

# Find the Culprit: Query-Driven Tracing of Cybersecurity Vulnerabilities using Python’s PyPI Dependencies

Cara Stievater  
cgs9212@rit.edu

Rochester Institute of Technology  
Rochester, New York, USA

## ABSTRACT

Programming packages simplify the development of high-level applications by allowing developers to integrate pre-built libraries into their projects. These libraries often rely on other packages, forming long and interconnected dependency chains. As a result, a single vulnerable or malicious module can compromise the security of an entire software system. The National Institute of Standards and Technology (NIST) catalogs threats in code through the open-source Common Vulnerabilities and Exposures (CVE) database, which documents known weaknesses across different software versions. This research combines these public sources to identify relationships between Python packages and known CVEs in a data-, query-driven fashion. We use a relational database to integrate the sources and SQL queries to expose direct and indirect risks across package networks. With this approach, researchers and developers can better understand how vulnerabilities spread and anticipate potentially exposed packages before they are exploited.

### ACM Reference Format:

Cara Stievater. 2025. Find the Culprit: Query-Driven Tracing of Cybersecurity Vulnerabilities using Python’s PyPI Dependencies. In *Proceedings of KDD ’25 Undergraduate and Master’s Consortium (KDD ’25)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 INTRODUCTION

In Python, common functionalities are stored in modules, and similar modules are grouped into packages. This structure leads to a chain of dependencies, where packages and modules can be bundled into libraries [14]. Many can be found through the Python Standard Library [5]. Python developers have created additional tools beyond the standard library, which can be shared by uploading their packages to public repositories. One of the most widely used repositories is the Python Package Index (PyPI), which has Python related software contributed by third-parties [4].

Opening Python to this kind of collaboration has led it to become one of the most popular and versatile programming languages. However, despite the widespread use of Python, there remains a lack of regulation over third-party libraries. This means that a single compromised module in a dependency tree can expose an entire application to attack. This is not uncommon – one study found that 80% of dependencies in software projects contain vulnerabilities [7]. Knowing why these vulnerabilities happen in the first place, how to avoid them, and the ability to track them remains a top priority in security.

*KDD ’25, Aug 03–07, 2025, Toronto, Canada*  
2025. ACM ISBN 978-1-4503-XXXX-X/2018/06  
<https://doi.org/XXXXXXX.XXXXXXX>

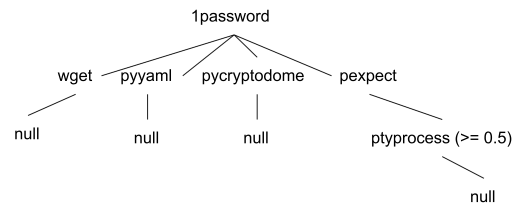


Figure 1: Tracing dependencies starting at 1password

To illustrate this concept, consider Figure 1, displayed is the dependency tree of the Python package *1password*. This package depends on *wget*, *pyyaml*, *pycryptodome* and *pexpect*. The first three packages are independent of any additional packages, while *pexpect* relies on any version higher than or equal to 0.5 of *ptyprocess*, which isn’t dependent on further packages. In this example, specific versions of *1password* have a medium vulnerability, found with CVE-2022-32550, and linked to numerous versions of the *1password* library, even for different platforms values like Android [11]. This is an example of a direct vulnerability, where there is a weakness detected at the first level. However, all the other dependencies of *1password* are indirectly affected by the same vulnerability. Unfortunately, these indirect vulnerabilities are only exposed when integrating data from package dependencies and vulnerabilities.

Similarly, *1password* may be indirectly affected by vulnerabilities in its dependencies, such as *pyyaml* and *wget*. Looking at *pyyaml*, it’s associated with CVE-2017-18342. If *1password* depends on that vulnerable versions of *pyyaml*, it would now be exposed to the vulnerability, while the other packages remain independent from this threat. However, if *1password* required a version of *pyyaml* after the vulnerability was resolved, it would not be affected anymore. Understanding this process is central to our data collection and analysis, as we discuss below.

But what leads to a compromised package? There are often errors or gaps in logic within the code that weaken its security which, once identified, attackers can exploit. One of the best-known examples of this is the 2017 Equifax breach, where over 148 million individuals’ personal data were compromised. The root cause was a vulnerability in Apache Struts, a widely used Java web application framework, which had improper exception handling [2]. Though not Python-specific, this breach highlights how overlooked flaws in third-party components can lead to massive data leaks. In addition to accidental flaws, intentional malware also poses a threat when attackers introduce malicious code.

