NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A RISK ANALYSIS OF SOFTWARE DEPENDENCIES FOR THE AI/ML SUPPLY CHAIN

by

Alexander S. Tum

September 2023

Thesis Advisor: Joshua A. Kroll
Co-Advisor: Britta Hale

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2023	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
A RISK ANALYSIS OF SOFTWARE DEPENDENCIES FOR THE AI/ML SUPPLY CHAIN

5. FUNDING NUMBERS

6. AUTHOR(S) Alexander S. Tum

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Artificial intelligence (AI) and machine learning (ML) offer new capabilities for the overall technology ecosystem. As it forms the foundation for new technology, the security of a final software product depends greatly on that of the underlying supply chain, including its software dependencies. This study examines a portion of the supply chain for AI/ML by mapping the dependencies of a select sample of ML libraries for vulnerabilities. We search for a relationship between the depth of a dependency within a sample library's dependency tree and the amount of vulnerabilities discovered within the corresponding library's supply chain. We consider multiple development tools and libraries and their software dependencies, all of which exist as open-source software. Understanding the potential risks, vulnerabilities, and dependency relationships present in the development supply chain will inform further efforts to securely develop AI/ML products and secure its supply chain.

14. SUBJECT TERMS
AI, ML, supply chain, package managers, risk analysis

15. NUMBER OF PAGES
93

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A RISK ANALYSIS OF SOFTWARE DEPENDENCIES FOR THE AI/ML SUPPLY CHAIN

Alexander S. Tum
Lieutenant Junior Grade, United States Navy
BS, United States Naval Academy, 2021

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN CYBER SYSTEMS AND OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
September 2023

Approved by: Joshua A. Kroll
Advisor

Britta Hale
Co-Advisor

Alex Bordetsky
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Artificial intelligence (AI) and machine learning (ML) offer new capabilities for the overall technology ecosystem. As it forms the foundation for new technology, the security of a final software product depends greatly on that of the underlying supply chain, including its software dependencies. This study examines a portion of the supply chain for AI/ML by mapping the dependencies of a select sample of ML libraries for vulnerabilities. We search for a relationship between the depth of a dependency within a sample library's dependency tree and the amount of vulnerabilities discovered within the corresponding library's supply chain. We consider multiple development tools and libraries and their software dependencies, all of which exist as open-source software. Understanding the potential risks, vulnerabilities, and dependency relationships present in the development supply chain will inform further efforts to securely develop AI/ML products and secure its supply chain.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AI	artificial intelligence
API	Application Programming Interface
APT	Advanced Package Tool
CISA	Cybersecurity and Infrastructure Security Agency
CPE	Common Platform Enumeration
CRAN	Comprehensive R Archive Network
CVE	Common Vulnerability Enumeration
DAG	directed acyclic graph
DFARS	Defense Federal Acquisition Regulation Supplement
DIC	Dependency Inspector for CVEs
EO	Executive Order
FAR	Federal Acquisition Regulatory
JSON	JavaScript Object Notation
LOC	lines of code
ML	machine learning
NTIA	National Telecommunications and Information Administration
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project

PyPI	Python Package Index
SaaS	Software as a Service
SBOM	Software Bill of Materials
SDLC	Software Development Lifecycle
SLOC	source lines of code
SSDF	Secure Software Development Framework
SV-AF	Security Vulnerability Analysis Framework
USG	U.S. Government

CHAPTER 1:
Introduction

Security experts have increased their attention on software supply chains and their vulnerabilities. The SolarWinds compromise in 2021 highlighted the severe damage that software supply chain attacks can inflict. The U.S. government and multiple cybersecurity firms believe an organized and dedicated attacker, specifically the Russian Foreign Intelligence Service, conducted the SolarWinds breach [1]. However, it does not necessarily take a well-resourced or even dedicated attacker to compromise critical software supply chains. In 2016, thousands of developers had their projects broken when one author's JavaScript module, named kik, originally published on the package manager npm (for a description on package managers see Section 1.1), was taken down and unpublished at the original author's request. Many news sources refer to this as the left-pad incident, due to the left-pad code that was part of the removed kik module. The removal of left-pad halted web development and took down many popular web apps that used the left-pad code as a dependency. The incident showed the fragility of the software supply chain, specifically in 2016 [2]–[5]. These two incidents, though separated in time by nearly 5 years, raise questions about the current state of the supply chain. Has it gotten more vulnerable over time and/or have the threat actors expanded their targeting to more of the supply chain?

While the vulnerabilities of conventional software supply chains have and are currently being studied, the security of the tools within the supply chain of artificial intelligence (AI) and machine learning (ML) remains under-explored. This lack of study comes despite the increasing utilization of AI and ML technology by many companies and the U.S.. government to achieve greater efficiency and new capabilities [6]. Developers for these entities rely on increasingly complicated software supply chains, including package managers, annotation tools, and AI/ML libraries, to develop their applications. Entities hoping to capitalize on AI/ML by using external resources currently accept a largely unknown or an inaccurate understanding of the risk to their supply chain. Developers frequently combine disparate tool-chains in order to achieve their objectives, which introduces a level of complexity to understanding the composition of an application's supply chain. Due to the emerging nature of AI/ML technology, the tools within the supply chain are not yet mature. Many tools and

the dependencies of these tools change regularly and updates can go easily unnoticed. In addition, malicious hijacking of packages has been seen in the wild, so even previously trusted packages could become a liability for developers and organizations. We study the supply chain of popular AI/ML tools in order to gain a more complete picture of the risks involved in AI/ML development and how to manage these risks.

Popular AI/ML libraries include Scikit-Learn, Keras, and TensorFlow [7]. These libraries can be downloaded from various package managers, such as pip, however package managers can and have held malicious packages [8], [9]. This is because package managers typically attest to integrity, users hold responsibility for ensuring they download the correct package and version, and maintainers must publish the intended package and not a malicious one.

The goal of this research is to examine a portion of the supply chain of AI/ML development and its risks. By studying the prevalence of vulnerable software packages present in AI/ML supply chains, a better picture can be formed of the development landscape and what to secure as well as how to secure them.

1.1 Research Questions

This thesis will perform a supply chain risk assessment of commonly used Python open source AI/ML tools and libraries and open source software dependencies used by the these libraries and managed by the package manager, pip. We evaluate the current state of the supply chain and consider the risks that AI/ML tools and libraries are exposed to through their dependencies. In addition, we consider the additional risks that AI/ML scientists, engineers, organizations, and models themselves are exposed to through the distribution of AI/ML tools and libraries by package managers.

Our study focuses on the likelihood of vulnerable dependencies/malicious packages in specific libraries that attackers could exploit rather than the prevalence of these packages across the entire repository. The intent is to understand the frequency of known vulnerabilities and/or malware being distributed in dependent packages installed as part of AI/ML libraries.

The majority of software developers utilize package managers in order to take advantage of the vast open source software ecosystem and incorporate existing software artifacts into

their applications. These artifacts can be anything from software libraries written for the intended language, binaries, containers, or packages. Essentially, we use artifacts as a broad term for any manner of code that can be distributed. Package managers typically host their own repositories for users to be able to download packages from. These repositories need to be updated whenever a new package or new version of a package is released. Within the ecosystem for package development, package maintainers hold responsibility for maintaining their packages and typically host their packages at multiple locations, or repositories. More details about their role can be found in Table 2.1. These other locations are not package managers, but solely hosting locations, such as GitHub and personal/corporate websites. Maintainers will also host on package managers for the intended language, such as the pip package manager for the Python Package Index (PyPI) for Python and the Comprehensive R Archive Network (CRAN) for R. Some packages also operate across languages to provide additional functionalities. For example, many Python packages are written in a combination of Python and C in order to provide better performance in certain aspects.

The convenience provided by package managers makes them the preferred choice for installing new libraries and applications and they are indispensable to software developers today. However, because package managers automatically install needed dependencies, many developers install new desired libraries without fully understanding what the other dependencies are. Installing a package manually typically requires manually satisfying required dependencies and this provides an opportunity for the developer to assess the dependency being installed. Package managers automate this process and the developer's role is less active. Some package managers, such as Advanced Package Tool (APT) will inform the user of the names of packages and associated dependencies being installed and request permission. These lists of packages could contain dozens or even hundreds of packages and the user can only choose to accept all or deny all of them. Unless a user researches all of the dependencies prior to installing the necessary packages, they have little knowledge of the risk they are accepting, as the package manager does not necessarily provide much information about the packages except names, version numbers, and that the installed state is functional.

An alternative to package managers for automating the installation of tools, includes using commands such as "curl [someScriptOnSomeSite] | sudo bash" which will download a

shell script and execute it with root privileges. While this is a valid method of installing software, the security risks this introduces are numerous as unverified scripts running with root privileges can wreak havoc on a system. An example of a possible attack is if an attacker compromises the aforementioned script to download malware.

The automated installation of applications and associated dependencies potentially exposes the software supply chain to risks emanating from these dependencies. Many popular software artifacts are developed by teams of people operating in an open-source environment or corporate teams which allow outside contributions by the open-source community. There exists an overall goal of honest behavior to create and improve projects, but these same projects can be malicious from the start or become corruptible by adversaries acting dishonestly and maliciously to subvert the software towards their own goals.

Our operating security model assumes an adversary who wants to install a specific artifact on a target machine without being detected, in order to accomplish their end-goals without impediment. This could be accomplished on any machine in an organization or that of a specific user's. The artifact may have an existing vulnerability, known to the adversary privately or even disclosed publicly, or may be a specially crafted malicious artifact that the adversary has modified to include the vulnerability. Alternatively, the malicious artifact may simply run a payload contained in the artifact. In order to deliver their target artifact, the adversary may insert artifacts into repositories, run a repository, become a package maintainer, contribute code to an open-source artifact, subvert a proprietary/closed-source artifact, or steal signing keys from maintainers. We assume the adversary cannot run installation commands on the target machine, modify files on the target machine outside the operation of their vulnerable artifact, or subvert the package signing infrastructure of pip.

The developers who use open source projects expect project maintainers to defend against these adversaries by reviewing their projects and any contributions made by outside contributors before releasing new versions. This expectation creates elements of trust within the ecosystem based on honest behavior, whether it is deserved or not. It occurs between project maintainers and contributors as well as between project maintainers and developers who use their software artifacts. Whether any of these agents are actually trustworthy is another matter. It is important to ensure that only those that are trustworthy become trusted and only when absolutely necessary. This especially applies for closed-source projects where users

cannot independently inspect or have evaluated the source code. The trust developers place in a project maintainer and their artifact not only applies to the original intended artifact, but also extends trust to its dependencies and those dependencies' maintainers. The artifact's maintainer may decide to update their dependencies at any point in the future and encourage it to be adopted through an update. They thus hold a critical role in selecting dependencies, monitoring for compromises in dependencies, and acting as a steward for developers that use their artifacts. Untrustworthy maintainers (i.e., malicious agents acting as maintainers) can abuse their roles to compromise unsuspecting users.

The practice of vendoring may provide a possible solution to reduce the risk of developers downloading malicious or compromised packages, but has yet to be studied in this context. Vendoring refers to an act of copying dependency code or creating private/internal repositories of artifacts, such as containers or packages, in order to pin specific versions of dependencies directly into a product rather than allowing package/dependency management tools to do so automatically [10], [11]. The exact practice varies across programming languages and many use it to solve issues arising from conflicting dependencies. The programming languages Go and Ruby on Rails provide mechanisms that support vendoring [11]–[13]. In contrast, Python does not officially provide a mechanism to support vendoring, but virtual environments provide a close alternative. The vendoring package available on PyPI is intended only for use with Python's pip package manager [14]. Vendoring can in some cases complicate efforts to apply security patches to dependencies [10], [15]. If a package vendors its dependencies, then any time a dependency releases a security patch then the entire package must be updated and rebuilt to incorporate the patch. Otherwise, the package will continue to use a vulnerable dependency.

Containers have managed to achieve some of the same effects that vendoring accomplishes, such as achieving deterministic build environments. However, containers also face some of the same issues, depending on the implementation, especially with regard to incorporating vulnerable dependencies.

Today's AI/ML tools and libraries make use of many dependencies that may vary in their overall state of use and current maintenance. The overall usage of these dependencies in the ecosystem may vary wildly from being known and considered an almost essential part of almost any application to being an obscure dependency that is only noticed when needed.

