additionally, other security sources assign their own IDs to vulnerability reports, such as OSVDB-ID and RHSA-ID. These sources also include a CVE-ID in their reports as an additional link to the official vulnerability database. For example, a Redhat security advisory may have an ID like "RHSA-2018:0548" associated with "CVE-2018-7262," while VULDB may have an ID like "VDB-116326" linked to "CVE-2018-9990." To extract the vulnerability ID from security reports, we utilize regular expressions as outlined in Table II. As a result, we establish an information connection among these vulnerability reports in the form of a two-tuple, such as (RHSA-ID, CVE-ID).

Dataset for Software Ecosystems: A package management system is a repository that automatically installs, upgrades, configures and removes software. Table III lists the package manager for different software packages in various ecosystems, including NuGet [16], PyPi [17], Maven [18], Gem [19], NPM [20], and Packagist [21]. In total, we collect 2,673,313 software package names and associated metadata. We also obtain the package identifiers name index database for each ecosystem. Those ecosystems cover 6 primary programming languages, including Java, Python, Ruby, Node.js, PHP and C#. The detailed naming scheme of each ecosystem is presented in Table III.

B. Quantitative Analysis About Missing Connections

Leveraging the aforementioned datasets, we conducted a comprehensive quantitative analysis to examine the missing information connections between vulnerability reports and packages from open-source ecosystems.

Enlarging Gap: The number of software packages and their vulnerabilities has been rapidly increasing in the past few

TABLE I
VARIOUS SOURCES OF VULNERABILITY REPORTS

Category	Security Sources	Description	Report ID	# Vul. Reports
Vulnerability Database	National Vulnerability Database	Government Repository	CVE-ID	147,245 (147,245)
	SecurityTracker	Public Archives for Vulnerabilities	SECTRACK-ID	14,470 (8,412)
	Open Sourced Vulnerability Database	Independent and open-sourced vulnerability database	OSVDB-ID	21,049 (19,946)
	VULDB	Community-driven Vulnerability Database	VDB-ID	100,000 (73,844)
Security Community	Vulnerability-lab	Vulnerability research, Bug Bounties & Vulnerability Assessment	VL-ID	1,639 (84)
	Securityfocus	A technical community for Symantec end-users, developers and partners.	BID	51,409 (48,324)
Vendor Alerts	IBM X-Force Exchange	Research, Collaborate and Act on Threat Intelligence	XF-ID	36,569 (31,518)
	Cisco Talos Intelligence Group	Commercial threat intelligence teams	TALOS-ID	992 (977)
Assessment Provider	Exploit-database	Non-profit public service	EDB-ID	10,832 (7,198)
	Metasploit	Penetration testing framework	MSF-ID	4,224 (2,069)
Security Advisory	Ubuntu Security Notices	Linux Distribution Maintainer & Community	USN-ID	3,470 (3,470)
	Debian Security Advisory		DSA-ID	4,311 (4,311)
	Fedora Announce List		Fedora-ID	4,838 (4,838)
	Red Hat Security Advisory		RHSA-ID	7,074 (7,074)
Ecosystem Tracker	Gemnasium	Dependency Monitoring Tool	Package Specific	5,329(4,312)
	GitHub Advisory Database	Multi-Ecosystem Vulnerability Database	GHSA-ID	4,497(4,351)
	Ruby Advisory Database	Ecosystem Specific Vulnerability Database	Package Specific	533(458)
	Node.js Ecosystem Security Working Group		Node.js Security WG ID	542(227)
	PHP Security Advisories Database		Package Specific	986(562)
	Safety DB		pyup.io-ID	1,799(510)

The numbers of vulnerability reports are until sep. 2020. For numbers in the X(Y) format, Y is the number of reports with CVE-ID and X is the total number.

TABLE II
REGEX INFORMATION FOR VULNERABILITY REPORTS FROM DIFFERENT SECURITY SOURCES

InfoSrc	Regex
CVE-ID	((?i)\bcve-(1999—2\d{3})-(0\d{2}[1-9]—[1-9]\d{3,}))
SECTRACK	(?i)\bsectrack:\d{6,7}
OSVDB-ID	(?i)\bosvdb-\d{3,5}
VDB-ID	(?i)\bvdb-\d{1,6}
VL-ID	(?i)\bvl-\d{1,4}
BID	(?i)\bbid:\d{1,6}
XF-ID	(?i)\bxf-\d{1,5}
TALOS-ID	((?i)\talos-(1999—2\d{3}):(\d{3,4}))
EDB	(?i)\bedb-\d{3,5}
MSF-ID	MSF:[^\s]*
USN-ID	USN-\d{4}-\d{1}
DSA-ID	(?i)\bdsa-\d{3,4}
Fedora-ID	((?i)\bfedora-(1999—2\d{3})-(0\d{2}[1-9]—[1-9]\d{3,}))
RHSA-ID	((?i)\brhsa-(1999—2\d{3}):(\d{3,4}))

TABLE III
SOFTWARE PACKAGES FROM DIFFERENT ECOSYSTEMS (UNTIL SEP. 2020)

Ecosystem	Lang.	Naming Convention	Num.
NuGet	C#	package:version	282,278
PyPi	Python	package:version	255,822
Maven	Java	group:package:version	346,260
Gem	Ruby	package:version	159,266
NPM	node.js	scope:package:version	1,357,880
Packagist	PHP	vendor:package:version	271,807

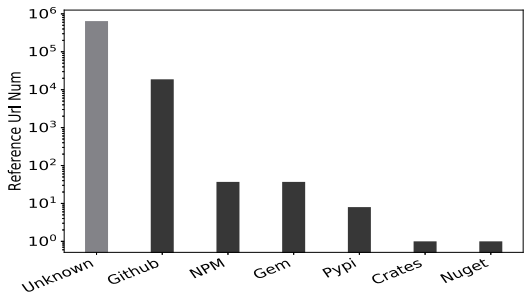

Fig. 4. Distribution of reference URL pointing to webpages.

decades. Fig. 2 depicts the timeline of the number of newly added software packages in the Maven ecosystem before Dec. 2020, showing that the number increases quickly over time. The number of newly added Maven packages is 4,219,314 in 2020, nearly 9 times of that in 2013. Fig. 3 plots the number of vulnerability reports from all these sources bucketed by the publication year. During the past two decades, the number of security reports in the first five years (2000-2005) is less than 26,757 and grows rapidly to 314,304 until Sep. 2020. In short, the increasing number leads to an enlarging information gap between vulnerability reports and software packages of ecosystems.

Missing URLs to Ecosystems: We conducted a detailed analysis to determine the extent of vulnerability reports lacking reference URLs to open-source software package webpages. We extracted a total of 663,886 reference URLs from vulnerability reports sourced from the NVD. Fig. 4 illustrates the distribution of these reference URLs, with the X-axis representing the URL

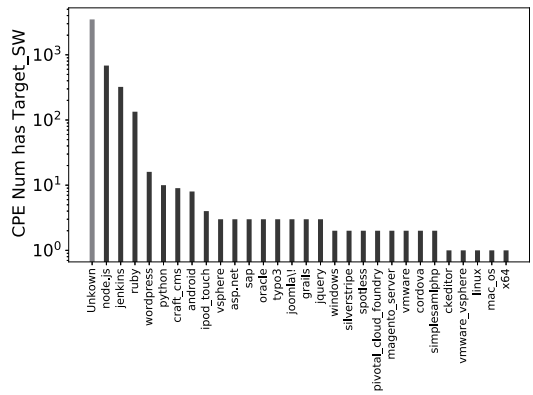

Fig. 5. Distribution of software attributes of CPE formatted strings.

type and the Y-axis denoting the number of URLs (scaled logarithmically). Our analysis revealed a significant observation: the vast majority of URLs (99.9%), the URLs in the "unknown" category, do not direct to webpages associated with open-source software packages. Since vulnerability reports cover a wide range of product types, the "unknown" category include URLs pointing to various hardware, operating systems, embedded devices, and non-open source software. Remarkably, only 61 URLs were found to point to software package webpages. Among these URLs, 33 were associated with NPM, 19 with GEM, 8 with PyPI, and 1 with NuGet. This observation underscores the pressing need to include the missing reference URLs in the corresponding vulnerability reports.