Even if every vulnerability does not lead to catastrophe, it is impossible to predict which one might trigger the next major breach.

With this in mind, this paper investigates how vulnerabilities originate and spread through Python packages. We aim to improve our understanding and reduce the associated risks with vulnerabilities.

## 1.1 Related Work

Prior studies have recognized similar issues, and have explored solutions through varying tactics.

Zimmermann et al. [17] collects data through npm API endpoints, and uses graphs to justify their concern of these third-party installations. Overall, Zimmermann et al. [17] does a phenomenal job of highlighting the larger issue over installation managers. They conclude that "npm is a small world with high risks," meaning that npm is small in perspective to the dense relationship between packages. This interconnectivity is what leads to a higher risk to jeopardizing one's security. Although providing solid reasoning as to why developers should be cautious of package and reporting the majority of their finding on graphs, the paper doesn't expand into tracing current or predicting future vulnerabilities.

Germán Márquez et al. [6], use Depex to draw conclusions from selected projects and create Satisfiability Modulo Theories models based on Depex's conclusions. The paper creates complex and incredibly insightful graphs that completely change our understanding of direct and indirect vulnerabilities. But, they acknowledge their limitations using Depex, like inconsistencies with collected data and slow execution times.

What differs the most is our strategy for collecting data. We implemented a query-based approach that integrates PyPI and CVE data to uncover both existing vulnerabilities and possible future exposures. As a web of dependencies form, our method keeps the dependencies traceable, combining SQL queries to follow relationships in Python packages. Most importantly our research allows for reusability and adaptability, allowing continuing research to use our methods and databases to expand into other theories or focus on other types of vulnerabilities.

The rest of the paper is organized as follows: Section 2 introduces the necessary background, Section 3 presents our data-gathering approach, Section 4 discusses query-driven analysis of vulnerabilities, and Section 5 presents our conclusions and future work.

## 2 BACKGROUND

To understand Python packages and how they can be prone to attack, one must understand the role of CVEs and CPEs, and their interaction with PyPI. This section introduces the origin, structure, and access methods for these resources, providing context for later discussions about how these resources are later implemented in this research.

### 2.1 Understanding CVEs

The National Vulnerability Database (NVD), developed by the National Institute of Standards and Technology (NIST), provides publicly accessible data that consolidates around 264,000 vulnerabilities found from 1998 to current times. According to NIST, a vulnerability is the following: A weakness in computational logic (e.g., code) found in software and hardware components that when exploited,

results in a negative impact on confidentiality, integrity, or availability [13]. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety). The cybersecurity vulnerabilities that fit this definition are organized in structured JSON files published by the Common Vulnerability & Exposure (CVE). Each CVE is assigned a unique identifier in the format CVE-[year]-[number], where the year is when the file was released, and code is a unique combination of numbers to distinguish files. The CVEs can be accessed through: <https://cveawg.mitre.org/api/cve/CVEidentifier>, where CVEidentifier is the previously stated combination. These files are represented using the Security Content Automation Protocol and highlight attacks that span various information technology systems, software, and package [3].

### 2.2 Understanding CPEs

Vulnerabilities of a program may be given a standardized code known as Common Platform Enumeration (CPE). Structured similarly to Uniform Resource Identifiers, CPEs are encoded as [12]:

```
cpe:/{part}:{vendor}:{product}:{version}:{update}:
{architecture}:{language}:{modifier}
```

The fields are as follows [15]: 1) **part**: the type of system detected (a: application, h: hardware, or o: operating system), 2) **vendor**: the organization that created the product, 3) **product**: the name of the product, 4) **version**: the version number for the product, 5) **update**: the update for the product and version detected, 6) **edition**: the edition of the software, 7) **language**: the language detected.

It is worth noting that there is a possibility of different CVEs being associated to multiple CPEs or none at all. Additionally, for our purposes, the most valuable information can be taken from the product portion, which will be explained further in Section 4.

### 2.3 Understanding PyPI

The Python Package Index is a searchable repository of packages in Python, encompassing around 580,000 projects, 6,000,000 releases, 12,000,000 files, and 870,000 users [4]. These attributes can be searched easily through their API for each package: <https://pypi.org/pypi/package/json>, <https://pypi.org/pypi/package/version/json>. Knowing the package name, these API's reveal vital information, such as what dependencies the provided package has.