We analyze the risks from dependencies for popular AI/ML packages using Common Vulnerability Enumerations (CVEs), a standardization used to uniquely identify vulnerabilities [16]. Then we consider whether a relationship exists between CVE presence, common dependencies between packages, and the depth of these dependencies in the parent packages' dependency trees (more on dependency trees can be found in Section 1.2). Through this we construct a better picture of the risk software dependencies create within the AI/ML supply chain and where the risks that do exist come from.

1.2 Scanning Popular Package Dependencies

In order to answer our questions, we developed a custom-built tool, named the Dependency Inspector for CVEs (DIC), to automate the generation of dependency trees for selected AI/ML packages, which were then cross-referenced with the National Vulnerability Database (NVD)s' CVE and Common Platform Enumeration (CPE) databases. More on the previously mentioned databases will be covered in Chapter 3, which enable our analysis of the relationships between packages, package dependencies, and known vulnerabilities.

In theory, a Software Bill of Materials (SBOM) should enable this deep analysis, as SBOMs contain information about software and all of the components that form it. More on SBOMs will be covered in Section 2.3.2, Here, we observe that current imaginings of SBOMs ignore important questions of use-in-context. An example would be creating lists of known vulnerabilities mapped to components, an underexplored area. This research aims to contribute to this area using dependency trees to visualize the structure of the information contained in SBOMs, specifically the codebase and its components.

The relationships between a package, its dependencies, its dependencies' dependencies and so on, can be represented in several different ways. We can represent the dependencies of a package as a directed acyclic graph (DAG), as these are essentially relationships between a single package and a set of packages. DAGs allow directional connections between nodes and no closed loops can be formed. For representing a package's dependencies, one node in the DAG represents the parent package and the other nodes represent the dependencies necessary to fulfill all requirements. The transitive closure of the graph captures when a package's dependencies are satisfied. The acyclic nature of a DAG applies to dependency relationships because any cycles that may exist in the dependencies will be broken and can

be ignored once all are installed. To illustrate this point, consider three artifacts, A, B, and C, where A depends on B, which depends on C, and C depends on A. Once all three artifacts are installed then the edge where C depends on A can be ignored, as all relationships are satisfied and the graph no longer needs to be traversed.

A tree can also be used to represent a package and its dependencies' relationships. Trees are defined as DAGs where the number of edges is equal to the number of nodes minus one. A distinguishing feature of trees is that a child node cannot have multiple parent nodes and there is a single root node that all children can be traced to. This can be useful for analyzing a characteristic of a specific subset of the relationships such as the furthest that a dependency exists from the parent package. In an unweighted DAG, the length of the path from an origin node to a destination node is calculated by the number of edges in the path. Similarly, in trees the depth of a node describes the number of edges from the root node of the tree to the leaf, or child node. Some DAGs are not naturally trees, but can be transformed into them. The dependencies in these DAGs may have cross-dependencies or shared dependencies, causing a child node to have multiple parent nodes, and possibly multiple depths associated. We transform these DAGs into trees by choosing a canonical way to attach shared nodes. More on this will be covered in Section 3.1.

The study of software vulnerabilities takes many approaches. The use of open source software components to develop products has only grown more widespread according to a study conducted by Sonatype, a company specializing in software supply chain management solutions [17]. That same study also found that of "the top 10% of the most popular Java, JavaScript, Python, and .NET projects, 29% of them contain at least one known security vulnerability" [17]. The security of modern software products depends not only on developing secure code for the product, but also on incorporating well-built and maintained software components.

Software can have known vulnerabilities and not-yet-discovered vulnerabilities which become discovered (i.e., 0-day vulnerabilities). Dealing with vulnerabilities in today's diverse package environment involves not only considering development practices but also dependency issues. Understanding the propagation and lifespan of vulnerabilities within a supply chain can allow developers to make educated decisions about the risks being accepted when incorporating new libraries into their applications. While they can be mitigated through

practices that promote secure software development, such as DevSecOps or secure development lifecycles, developers who use third-party libraries should consider other factors besides the development practices used. Alfadel et al. argue that the "two most critical aspects in dealing with package vulnerabilities are how fast developers can discover and fix the vulnerability, and how fast the applications update their packages to accommodate the fixed versions" [18].

When analyzing supply chain risk, developers must consider the risks of supply chain compromise that may come from both supply chain attacks and supply chain exploits. These two categories of supply chain compromise differ in that we define supply chain exploits as those attacks that take advantage of vulnerabilities already present in a system's supply chain whereas Cybersecurity and Infrastructure Security Agency (CISA) defines a supply chain attack as occurring "when a cyber threat actor infiltrates a software vendor's network and employs malicious code to compromise the software before the vendor sends it to their customers" [19]. Landwehr et al. in the creation of their taxonomy of computer program security flaws chose to consider both intentionally inserted and accidental flaws, or vulnerabilities [20]. The main difference between our two categories of supply chain compromise is whether the compromise occurs because of intentional security flaws inserted by an attacker, or through an attacker's use of unintentional flaws, which exist because of errors made on the part of developers. The end result in both cases is that supply chain attacks and supply chain exploits expose developers and users who utilize a specific software supply chain to the risk of being compromised by the third-party tools and libraries used. For the purposes of defending against supply chain risks, the mitigations and risk analyses for individual developers defending their own systems are largely the same as it involves understanding the underlying system's artifacts (e.g., containers, libraries, and binaries) and applying mitigations to them so for our purposes, understanding supply chain exploits help us to defend against both.

CHAPTER 2:
Background

Concerns about supply chain security date back to the earliest efforts of security, being mentioned at least as early as Karger and Schell's [21] efforts in 1974. Their critical work, *Multics Security Evaluation: Vulnerability Analysis*, was republished in 2002 due to its continuing relevance today [21]. In their security evaluation of Multics, they discussed the insertion of trapdoors at various stages of a systems life-cycle, from design to distribution to replacement. Another author, Myers, performed an in-depth analysis on the practice of subversion, the organized efforts to compromise computer systems by clandestinely inserting mechanisms, e.g., trapdoors and Trojan horses, that can be used by an attacker later on [22]. One of the key observations he made is that subversion can occur anywhere in the life-cycle of a computer system or software. Thompson's demonstration of compiler insertion of a backdoor into code demonstrates one of the earliest supply chain attack techniques. Thompson acknowledged in his famous "Reflections on Trusting Trust" Turing Award Lecture, that he took inspiration from Karger and Schell's work [21], [23].

For modern software development, package managers have become essential infrastructure, particularly for open source software, as they provide distribution for the majority of commercial-off-the-shelf packages. They provide an efficient and convenient method of distributing and obtaining software. Package managers usually support the package base for a specific programming language or operating system, such as PyPI for Python or APT for Debian systems.

Unfortunately, the critical role of open-source software and package managers also makes them a prime target for attackers. As we stated in our security model in Section 1.1, we assume attackers want to place a specific package on a victim's system. This specific attack package could already have vulnerabilities known only to the attacker or be a specially crafted malicious package. The delivery of the attacker's intended package abuses the mechanisms in the software supply chain that are intended for the delivery of legitimate software. Mechanisms that allow users to place a certain level of trust that the software artifacts they download have been created and distributed with honest intentions. The

attackers we consider in this work differs from that of previous work studying the software supply chain. In previous work, the threat model is limited to an attacker that either takes advantage of vulnerabilities in legitimate packages or places malicious packages, but not both simultaneously. Our attacker model includes both types of attacks.

Our scope of supply chain attacks in this work also includes those attacks that exploit vulnerabilities within the dependencies of a target application. A notably recent example of this being the "Log4Shell" vulnerability **(CVE-2021-44228)** in the Apache Log4J library [24]. Attacks which utilize this vulnerability may not fall into a strict definition of a software supply chain attack. As we stated earlier at the end of Section 1.2, CISA defines supply chain attacks as occurring when a threat actor infiltrates a software vendor to compromise the software before the vendor distributes it commercially. For example, the insertion of malware into software to be hosted on package registries and later download by unsuspecting users falls under the definition of a supply chain attack.

For the purpose of evaluating risk and security within the software supply chain of legitimate packages we consider supply chain compromise to include the types of attacks defined by CISA as well as attacks by threat actors that utilize vulnerabilities found within software artifacts. Vulnerabilities within the supply chain can have the same effects as "supply chain attacks" in that they compromise large amounts of software through dependencies. Studying supply chain compromise allows us to provide examinations evaluating security of the software supply chain for broad portions of the ecosystem.

2.1 Supply Chain Security

Several recent studies have examined aspects of supply chain security. Some focus on empirically examining the prevalence of attackers conducting supply chain attacks and the risk of attackers inserting malware. The threat models in these studies depict an attacker that places malware on package repositories for later download by users. Examples of attacks include typosquatting attacks, hijacking of accounts that control packages, and social engineering of both package maintainers and repository maintainers [9].

2.1.1 Empirical Studies of Attack Prevalence

One method to evaluate the security risks present in a system involves building a picture of the attacks and attackers being faced by defenders. For the software supply chain, this has entailed studying the prevalence of malware distributed via package managers.

Open source software developers typically distribute their products in a combination of one or multiple methods: privately hosted websites dedicated to the software, Github repositories, and package manager repositories. Package managers are a useful tool that streamlines code sharing and promotes efficiency in code development, as it allows developers to freely share code with their community as well as provide a central location for developers to find code. For users, the benefits of using a package manager can include automatic installation, dependency resolution, and updating of prior installed packages [9], [25]. Communities have been built around package managers and this in turn encourages active development in that ecosystem which benefits all stakeholders. Unfortunately, this is not without trade-offs, specifically from a security perspective. Package managers and the people involved in the operation of the ecosystem function as trusted intermediaries for its stakeholders (i.e., the users that download packages). This position of trust provides an opportunity for attackers to abuse the trust that exists in order to insert themselves into the supply chain.

Duan et al. [9] characterize the primary stakeholders' roles within the ecosystem for package managers. The stakeholders include registry maintainers, package maintainers, developers, and end-users. Table 2.1 describes these roles. These stakeholder roles may overlap (i.e., an entity may act in multiple roles within the package manager ecosystem). Developers of open source software frequently perform the role of package maintainer for their product. Understanding the relationships between stakeholders provides a path towards examining where attackers may insert themselves into the supply chain and how to defend against them.

Table 2.1. Package Manager Stakeholder roles and Descriptions. Adapted from [9].

Stakeholder	Description
Registry Maintainers	They maintain the registry of packages as well as the infrastructure involved in hosting and distribution. Also known as package managers, they typically operate web applications to manage and serve packages as well as client applications for users to interface with and download packages
Mirror Maintainers	They maintain mirrors of the registry of packages as well as the infrastructure involved in hosting and distribution from their mirror. Typically they sync their mirror to the official registry for users to interface with and download packages from. Mirrors allow the traffic load to be distributed across multiple servers.
Package Maintainers/Developers	They create and maintain their respective packages. They then choose to host their packages on platforms such as a package registry. Open source package maintainers typically utilize a hosting platform such as Github for collaboration with the community.
Developers	They typically use packages developed by others for their own applications. They must choose the packages necessary for building their own applications.
End-users	They are the users of software.

Deeper understanding of the relationships between stakeholders helps to identify how current practices in the package manager ecosystem may lead to security vulnerabilities, which attackers then exploit to insert themselves into the supply chain. Duan et al. [9] use a comparative framework to analyze registries and identify security gaps within package managers. They identified mechanisms that can exist on package managers which can provide functional, review, and remediation features. These include signing and verification of packages, access control measures, and removal of packages from either the registry or user systems. Many times these features were not available, but even when available they were not used. Based on which features were present within each manager's ecosystem, possible vulnerability issues could be identified.

The threat models the authors consider focus on how attackers could exploit package managers and their practices to distribute the attackers malware and compromise downstream stakeholders. Possible examples of these threat vectors include registry exploitation, typosquatting, malicious publishing, account compromise, infrastructure compromise, disgruntled insider, malicious contributor, and ownership transfer [9]. All of these attack vectors ultimately lead to a malicious package being placed on a repository for download by developers who trust the repository to serve legitimate packages, but instead are served malware.