Missing Software Attribute in CPE Strings: Now we focused on examining the software attributes present in the "target_s" field of CPE strings associated with vulnerabilities. The software attribute, also known as software type, provides information about the specific software mentioned in the CPE strings. To conduct our analysis, we extracted a total of 532,481 CPE strings from 147,245 CVE items sourced from the NVD. Fig. 5 visually represents the distribution of software attributes within the CPE formatted strings. The X-axis denotes the software type, while the Y-axis represents the number of vulnerabilities associated with each corresponding software type, using a logarithmic scale (base 10) on the Y-axis. Our findings indicate that a significant majority of vulnerability CPE strings (522,481 or 95%) have "NA" (Not Available) in the "target_s" field, indicating missing or unspecified software attributes. It is common for CPE formatted strings to have the "NA" or "*" notation when the software attribute is absent. Among the non-"NA" attributes, the software attribute 'node.js' exhibits the largest number of occurrences with 684 CPE strings, followed by 'jenkins' with 322, and 'ruby' with 134. Considering the vast number of software packages present within ecosystems, the substantial amount of missing software attribute information in CPE strings is noteworthy. Addressing this missing information is crucial for a more comprehensive understanding of vulnerabilities within these ecosystems.

Naming Conventions for CPE Strings: In addition to analyzing software attributes, we further investigated the naming conventions present within the 532,481 CPE strings, focusing on the vendor and product fields. Our analysis revealed that 107,436 CPE strings exhibit identical vendor and product names. This finding is intriguing as it implies that certain products share the same name as their respective vendors. Upon closer examination, we identified two underlying causes for this phenomenon. First, in some cases, a product assumes the same name as its vendor because it is the sole offering in the market. This scenario often occurs with smaller corporations, such as the opendnssec corporation, where their product bears the name "opendnssec." Second, there are instances where open-source packages are developed by small groups without a formal vendor or manufacturer. Numerous projects hosted on platforms like GitHub fall under this category, resulting in missing vendor names within the CPE strings. For example, the project "omniauth" [24] lacks a vendor name. Consequently, there is a pressing need to automatically rectify CPE strings by aligning them with the respective affected software names within open-source ecosystems. This calibration process will enhance the accuracy and completeness of vulnerability information in relation to software packages.

III. Design of *PKVIC*

To address aforementioned challenges, we propose *PKVIC* to build the information connections between vulnerability reports and affected software packages. Fig. 6 shows the overview of PKVIC, including an ecosystem classifier (EC), a package-related entity extractor (PET), a package-related entity search and extend (PESE) module, and a package identifier generator (PIN). The input is a vulnerability report, and the output is its package identifier in ecosystems.

Architecture: First, given a vulnerability report written in natural language, we design an EC module to map it to a corresponding ecosystem through a set of classification models. Different ecosystems have different naming schemes, and many software package names might be similar or overlapped across different ecosystems. For accurate matching, it is necessary to determine which ecosystem the vulnerability report belongs to. Second, we build an NER model for every software ecosystem in the PET module. The NER model is to extract package-related entities from the vulnerability report. Note that existing NER techniques are highly domain-specific, and there is no existing corpus for training a model to extract package-related information from the texts of vulnerability reports. By using indexes from different software ecosystems, PKVIC automatically annotates package-related entities in vulnerability reports and constructs a high-quality corpus. Third, the PESE module further extends those package-related entities by leveraging today's search engines. The reason is that extracted entities in reports cannot provide a full coverage on package names in ecosystems, due to different naming conventions, abbreviations and errors. We search the extracted entities and generate a richer candidate entity set to cover all possible entities in package identifiers. Fourth, to locate the affected package, the PIN module generates package identifiers based on the naming scheme and the candidate entity set. Every software ecosystem has a unique naming scheme, and candidate entities will be used to construct validate package identifiers. Then, the PIN module searches those package

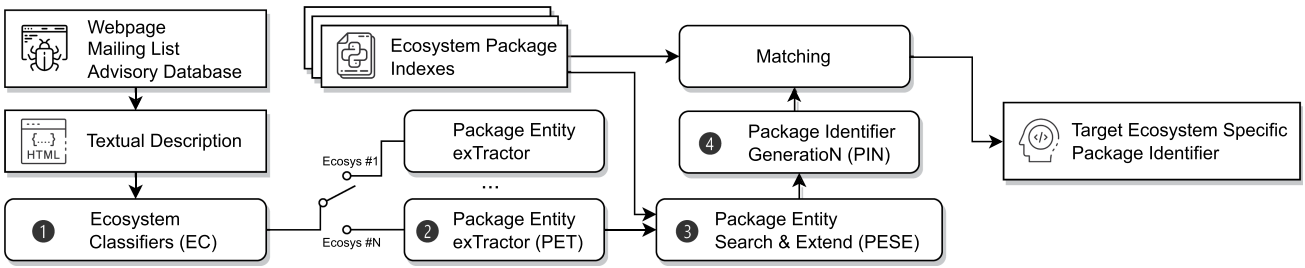

Fig. 6. Overview of PKVIC: build the mapping between security reports and software packages from various ecosystems.

Algorithm 1: Algorithm of PKVIC.

Input : Package full names of multiple ecosystems: Φ; Set of vulnerability reports: $\mathcal{R}$; Ecosystem specific high freq. conjunctions: C.

Output : Report set and Package identifier set: $(\mathcal{R}, \mathcal{O})$

for *one ecosys* $\phi \in \Phi$ **do**
  current ecosys tokenset $T_\phi = \emptyset$
  for *one package* $p \in \phi$ **do**
    cur package tokens T_p = Tokenize(p)
    T_ϕ.add(T_p)
$\mathcal{T} \leftarrow$ Generate a trie tree for each ecosystem package index
for $r \in \mathcal{R}$ **do**
  ϕ = Classifier(r) ❶
  E = NER(r) ❷
  E = Search(E) $\bigcap T_\phi$ ❸
  Candidate Identifier E^* = $E \bigcup C_\phi$
  $\mathcal{W} = Rank(E^*, \mathcal{T})$
  for $w \in \mathcal{W}$ **do**
    G = null
    $\mathcal{O}[r]$.append(Gen(G, E^*, w)) ❹
return $\mathcal{O}$

identifiers in the ecosystem package dataset. If there is a match, the information connection between a vulnerability report and an affected software package is built. The procedure of PKVIC is listed in Algorithm 1. Below, we present the details of the four modules.

A. Ecosystem Classifier

A vulnerability report is written in natural language, which is too complicated for handcraft features to classify its ecosystem. To this end, we develop a deep neural network (DNN) model for each ecosystem to capture long-term dependencies of textual words and classify reports.

Preprocessing: We use the CPE string and a textual description in the report as the input. The report description is to detail and summarize the relevant information of the vulnerabilities. We convert every word in the plaintext into a token. The CPE string has 12 fields, each of which is divided by a colon, such as the "vendor" field and "product" field. Every field is converted into a word token. Since the DNN classifier cannot directly process text inputs and requires input in vector format, we utilize the pre-trained Bert model [25] to convert those word tokens into vectors, called word embedding. Bert is a powerful language model that has been trained on a large corpus and can effectively encode textual information into meaningful vector representations. By leveraging the pre-trained Bert model, we are able to transform the text inputs into vector representations that can be easily consumed by the DNN classifier. In short, the preprocessing converts a textual input into a vector for the classification model. The cost for preprocessing is presented in Table VII. It takes only 1.2–7.1 ms to process each package. The software packages that have already been processed do not require repetitive processing. As the number of packages within each ecosystem increases, we can periodically execute the preprocessing step on the newly added packages, making the system stay up-to-date with the evolving ecosystems.

DNN Model: We utilize a recurrent neural network (RNN) to learn and derive the classification. RNN leverages a long short-term memory (LSTM) in the network to capture the inputs' long-term dependencies. Particularly, LSTM uses three gates (input gate, forget gate, and output gate) to store long-term dependencies of textual words. We connect an LSTM with a fully-connected layer, followed by a softmax layer, which outputs the ecosystem category's probability. Here, we train the models for each ecosystem individually, including NuGet [16], PyPi [17], Maven [18], Gem [19], NPM [20], Packagist [21], and unknown. Here, "unknow" is the category for reports that do not belong to any of the six ecosystems under consideration. We choose the maximum probability as the output for the vulnerability report.