### 2.4 Correlation between CVE, CPE, and PyPI

CVEs are made to identify packages with vulnerabilities. Some CVEs have one or multiple CPEs, which can be used to group together similar attacks. Breaking up the CPE, and accessing the element in the fourth portion, the product section, will reveal the name of the vulnerable package. As mentioned in Section 2.3, we use the package name and the version, if available, from the CPE to retrieve the package information using PyPI's API. If found, then the product is assumed to be a product from Python. Using this formula results in a single tracing, finding what CVE files correlate to what Python packages. To go a step further, starting with a package, there can be full trace for all dependencies, determining the CPE and CVEs that correlate, ultimately being traced back to the starting package.

### 3 DATA GATHERING PROCESS

Data is collected primarily through permitted API endpoints of various sites. NIST's APIs offer access to all CVEs, while PyPI's provides package details.

The initial set of files is found using the following URL: `https://services.nvd.nist.gov/rest/json/cves/2.0?startIndex=cidx` where `cidx` is an integer value to search a specified CVE, and then 2,000 CVEs follow. The contents of this link were saved in a JSON, later stored as a ZIP file (containing more than 100 JSON files) and interpreted locally, leading to 264,366 unique CVEs. We process these files as well as PyPI's package information and integrate them under the same database presented in Figure 2.

The rest of this section will discuss in what ways the programs are catered to processing these JSON structures, how the data for each table is retrieved. When referring to the schemas, all capitalized tables are related to CVEs, e.g., **CVE**, table names beginning with "PyPI" are associated with PyPI, e.g., **PyPI\_versions**, and any italicized tables combine both datasets, e.g., *CPE\_PyPi\_ref*.

**CVE.** Each CVE's metadata is saved in the data table, **CVE**. A walk through of the data can be followed using Figure 3. The unique title column can be found from: "id": "CVE-2021-30506". The URL showing the source of the vulnerability is saved in `vulnstatus` from line: "sourceIdentifier": "chrome-cve-admin@google.com". The date the CVE was published is saved in the `published` column, found by: "published": "2021-06-04T18:15:07.720". The date the CVE was last modified, is saved under `lastModified`, and found from: "lastModified": "2023-11-07T03:33:02.610".

**Weaknesses.** Because weaknesses in software packages can lead to vulnerabilities, many CVE records categorize them by their Common Weakness Enumeration (CWE). CWE often appear in the form 'CWE-[number]' where 'number' is a specific category of weakness, and are interpreted similar to CPEs (see Section 2.2). For example, CWE-74 indicates "Improper Neutralization of Special Elements in Output Used by a Downstream Component" [10]. Each CVE may include multiple sources (the contributor of the weakness information) and types, which can be either Primary or Secondary. Primary sources include the NVD and CNAs (CVE Numbering Authorities) that have reached the "provider level" in the Common Vulnerability Mapping and Analysis Program. If a "provider level" CNA's submission has been reviewed by the NVD, then the NVD appears as the Primary source, and the CNA is marked as Secondary. According to the NVD, only 10% of weaknesses files encounter this condition [9]. Looking at Figure 3, the source is listed as `nvd@nist.gov` and the type is Primary, meaning the NVD provided and validated this weakness information. Some examples of Secondary sources include `security-advisories@github.com` or `contact@wpscan.com`. These sources and types are stored in **CVE\_WEAKNESSES** under their respective columns. Then, contained in another list within weaknesses called "description" contains the CWE value. This value is stored in **WEAKNESSES\_DESCRIPTIONS**, under column "value". A CVE file can have multiple weakness sources and types and that weakness can have one or many CWE codes. To mimic this relationship, the table **CVE\_WEAKNESSES** inherits an id from the `cve` table, and **WEAKNESSES\_DESCRIPTIONS** inherits an id from the **CVE\_WEAKNESSES** table.

**References.** If "references" is present, the document includes a URL with additional insight on the vulnerability reported. Following Figure 3, there is a URL to the organization that sourced the information as follows: "url": "https://chromereleases.googleblog.com/(...)". This URL is saved in the table **CVE\_REFERENCES** in column `url`, which corresponds to "chrome-cve-admin@google.com" in this case. We can find tags in the same section of the CVE, used by NVD for more efficient searching capabilities. The **TAGS** table is saved similar to **CPE**, where there is a reference to compare all repeating codes. Unique tags can be found in the JSON files in the "tags" category, given as a list. The unique values are saved in the table **TAGS\_REF**. An example is seen in Figure 3 where two tags are given, "tags": ["Release Notes", "Vendor Advisory"]. These unique values are then paired with `cve` reference id's from **CVE\_REFERENCES**.