Package managers frequently use mirror servers, either public or private mirrors hosted by third-parties, to distribute packages. They allow users to download packages from different servers than the one hosting the main repository and provide benefits that reduce costs and overhead for the package manager. Public mirrors can be endorsed by the repository maintainers thus making the mirror an official mirror. Endorsement requirements vary, but typically mirror maintainers must be in contact with the repository maintainers [8]. Official mirrors hosted by third-parties serve an essential role and provide distribution capabilities for many package managers. Attackers who use or control officially endorsed mirrors can potentially gain access to many targets due to this. Unofficial mirrors provide distribution capabilities for software packages, but generally for a different audience. These unofficial mirrors may be public or private mirrors hosted by individuals or corporations for a wide range of reasons. For example, a corporation may desire to maintain control over a private mirror's repository so they can host internal as well as external resources, restrict access, and control what packages are allowed on their internal network. Public unofficial mirrors also

provide services that an official mirror may not provide such as hosting package versions unavailable on an official mirror. This potentially offers attackers an opportunity to distribute their surreptitious attack package for users to download without going through the process of obtaining an official mirror.

Attacking the mirror infrastructure used by package managers provides an opportunity for attackers to dictate the packages installed by developers. Cappos et al. [8] developed a proof of concept attack which looked at how an attacker could circumvent security practices and checks in order to serve known vulnerable package versions from an attacker-controlled mirror. While their research focused mainly on package managers that serve software for operating systems, it applies for package managers used to install software libraries. Because of technical advances and ecosystem changes, many package manager maintainers have increased requirements to become an official mirror and applying for and hosting official mirrors has gotten more difficult in today's environment. Despite this, the proof of concept attack demonstrates that attackers can insert themselves in the role of mirror maintainer. As depicted in Table 2.1, Mirror maintainers are an intermediate role meant solely to maintain a mirror and its services. These mirror maintainers exist between registry maintainers, who consolidate and serve packages, and developers and users, who use them. This unique role places the attacker in a portion of the supply chain unexpected by users and allows them to insert themselves as an easily overlooked intermediary. This role gives them the capability to dictate packages installed by clients by making available certain versions for download, these may be vulnerable versions of legitimate packages or attacker crafted malware. The malware may either embed itself within legitimate packages, be a separate dependency or be installed by the attacker later if a vulnerable package is used to provide an entry point for malware.

Malware that embeds itself in a package often performs additional behavior entirely unrelated to the original intended purpose of the package. These behaviors include making unusual network connections and sending sensitive data from paths unrelated to the software when the original package operates solely locally. An example of this can be seen when an attacker compromised the packages eslint-scope and eslint-config-eslint. New versions of these packages, originally intended to support the ESLint project which statically analyzes JavaScript code, had been published using an ESLint maintainer's stolen account access credentials. The attacker included a malicious postinstall script into each of these pack-

ages. When a user installs the compromised version of these packages, the postinstall script would run and exfiltrate over the network the user's .npmrc authentication token [26], [27]. Another incident occurred in late 2020, when a legitimate ownership transfer occurred for the Google Chrome extension The Great Suspender. The new maintainer of the software package inserted code that potentially conducted user tracking and advertising fraud [28], [29]. Lastly, in 2022 the maintainer for the node-ipc package, which receives millions of downloads, modified his package to include a dependency written by him. This dependency contained code which overwrote computers located in Russia and Belarus with text files containing a protest against Russia's invasion of Ukraine [30].

Duan et al. [9] studied the frequency of these types of malware disguised within and as packages on package repositories. Using their tool MALOSS, they automated program analysis using metadata, static, and dynamic analysis techniques to identify malware. The metadata analysis involved identifying suspicious traits of packages. This includes filtering for package names similar to popular trusted packages, which indicates possible typosquatting attacks. Other characteristics the authors considered include comparing authors of packages to known malware authors as well as examining file types within packages for suspicious file types, such as embedded binaries [9]. The static analysis mainly focused on tracking API usage and data flow analysis. This allows identification of suspicious API calls and data flows to suspicious sinks such as HTTP network sinks. The dynamic analysis tracks suspicious system calls. Overall, these techniques allow for unusual behaviors by a package to be flagged and further analyzed.

Duan et al.'s [9] study focuses on identifying root causes and summarizing attack vectors and threat behaviors within the package manager ecosystem. This means their empirical efforts focus on attackers whose malware already presents itself within the ecosystem and uses package managers as a distribution mechanism. Additional results of their efforts included several of the packages discovered by them receiving CVE numbers for the purposes of expediting notification to users and removal of the packages as they introduce significant vulnerabilities into the supply chain. Cappos et al.'s [8] proof of concept attack also looks at the distribution mechanism aspect by studying the feasibility of using mirrors to distribute vulnerable packages and malware. Our study differs in that we look at the proliferation of malware and vulnerabilities due to the usage of packages as dependency requirements for AI/ML packages. Our study examines the effect on subsequent stages of the supply chain

following the threat actors' usage of package managers' distribution mechanisms to infiltrate the supply chain. This allows us to consider the extent to which package distribution can lead to compromises in other packages' dependencies.

The study of package managers as distribution mechanisms for malware within the software supply chain provides crucial background for our study as we also compare the names of dependencies downloaded from PyPI to those provided on a package's official GitHub page. This allows us to verify that packages being installed through pip correspond to another official distribution channel. Compromising package distribution exemplifies one family of supply chain attacks, however similar effects to supply chain attacks can be achieved by exploiting vulnerabilities within dependencies located on downstream targets.

2.1.2 Vulnerability Studies of the Supply Chain

Various authors perform studies measuring and analyzing vulnerability presence within the software supply chain. One study into the relationship between software reuse and security vulnerabilities by Gkortzis et al. [31] finds an association between source code size, or source lines of code (SLOC), and the amount of potential vulnerabilities as well as an association between higher numbers of dependencies and vulnerabilities. It is no surprise that these two relationships are similar as dependencies increase the total lines of code (LOC) for a project. Other studies on vulnerability presence such as Alfadel et al. [18] focuses on understanding vulnerability propagation and lifespan within a number of Python packages. Besson et al. [32] studies vulnerability detection techniques through analysis of the packages themselves. In a different vein, a study by Imtiaz et al. [33] compares analysis reports by nine industry-leading software composition analysis tools, which typically generates inventories of the open source components in a software product and the presence of any known vulnerabilities.

Several studies find that a small number of popular packages could potentially affect large parts of several software ecosystems, such as the JavaScript, Ruby, and Rust ecosystems. Kikas et al. [34] analyze the dependency network structure of the aforementioned ecosystems and found that the amount of packages vulnerable due to a popular dependency is increasing, but most other dependencies have a decreasing impact as popularity decreases. Zimmermann et al. [35] find that packages in the npm ecosystem are densely connected via dependencies

and compromises in just a few popular packages could affect hundreds of thousands of others in the ecosystem.

Scanning software components for known vulnerabilities is another practice used to study vulnerability characteristics. Williams and Dabirsiaghi with their work on scanning dependencies for vulnerabilities, inspired a plethora of additional work on the subject [36]. The Open Web Application Security Project (OWASP) Foundation built its Dependency-Check tool to solve the issues laid out in Williams and Dabirsiaghi's paper. Several papers utilize Dependency-Check to perform vulnerability studies of several software development supply chains, notably in the areas of tool feasibility studies and vulnerability tracking. Pathirathna et al. [37] create a technical solution for scanning web applications using an automated multi-part Software as a Service (SaaS) scanner. Their scanner utilizes Docker containers that run various sub-scanners, specifically Dependency-Check, Zed Attack Proxy, and FindSecBugs. Cadariu et al. [38] also utilizes Dependency-Check in their Vulnerability Alert Service tool to track vulnerabilities in Java projects built with Maven, a software tool used by Java developers to automate the build process.

Alqahtani et al. [39] create an analysis framework, the Security Vulnerability Analysis Framework (SV-AF), in order to "establish bi-directional traceability links between security vulnerabilities databases and traditional software repositories." They perform perhaps the study most similar to ours, but it differs in several key aspects. Firstly, they limit their scope to projects built with the Java programming language and Maven, whereas ours focuses on Python based projects. Similarly to ours, their approach combines data from the NVD and Maven central repository to create custom ontologies which allows them to determine direct and transitive dependencies[1] between reported vulnerabilities and potentially affected Maven projects. Alqahtani et al.'s [39] work also compares their SV-AF to the OWASP Dependency Check tool and achieved better results with their own tool. Our work extends on their study and looks for vulnerability patterns in the dependency trees of popular Python AI/ML packages.

[1]Transitive in this context refers to dependencies that indirectly affect the target project, such as second or third order dependencies, which exist in the dependency tree due to more direct dependencies.

Our method of evaluating dependencies' vulnerabilities and relationships to the parent package introduces a new direction, one that should allow for a better understanding of the risk specific to using these AI/ML packages. We contribute towards this understanding of risk in the supply chain by examining the propagation of vulnerabilities in AI/ML dependencies to allow for a better understanding of the risks possibly being accepted when utilizing these packages.

Overall, the modern software developer must be cognizant of what forms their software supply chain, including the libraries incorporated, package managers used in their workflow, and potential vulnerabilities that exist. Much work has been done to examine the security of some of the portions of the overall software supply chain such as examinations of supply chain attacks in Ohm et al. [40] as well as the risks of supply chain exploits. Alfadel et al. [18] specifically study the characteristics of vulnerabilities present in a select number of Python packages in the PyPI ecosystem. Mitigating these risks can include utilizing secure enclaves, which allows for secure execution environments that segregates and protects data in untrusted environments [41]. Another method to mitigate the risks involves conducting program analysis of third-party code to identify potential security defects which can then be mitigated during incorporation [32].

2.2 Policy Guidance for Supply Chain Security

In the U.S., the security and durability of the software supply chain has received remarkably increased attention beginning in 2021. On May 12, 2021, President Joseph R. Biden Jr. signed Executive Order (EO) 14028. Section 4e of that executive order directs the National Institute of Standards and Technology (NIST) to "issue guidance identifying practices that enhance the security of the software supply chain" [42, p. 1]. Another portion of the same EO also requires government agencies to comply with the resulting guidance released by NIST by May 2023. Prior to the order, NIST had released the initial Secure Software Development Framework (SSDF). Due to the EO, NIST revised the SSDF to comply with the order and released SP800-218, *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities* [43].

The SSDF describes at a high level the practices that should be implemented to securely develop software. The SSDF covers its intended topics broadly and thus may not be the

most useful for those without prior expertise. The authors of the SSDF acknowledge these limitations, as they recommend that those intending to implement the practices described within should have expertise in secure software development practices [43].

Organizations that are inexperienced in these subjects may find it difficult to achieve these recommended practices, whether due to a lack of expertise or a shortage of resources necessary to hire the expertise required. Consider, for example, a single author of a software library who maintains their package and open-sources the project for use by anyone. This scenario occurs often and many larger organizations may utilize this author's library for their own projects. As discussed earlier, the same scenario occurred for the kik package located on the npm repository [2]–[5]. For our example, it is likely the author of the library may not attempt to adopt SSDF recommended practices because of the burden it creates which may or may not have a benefit to security. Adopting SSDF practices could be costly and challenging for small businesses [44]. As a result, any organization subject to complying with the SSDF may be unable to use the lone author's library without also going through a review process. For those that do conduct the review process, it may be prudent to also recognize that small organizations and development shops may just meet the minimum standards for compliance in the review process.

The compliance framework that the SSDF will fall under is FedRAMP, the set of infosec controls that need to be met to sell to the U.S. Government (USG). The Federal Acquisition Regulatory (FAR) council is supposed to meet and add the SSDF to FedRAMP within the mandated deadline of May 2022 and update the standard acquisitions clauses (Federal Acquisition Regulation/Defense Federal Acquisition Regulation Supplement (DFARS)), if necessary [42]. The compliance burden introduced by the SSDF may soon be limited to those organizations conducting business with the U.S. government, but it likely will expand in the future and be adopted by industry as other NIST standards and publications have been, such as the Cybersecurity Framework [45]. Security often introduces a burden on organizations. That burden can and should be lessened with appropriate tools, as insecurity also presents costs. Trade-offs must be made and risk management applied to maximize benefit. Increased adoption of secure development frameworks by organizations can lead to a greater market for tools created to solve the issues of securing the supply chain.