B. Package-Related Entity Extractor

PKVIC utilizes name entity recognition (NER) to extract package-related names from the texts of vulnerability reports. Those names are not the package identifiers in software ecosystems because of different naming conventions.

The NER model belongs to sequence labeling, which maps an observed sequence to a sequence of labels, denoted as $f: x \rightarrow y$, where x and y have the same length. Fig. 7 provides a graphical illustration of the NER model that tags a sentence with a label sequence. First, words and characters in the sentence are embedded into vectors, which is necessary for NER tasks. Then, these

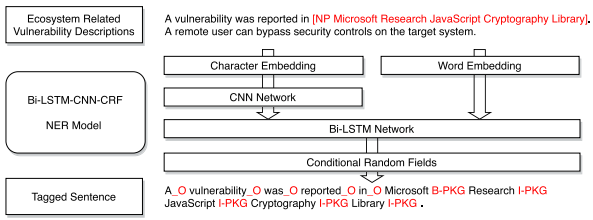

Fig. 7. NER extracts software package-related names in the vulnerability report.

vectors are fed into a Bi-LSTM network to capture the context of words in their sentence. Given the training set (X, Y), the NER needs to learn the model ($\theta = \arg\min_\theta \Sigma_{X,Y} L(y, f(x, \theta))$), where θ is the model parameter and L is the loss function. After that, this word-contextual network's output can be directly fed into a fully connected layer followed by a softmax layer to output the probability distribution over the possible tags. We provide labels to every word in a sentence for maximizing the conditional probability $P(y|x, \theta)$. Below, we introduce the details of three components of NER tasks: embedding, Bi-LSTM, and tag sequence.

Embedding: In our work, PKVIC utilizes two embedding methods to encode plaintext, including character-level and word-level. (1) The character-level embedding is to capture the character information of a word, e.g., prefix [26] and suffix [27]. The reason is that many software package names do not exist in the word vocabulary, due to the limitations in building the dictionaries and pre-trained model. Such words have to be replaced by a special word (e.g., unknown), and the character-level embedding method encodes morphological features of characters. (2) The word-level embedding is to encode the semantic content of words. We use the Bert model [25] as the embedding model, which is a well-known text embedding model, and directly concatenate those two (word and character) embeddings to create a vector that represents rich semantic and grammatical aspects of the input sentence.

Bi-LSTM: We utilize the Bi-LSTM to extract semantic meaning from the (word&character) vector, identifying the sequential relationship among words. For example, CVE-2020-28052 states, "An issue was discovered in Legion of the Bouncy Castle BC Java 1.65 and 1.66", where "Bouncy Castle" is behind "in Legion of" and "discovered". To obtain a backward sequential relationship among words, we use LSTM to conduct a bi-directional scan of the sequence. In our implementation, we use Bi-LSTM [28] to extract the sequential relationship of words in combination with their left and right contexts. Bi-LSTM checks the sentences in both directions, forwards and backward, using two parallel LSTM networks, and combining their outputs.

Tagged Sentence: The input to Bi-LSTM is the (word& character) vectors for sentences, and the output is the conditional probability distribution over the output vocabulary. We utilize a fully connected layer followed by a softmax layer to output the probability distribution over the possible tags. Recall that our task is to identify the entities of our interest, which pertain to software packages. To achieve this, we assign each word with one of the following labels B (begin of tag), I (inside of tag), and O (outside of tag).

Based on our network design, we consider two types of NER approaches. The first one, NER-Package, represents the conventional approach that aims to extract the complete package identifier, such as 'zhmc-ansible-modules', that is labeled as "B I I I" directly. This method serves as the baseline for comparison. On the other hand, we introduce our proposed solution, NER-Token, which focuses on extracting package-related tokens, for example, 'zhmc' and 'ansible', which are labeled as "B". These extracted tokens are then utilized as input for the PESE module. We compare the performance of NER-Package and NER-Token in Section IV. The results clearly indicate that our NER-Token design outperforms NER-Package significantly. Specifically, NER-Token achieves an F1 score of 91.64%, whereas NER-Package only achieves an F1 score of 24.23%. The main reason NER-Package performs poorly across all ecosystems is that many reports do not contain the complete package identifier. This lack of complete identifiers creates two challenges: first, there is a scarcity of labeled training data that can assist the NER-Package model in learning the position and contextual relationships of complete identifiers in text. Second, in practice, it becomes infeasible for NER-Package to extract package-related information from reports that lack complete identifiers. Moreover, the complex naming conventions within open-source ecosystems further increase the difficulty of NER models in extracting complete package names. Maven and Packagist, for instance, have unique naming conventions. In contrast, vulnerability reports consistently contain entities related to package names. This substantial difference in performance highlights the superiority of our proposed NER-Token approach in accurately identifying and extracting package-related information.

C. Package-Related Entity Search and Extend

Aforementioned, package names of software ecosystems might not exist on a vulnerability report, due to different naming conventions among various sources. Our package-related entities extracted by PET might not be enough for generating a full coverage on package identifiers.

To fill in the vulnerability report's missing information, we have a practical observation that today's search engines rank web pages based on the correlated degree between a search query and a search engine result, e.g., PageRank [29]. If we utilize those entities extracted by the PET module as the search query to a search engine, the search results may contain the missing package names of software ecosystems. Then, the problem is how to identify which part of the search results are the missing package names. Here, we leverage a trick that a software ecosystem (e.g., NPM [20]) provides the package index dataset to the public. If the content of the search result appears in the package index dataset, we extract the related words as the package-related entities for filling in the missing information.

Fig. 8 depicts how the PESE module find more package-related entities for generating package identifier of a software ecosystem. First, we use all package-related entities extracted from a vulnerability report as the search query. We send the

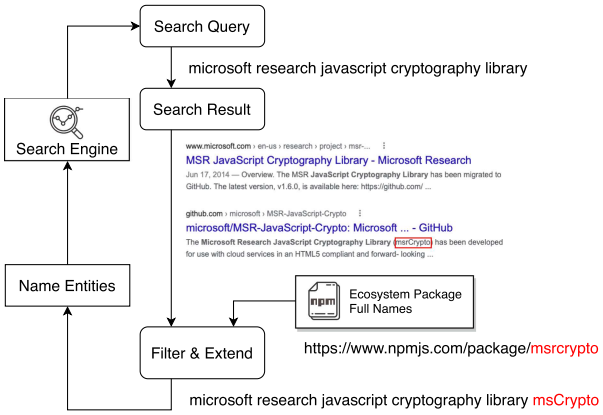

Fig. 8. PESE extends package-related entities.

search query to a search engine and gain the search result that has a high correlation with package-related entities. Second, we collect the ecosystem package index dataset. For instance, NPM [20] provides 1,357,880 full package names. Third, we use the package name list to filter out the search result and extract those matched package-related entities. If a word is matched, we directly add it into the set of candidate entities as the PIN module's input. In our implementation (Section IV), we detail the specific parameters for our PESE module.

Example: As shown in Fig. 8, we provide an example to illustrate the PESE module. The entity set is "microsoft research javascript cryptography library", encapsulated into a query. We send it to the Google search engine, in the form of "search engine/search?q=microsoft+research+javascript+cryptography +library+&btnG=Search", where the mark (?) indicates the end of the URL and (&) separates arguments, q is the start of the query, the plus mark (+) represents space, and "btnG = Search" denotes that the search button is pressed on the web interface. We obtain the search result that may include package-related names. The word "msCrypto" also appears in the NPM [20] package index dataset. We extract this word and add it into the set of package-related entities, as "microsoft research javascript cryptography library msCrypto".

D. Package Identifier Generation

A package identifier is unique for representing a software package in an ecosystem. Every software ecosystem contains its own namespace, and we use the Trie tree [30] to index the ecosystem package name dataset. Given an ecosystem's namespace, the PIN module generates package identifiers based on those package-related entities from the PESE module.

Algorithm 2 describes how to generate package identifiers in terms of package-related entities. Our idea is to enumerate all possible combinations of those entities to find a matched package identifier. If we find a matched w (Line 4), we use it to represent the vulnerability report's package identifier. Our matching process is a strict string match rather than approximation matching. The reason is that various package names are similar. Approximation matching could lead to a high false positive rate for finding package identifiers. For instance, in the

Algorithm 2: Generating Package Identifier of Software Ecosystem Based on Package-Related Entities.