**Vulnerability description.** The section "descriptions" in the JSON file provides details of the vulnerability and its effects on the platform at hand. One example can be found in Figure 3 in English (indicated by "lang": "en"). Looking at the English version, the section "value" provides a description for attack: "value": "Incorrect security UI in Web App Installs in Google Chrome on Android prior to 90.0.4430.212 allowed an attacker who convinced a user to install a web application to inject scripts or HTML into a privileged page via a crafted HTML page." This information is saved in the table **CVE\_DESCRIPTIONS**, along with the language it is written in.

**Configurations.** Some common vulnerabilities are categorized through CPEs (see Section 2.2). These unique product codes are saved in the table **CPE\_REF** and are given a unique ID. This unique ID is then paired with the corresponding CVE's id that it was found in. These values are inserted into the **CPE** table. Keep in mind that CVEs can refer to multiple CPEs- meaning, a single vulnerability may affect more than one product or version. In the CVE file in Figure 3, the section "cpeMatch" within "configurations" illustrates this part. Each "nodes" field can contain a single section or an array with several sections that specify multiple criteria referring to CPEs. An example in Figure 3 is the CPE that refers to Google Chrome. The criteria is a Boolean expression with AND and OR operations. This information is saved in the **CPE\_REF** table, and referenced by the **CPE** table to connect CVE files with CPE codes.

**PyPI.** Using the method described in Section 2.4, we search for packages using PyPI and we update dependency information in the corresponding tables. Following Figure 3, one of the CPEs saved is, `cpe:2.3:a:google:chrome:*:*:...`. Because there is no package in PyPI named "chrome," it is assumed this CPE has no association with Python. However, this is not the case for the package "gpac" taken from the following CPE: `cpe:2.3:a:gpac_project:gpac:*:*:...`. In this case, we retrieve a JSON file; an excerpt of it is found in Figure 4.

**Info.** If dependencies are present, a list of those dependencies is provided with version constraints. In this example, the dependencies can be found in the "requires\_dist" field. Dependencies include the name of the package and the version it requires, e.g., "openpyxl>=3" implies that it requires at least version 3 of openpyxl. Dependencies can impose extra constraints, e.g., "mkdocs>=1.4.0; extra=="docs"" requires mkdocs and its documentation.

To show how these values are stored in our database, let's consider the dependency "openpyxl>=3." The initial step would be