2.3 Other Software Development Security Frameworks and Practices

The following section will discuss various other security frameworks and software development practices used to secure aspects of the software supply chain.

2.3.1 Reproducible Builds

Another framework that exists includes the Reproducible Builds project [46]. This project pushes for software that one can verify came from a specific source both pre and post-compilation. The efforts by the Reproducible Builds project support software transparency, which describes the ability to observe and understand the composition and functionality of software, especially its components. Many developers release software in the form of pre-compiled executables, which can make transparency difficult. This includes open-source software which may or may not compile to a binary identical to the one being distributed, depending on how a user builds it. The Reproducible Builds project aims to allow one to verify that no vulnerabilities have been inserted during compilation and to assume that the post-compilation artifact corresponds with known code e.g., open-source code [46]. Thompson's demonstration of a compromised build tool, the compiler, provides an example of the type of attack the Reproducible Builds project tries to defend against [23]. Additionally, Myers' work in 1980 gives an overview of techniques that can be used to address many of these problems [22].

Ideally, any abnormalities between a distributed binary and one compiled by a user from source code can give an indication that the build process has been compromised and allow for response and recovery. Abnormality detection typically is achieved through the use of hashes as differing hashes between binaries will indicate changes between the copies of the source being compiled. However, hashing does not provide a complete solution, especially when third parties become involved, as users also need to be able to trust the entity they compare hashes with to provide valid and correct hashes for the corresponding artifact. Assume that we download an artifact from a package manager and compare the hashes we obtain from the artifact to those the package manager claims to have. Using solely hashes makes it difficult to verify the hash being received has also been received by others, and that we can trust the hash provided by an entity is the same as the one they obtained. This issue

can be mitigated through the use of signatures, as they allow the claims that hashes make to move across time and space. Specifically, signatures provide verification the distributing entity made a claim and non-repudiation that they served the artifact.

Achieving a reproducible build requires that several practices be incorporated into the development process. The first is that builds should be deterministic, allowing one to repeatedly compile source code and achieve the same results every time [46]. Recording random or non-deterministic values, such as dates or data encoded with random values, into a build will not allow it to be reproduced later on. Another practice is having a well defined build environment and tool-chain. The Reproducible Builds Organization states that "the set of tools used to perform the build and more generally the build environment should either be recorded or pre-defined" [46]. This is reminiscent of a practice that NIST included in the SSDF, namely the Software Bill of Materials SBOM, covered in Section 2.3.2. Containers also provide automation that supports a repeatable, defined build environment [47]. The last requirement to accomplish a reproducible build involves providing users a method of replicating the developer's build environment, building the software and validating the builds match [46]. The process of validating should include the use of cryptographic checksums and signatures to verify the build came from a trusted source.

The principles behind these practices arise from solutions faced by developers attempting to create reproducible builds for Debian packages. For example, in a history given by the Reproducible Builds project of efforts to make Debian and software built around Debian reproducible, they described how one developer built a tool to strip sources of non-determinism from files [46]. As work continues on making more software reproducible, more tools customized for different operating systems and environments to support creating and checking reproducible builds will be created.

For software supply chains, specifically AI/ML supply chains, reproducible builds provide a useful tool for verifying integrity of compiled software and the tools used in software development. However, it only solves a portion of the issues that can plague software supply chains. For example, it does not apply to libraries that are released source-only, non-compiled programs, or ML models themselves. Thus, as a framework its applications are limited and others may be better suited for other software development pipelines to achieve security. Reproducible Builds are only a part of the answer.

2.3.2 Software Bill of Materials

In mid-2018, the National Telecommunications and Information Administration (NTIA) convened a multistakeholder process to support software transparency. This process led to the concept of an SBOM, an inventory of software components and dependencies, with the goal of providing transparency to stakeholders [48]. Multiple working groups formed as a result of this process, each addressing different aspects of the issue with the goal of developing a solution, and operationalizing it into workflows. These working groups included the Framing Working Group, Use Cases and State of Practice Working Group, Standards and Formats Working Group and a Healthcare Proof of Concept Working Group. The working groups operated in a limited scope focused on the creation of a model for SBOMs that aids in sharing information on software components.

The current proposal for an SBOM requires that it be a formal, machine-readable inventory. There currently is no official analogue to what an SBOM would accomplish. The closest would be dependency lists in software, however they typically only include the first level of dependencies. An SBOM however should enumerate all of the software components and their dependencies, as well as include information about the components and relationships to each other. Enough information must be detailed within so that individual components can be uniquely identified using a standard format. Ideally, creating an SBOM can be automatic, but this may not always be the case, especially for older software [49]. Automation provides a crucial capability for use in modern DevSecOps pipelines, which we detail further in Section 2.3.3.

The Use Cases and State of Practice Working Group framed the software supply chain through three main perspectives: *producers*, *choosers*, and *operators* of software [50]. Through the characteristics of SBOMs, they posited various benefits that each of these perspectives stand to gain from the use of SBOMs. For example, producers of software products can use SBOMs to better design software that avoids risky dependencies and libraries that may be rife with vulnerabilities. Choosers of software should be able to build a clearer picture of the components of the software, allowing better due diligence to be performed. Operators can use an SBOM to identify and evaluate their potential risk to newly discovered vulnerabilities in their software and the corresponding supply chain [50]. Table 2.2 from the working group summarizes their work.

Table 2.2. Software Bill of Materials – Perspectives and Benefits. Reproduced from source: [50].

	Perspective on Software		
Benefit	**Producer**	**Chooser**	**Operater**
Cost	Less unplanned, unscheduled work	A more accurate total cost of ownership	More efficient administration
Security Risk	Avoid known vulnerabilities	Easier due diligence	Faster identification and resolution. Know if and where specific software is affected
License Risk	Quantify and manage licenses and associated risk	Easier due diligence	More efficient, accurate response to license claims
Compliance Risk	Easier risk evaluation. Identify compliance requirements earlier in lifecycle	More accurate due diligence, catch issues earlier in lifecycle	Streamlined process
High Assurance	Make assertions about artifacts, sources, and processes used.	Making informed, attack-resistant choices about components.	Validate claims under changing and adversarial conditions.

Though a relatively recent development, the SBOM has begun to be adopted as a component piece of securing the overall supply chain. In the SSDF, NIST advocated for the use of SBOMs as a method to obtain provenance information for software components, which through further analysis will allow one to better assess the risk that components may introduce [43]. Notably, the scope identified by the SBOM Framing Working Group explicitly excluded deep analysis related to supporting activities such as creating lists of known vulnerabilities mapped to components, an under-explored area [48]. In our research we utilize an analogous practice to SBOMs, dependency trees. They can accomplish some of the

functions that SBOMs provide, though dependency trees simply model existing dependency relationships rather than describing them explicitly. Nonetheless, dependency trees provide a structure for obtaining information on software dependencies which we can then evaluate.

2.3.3 Software Development Lifecycles

There are several development methodologies that focus on approaches to development and the Software Development Lifecycle (SDLC), such as Agile [51], DevOps [52], and DevSecOps [53]. Agile focuses on improving the processes and organizational practices around software development to optimize product delivery. DevOps aims to promote collaboration between the software development (Dev) and Operations (Ops) teams to improve the planning, design, and release of products. This also allows developers to operate on integrated platforms like Kubernetes. DevSecOps incorporates security into the entire SDLC, with emphasis on it being a consideration in the early development stages, as opposed to traditional development practices that conduct security checks and add features near the end of the development cycle [53]. Overall, these methodologies mainly focus on affecting the culture and mindset of people in organizations. For DevSecOps, this entails making security a significant part of the culture and development process.

In a multivocal literature review conducted on DevSecOps, Myrbakken and Colomo-Palacios [54] identified several characteristics that distinguish DevSecOps, namely the principles behind it and the practices implemented. These principles are summarized below:

- Culture: to promote collaboration between development teams, operation teams, and security teams who all integrate security in their work [54].
- Automation: to achieve DevOps objectives of rapid building and deploying, but also for security.
- Measurement: to enumerate threats and vulnerabilities discovered throughout the development process.
- Sharing Information: to ensure development teams, operation teams, and security teams understand the environment.
- Shifting Security Left: to make security considerations at earlier parts of the SDLC .

The practices the authors of the literature review enumerate for DevSecOps focuses on some of the security practices that may be added or modified in DevOps rather than all of the practices that an organization might conduct. The following outlines the practices the authors discovered:

- Threat Modeling and Risk Assessment: which allow the teams to identify how attackers could affect the system and how best to mitigate and defend [54].
- Continuous Testing: which includes automating security controls and testing in order to scan code, the environment and protect from possible issues.
- Monitoring and Logging: to collect evidence and data to track the performance of security controls.
- Security as Code: by defining security policies and then writing scripts and/or creating configuration file templates to allow codified security policies to be implemented at the start and reused across projects.
- Red-Team and Security Drills: to attack the system in order to find vulnerabilities, improve processes, and find solutions.

The flexibility of DevSecOps allows it to also incorporate other frameworks and practices such as the SBOM, though the category of practices to incorporate changes as fluidly as the practices being created.

We know of no formal studies measuring the effectiveness of SBOMs in DevSecOps workflows for vulnerability analysis, though there does exist work analyzing it theoretically and incorporating practices that contain similar concepts to those of SBOMs. Additionally, vendors have performed private studies to support their products that perform vulnerability analysis using public and private data [17].

Graham conducted an informal examination on how the current environment around SBOMs may affect DevSecOps [55]. Overall the analysis took a negative view of SBOMs as they existed in January 2019 and claimed that they did not currently provide a benefit due to insufficient practices and standards. One of the central arguments claims that SBOMs do not have enough granularity to be useful. Specifically, they currently track main components and focus on the license level, rather than tracking sub-components of components (i.e., dependencies and libraries) or operating at more finely-detailed levels such as patch

versions. We study whether this sub-component tracking is needed by looking at the level that vulnerabilities are found and whether it is worth the effort to track these inner sub-components. We obtain data on sub-components in part by using the NVD's CPE and CVE databases. More details on this are given in Chapter 3. Graham's examination does point out that even if vulnerabilities are found in a component, they may not always be reachable and thus may not significantly affect the risk involved in using a component with known vulnerabilities. It also makes the claim that DevSecOps practitioners may not be well-served analyzing every package and sub-components' vulnerabilities [55]. Our study supports this conclusion, as many of the vulnerabilities found existed in the immediate dependencies of the supply chain. Overall, there are several important issues SBOMs and their use in DevSecOps currently face. These same issues may not only apply to their use in DevSecOps, but also any application of SBOMs. It will be crucial to perform studies examining their effectiveness based on appropriate data and measurement techniques.

CHAPTER 3: Methodology

The primary goal of this thesis is to study the propagation of vulnerabilities within the dependency chain of a sample of AI/ML libraries. We accomplish this using a purpose built Python prototype tool: Dependency Inspector for CVEs (DIC). It builds dependency graphs for each package using the Python package, pipdeptree. The OWASP Dependency-Check tool performs similar functions as our DIC with a few key differences. Both Dependency-Check and our DIC utilize the NVD repository to gather vulnerability data. However, our DIC goes further and builds dependency trees where each node is a dependency, which we then use to associate vulnerability data with each node. For our study we consider AI/ML tools and libraries available on PyPI through the pip installer. The Python programming language has become a popular tool for the development of AI/ML applications precisely because of the various advanced open source tools that support AI/ML development.

By studying the number of vulnerabilities within dependencies for a relationship to propagation measurements, we can highlight where efforts to secure the dependency supply chain may be better focused. This could entail focusing efforts on rigorous evaluation of upstream dependencies before incorporating them, securing downstream parent packages from expected vulnerabilities, or both. Securing downstream parent packages may include measures to sandbox dependencies or limiting data flows.

This chapter covers the DIC in further detail, the package sample we use in our study, and the NVD and the CPE ontology we use for our vulnerability reference data.