Input : Package identifier: w
Package-related entity set: E
Current enumerated package: g
Output : Matched package identifier: $result$

```
1  Function Gen(g, E, w)
2      result = ""
3      for e_i ∈ E do
4          if g equals to w then
5              Return g
6          else
7              g* = g + e_i
8              if w start with g* then
9                  result += Gen(g*, E, w)
10     return result
```

Pypi ecosystem [17], the package "color-palette" (a package for easy coloring) is very similar to the package "color-pallete" (creating color pallete of the provided input image), where fuzzy matching will cause a false positive.

Pruning: Our enumeration procedure belongs to the depth-first search (DFS) but with expensive time cost, as the running time grows exponentially with the problem size. Given n candidate entities, the total number of enumerations is $\Theta((n+1)^L)$ for a vulnerability report, where L is the maximum length of package names of software ecosystems. The algorithm time cost is related to two factors: the branching factor n of the tree, which is the maximum number of children generated at any node, and the search depth L of the tree, which is the longest path from the root to a leaf. Directly searching through the entire space of package name entities would result in a time cost of over 15 minutes when the search depth reaches 6. Such a time expense is not acceptable in practical applications. Therefore, PKVIC incorporates pruning methods to significantly reduce the search time cost. The n is the package-related entity set from the PESE module, and we cannot revise it. Instead, we use a heuristic pruning policy to reduce the longest path L for reducing the time cost. The first pruning rule is: if a generated enumerated identifier g is not the package identifier's prefix, we skip the corresponding branch (Line 8). The second pruning rule is: if we find a match, we skip the rest search space and return the matched identifier (Line 4); otherwise, return a leaf node (Line 10). Such a pruning rule won't cause the occurrence of missing the correct package identifier. First, a vulnerability report usually refer to only one software package. We further analyze the identifiers of 2,673,313 packages across 6 ecosystems and present the statistics in Table IV. It shows that the majority of packages possess unique names. In other words, their identifiers have unique sets of tokens. Hence, the percentages of missing the correct package identifier due to the pruning rule range from 0.04% to 1.6% across various ecosystems.

Furthermore, if we compare the generated package identifiers with all full package names of a software ecosystem (e.g., 1,357,880 packages in NPM), the time cost is prohibitively high

TABLE IV
POSSIBLE FALSE NEGATIVE (FN) RATE CAUSED BY PRUNING

Ecosystem	# Package	# Package with Unique Token Set	# Package Sharing Token Set	Possible FN Rate
Nuget	282,278	279,365	2,913	1.03%
Pypi	255,822	255,201	621	0.24%
Maven	346,260	344,785	1,475	0.43%
Gem	159,266	156,714	2,552	1.60%
Npm	1,357,880	1,336,695	21,185	1.56%
Packagist	271,807	271,700	107	0.04%

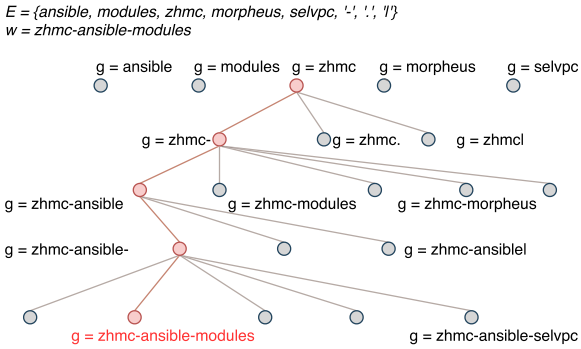

Fig. 9. PIN generates a package identifier by enumerating all possible combination and pruning unnecessary parts.

in practice. Hence, it is crucial to significantly reduce the search space of software package names to ensure practical computational cost. To achieve this, we propose another pruning policy is that we only compare a small set of full package names (w) with our generated identifiers. Specifically, we first index the software package names using TF-IDF. Then, for the input entity list, we obtain sorted package names from the TF-IDF index based on the similarity between the package names and the input entity list, sorted in descending order. We select the top 30 package names as the search space for the generated package identifiers. The utilization of TF-IDF indexing makes the aforementioned similarity based ranking and retrieval process more efficient. If Algorithm 2 can find a matched identifier, the PIN module outputs it as the package identifier for the vulnerability report. The parameter 30 is chosen based on the following rationale. As shown in Table IV, software package names exhibit strong specificity, with the majority being unique. After generating package identifiers using the PIN algorithm and performing similarity-based searches using tf-idf, in most cases, there is only one full package name (i.e., the correct target package) with a 100% similarity to the generated identifier. Additionally, there are fewer instances where multiple package names are highly similar, and the similarity decreases rapidly when full package names are sorted by similarity using TF-IDF. We randomly selected 100,000 packages from six ecosystems for analysis and found that, on average, there were 18.08 packages with a cosine similarity greater than 0.5 to the PIN-generated identifier. Therefore, the target full package name is highly likely to appear within the top 20 names. Selecting the top 30 package names based on similarity provides a safe coverage of the target package.

Example: As shown in Fig. 9, we provide an example to illustrate how to generate a package identifier. The full package name is "zhmc-ansible-modules", and our candidate set is the {ansible, modules, zhmc, morpheus, selvpc} and the naming scheme regex is {'.', '-', '|' }. We enumerate all possible combinations and prune unnecessary branches. Once a match is found, the "zhmc-ansible-modules" will be returned.

IV. EXPERIMENT EVALUATION

In this section, we present the experimental evaluation for validating the efficacy of PKVIC.

A. System Implementation

We implement a prototype of PKVIC as a self-contained piece of software, including four key functional components: the ecosystem classifier (EC), the package-related entity extractor (PET), the package-related entity search and extend (PESE) module, and the package identifier generation (PIN) module. Those components are extensively used across the whole prototype, and their implementations are described as follows.

1) *The EC module:* We use the Bert model as the pre-trained model for our ecosystem classifier. We utilize the deeppavlov framework [31] to process a vulnerability report, including text tokenization, encoding with the pre-trained model, creating segment masks. We use the TensorFlow framework [32] to implement the ecosystem classification.

2) *The PET module:* We implement the Bi-LSTM-CNN-CRF (NER) model by leveraging the deeppavlov framework [31]. Given a report, it is first segmented into word tokens, and then is converted to vectors. For every word token, the NER provides a tag "B"/"I"/"O" to it. To find more package-related entities, we collect words with "B" tag.

3) *The PESE module:* We use GoogleScraper [33] to query output entities of the package entity extractor and save the top 30 search results. Based on our experiments, top 30 search results are sufficient for package entity searching and extending. For each search result, we only extract the title and brief description of the result item. We use the regex to identify and extract package-related entities from the search results. If a string in the search result matches any package identifier tokens in the ecosystem index dataset, the PESE module will add it into a package-related entity set.

4) *The PIN module:* For each ecosystem (NuGet [16], PyPi [17], Maven [18], Gem [19], NPM [20], and Packagist [21]), we build a Trie-Tree to build their package name indexes. Specifically, we use Elastic Search [34] to store the ecosystem's package identifiers. Our enumeration algorithm is a custom Python script to generate a package identifier of a vulnerability report.

TABLE V
LABELED DATASET FOR VULNERABILITY REPORTS

Ecosystem	Vulnerability Report #	Package Identifier #
Maven	818	818
Pypi	375	375
Gem	250	250
NPM	813	813
Packagist	278	278
Nuget	41	41
No Ecosystem Related	1,000	0

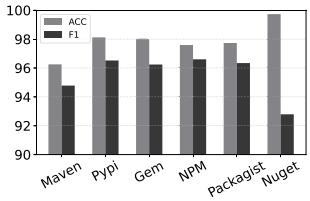

Fig. 10. Performance of ecosystem classifiers.

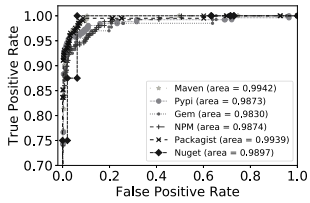

Fig. 11. ROC of ecosystem classifiers.

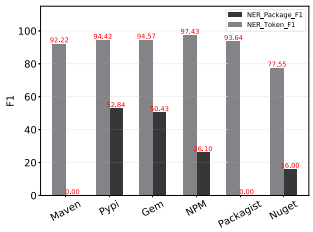

Fig. 12. Performance of PET over six ecosystems.