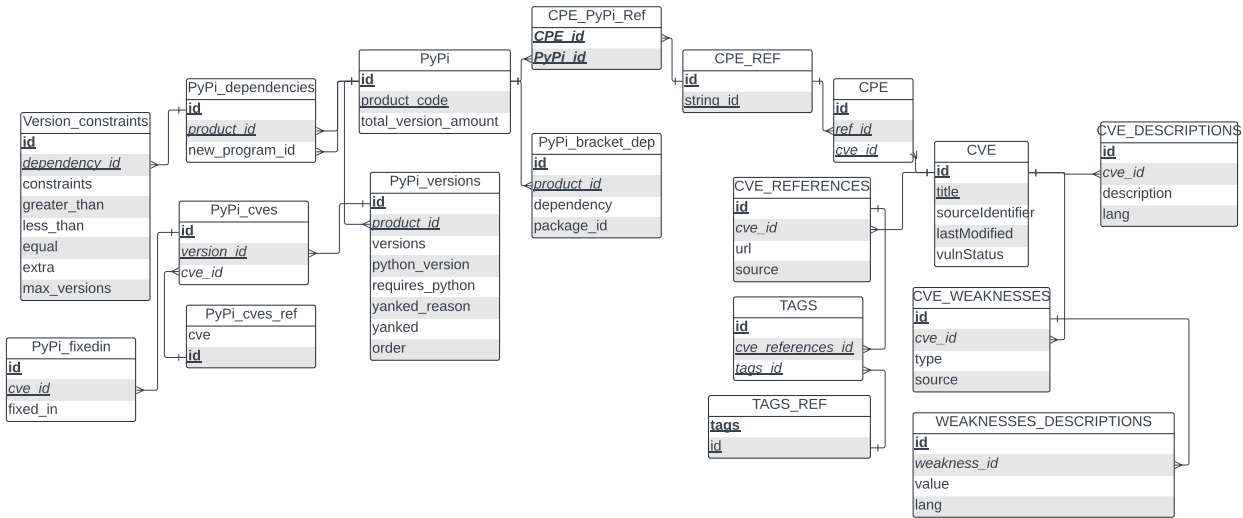


Figure 2: Connecting PyPI and CVE data (pk bold, fk italicized & arrow from reference, unique values underlined)

```

370 "cve": {
371   "id": "CVE-2021-30506",
372   "sourceIdentifier": "chrome-cve-admin...",
373   "published": "2021-06-04T18:15:07.720",
374   "lastModified": "...",
375   "vulnStatus": "Modified",
376   "descriptions": [{"lang": "en",
377     "value": "Incorrect security UI..."},...
378 ],...
379   "weaknesses": [{"source": "nvd@nist.gov",
380     "type": "Primary",
381     "description":
382       [{"lang": "en", "value": "CWE-74"}]}
383 ],
384   "configurations": [{"operator": "AND",
385     "nodes": [{"operator": "OR",
386       "negate": false,
387       "cpeMatch": [
388         "cpe:2.3:a:google:chrome:*:*:*",
389         ... ]}],... ]
390 }, {"nodes": [{"operator": "OR",
391     "negate": false, ... }... ]}],
392   "references": [{"
393     "url": "...",
394     "tags": [
395       "Release Notes", "Vendor Advisory" ] }, ... ]}

```

Figure 3: Dissecting the CVE JSON

inserting the dependency name, 'openpyxl' to the table PyPi if it does not already exist. A new row is then inserted into the table PyPi\_dependencies, where the id of the initially searched product, in this case 'gpac,' is paired with a new product id, like 'openpyxl.' When a dependency includes additional constraints following it, a new row is inserted into Version\_constraints, where the version's id and unique id are saved. In both columns greater\_than and equal, 3 would be inserted. In another example, using package mapproxy's

```

428 "info": { ...
429   "release_url": "https://pypi.org/project/gpac/1.0.14/",
430   "requires_dist": [
431     "openpyxl>=3", "tabulate>=0.9", "scipy>=1",
432     "sympy>=1", "numpy>=2", "matplotlib>=3",...
433 ], "requires_python": ">=3.7",
434   "summary": "A Python package for extracting...",
435   "version": "1.0.14",
436   "yanked": false,
437   "yanked_reason": null },...
438   "releases": {
439     "0.0.1": [ {
440       "comment_text": "",
441       "digests": {
442         "blake2b_256": ...,
443         "md5": ...,
444         "sha256": ... },
445       "downloads": -1,
446       "filename": "gpac-0.0.1.tar.gz", ...
447       "packagetype": "sdist",
448       "python_version": "source",
449       "requires_python": ">=3.7",
450       "size": 16385, ...
451       "yanked": false, "yanked_reason": null },...
452     "1.0.9": [... ]},
453   "urls": [{...
454     "filename": "gpac-1.0.14-py3-none-any.whl",
455     "has_sig": false,
456     "packagetype": "bdist_wheel",
457     "python_version": "py3",
458     "requires_python": ">=3.7",...
459     "yanked": false,
460     "yanked_reason": null},... ],
461   "vulnerabilities": []

```

Figure 4: Dissecting the PyPI JSON

```

465 "info": {
466   "author": "The Borg Collective (see AUTHORS file)",
467   "author_email": "borgbackup@python.org",
468   "bugtrack_url": null,
469   "name": "borgbackup", ...
470   "version": "1.2.4",
471   "yanked": false, "yanked_reason": null
472 },
473 "urls": [...],
474 "vulnerabilities": [ {"aliases": ["CVE-2023-36811"],
475   "details": ...
476   "fixed_in": ["1.2.5"],
477   "id": "GHSA-8fjr-hghr-4m99",
478   "link": ..., "source": "osv",
479   "summary": null, "withdrawn": null},
480 {"aliases": ["CVE-2023-36811", "GHSA-8fjr-hghr-4m99"],
481   "details": ..., "fixed_in": ["1.2.5"], ...} ]

```

Figure 5: Searching package versions in PyPi

dependencies: "Pillow!=2.4.0,! =8.3.0,! =8.3.1." From the constraints '! =2.4.0,! =8.3.0,! =8.3.1', we can interpret that all versions are valid except versions 2.4.0, 8.3.0, and 8.3.1. In columns less\_than and greater\_than, the values would be inserted. Another dependency type not shown in the example is as follows: "requires\_dist": [ "ipython[doc,matplotlib,test,test\_extra];... This indicates that the package dependency depends on other modules/packages. In the example, ipython depends on packages: doc, matplotlib, test, and test\_extra. When saving these values in PyPi\_bracket\_dep, it is also important to keep track of what dependencies are grouped together, since one package can have different dependency requirements.