3.1 Dependency Inspector for CVEs

Our custom tool, the DIC, builds links between vulnerability data and a specified project's dependency data. It is an early unvalidated, prototype that consists of a set of Python scripts. These can be found in the link below,[2] along with our raw result data and Python virtual environments. What follows is a high-level overview. First, we retrieve the vulnerability data

[2]https://github.com/babadum/Dependency-Inspector-for-CVEs.

for each package version, including dependency packages, from the NVD's CVE database using an Application Programming Interface (API) provided by the NVD. We then store the resulting output in either one of two applicable dictionaries, one contains vulnerabilities that apply to the current package version and the other contains those that apply to any package version. These dictionaries are then both associated with the corresponding package object for later reference. These package objects form the nodes of a dependency tree for the original target package. Once the dependency tree is built, we analyze the data and structure of the tree for relationships and patterns. The DIC is run individually on each package in our study sample.

The ML libraries we study all rely on external dependencies in some manner. To collect data on these dependencies, we utilize the open-source Python package pipdeptree, available on PyPI [56]. pipdeptree builds dependency graphs for a requested package and includes version numbers for installed packages and their dependencies. It relies on the internal API of pip to retrieve information on installed packages, such as package names, installed versions, and version constraints. Pipdeptree's reliance on pip requires us to install the necessary packages on our own and then use pipdeptree to retrieve data on installed dependencies, as pip will only resolve the entire set of dependencies when installing packages. Thus, in order to use the DIC, we first download a single target package from our study sample and its associated dependencies into an isolated Python virtual environment. The versions of the dependencies we download are determined by the default configuration of pip's dependency resolver.

We use pipdeptree and thus pip's dependency resolver to gather the dependencies and chose not to consider dependencies listed in a package's requirements file.[3] We decided not to resolve dependencies ourselves in order to take advantage of pipdeptree's capabilities to build JavaScript Object Notation (JSON) formatted trees. In addition, we wanted to consider what end users would most likely see as their dependencies, as most would simply look at the output that pip provides on the dependencies installed. Any vendored packages or those seen in a package's requirements file would not be displayed by pip, thus not seen by the average user and ultimately the DIC. Unfortunately, this also prevents us from fully understanding the vulnerability picture and dependency trees of those packages in our

[3]A requirements file refers to a file that packages have that is typically a text file containing a list of dependencies and version constraints, if any.

sample that do perform vendoring. We acknowledge that examining requirement files could be a useful extension of this work.

The DIC utilizes pipdeptree's JSON output functionality, which returns a DAG of the requested target package's dependencies. We use the resulting JSON DAG to build internal representations, where each node is a package object. Object characteristics at this point include the current package version and dependency depth. Later, when the full dependency tree is built and we traverse the tree, we dynamically update the dependency depth to its longest path value from the root node.

We then run search queries on the NVD's CVE database using its API to retrieve vulnerability data on each package in the set of dependencies, which ultimately is stored in dictionaries as part of a corresponding package object. There are two dictionaries of vulnerability data in each object, one is data that applies to the current version of the package, and the other set contains all data that applies to any version of the package. The API utilizes search attributes based on the CPE ontology (see Section 3.3) and when a request returns results, the output includes CVE entries that match the specified search criteria [57]. In the case of the DIC, this search criteria consists of the package name, as this allows us to retrieve data for all package versions which the DIC then places into the appropriate dictionary. We manually verified several of our results to ensure our search queries were generated correctly.

Once the Dependency Inspector collects all data, it builds a canonical tree of a target package's dependencies. We accomplish this by recalculating the number of vertices that a dependency tree has based on the longest path. We then manually analyze the data for each package in our study sample in order to track the propagation of common dependencies across the various library supply chains.

For the purposes of studying where vulnerabilities exist within a package's dependencies, we consider a dependency tree constructed from the DAG of a package's dependencies (For more on DAGs see Section 1.2). This dependency tree places dependent packages at the furthest depth possible from the parent node by running a longest path algorithm on the DAG. We chose to use the longest path instead of the shortest path as it illustrates the least control the parent package has over the dependency required.

While users are able to specify directly which package and version to install, many choose to specify only the name of the package. This does not guarantee that the package version installed is that intended by the parent package. For example, when only the name is provided, pip will match the name of the desired package and install the available version that satisfies version constraints in that environment, not necessarily the latest version.

Consider Figure 3.1, where the numbers in the figure indicate version numbers. The version numbers appearing inside the circles are the package versions installed from pip. Other trees that we study may have an additional qualifier inside the circle that describes extra characteristics. Version numbers appearing next to arrows with relational operators, e.g., ">=", "==", and "<", describe version constraints set on the package pointed to by the arrow that will meet dependency requirements for the package at the origin of the arrow. We see that the version of package pyasn1 that is installed, when a user installs the google-auth package, is not decided by google-auth directly but rather by the packages pyasn1-modules and rsa. Each of those packages has its own version requirement for pyasn1.

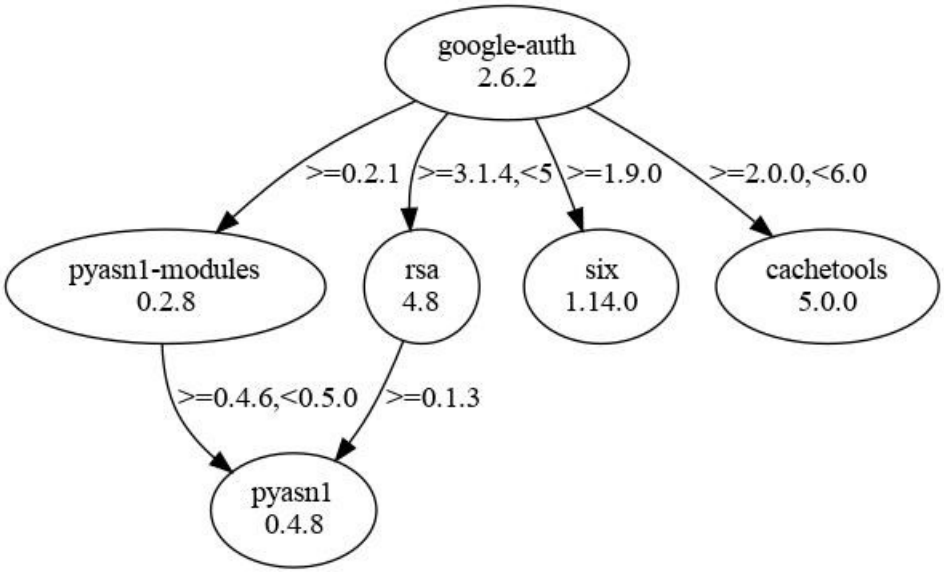

Figure 3.1. This is an example of a DAG that must be transformed into a tree as pyasn1 has two parent nodes. If pyasn1 only belonged to pyasn1-modules for example, then this would be an example of a tree with a height of three. The dependency depth of pyasn1 would also be two as it is on the second level from the root node. Adapted from [58].

The DIC builds dependency trees that allow us to perform a depth-first search and calculate the height of each node. It also allows us to manually identify common package relationships between parent libraries and calculate the frequency of dependency usage between packages for a relationship between commonality and vulnerability within the sample.

We study a small directed sample of packages, which we discuss further in Section 3.2. This small sample gravitates towards some of the more popular packages. We look at the prevalence of dependencies within a narrowly chosen set of packages rather than examining the ecosystem as a whole in order to focus on popular emerging tools and applications for AI and ML. While randomly choosing a sample is possible, we selected our sample packages because of their ubiquitous use in AI/ML applications. In the following section, we organize the sample by the number of packages in their respective dependency trees.

3.2 AI/ML Tools and Libraries

We chose six Python libraries which support ML to evaluate the dependencies of: TensorFlow, PyTorch, SciPy, Scikit-learn, Pandas, and Matplotlib. These libraries ad hoc were chosen from an informal survey of several professors at the Naval Postgraduate School. We asked these professors to provide lists of what they believed to be the current most popular/important libraries for AI/ML and from their responses we chose libraries that appeared most often on the lists or that we felt were especially related to our subject. At the time of data collection, the latest version of each package was installed and vulnerability data associated. We also enumerated vulnerability data for all existing versions of the sample packages and respective dependencies, in order to obtain a maximum number of vulnerabilities that exist in the total set. This specific portion of data could be considered to be an upper bound for a future analysis approach to explore maximum and minimum vulnerability presence in the supply chain based on version constraints set by parent packages.

3.2.1 TensorFlow

Developed by Google researchers, TensorFlow has emerged to become a leading platform for developing ML applications as many companies and researchers have utilized TensorFlow [59], [60]. The project's GitHub page indicates that more than 192k repositories utilize TensorFlow, however this number most likely underestimates as not all projects that utilize

TensorFlow appear on GitHub [59]. TensorFlow itself provides a framework for creating large-scale ML models and then training these models on large datasets. It also can be used to create deep learning models and neural networks. The aspects of how TensorFlow achieves its capabilities are outside the scope of this paper, but it does utilize external libraries for some of its capabilities [61]. One of the most significant libraries it uses as a dependency is NumPy, which we cover further in Section 4.2.1.

3.2.2 PyTorch

PyTorch provides an alternative to the TensorFlow ecosystem. It has been used extensively for development in the areas of computer vision and natural language processing. Developed by a team at Facebook (now Meta) in 2016, it currently exists as an open source project [62], [63]. The PyTorch ecosystem has many dependencies and there exist many tools and libraries intended to extend the usability of PyTorch [64]. This extensability has led to many utilizing PyTorch for research [62].

We examine the PyTorch ecosystem that exists on pip's ecosystem using installation instructions provided by their official website [64]. The exact command varies by the system chosen to run PyTorch, specifically the computing platform. The options provided for computing platform essentially distinguish between CUDA capable platforms and CPU only. CUDA was developed by the technology company NVIDIA, a major graphics card manufacturer, as a platform to enable parallel computing on graphical processing units [65]. CUDA allows for computing-intensive AI/ML models to be run efficiently. All of the commands install three packages, Torch, Torchaudio, and Torchvision. Torchaudio provides capabilities for language processing, and Torchvision is used for computer vision. For our purposes, we consider the Torchaudio and Torchvision packages to be dependencies for Torch because the development team's install instructions include all three.

3.2.3 Matplotlib

Matplotlib, a crucial piece of infrastructure for software applications, allows developers to create static, animated, and interactive visualizations of data in Python [66]. It is a competitor to the proprietary MATLAB software and is perhaps the most popular library in Python for data visualization. For the purposes of developing ML applications, it allows the

visualization of the massive amounts of data ML models typically utilize [67]. This gives developers the crucial ability to better understand the data they work with.

3.2.4 Scikit-learn

Scikit-learn originated out of a Google Summer of Code project and developed into a Python module to create medium scale ML applications for supervised and unsupervised problems [68]. It aims to be accessible to allow non-specialists to easily incorporate ML into their applications, as it provides several ML algorithms. This eliminates the need for the user to implement their own. Its original design also attempts to rely on minimal dependencies such as NumPy, SciPy, and matplotlib (see Section 3.2.3) to facilitate easy distribution [68]. These libraries are often installed, as many other AI/ML frameworks depend on them. Scikit-learn builds on the capabilities of the aforementioned foundational libraries and provides tools directly to developers to perform classification, regression, clustering, dimensionality reduction, model selection, and data pre-processing. Scikit-learn does not have the extensibility of TensorFlow, as it aims for simplicity; however it provides a great tool for users incorporating ML into products.

We note that it utilizes SciPy as a dependency and thus its true dependency graph should differ from ours generated in Figure A.8 for reasons that we cover in Section 3.2.6.

3.2.5 Pandas

Beginning development in 2008, Pandas has evolved into a high-performance open source project that provides data analysis and manipulation capabilities for a variety of applications [69]. Similar to SciPy, it aims to be a fundamental building block for conducting data analysis in general, and as a result has been found in the ML domain.

3.2.6 SciPy

The SciPy library consists of mathematical algorithms and functions built on top of NumPy (see Section 4.2.1). It specifically builds on NumPy's capabilities by providing additional tools for array computing, and specialized data structures, such as sparse matrices and k-dimensional trees. These allow python developers to conduct advanced data-processing and prototype systems [70], [71]. The SciPy library differs from the Scikit-learn and Pandas,

as it provides baseline infrastructure tools for mathematics, science, and engineering. For the purposes of the AI/ML supply chain, it acts as a foundational library which other infrastructure builds upon. Specifically, it provides several algorithms useful for different computational areas in AI/ML such as array optimization and linear algebra.