B. Experimental Setting

Our experiments are conducted on Ubuntu 18.04, Intel i9-9820X 3.30 GHz, 64 G memories, 4*GeForce RTX 2080 Ti. Here we explain the datasets used in the study.

Labeled Datasets: We utilize a set of vulnerability tracker databases to manually label the vulnerability reports, including ruby-advisory-db [35], GitHub Advisory [36], gemnasium [37], safety-db [38], nodejs-security-advisories [7], and php-db [39]. The file formats in the tracker databases include. toml,. yaml,. yml, or json. Note that their package identifiers are manually calibrated by software developers. We manually identify and extract full package identifiers to create a database as the ground truth of our experiments, where each package identifier is labeled by the BIO (Beginning, Inside, Outside) scheme. We obtain 2,575 ground truths from 6 different ecosystems and 1,000 reports that do not belong to any ecosystem. We use the labeled database as the training dataset for both NER-Package and NER-Token approaches. These two approaches differ in their use of this training set. NER-Package aims to recognize the complete package identifiers and directly uses the standard 'B', 'I' and 'O' labels. NER-Token aims to recognize the tokens in package identifiers, therefore, it splits each package identifiers into multiple tokens and labels each token as 'B'. In the controlled experiments, we use 80% of data for training data, 10% for validation data, and the rest 10% for test data, and conduct 5-fold cross-validation. Table V lists the number of vulnerability reports from different ecosystems.

C. Performance

We first validate the performance of the ecosystem classifier. Fig. 10 shows two performance metrics for the EC module, including the accuracy (ACC) and F1-score. Overall, our classification results are promising for identifying 6 different ecosystems, where the average classification accuracy is 97.9%, and the average F1 score is 95.5%. The classifier for Nuget ecosystem [16] has worst performance (92.79% F1-score) compared with others. The reason is that its training dataset is small, with only 41 vulnerability reports. We also depict receiver operating characteristic (ROC) of 6 ecosystem classifiers, as shown in Fig. 11. The ROC value is the area computed based on the true positive rate (TPR) and the false positive rate (FPR). The average AUC score is larger than 98%. In short, our ecosystem classification model achieves high accuracy to determine whether a vulnerability report involves a specific ecosystem.

Then, we evaluate the performance of the PET module. We also use the F1-score as the evaluation metric. As introduced in Section III-B, we measure two types of NER approaches: NER-Package that extracts full package names and NER-Token that extracts package-related tokens. Here NER-Package serves as the baseline and our PET module utilize NER-Token.

We train an NER model for each ecosystem. Fig. 12 depicts the performance of two NER approaches across six ecosystems. The average F1 score for NER-Package is only 24.23%. As mentioned in Section III-B, the main reason NER-Package performs poorly across all ecosystems is that many reports do not contain the complete package identifier. The complex naming conventions within open-source ecosystems further increase the difficulty of NER models in extracting complete package names. For example, we cannot even find a complete package name (0% F1 score) in the vulnerability reports for the Maven [18] and Packagist [21] ecosystems. Maven and Packagist use a more complicated naming scheme, leading to that their complete package identifiers never exist in any vulnerability report. Therefore, conventional NER techniques are unable to extract package identifiers with high precision and recall. In contrast, the average F1 score for NER-Token reaches 91.64%, indicating that NER-Token can effectively extract entities belonging to package identifiers. In other words, we can leverage those correct tokens to generate a package identifier in the subsequent process. Among six different ecosystems, the NER-Token model achieves better accuracy for NPM, Pypi, and Gem ecosystems because they have more training data and their package naming

TABLE VI
COVERAGE RATE IMPROVEMENT BY THE PESE MODULE

Ecosystem	P-R	R/Google	R/Bing	R/DuckDuckGo
Maven	0.00	37.95%	36.96%	36.57%
Pypi	71.59%	98.11%	95.53%	94.53%
Gem	67.88%	98.54%	96.42%	94.95%
NPM	69.11%	96.91%	94.36%	93.38%
Packagist	39.92%	90.12%	88.44%	86.86%
Nuget	20.0%	90.00%	87.63%	86.72%
Average	44.75%		85.27%	

P-R indicates prior $R_{coverage}$ without the PESE module, and R indicates $R_{coverage}$ with the PESE module.

conventions are clearer and shorter. The results demonstrate that the PET model with NER-Token is capable of obtaining a sufficient set of package-related tokens for generating an ecosystem's package identifiers.

Third, we evaluate the performance of the PESE module. Aforementioned, the PESE module is to extend more package-related entities for filling in the missing information of a vulnerability report. We use the coverage rate to measure the effectiveness of the PESE module as follows:

$$R_{coverage} = \frac{\Sigma(e|e \in E \& e \in w)}{\Sigma(token|token \in w)},$$

where E is the entity set extracted by the PET module, and w is a package identifier of an ecosystem. The range of $R_{coverage}$ is [0.0, 1.0]. The larger $R_{coverage}$ is, the higher chance of generating a correct packager identifier. When $R_{coverage}$ is equal to 1.0, it indicates that the package-related entities cover a package identifier. By contrast, when $R_{coverage}$ is equal to zero, our package-related entities are missing the information of a package identifier.

Table VI lists the coverage rate improvement by the PESE module. For vulnerability reports from different ecosystems, the PESE module can significantly improve the entity coverage rate by 40.52% on average. In terms of those extended entities, it has a higher probability to generate a package identifier. In particular, Maven [18] has zero coverage for package-related entities extracted by PET. The reason is that, in the Maven ecosystem, the identifier of a complete package often includes groupID and artifactID. Here we take "org.dynamoframework|dynamo-impl" as an example, which is a software development framework. The entities "org"and "impl" are very ecosystem-characterized words and often do not appear in vulnerability reports, resulting in a low coverage rate before search. The PESE module increases the coverage rate from 0% to 37.95% for Maven. One concern is that today's search engine is a black box for finding relevant entities for software identifiers. We conduct experiments to assess the impact of different search engines, including Google, Bing, and DuckDuckGo. Our experiments show that the search engine selection has little impact on the performance of the PESE module.

Overall Performance: In terms of extended entities (Table VI), the PIN module generates package identifiers of corresponding software ecosystems. As the PIN module is the last component in the workflow of PKVIC, the overall performance of PKVIC is represented by the output of PIN module, which also depends on the first three modules.

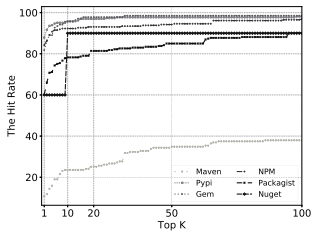

Fig. 13. Overview performance of the PKVIC.

TABLE VII
TIME COST FOR PREPROCESSING

Ecosystem	PKG Num.	Token Num.	Avg.(ms) per PKG
Maven	346,260	81,535	1.5
Pypi	255,822	172,838	2.5
Gem	159,266	80,384	1.2
NPM	1,357,880	423,715	7.1
Packagist	271,807	147,732	2.1
Nuget	282,278	143,616	2.2

PKG is short for package.

Fig. 13 shows the performance of generating package identifiers. For Pypi, Gem, and NPM, the hit rate of the top-1 generated identifier is 85%, and the hit rate of the top-5 generated identifier is 93%. For Packagist and Nuget, the top-1 hit rate is around 62%, and the top-10 hit rate is 86%. The Maven's accuracy is much lower than that of other ecosystems. The reason is that Maven's naming scheme of "group:package:version" is the most complicated. So far, the state-of-art tools cannot extract any Maven full package name. Note that some software ecosystems' hit rates cannot reach their $R_{coverage}$ scores. The reason is that there exist software packages with similar names, and a simple similarity-based match algorithm will hit false-positive packages with similar names.

As we mentioned before, a package identifier in a specific software ecosystem is a unique and unambiguous index, and we can use it to locate the corresponding software package accurately. Overall, our experiment results show that PKVIC extracts package identifiers and use them to build the accurate information connection between vulnerability reports and software ecosystems.