**Releases.** Within the JSON file retrieved from PyPi, all past releases are organized in order from oldest to most recent in a number of nested JSON documents. The following discusses how the values from Figure 4 are stored in the table PyPi\_versions. The value "0.0.1" refers to the release version and is placed in the column, versions. The column, requires\_python, stores the python versions that are compatible with the package. In this case, the Python version must be greater or equal than 3.7. If the version is inaccessible due to vulnerabilities, then the yanked column is set to true. And if yanked is true, then the yanked\_reason column stores the reason that the version is unavailable. Lastly, the column, order, keeps track of the order of releases (from oldest to newest). The column order is mainly important when considering dependencies. For instance, when there is a dependency requiring package gpac, greater than version 1.0.3, we know 1.0.9 is a later version because it comes after 1.0.3 in this nested JSON document, and is therefore unaffected by version 1.0.3.

**Vulnerabilities.** Using the PyPi URL that requires both package and version (this data is found in PyPi and PyPi\_versions), Section 2.3, may reveal the CVE files associated with specific PyPi packages. In the example URL, <https://pypi.org/pypi/borgbackup/1.2.4/json>, version 1.2.4 of package borgbackup is searched for. The results are in Figure 5. In this case "CVE-2023-36811" has ties to borgbackup version 1.2.4. This CVE is stored in the table PyPi\_cves, which inherits the id from table PyPi\_versions. In the same section of the JSON, the table PyPi\_fixedin, saves values from the list "fixed\_in",

```

523 SELECT DISTINCT p.product_code, p.id FROM PyPi p
524 JOIN PyPi_versions v ON p.id = v.product_id
525 JOIN PyPi_cves c ON v.id = c.version_id
526 LEFT JOIN PyPi_fixedin f ON c.id = f.cve_id
527 WHERE f.cve_id IS NULL
528

```

Figure 6: Retrieve currently vulnerable product codes

```

529 SELECT p.product_code, p.id FROM PyPi p
530 JOIN PyPi_versions v ON v.product_id = p.id
531 JOIN PyPi_cves c ON c.version_id = v.id
532 JOIN PyPi_fixedin f ON f.cve_id = c.id
533

```

Figure 7: Retrieve resolved vulnerable product codes

declaring if and when the vulnerability was fixed. Although the information provided in section vulnerabilities is useful, the JSON files are not necessarily up-to-date, and not all CVEs are listed for each product. Thus, it cannot be the only method used when searching for connections between CVEs and Python packages.

**Combining CPE and PyPi.** If we conclude that the CPE is associated with Python (see Section 2.4), then the CPE value is matched with its product code in PyPi. They are organized in the table as *CPE\_PyPi\_Ref*, where the id from the CPE\_REF table is paired with the id of the PyPi table. The pair must be unique.

## 4 DATA ANALYSIS

As previously discussed, the large network of Python package dependencies makes it difficult to identify and trace vulnerabilities. To support vulnerability analysis, we developed a number of SQL queries that allow analysts to explore the status of packages – whether they are currently vulnerable, have a history of resolved issues, or have remained unaffected.

The first question an analyst may ask is packages that have unresolved vulnerabilities. Figure 6 presents the SQL query to retrieve packages that can lead back to CVEs, which shows that any retrieved package was once (if not still) vulnerable. The query uses the PyPi\_cves table linked with fix data from PyPi\_fixedin to filter for unresolved vulnerabilities. Furthermore, it is important to evaluate resolved vulnerabilities. This can be achieved by combining products from the PyPi\_cve table and solutions from the PyPi\_fixedin table. If there is a match, it can be assumed that the vulnerability at hand was fixed. Assuming all vulnerabilities for a product are resolved, that package is now free from error. Figure 7 retrieves packages that had vulnerabilities, but they have been resolved based on the criteria stated previously (there is at least one match in the PyPi\_fixedin table). Figure 8 retrieves fixed CVEs for a specific package. The query helps assess a package's history of vulnerabilities that have already been resolved.