We discussed earlier in Section 3.1 that some packages we study utilize vendoring for managing dependencies and due to the design of the DIC, we do not examine those vendored dependencies. SciPy is one of the packages in our sample that performs vendoring, and thus when we look at its dependency graph later in Figure A.6 we will see a much smaller one than its true graph.

3.3 The Common Platform Enumeration

The National Vulnerability Database (NVD) makes available the Common Platform Enumeration (CPE) as a search criteria for the purpose of matching platforms. The CPE framework is intended as a standard ontology of platform components such as software products, operating systems, and hardware devices [72]. A CPE itself is a structured name that allows software and packages to be identified through a combination of various unique attributes such as component name, developer, and version to name a few. The other part to obtaining component data involves using a component's CPE to search the NVD's CVE database.

We take advantage of this ontology by specifying a target package name in the CPE format to retrieve CVE results for, these targets include the sample packages and their dependencies. Utilizing the ontology reduced the amount of extraneous CVEs that did not apply to any of our studied packages. The gathered CVE data we collect for each package then gets associated with the corresponding package object in our dependency graph.

CHAPTER 4:
Results and Analysis

Using our tool, the DIC, we discovered that the number of CVEs in a package's dependency tree grew proportionally to the total amount of dependencies, and CVEs occurred more frequently in the shallower dependencies (closer to the parent package). We performed our data collection in early July 2022. Table 4.1 provides a high-level synopsis of some of our findings. Further analysis for each package examined can be found in Section 4.1 and its subsections.

Overall, our results indicate a potential path towards allowing developers to secure their AI/ML supply chain and possibly the software supply chain in general by acknowledging where vulnerabilities exist and what could be in a developer's control. We found that the overall number of CVEs in each sample package's supply chain comes from a small subset of dependencies of the entire dependency list. In addition, many of the same packages with vulnerable versions appear frequently as dependencies for most of the sample packages. The small set of common vulnerable packages and the relative shallowness of CVEs within the dependency tree provides reassurance on the ability to secure the supply chain with practices such as vendoring. The locations of packages with CVEs in the dependency trees as well as the limited set affected, allows the developer greater control over what package versions to install to fulfill requirements, satisfy dependencies and mitigate risks, and also suggests that complete understanding of the formation of the supply chain may not be required to mitigate most of the known risks.

Table 4.1. This table summarizes our main findings for our sample of packages.

Packages	Number of Packages in Dependency Tree	Number of Vulnerable Packages	Number of CVEs in Package Ecosystem
Tensorflow	44	10	57
Torch	15	4	44
Matplotlib	10	2	28
Scikit-learn	5	3	11
Pandas	5	2	9
SciPy	2	2	9

4.1 AI/ML Development Tools and Libraries

As discussed in the previous Chapter, we chose six Python packages to analyze the CVE presence within their corresponding dependency graphs. Though these six tools represent a small sample of packages, they are significantly popular within the development community. Out of these tools, only one, TensorFlow v2.9.1, still used a vulnerable package as a dependency, FlatBuffers v1.12, as of July 4, 2022. Thus, the majority of our analysis will be on total CVEs across all previous versions of packages within the respective dependency trees, including the parent package.

We compile data on all versions of packages, as not all packages in the supply chain specify required versions for dependencies, and thus, for certain dependencies, any version could satisfy the parent's requirements. These inconsistencies in version requirements could allow an attacker to conduct attacks that intentionally serve older and vulnerable versions of packages, as described in Cappos et al. [8]. Though we compile data on all versions of packages, we do not attempt to create potential sets of packages that could be formed based on varying dependency requirements. This simplifies our analysis. Our work could instead serve as a basis for future work that explores potential maximum and minimum vulnerability presence in the dependency tree.

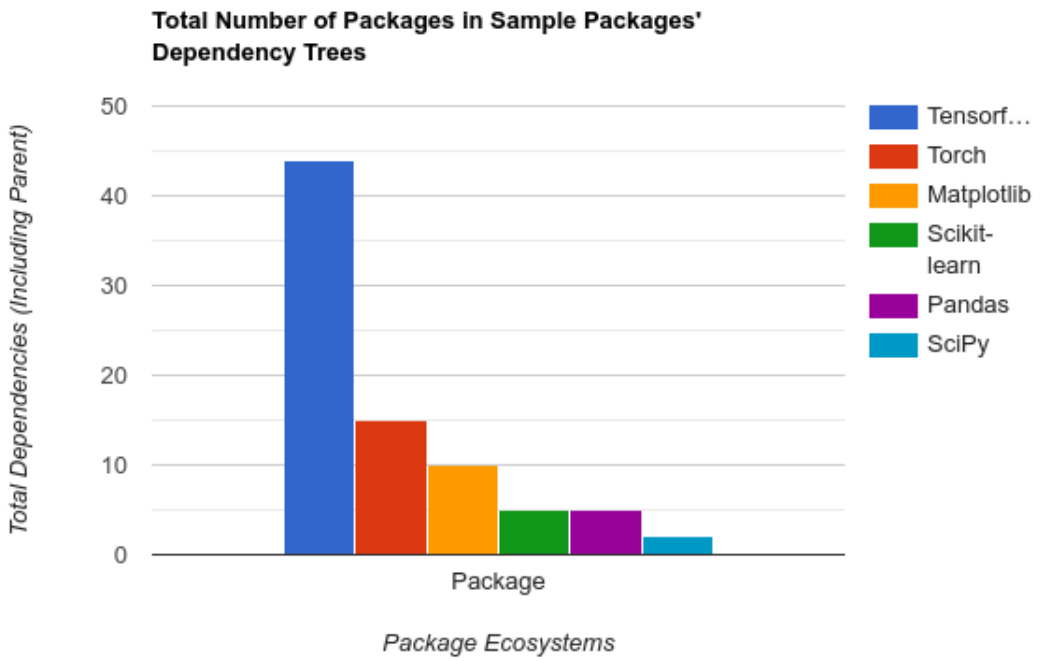

Figure 4.1. This bar graph displays the total number of packages in the selected sample ecosystems. The DIC utilized the pipdeptree library to obtain dependency data for each package.

Figure 4.1 shows the total number of dependencies (including the parent package) for each of our sample packages. More on each individual package's ecosystem will be covered in the following paragraphs and in the Appendix. These overall results confirm the expectations we have of these packages in terms of their code-base. TensorFlow and PyTorch (or Torch, as this is the name of the package on pip) intend to be large frameworks that provide functionality needed to develop ML models, thus they have more total packages. The other packages provide infrastructure capabilities, such as data structures, algorithms, and data visualization. The exception to both of these categories, Scikit-learn, matches the infrastructure packages in number of dependencies because its developers intended it to rely on as few dependencies as possible.

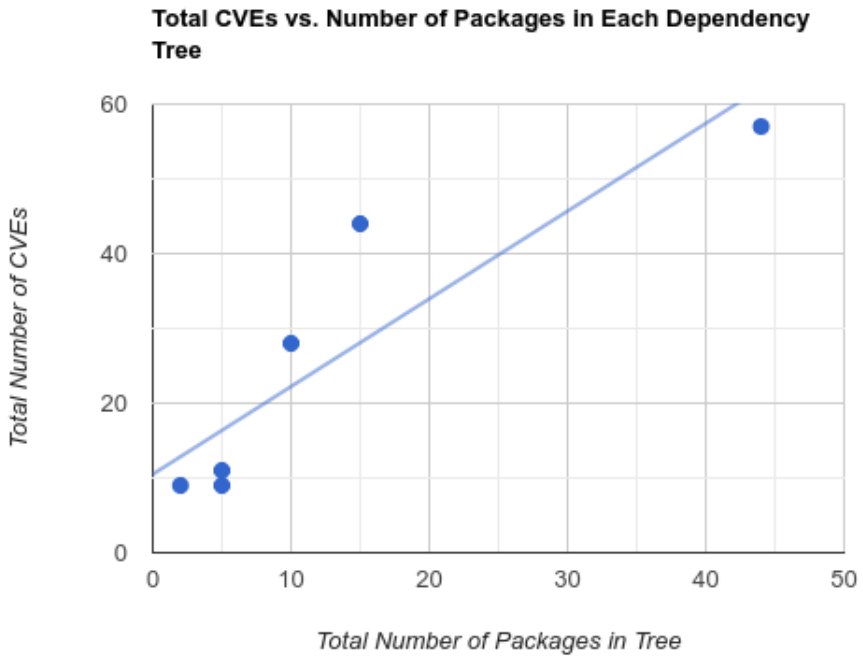

Figure 4.2. This scatter plot displays the total number of CVEs in each sample packages' dependency tree based on the total number of dependencies in each tree.

In Figure 4.2, we see that the number of CVEs in each sample package's ecosystem roughly increases with the total number of packages in each dependency tree. This data potentially reveals that a complex code-base with an increased number of dependencies may be an indicator of increased risk from the supply chain. This increased risk could also be attributed to the overall sizes of projects, as total LOC would generally increase with an increase in dependencies. We do not study our sample packages LOC directly, but others, such as Gkortzis et al. [31], have already established a relationship between LOC and vulnerabilities for other projects. Further analysis into the specific dependencies with vulnerabilities in the sample ecosystems reveals that this relationship between dependency count and vulnerability may be more nuanced.

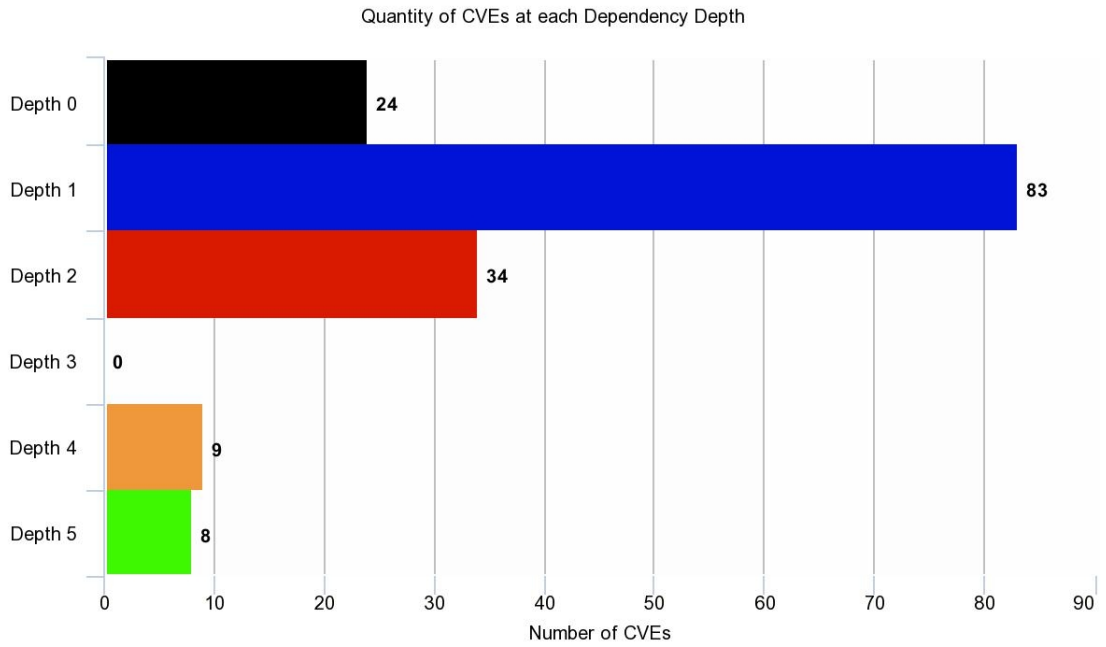

Figure 4.3. This chart displays the quantity of CVEs at each dependency depth across all ecosystems. Depth 0 is where the parent package is located and thus the root of the dependency tree. CVEs located at depth 0 indicate vulnerability in a version of one of our sample packages.

As seen in Figure 4.3 above, we find a higher frequency of CVEs occurring at shallower depths (depths zero, one, and two) versus deeper in the dependency tree. Though CVEs appear more at shallower depths this may be due to a lack of security vulnerability research into the deeper and possibly more obscure dependencies, rather than a legitimate lack of vulnerability. In addition, our small sample size biases this data, as only TensorFlow contained dependencies that existed at a depth deeper than three using a longest path search on the dependency graph.