D. Overhead

We first processed all 2,673,313 packages and measure the time cost for the preprocessing phase (Line 1–5) in Algorithm 1. Table VII provides the cost breakdown and the preprocessing time for each package is only 1.5–7.1 ms. It's worth noting that the preprocessing phase can be carried out offline. As the number of packages within each ecosystem increases, it is efficient to periodically process the newly added packages. Therefore, the system can stay up-to-date with the latest additions to the ecosystem with reasonable processing time. Furthermore, we randomly selected 1,000 vulnerability reports that related to open-source software packages from 6 ecosystems. We sequentially ran the

TABLE VIII
TIME COST OF EACH MODULE OF PKVIC

	Avg.(s)/Report	Std.
Ecosystem Classification (EC)	0.0064	0.0129
Package Entity Extraction (PET)	0.0711	0.0547
Package Entity Search (PESE)	2.0692	0.4843
Package Identifier Generation (PIN)	0.6138	0.6885

four modules of PKVIC on these reports and record the time cost of each module. The detailed implementation of each module is introduced in Section IV-A and the hardware environment is described in Section IV-B. Table VIII lists the time cost for the four modules, i.e., the EC, PET, PESE, and PIN modules. The primary time cost lies in that PKVIC needs to use a web crawler to search and extend the package-related entities. In short, the time overhead of PKVIC is moderate and easily affordable in practice.

E. Building the Connections

In this subsection, we utilize PKVIC to process the extensive datasets that we have collected. For specific information regarding the datasets used, please refer to Section II-A. We ran PKVIC over 421,808 vulnerability reports from 20 well-known sources of security vulnerabilities and found a total of 11,279 unique vulnerability reports related to open-source software. It is worth noting that the majority of the vulnerability reports do not pertain to open-source software. Since different reports can refer to the same affected software, upon further examination, we discovered that the 11,279 reports contain duplicated software packages. After eliminating these duplicates, we were left with 2,703 unduplicated software packages. PKVIC accurately located reference URLs for these 2,703 software packages across six different open-source ecosystems. Table IX lists the connections between IDs in vulnerability reports and software packages of ecosystems. The most related work is a commercial vulnerability database – Snyk [40], which disclosed 1177 reports for npm, maven and rubygems, and provided manually built connections to affected packages. We manually verified that all 1177 packages have also been identified by PKVIC. Note that, PKVIC has located more vulnerable packages in an automatic manner, which proves the great value of PKVIC.

CPE String Revision: We propose to revise CPE strings to correctly represent those software packages, including: (1) renaming the product name in CPE strings with the same name of the software package in their ecosystems; (2) supplementing the missing software attribute information in the"target_sw" field of CPE strings. Note that our revised CPE strings need to be compatible with the original ones.

URL Linked to the Affected Software: Given a package identifier, we leverage structured information (e.g., URL prefix) from software ecosystems to automatically generate a package-specific reference URL for the vulnerability. We use regular expressions to generate different kinds of structured information for URLs. For each URL of the software ecosystem, we build a parser to transform it into a general template: *https://prefix/*{*parameterspkg*/, where prefix is the domain name of the software ecosystem, pkg is the package entity calibrated by PKVIC, {parameters} is the directory structure of websites. For each generated URL, we check whether it is accessible and available.

Table X presents an example of generating URLs for software packets in ecosystems, showing how we generate a reference URL based on package entity extracted by PKVIC for a vulnerability report. The table's rightmost column shows the examples of generated URLs based on the structured information and package entities. The rest of columns are the structured information, which is determined by package ecosystems. Every row shows a specific URL based on its corresponding ecosystem. Table XI lists the ratio of software packages with missing links in CPE strings under different open-source ecosystems. In this way, we build a reference URL for a vulnerability report that lacks a relevant URL to its software package. Note that our template can be updated with a little modification when the structured information of websites is changed.

Calibration Dataset: We will distribute the calibrated dataset to the research community. The dataset consists of a set of 4-tuple records (CVE ID; Other ID; Package name; URL). Each entry of the dataset represents a vulnerability from security sources, and thus we assign it to relevant project URL. The dataset is released under the Apache 2.0 license, available for free download.

F. Case Study

We present a simple case study to demonstrate how PKVIC enhances software security, besides directly informing the security community of those vulnerable software packages.

Package Dependency Defense: The reuse of software packages is pervasive in today's software community, where numerous applications have been developed by reusing packages in the package manager. Here we draw 20 vulnerability reports from NVD, Securityfocus, and Ubuntu Security Notices. PKVIC builds the information connections between those reports and software packages in ecosystems, revealing that the following three software packages are vulnerable, "org.apache.storm|storm-core" in Maven, "urllib3" in Pypi, and "prismjs" in NPM. A simple case study in Fig. 14 depicts that many libraries are directly dependent on those three vulnerable software packages. When a reused software package involves a vulnerability, its dependent projects will likely suffer a similar risk, implying the need to ensure the security of directly and indirectly reused packages. Therefore, our calibration connections provided by PKVIC can help package managers to isolate untrusted packages, evaluate risks, and remediate issues.

V. DISCUSSION

While PKVIC is the first work to fill in the missing connection between vulnerability reports and software packages. In this section, we first discuss the validity and then the limitations of PKVIC.

TABLE IX
CONNECTION BETWEEN IDS IN SECURITY REPORTS AND SOFTWARE PACKAGES OF ECOSYSTEMS

	CVE	SECTRACK	OSVDB	VDB	VL	BID	XF	TALOS	EDB	MSF	USN	DSA	Fedora	RHSA
Maven	1378	7	14	1516	-	316	14	4	4	22	26	57	62	550
Pypi	714	8	16	346	1	126	27	2	-	9	103	74	167	207
Gem	540	46	26	375	-	139	42	2	4	17	39	73	90	126
NPM	1074	-	-	751	-	64	1	-	-	9	12	17	39	106
Packagist	1023	-	-	521	3	128	2	-	1	12	1	35	76	5
Nuget	77	-	-	21	-	9	-	-	-	1	-	1	-	1

TABLE X
EXAMPLES OF GENERATING URLS

Ecosystem	prefix	parameter	pkg.	Generated URL
Maven [18]	https://search.maven.org	groupid/ artifactid	org.apache.santuario/ xmlsec	https://search.maven.org/artifact/ org.apache.santuario/xmlsec
PyPi [17]	https://pypi.org	project	html5lib	https://pypi.org/project/html5lib/
Gem [19]	https://rubygems.org	gems	rubygems-update	https://rubygems.org/gems/rubygems-update
NPM [20]	https://www.npmjs.com	package	https-proxy-agent	https://www.npmjs.com/package/https-proxy-agent
Packagist [21]	https://packagist.org	vendor/ package	typo3/cms	https://packagist.org/packages/typo3/cms
NuGet [16]	https://www.nuget.org	package	Sustainsys.Saml2	https://www.nuget.org/packages/Sustainsys.Saml2

For software packages in ecosystems.

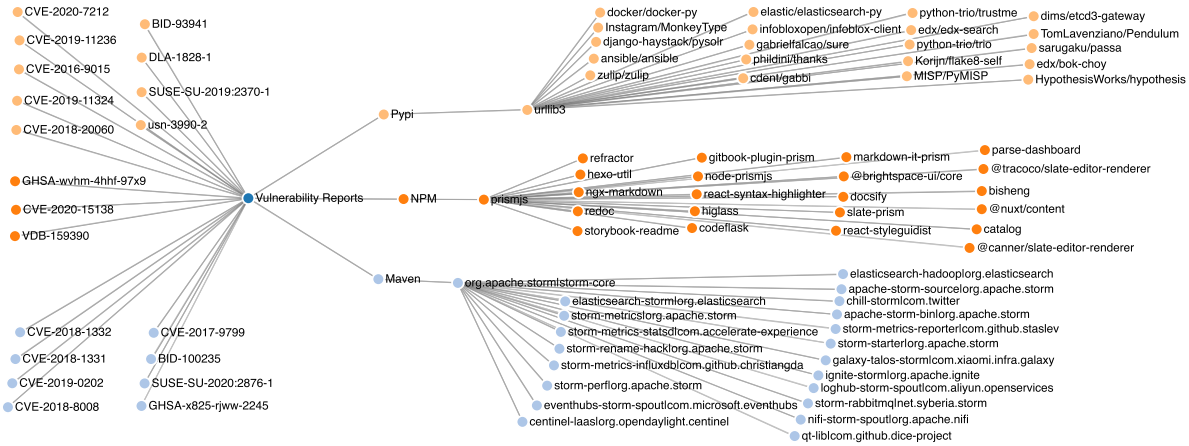

Fig. 14. Graph of software package dependency.