Finally, Figure 9 links relevant Python packages to their CVE descriptions and related weakness categories. This is to provide a description of all current and past weaknesses and vulnerabilities associated with Python packages. Ultimately allowing for a more semantic understanding of the packages exposure history. Running

```

581 SELECT cve FROM PyPi_cves c
582     JOIN PyPi_versions v ON c.version_id = v.id
583     JOIN PyPi p ON v.product_id = p.id
584     JOIN PyPi_fixedin f ON f.cve_id = c.id
585     WHERE p.id = %s
586
587
588
589

```

Figure 8: Retrieve resolved vulnerable CVEs

```

590 SELECT DISTINCT p.product_code, c2.title,
591     c3.descriptions, d.value_ FROM PyPi p
592     JOIN PyPi_versions v ON p.id = v.product_id
593     JOIN PyPi_cves c1 ON c1.version_id = v.id
594     JOIN CVE c2 ON c1.cve = c2.title
595     JOIN CVE_DESCRIPTIONS c3 ON c2.id = c3.cve_id
596     AND lang = 'en'
597     JOIN CVE_WEAKNESSES w ON c2.id=w.cve_id
598     JOIN WEAKNESSES_DESCRIPTIONS d ON
599     d.weakness_id=w.id AND lang_ = 'en'
600
601
602
603

```

Figure 9: Retrieve CVE descriptions and product codes

Table 1: Runtimes in seconds of the queries discussed

	Figure 6	Figure 7	Figure 8	Figure 9
Test 1	6.412	6.313	0.007	56.844
Test 2	6.351	6.232	0.005	53.701
Test 3	6.675	6.402	0.005	54.297
Average	6.479	6.315	0.006	54.947
# Rows	150	631	1	7,335

this query we retrieved that Tensorflow is the Python package with the most vulnerabilities, having 538 product codes.

Table 1 presents three rounds of retrieval times with the different queries previously described. These times are greatly impacted by how many rows are returned. Thus, Figure 8’s query has the best runtimes by far, only returning one row. This is indicative of the efficiency of our queries, shown by how fast it is to search through the tens of millions of rows of data. The worst runtime is achieved by Figure 9’s query, caused by the amount of rows returned and not necessarily because it queries through the most amount of tables.

**Limitations.** The assumption that a CVE affects a Python package if it can be found using the PyPI API may not always be correct. Packages for different programming languages like Java could use the same name, meaning the Python package was actually never linked to vulnerability. Additionally, the data was collected by taking the initially marked “Python” files and finding their dependencies. These dependencies were then stored in the database and then were searched for. This recursive method may have missed some relevant Python packages, particularly those not directly linked from initial seeds. Although it is not a requirement to have every Python package and their dependencies, it could be useful when finding packages that have no history of vulnerabilities.

When collecting PyPI’s release data, we realized our program collecting this data never seemed to end. The cause being that

new versions are uploaded every few minutes. To avoid repeatedly reloading the entire dataset, we built a resume capability: using the data we already had, any values with a timestamp after November 1, 2024, were removed. Then, the program was re-run, only look at packages that we had yet to trace and only accepting values from before November 1, 2024. This process let us keep our data current without re-collecting unchanged releases or reprocessing hundreds of thousands of existing entries.

## 5 CONCLUSIONS

Through the permitted use of APIs from trusted sources such as NIST and PyPI, the research we have presented in this paper successfully retrieves and processes a comprehensive dataset of around 264,000 CVEs. By structuring the extracted JSON data into a database the system enables efficient storage, searchability, and future analysis of software vulnerabilities.

Ultimately, this work aims to deepen our understanding of vulnerabilities from third-party sources and enhance future tools for vulnerability tracking. By making vulnerability data easier to query, categorize, and analyze, both researchers and developers are better equipped to respond to emerging threats in these open-source environments and ensure safe collaborative opportunities for developers. As a final note, while our findings emphasize the risk of using third-party packages, there is no reason to avoid exploring beyond the Python Standard Library or contributing to the open-source community. The following section outlines safe practices for responsibly using third-party packages.

**Safe Practices.** Developers concerned by these findings can adopt several best practices to ease any fears: 1) Use specific distributions such as Debian, Ubuntu, Fedora, or SuSE, which only allow certified developers to upload packages and include regular security checks and threat warnings [8]. 2) Sandboxing—installing packages in a container or VM—helps ensure that if a package is compromised, only the isolated environment is affected rather than the entire system [8]. 3) Use stricter dependency managers, such as pipenv or poetry, which verify packages before installation [1].

**Future work.** Searching for similar vulnerabilities within related programming languages, such as Java, is one of the main goals in our future work. Comparing vulnerabilities among languages is both interesting and insightful, and due to the reproducible nature of our research, additional data gathering and analysis processes would be similar. Another direction is to represent the relational database presented in this paper as a knowledge graph. A knowledge graph is, on one hand, more flexible to store and analyze vulnerabilities. Furthermore, a knowledge graph representation can be used to predict new vulnerabilities in both transductive (existing packages and vulnerabilities) and inductive (unseen packages and vulnerabilities) settings. Finally, a knowledge graph can be used in retrieval-augmented generation tasks, aiming to improve the accuracy of large language models.