TensorFlow contains the largest ecosystem that we studied, in terms of the size of its dependency tree which contains forty-four total packages. Of these forty-four packages, ten previously had versions released for which CVEs have been found. Table 4.2 shows the vulnerable packages, the number of CVEs for each package, and the deepest depth within the TensorFlow ecosystem they can be found at.

Table 4.2. TensorFlow Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
tensorflow	20	0
flatbuffers	2	1
numpy	8	2
markdown	1	2
protobuf	3	2
setuptools	1	2
werkzeug	5	2
requests	8	4
rsa	1	4
urllib3	8	5

Using the DIC, the results obtained for the Torch package seem to show that it has few dependencies, as seen in Figure A.2; however this is not exactly the case. Checking the application's GitHub page shows that PyTorch has a number of dependencies which are vendored directly as part of the package [63]. Because vendoring renders dependencies invisible to pip, they are also not visible to our tool. Further study on PyTorch and its dependencies may be needed to fully understand the vulnerabilities in its ecosystem. This also highlights a weakness in our method of evaluating dependencies as we expect the dependencies to be resolved by pip. Thus, any dependencies resolved in another manner are not examined. Practices such as the SBOM are meant to solve this issue of "missing" dependencies.

We attempted to gather and examine the CVE data for both Torch and its dependency, Typing-Extensions. Unfortunately, no data seems to be available. It is highly unlikely that vulnerabilities do not exist in these applications. Further vulnerability research is required to better understand the Torch ecosystem. For example, more research could be conducted into the Torch's vendored dependencies in order to see if vulnerabilities exist in any of those packages.

Part of the Torch ecosystem is the Torchvision package which provides functionality intended for use with computer vision applications. Specifically, it contains "popular datasets, model architectures, and common image transformations for computer vision" [73]. Unlike Torch and Torchaudio, this package has significantly more dependencies, as seen in Figure A.4. The increased number of dependencies introduces complexity in the supply chain for Torchvision. Though it only specifies five packages directly as top-level dependencies, several of them specify their own dependencies which nearly doubles the number of packages in the ecosystem as well as increases the maximum depth of the dependency tree. The deepest dependencies for Torchvision exist at depth two and the parent package exists at depth zero.

As can be seen in Table A.6, only two packages in the Matplotlib ecosystem have CVEs associated with them. NumPy has been seen frequently already in our other sample packages. The Pillow package has also been seen before in Torchvision's ecosystem. Both of these packages resided at a depth of one in Matplotlib's dependency graph. Due to this, Figure A.13, shows simply that a hundred percent of the total CVE quantity of twenty-eight existed at depth one.

In Scikit-learn's ecosystem, we found CVEs for the Scikit-learn, SciPy, and NumPy package. The NVD's database does not contain any CVE data for its other dependencies, Threadpoolctl or joblib. As we discussed earlier for Torch's results, we do not believe that these other packages contain no vulnerabilities, rather they have yet to be discovered. A possible reason for the lack of CVEs may be related to the package's popularity as they have not appeared before, yet NumPy appears in almost all of our sample packages' dependency lists.

Pandas has a smaller ecosystem than most of the others in our sample. It contains only three direct and one transitive dependency. Of these we only found CVEs for one dependency, NumPy. The Pandas package itself has one CVE associated with a previous version.

As discussed earlier in 3.2.6, SciPy acts as an infrastructure library for AI/ML. It utilizes minimal dependencies to accomplish its goals as seen in its dependency graph in Figure A.6. Though pip only resolves one package, NumPy, as a dependency for SciPy, its doc_requirements.txt file (a file in a package used to specify dependency and version requirements) lists several others. This illustrates one of the inconsistencies that we find

between gathering dependency data using pip based tools like pipdeptree and manually inspecting package files. For the reasons we outline in Section 3.1, we chose not to consider the dependencies listed in the package's requirements file.

4.2 Dependencies

Shared vulnerable dependencies are a common trait we find among our sample packages. This implies that the ML supply chain for Python applications, as it currently exists, may be limited in size. Thus, efforts to secure the entire supply chain may be more achievable. This is possible because the dependency set for ML projects is small and previous research mentioned in Chapter 2 has looked at vulnerability across entire software ecosystems and the thousands of packages involved, rather than particular application domains.

4.2.1 NumPy

One package we would like to specifically highlight of the shared dependencies, NumPy, appeared as a dependency in nearly all of our sample packages, except for Torch and Torchaudio. As it appears for Torchvision though, we do consider it part of the Torch supply chain since developers typically install Torch and its related Torch packages. In addition, Torch does utilize NumPy in its requirements.txt file which we did not consider for our dependency data collection. The prevalence and importance of NumPy in the AI/ML supply chain cannot be understated. Based on the data we collected, it has the greatest dissemination across our sample of packages, which all in some way relate to the AI/ML supply chain.

This reach that NumPy has across the supply chain potentially makes it an appealing target for attackers looking to conduct supply chain attacks. Eight CVEs apply to various versions of NumPy, and as discussed in Section 2.1.1, Cappos et al. [8] demonstrated an attack where attackers could specify known vulnerable package versions. In this case, an attacker controlling a mirror that contains vulnerable versions of NumPy could potentially compromise many downstream ML applications as well as other types of applications. This includes SciPy and Pandas as they provide infrastructure for other applications.

Though we chose NumPy to highlight, we found several other packages that appeared in several of our sample's dependency lists.

4.2.2 Other Overlapping Dependencies

We find multiple dependencies that overlapped across our sample include Pillow, Requests, and urllib3. By finding these common dependencies across supply chains we potentially find weak points of the supply chain that could have catastrophic cascading effects.

Matplotlib utilizes many dependencies as seen in Figure A.12. We see many of these packages appear in the supply chains of our other sample packages, including python-dateutil, NumPy, Pillow, and Six. In total, ten packages exist in Matplotlib's ecosystem and of these forty-percent can be found in the rest of our sample's dependencies.

The Pandas package relies mostly on packages seen in our other sample's dependencies, as well as one, pytz, that we do not observe in any besides Pandas. As can be seen in Figure A.10, Six exists in Pandas ecosystem only because python-dateutil relies on it as a dependency. NumPy appears in nearly all of our other sample packages, while we observed the Six package previously in TensorFlow's ecosystem. Python-dateutil and pytz provide support for date, time and timezone functions [74], [75].

4.2.3 Version Constraints

As stated earlier in Section 4.1, only TensorFlow, out of our chosen sample of packages to examine, currently uses a vulnerable version of a dependency. We utilized the latest version of TensorFlow (v2.9.1), available as of May 23, 2022. The vulnerable dependency, FlatBuffers, does have a new version (v2.0.0) available, however TensorFlow directly pins the required version of FlatBuffers to be >=1.12.0 and <2.0 [76]. The next version of TensorFlow does update the pinned requirement, however it has not been officially released yet [59].

This situation with FlatBuffers, highlights one of the dangers of vendoring code or pinning versions. This especially would apply to unmaintained packages that choose to vendor code, as those packages may not update its vendored code or pinned package versions for significant periods thus leaving the supply chain vulnerable. However, TensorFlow is

well-maintained and the developers most likely want to ensure compatibility with the new version of FlatBuffers or propose a patch to the old version. Additionally, both products are developed by Google, which could indicate less risk involved, as both development teams are likely aware and conscious of the issue. While this could be one of many reasons, until its next release the TensorFlow ecosystem is possibly vulnerable to FlatBuffer's vulnerability (CVE-2020-35864) [77]. A possible solution to this risk that vendoring code or pinning versions introduces may be to check for vulnerabilities before choosing to vendor or ensure maintainers are actively involved in their project.

Scikit-learn also sets version constraints that exposes its supply chain to risks. Scikit-learn has four total dependencies, as seen in its graph, Figure A.8. The version requirement set for NumPy by Scikit-learn v1.1.1 specifies that NumPy's version should be greater than v1.17.3, which allows any newly released version of NumPy to be utilized. Though Scikit-learn allows for newly released versions of NumPy, SciPy establishes an upper-bound on allowed versions of NumPy to be v1.25.0. Due to SciPy's limit on NumPy version, this effectively also limits NumPy's version in Scikit-learn's supply chain. Though no current CVEs exist for NumPy v1.23.0 (the latest version of NumPy), in the event SciPy fails to update its requirement past v1.25.0, Scikit-learn could potentially be at risk to vulnerabilities in versions of NumPy prior to v1.25.0 despite allowing newer versions itself.

The situation seen with Six in Pandas' ecosystem also illustrates the risk that version constraints pose. We discussed earlier how SciPy in Scikit-learn's ecosystem set constraints for allowed versions of NumPy, which indirectly restricts Scikit-learn's ability to use versions of NumPy outside of the constraints. In Panda's ecosystem, only python-dateutil specifies six as a dependency, so Pandas has no control currently as to what versions of Six enters the supply chain. Even if the maintainers of Pandas in the future decide to specify Six, it still would be restricted to what would be compatible with python-dateutil.

One of Matplotlib's dependencies, pyparsing, has multiple version constraints placed on it, similar to what we saw earlier with NumPy in Scikit-learn and Six in Pandas. For pyparsing, the constraints essentially elevate the minimum version requirements while disallowing a specific past version. We discussed before how maximum version requirements could prevent security updates. The difference in this case comes from how the minimum requirements do not expose the ecosystem to the same issues, as newer versions can be

installed without much issue. This represents a case where version constraints do not increase risk to the ecosystem.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions

AI and ML have become incredibly popular areas of research and development. Many entities including tech companies, traditional industries wishing to innovate, and governments have tried to adopt these technologies in some fashion. The promise of new capabilities through exciting technology does not come without risks. We have seen already several instances where supply chain compromise has led to disastrous consequences and this will not go away. Despite people's best efforts to secure their systems and supply chains, vulnerabilities and the attackers that exploit them will continue to exist, and the threat of compromise will always be there.

Securing AI/ML supply chains will require understanding its many facets, such as where vulnerabilities exist, how attackers utilize them, and how agents in supply chains mitigate the risks. Through understanding, we can then arrive at a solution to address issues found. Our work focuses on understanding where vulnerabilities lie in the supply chain. Though not all vulnerabilities for packages are necessarily known, CVEs provided an excellent tool for measuring the frequency of vulnerabilities that have been found.

As discussed in Chapter 4, we found an increase in the number of CVEs in each supply chain as the number of dependencies for a package in our sample increased. Most vulnerabilities in our chosen sample of packages existed at dependency depths of zero, one, or two, where zero is the parent package. As we examined deeper levels of the trees, we found that few packages had deep dependencies and with that fewer CVEs. In addition, we saw many of the same vulnerable packages among the dependencies of each of our sample packages, with the most frequent package being NumPy. This does not come as a surprise as it also is the most common package across each of our sample's dependencies as well.

Though the DIC has not been validated yet, the results it provides allows us to make observations we believe should be further explored. The largest ecosystem in our sample was TensorFlow. Examining the data for TensorFlow, we find that only about a quarter of the dependencies for TensorFlow have been discovered to have vulnerable versions. First, it is highly unlikely that no vulnerabilities exist in the other thirty-four packages in the

TensorFlow ecosystem. Second, our results support other findings in the literature, such as Gkortzis et al. [31], that increased numbers of dependencies are associated with more vulnerabilities. Third, some transitive dependencies are associated with more vulnerabilities than others.

5.1 Future Work

We collected our data using our tool the DIC which relied on several Python packages including pipdeptree, requests, and packaging. pipdeptree provided crucial functionality in generating dependency graphs of each of our sample packages in a JSON format, which we then used to generate CPE search queries on the NVD's CVE database. Using the CPE ontology worked well for our purposes. We were able to obtain vulnerabilities relating to all versions of a specific package. This is a result of our use of the NVD's API which allows us to provide a CPE-formatted string to be matched, which then returns CVE entries associated with the search query. We manually verified several of our results to ensure our queries were generated correctly.

Unfortunately, our approach limited our ability to dive deeper into dependencies, especially those that pipdeptree does not find. Pipdeptree relies on pip's internal API and pip only resolves dependencies when a package requires it to. Thus, it does not necessarily capture dependencies that have been vendored and are downloaded with the target package. An example we found involves the absence of NumPy from the Torch package's dependency tree. Torch's dependency tree as built by pipdeptree lacks NumPy as a dependency. However, in manually examining Torch's requirements.txt file, we were able to see that Torch does depend on NumPy as well as many other packages that pipdeptree, and by extension pip, do not provide to us.