TABLE XI
RATIO OF SOFTWARE PACKAGES WITH MISSING LINKS IN CPE UNDER DIFFERENT OPEN-SOURCE ECOSYSTEMS

Ecosystems	# pkgs w link in CPE	# pkgs w/o link in CPE	Missing rate
Maven	19	857	97.8%
Pypi	55	402	86.3%
Gem	90	259	65.3%
NPM	686	862	20.4%
Packagist	4	281	98.6%
Nuget	7	42	83.3%

A. Validity

Internal Validity: In our work, we employ classification and embedding deep learning models as our building blocks. In our implementation, we utilized the popular LSTM, Bert, Bi-LSTM. For the ecosystem classification, we have tried a series of learning models, including logistic regression, SVM, LSTM and Bert. The results show that the deep learning models achieve better and similar performance. For example, the F1 score is 0.95 when we use LSTM and 0.93 if we use Bert instead, which are very close. Therefore, the choice of classification models doesn't have a significant impact on the effectiveness of PKVIC. We use LSTM in the implementation due to its slightly better performance, which may be attributed to LSTM's ability to effectively capture both long-term and short-term dependencies in the input reports. With the continuous advancement of machine learning models, PKVIC can leverage any advanced text classification and embedding models to further enhance its performance. Regarding the dataset, we collected over 421,808 reports covering 20 popular online sources and 2,673,313 software packages across 6 popular open-source ecosystems. Therefore, these datasets offer extensive coverage, substantial volume, and strong representativeness, effectively demonstrating the effectiveness of PKVIC. For the PESE module, we compared its performance using three mainstream search engines (Google, Bing, and DuckDuckGo), as shown in Table VI. The results indicate that the choice of search engine has minimal impact on the module's performance.

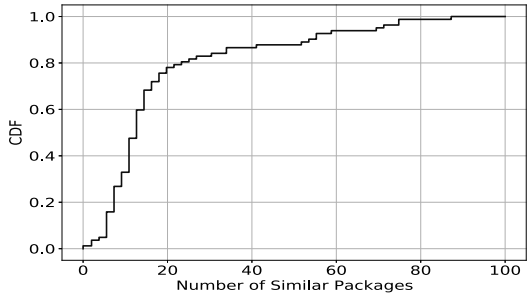

Fig. 15. Name similarities for top 20 packages in 6 ecosystems.

Construct Validity: To evaluate the performance of PKVIC, we employed several standard and widely-used metrics. These metrics include accuracy (ACC), F1 score, ROC curve, and hit rate. These metrics are well-established and capable of sufficiently reflecting the performance of PKVIC. Additionally, we used runtime as a metric to measure the overhead introduced by our approach.

B. Discussion

No-CVE-ID Reports: The vulnerability reports come from different security sources (listed in Table I). Not all reports from those sources that contain a CVE-ID due to various reasons. 28.04% of our collected vulnerability reports do not have corresponding CVE-IDs. We conducted experiments and analysis to evaluate the influence when there is no CVE-ID. First, for the ecosystem classification, we compared the performance of classification with and with not CVE-ID. For nuget, packagist, and maven, the classification accuracy is 0.998, 0.976 and 0.983 with CVE-ID, and decreases slightly to 0.997, 0.973 and 0.981 without CVE-ID, respectively. For pypi, npm and gem, the classification accuracy remains unchanged. Overall, the classification performance is virtually unchanged even there is no CVE-ID, which means the textual descriptions of reports already provide sufficient information for ecosystem classification. Second, for the NER and subsequent steps, 90.5% reports in six ecosystems already have all tokens in the CVE-ID fully included in their textual description. Hence for 90.5% reports, the absence of CVE-ID has no effect on the subsequent performance of PKVIC. For the remaining reports, PESE is designed to significantly reduce the impact of missing CVE-ID by searching and extending the potentially missing tokens, as shown in Table VI. Therefore, PKVIC is still capable of building the information connection between non-CVE-ID reports and software packages.

Package Name Similarity: With the rapid growth of the number of software packages, more similar package names appear in the ecosystems. We manually selected the top 20 recommended software packages in the six ecosystems and used Jaccard distance to search for similarity among the million-level software packages' names. As shown in Fig. 15, if the similarity is higher than 90%, we consider the two packages' names to be similar. We observed that more than 60% of the software packages' names are similar to at least 10 other software packages' names. Further, malicious users intentionally create a similar name against an original package, so-called "Bogus Packages". For instance, crypt impersonates crypto, urlib3 impersonates urllib3, and jeIlyfish impersonates jellyfish. As we mentioned before, the package name similarity would decrease the effectiveness of PKVIC, but it will has the same negative impacts on all NLP techniques.

Extension: So far, PKVIC does not support C and C++. However, there are many vulnerabilities exposed to software packages written in C and C++. Conan [41] is the software ecosystem for C/C++. However, we do not have its package index dataset. Even so, we provide the largest and most comprehensive dataset, covering 20 security sources and 6 popular software ecosystems. In our future work, we will apply PKVIC for other project management tools.

Errors and Misses: PKVIC provides the calibrated mappings between vulnerability reports and software packages of ecosystems, which inevitably suffer errors and misses when applying to the full dataset across different types over time. In our future work, we will apply crowdsourcing and peer-review calibrations for generating more accurate and complete information connections between vulnerability reports and software packages.

VI. Related Work

Online Security Sources: Today, professionals provide a variety of online security sources to the public, and prior works [12], [42], [43], [44], [45], [46], [47] have made great efforts of distributing online security information, such as vulnerability patch, exploit behavior, and threat intelligence. Sabottke et al. [42] extracted tweets of Twitter to assess the vulnerability impacts (similar as CVSS) and alert users. Liao et al. [43] proposed to extract machine-readable threat intelligence information from thousands of technical articles. Li and Paxson [48] utilized the NVD and Github websites to investigate the life cycles of security vulnerabilities and their patches. Feng et al. [46] crawled multiple online security sources for generating specific signatures to detect exploit attempts on embedded devices. Dong et al. [12] built the customized NLP models to extract vulnerable software names and versions from vulnerability reports. Due to different naming schemes and missing information, the techniques of prior works [12], [43], [46] cannot be applied for addressing our problem. Li et al. [45] leveraged several online sources (e.g., Facebook ThreatExchange and AlienVault) to conduct the analysis of threat intelligence. Bouwman et al. [47] assessed the empirical quality of online sources (threat intelligence) by interviewing 14 security professionals.

Similarly, our work also leverages 20 security sources for obtaining vulnerability reports. By contrast, we are the first to fill in the information gap between today's online security sources and ecosystems' software packages.

Software Ecosystem: Numerous applications have been developed by reusing open-source libraries from software ecosystems, and there are previous works [49], [50], [51], [52] studying on vulnerabilities caused by those reused components. Neuhaus et al. [49] proposed combining software version archives and vulnerability databases to detect vulnerable components in the Mozilla software system. Zimmermann et al. [50] conducted a large-scale empirical study on the Windows Vista system,

indicating that metrics such as code churn, code complexity, and organizational measures allow vulnerabilities to be detected with high precision but low recall rates. Edwards and Chen [51] explored the version archive of Sendmail, Postfix, Apache HTTPd, and OpenSSL and utilized the CVE information to identify exploitable bugs/vulnerable components. Scandariato et al. [53] 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. [54] proposed to utilize metadata features of a project on GitHub, including programming language, author credit, keyword, code size, and star and fork counts, in order to find vulnerabilities in open-source projects.

Gkortzis et al. [55] manually built a dataset of security vulnerabilities in open-source systems. Duan et al. [14] proposed a patching mechanism to fix and maintain open-source software applications. Zimmermann et al. [13] studied the NPM ecosystem's threats, such as malicious packages, package takeover, and account takeover. Quiring et al. [52] designed a set of simple mutations to change the programming style of software code to determine authorship attributes. Duan et al. [56] investigated the supply chain attack of the NPM ecosystem and proposed several mitigation strategies to packages with public and disclosed CVEs.