Lastly, NIST announced recently that every CVE published before January 1, 2018 is now considered deferred to the NVD dataset. This is because NIST is currently backlogged in processing vulnerability reports. This will only limit the present research, as not all vulnerabilities will be identified [16].

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2346959.

## REFERENCES

- [1] Bolster AI. 2023. *PyPI Supply Chain Attacks: The Rise of Malicious Packages*. <https://bolster.ai/blog/pypi-supply-chain-attacks> Accessed: 2025-04-25.
- [2] Bank Info Security. 2019. *Equifax's Data Breach Costs Hit \$1.4 Billion*. <https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473> Accessed: 2025-04-25.
- [3] MITRE Corporation. 2023. *CVE - Common Vulnerabilities and Exposures*. <https://cve.mitre.org/index.html> Accessed: 2025-04-25.
- [4] Python Software Foundation. 2023. *PyPI - The Python Package Index*. <https://pypi.org> Accessed: 2025-04-25.
- [5] Python Software Foundation. 2023. *Python Standard Library Documentation*. <https://docs.python.org/3/library/index.html> Accessed: 2025-04-25.
- [6] A. Germán Márquez, Ángel Jesús Varela-Vaca, María Teresa Gómez López, José A. Galindo, and David Benavides. 2024. Vulnerability impact analysis in software project dependencies based on Satisfiability Modulo Theories (SMT). *Computers Security* 139 (2024), 103669. doi:10.1016/j.cose.2023.103669
- [7] Serena Elisa Ponta Antonino Sabetta Fabio Massacci Ivan Pashchenko, Henrik Plate. 2018. Vulnerable open source dependencies: counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 1–10. doi:10.1145/3239235.3268920
- [8] Codethink Ltd. 2023. *The Safety of PyPI: Is It Secure Enough?* <https://www.codethink.co.uk/articles/2023/pypi-safety/> Accessed: 2025-04-25.
- [9] National Institute of Standards and Technology. 2024. *Vulnerability API - National Vulnerability Database*. <https://nvd.nist.gov/developers/vulnerabilities> Accessed: 2025-05-14.
- [10] National Institute of Standards and Technology. 2024. *Vulnerability Categories - NVD*. <https://nvd.nist.gov/vuln/categories> Accessed: 2025-05-07.
- [11] National Institute of Standards and Technology (NIST). 2022. *CVE-2022-32550: AgileBits 1Password Apps and Integrations Vulnerability*. Accessed: 2025-05-08.
- [12] National Vulnerability Database (NIST). [n. d.]. *CPE Search Results: pip*. [https://nvd.nist.gov/products/cpe/search/results?keyword=cpe:2.3:a:pyppa:pip:\\*:\\*:\\*:\\*:\\*:\\*:\\*&status=FINAL,DEPRECATED&orderBy=CPEURI&namingFormat=2.3](https://nvd.nist.gov/products/cpe/search/results?keyword=cpe:2.3:a:pyppa:pip:*:*:*:*:*:*:*&status=FINAL,DEPRECATED&orderBy=CPEURI&namingFormat=2.3) Accessed: 2025-04-25.
- [13] National Institute of Standards and Technology. 2023. *National Vulnerability Database (NVD)*. <https://nvd.nist.gov> Accessed: 2025-04-25.
- [14] Shiksha Online. 2024. *Difference Between Module and Package in Python*. <https://www.shiksha.com/online-courses/articles/difference-between-module-and-package-in-python/> Accessed: 2025-04-25.
- [15] Nicholas Sciberras. 2014. *Common Platform Enumeration (CPE) Explained*. <https://www.acunetix.com/blog/articles/common-platform-enumeration-cpe-explained/> Accessed: 2025-04-25.
- [16] Alex Scroton. 2025. *NIST calls time on older vulnerabilities amid surging disclosures*. <https://www.computerweekly.com/news/366622153/NIST-calls-time-on-older-vulnerabilities-amid-surgings-disclosures> Accessed: 2025-04-25.
- [17] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with high risks: a study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) (*SEC'19*). USENIX Association, USA, 995–1010.

## A REPRODUCIBILITY

The data discussed in the study can be accessed through the GitHub link: <https://github.com/cgs9212/Fall-Research-2024/>.