Similarly, our work also focuses on open-source packages in software ecosystems. Different from prior works, the objective of PKVIC is to fill in the missing information, i.e., establish the information connection, between vulnerability reports and software packages of ecosystems, which is the prerequisite for detecting and securing software packages in the real world. There are six popular software ecosystems being thoroughly studied in our work, including NuGet [16], PyPi [17], Maven [18], Gem [19], NPM [20], and Packagist [21].

VII. Conclusion

The information of open-source software packages in vulnerability reports is critical for software developers to build a secure software. However, in this work, we reveal that current vulnerability reports are mainly from a commercial vendor-centric point of view, missing the the information of the affected software packages in open-source ecosystems. This observation is based on 421,808 vulnerability reports collected from 20 security sources and 6 open-source software package ecosystems. To address this problem, we develop a framework PKVIC to automatically fill in the missing information of software packages in vulnerability reports. We validate the effectiveness of PKVIC by building the accurate information connections between 421,808 vulnerability reports and 2,673,313 packagers from 6 ecosystems, generating 2,703 reference URLs and 11,279 package identifiers, which fill in the missing information in vulnerability reports and significantly enhance software development security.

Acknowledgment

We would like to thank the anonymous reviewers for their detailed and insightful comments, which have helped to greatly improve the quality of the paper.

References

[1] J. Foundation, "Lodash," 2021. [Online]. Available: https://github.com/lodash/lodash
[2] J. community, "jackson_databind," 2021. [Online]. Available: https://github.com/FasterXML/jackson-databind
[3] A. License, "Htmlunit," 2021. [Online]. Available: https://github.com/HtmlUnit/htmlunit
[4] U. N. I.for Standards and T. (NIST), "NVD-CPE, Computer Security Resouce Center for Official Common Platform Enumeration (CPE) Dictionary," 2017. [Online]. Available: https://nvd.nist.gov/products/cpe
[5] NVD, "U. S. National Institute of Standards and Technology National Vulnerability Database," 2007. [Online]. Available: https://nvd.nist.gov/home.cfm
[6] SecurityGlobal.net, "Securitytracker," 2018. [Online]. Available: https://www.securitytracker.com
[7] I. npm, "NPM Security advisories," 2021. [Online]. Available: https://www.npmjs.com/advisories
[8] V. H. Nguyen and F. Massacci, "The (un) reliability of NVD vulnerable versions data: An empirical experiment on Google chrome vulnerabilities," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 493–498.
[9] C. Sabottke, O. Suciu, and T. Dumitras, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting Real-World exploits," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 1041–1056.
[10] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2201–2215.
[11] D. Mu et al., "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 919–936.
[12] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 869–885.
[13] 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 *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 995–1010.
[14] R. Duan et al., "Automating patching of vulnerable open-source software versions in application binaries," in *Proc. Netw. Distrib. Syst. Secur. (NDSS) Symp.*, 2019.
[15] T. O. Foundation, "Dependency-check tool," 2020. [Online]. Available: https://owasp.org/www-project-dependency-check/
[16] Microsoft, "Nuget is the package manager for.net," 2020. [Online]. Available: https://www.nuget.org/
[17] P. S. Foundation, "The Python package index (PyPI) is a repository of software for the Python programming language," 2020. [Online]. Available: https://pypi.org
[18] T. A. S. Foundation, "Apache maven is a software project management and comprehension tool," 2020. [Online]. Available: https://maven.apache.org/
[19] R. community, "Rubygems.org is the ruby community's gem hosting service," 2020. [Online]. Available: https://rubygems.org/
[20] I. npm, "npm is the package manager for node.js," 2020. [Online]. Available: https://www.npmjs.com/
[21] P. PACKAGIST, "Packagist is the main composer repository," 2020. [Online]. Available: https://packagist.org/
[22] Zyte, "Scrapy, An open source and collaborative framework for extracting the data you need from websites," 2021. [Online]. Available: https://scrapy.org
[23] BeautifulSoup, "A Python package for parsing html and xml documents," 2012. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/
[24] M. Bleigh and Intridea, "A flexible authentication system utilizing rack middleware," 2022. [Online]. Available: https://github.com/omniauth/omniauth
[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[26] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," 2015, *arXiv:1509.01626*.
[27] Y. Kim, Y. Jernite, D. Sontag, and A. Rush, "Character-aware neural language models," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 2741–2749.
[28] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF models for sequence tagging," 2015, *arXiv:1508.01991*.
[29] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[30] C. Douglas and S. Ravi, "17th annual symposium on foundations of computer science SFCs 1976," 1976. [Online]. Available: http://IEEExplore.ieee.org
[31] M. Burtsev et al., "DeepPavlov: Open-source library for dialogue systems," in *Proc. ACL Syst. Demonstrations*, 2018, pp. 122–127.
[32] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.
[33] N. Tschacher, "Googlescraper, GoogleScraper parses Google search engine results easily and in a fast way," 2020. [Online]. Available: https://github.com/NikolaiT/GoogleScraper
[34] elastic.co, "elasticsearch: Free and open, distributed, restful search engine," 2021. [Online]. Available: https://github.com/elastic/elasticsearch
[35] rubysec, "Ruby advisory database, community effort to compile all security advisories that are relevant to ruby libraries," 2021. [Online]. Available: https://github.com/rubysec/ruby-advisory-db
[36] GitHub, "Github security advisory database," 2018. [Online]. Available: https://github.com/advisories
[37] GitLab, "Gitlab secure, maintenance and update of the vulnerabilities database," 2019. [Online]. Available: https://docs.gitlab.com/ee/user/application_security/
[38] pyup.io, "Safety DB, A database of known security vulnerabilities in python packages," 2021. [Online]. Available: https://github.com/pyupio/safety-db
[39] FriendsOfPHP, "PHP security advisories database," 2021. [Online]. Available: https://github.com/FriendsOfPHP/security-advisories
[40] Synk, "Snyk vulnerability database," 2019. [Online]. Available: https://security.snyk.io/disclosed-vulnerabilities
[41] Conan.io, "Conan, The C/C package manager," 2020. [Online]. Available: https://conan.io/
[42] C. Sabottke, O. Suciu, and T. Dumitraş, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits," in *Proc. 24th USENIX Secur. Symp.*, Berkeley, CA, USA, 2015, pp. 1041–1056. [Online]. Available: [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831209
[43] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah, "Acing the IOC game: Toward automatic discovery and analysis of open-source cyber threat intelligence," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 755–766.
[44] X. Shu et al., "Threat intelligence computing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1883–1898.
[45] V. G. Li, M. Dunn, P. Pearce, D. McCoy, G. M. Voelker, and S. Savage, "Reading the tea leaves: A comparative analysis of threat intelligence," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 851–867.
[46] X. Feng et al., "Understanding and securing device vulnerabilities through automated bug report analysis," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 887–903.
[47] X. Bouwman, H. Griffioen, J. Egbers, C. Doerr, B. Klievink, and M. van Eeten, "A different cup of TI? The added value of commercial threat intelligence," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 433–450.
[48] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2201–2215. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134072
[49] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 529–540. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315311
[50] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 421–428.
[51] N. Edwards and L. Chen, "An historical examination of open source releases and their vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 183–194.
[52] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 479–496.
[53] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
[54] H. Perl et al., "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813604
[55] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "VulinOSS: A dataset of security vulnerabilities in open-source systems," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 18–21. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196454
[56] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proc. Netw. Distrib. Syst. Secur. (NDSS) Symp.*, 2021.

Jinke Song received the master's degree in computer science from Beijing Jiaotong University, in 2017, and is now working toward the PhD degree with the School of Computer and Information Technology, Beijing Jiaotong University, China. His main research areas are cyberspace security and Internet of Things security.

Qiang Li received the PhD degree in computer science from the University of Chinese Academy of Sciences, in 2015. Currently, he is an associate professor with the School of Computer and Information Technology, Beijing Jiaotong University, China. His research interests revolve around Internet of Things, networking systems, network measurement, machine learning for cybersecurity, and mobile computing.

Haining Wang received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, Michigan, in 2003. Currently he is a professor with the Department of Electrical and Computer Engineering, Virginia Tech. His current research interests include security, networking systems, and cloud computing.

Jiqiang Liu received the BS and PhD degrees from Beijing Normal University, in 1994, and 1999, respectively. He is currently a professor with the School of Computer and Information Technology, Beijing Jiaotong University. His current research interests include cryptographic protocols, privacy preserving, and network security.