This limitation of our tool, the limited ability to gather all relevant dependencies, demonstrates one of the missing capabilities that the current Open-Source community does not provide explicitly in an API. An ability to generate comprehensive dependency lists in a useful format, specifically for Python. Github provides a limited version of this ability, however features continue to be actively developed [78]. In order to further understand and secure the software supply chain, more robust capabilities to generate and analyze dependency data will become necessary.

5.1.1 Vendoring

The ability to incorporate new versions of dependencies into the supply chain is crucial for mitigating vulnerabilities. Most security advisories today almost always recommend keeping software updated in order to mitigate risk from vulnerabilities found in past versions. This is partially because attacking known vulnerabilities generally presents an easier attack method than discovering or developing new exploits and technologies. Those packages that do not allow new versions and fail to update their dependency requirements potentially expose themselves to vulnerabilities found in outdated packages. For example, in Figure A.6 we see that SciPy sets a constraint on new versions of NumPy past v1.25.0. This constraint, at the time of our study, potentially prevents SciPy and any product that uses SciPy as a dependency from updating their version of NumPy past the constraint.

While constraining versions through pinning is not quite the same as vendoring software, it presents an analogous method for us to analyze vendoring's potential effects. Our results show that one of the drawbacks of constraining versions means that TensorFlow still used a vulnerable dependency (e.g., flatbuffers) whereas without constraining it may not have. Though vendoring does have its uses, it should not be a default behavior and developers on both sides (both vendors and vendored) should strive to make their code robust. Vendoring packages provides a guarantee to developers that their project will not suffer a catastrophic breakage from a dependency updating or disappearing, such as in the left-pad incident we discussed in Chapter 1. However, this guarantee comes at the risk of using outdated versions of packages which could expose their projects to vulnerabilities that would have been patched in a security update. In order mitigate the risks of outdated versions, developers who practice vendoring must be active maintainers of their projects and ensure they consistently monitor for updates. This introduces increased workload on the part of developers.

We explained in Chapter 2 that attackers may introduce vulnerabilities to the supply chain in new versions of software purposely, such as in the case of the Eslint attack [26], [27] covered in Section 2.1.1. Due to the risks present from outdated packages as well as the risks from compromised new versions or loss of a version, such as in the case of the left-pad incident, we believe that vendoring, restricting updates, or pinning dependencies can play a role in a developer's security practices. If the developer has the capacity available to ensure their vendored dependencies stay up to date, then the risk from an outdated dependency would be

minimal and the benefits of vendoring could be maintained with minimal risk. However, as the number of dependencies increase, the workload increases. Thus, small-scale developers may not have the ability to ensure their vendored packages stay updated. A solution for these smaller scale developers would be to adopt a hybrid approach where direct and well-maintained dependencies, which garner a high level of trust from developers, do not need to be vendored. In the hybrid approach, transitive dependencies in addition to ones that are more obscure and less maintained should be vendored to ensure high levels of availability as well as allowing closer monitoring by the developer for legitimate updates.

Though not vendoring introduces the risk of malicious dependencies through hijacked maintainer accounts, solving this may not require significant action on the part of developers that depend on the hijacked package. Instead, maintainers should better protect their accesses to their code-base. In addition, package managers may need to take a more active role to combat active supply chain attacks, such as those which take advantage of a lack of access control mechanisms by the package manager.

5.1.2 Software Bill of Materials (SBOMs)

SBOMs for highly complex software may generate large amounts of data on dependencies. But, this is necessary as our research shows that vulnerabilities can be found in almost any level of the dependency tree. Mitigating the risks from vulnerable dependencies requires understanding where they exist and SBOMs provide a path to this knowledge. As our data shows in Figure A.1, 70.2 percent of the vulnerabilities in the TensorFlow ecosystem are found within the first three depth levels. However, a still significant number, 29.8 percent, of the vulnerabilities in TensorFlow's ecosystem are found at the deepest levels. The other sample packages we examine also have CVEs in nearly all levels of their dependency tree. However, many of the other sample packages only have three levels of dependencies. Further studies should be performed on larger sample sizes that looks for deeper trees to see the frequency of CVEs at deeper levels.

From a risk management point of view, achieving a reasonable level of security in the supply chain may not require an extremely detailed SBOM to identify vulnerable products especially when vulnerability data may not exist for the majority of those products. Highly detailed SBOMs may still be useful for other uses, but identifying products in the supply

chain only provides a portion of the risk picture. As evidenced by our data, a majority of the packages being used as dependencies in our sample's ecosystems do not have any vulnerability data available yet on the NVD's database.

Finally, our research does not utilize an actual SBOM, instead we consider a close equivalent to what SBOMs have been described to be. Future work in this field could also replicate our efforts utilizing SBOMs to determine the actual effectiveness SBOMs have.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Additional Analysis

A.1 TensorFlow

Table A.1 provides more detail than Table 4.2 in Chapter 4. It shows the vulnerable packages, their CVE numbers, and the deepest depth within the TensorFlow ecosystem they can be found at. Following that is a pie chart, Figure A.1, that visualizes the relationship between the CVEs and the depth they are located at in the TensorFlow ecosystem.

Table A.1. TensorFlow Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
tensorflow	CVE-2022-29213, CVE-2022-29212, CVE-2022-29211, CVE-2022-29216, CVE-2022-29210, CVE-2022-29209, CVE-2022-29208, CVE-2022-29206, CVE-2022-29205, CVE-2022-29204, CVE-2022-29203, CVE-2022-29202, CVE-2022-29201, CVE-2022-29207, CVE-2022-29200, CVE-2022-29194, CVE-2022-29192, CVE-2022-29191, CVE-2022-29199, CVE-2022-29198	0
flatbuffers	CVE-2020-35864, CVE-2019-25004	1
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	2
markdown	CVE-2018-1000874	2
protobuf	CVE-2021-22570, CVE-2021-3121, CVE-2015-5237	2
setuptools	CVE-2013-1633	2
werkzeug	CVE-2022-29361, CVE-2019-14322, CVE-2020-28724, CVE-2019-14806, CVE-2016-10516	2
requests	CVE-2021-21676, CVE-2021-21674, CVE-2021-21675, CVE-2021-29476, CVE-2018-18074, CVE-2015-2296, CVE-2014-1830, CVE-2014-1829	4
rsa	CVE-2016-1494	4
urllib3	CVE-2021-33503, CVE-2021-28363, CVE-2020-26137, CVE-2019-11324, CVE-2019-11236, CVE-2018-20060, CVE-2020-7212, CVE-2016-9015	5

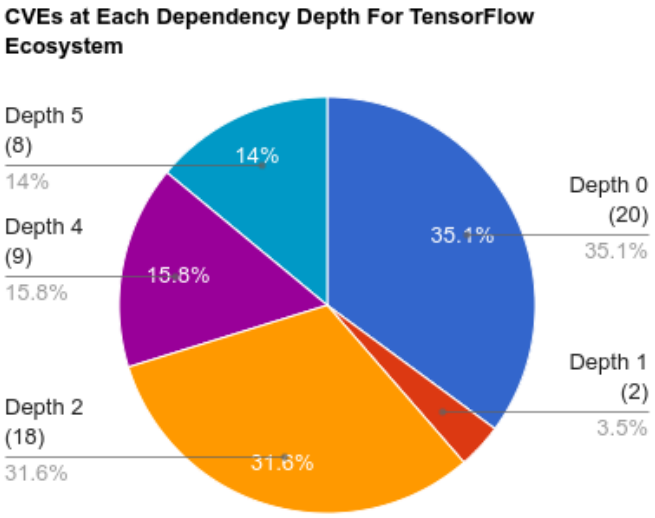

Figure A.1. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 57 total CVEs in the TensorFlow Ecosystem. Depth 3 does not appear as there are no dependencies at that depth with CVEs.

A.2 PyTorch

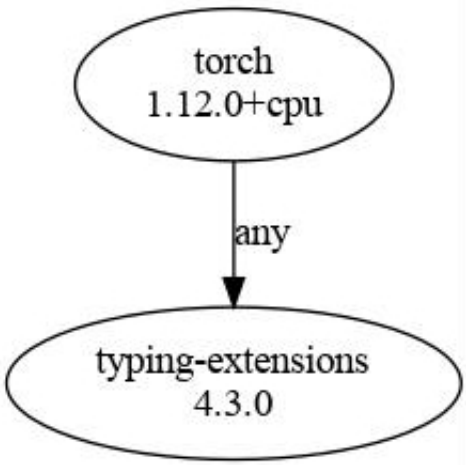

Figure A.2. This directed acyclic graph displays the packages in the Torch ecosystem installed using pip. As can be seen, Torch only has one dependency.

We found no CVEs associated with the Torch package or its dependency, typing-extensions.

A.2.1 Torchaudio

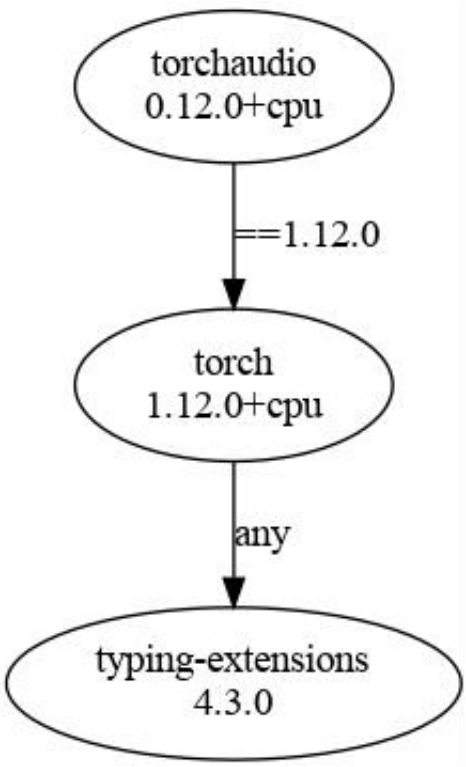

Figure A.3. This directed acyclic graph displays the packages in the Torchaudio ecosystem installed using pip.

When installing PyTorch, users that follow the developer's instructions also install the Torchaudio package, a library for processing audio and signal data [79]. This package has Torch as a dependency and as a result also adopts Torch's dependencies as its own, in this case Typing-Extensions. An illustration of this relationship can be seen above in Figure A.3.

Examining the results of the DIC on the Torchaudio ecosystem, we receive a similar result as before with Torch, with no CVEs entries in the NVD at the time of our data collection.

A.2.2 Torchvision

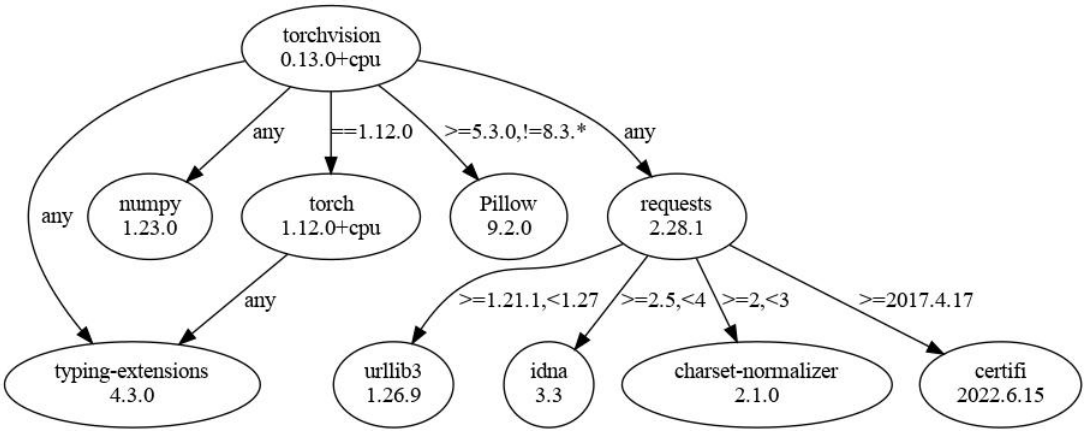

Figure A.4. This directed acyclic graph displays the packages in the Torchvision ecosystem installed using pip. This package contains significantly more dependencies than its relatives with a total of ten packages in the ecosystem. It also contains packages with known CVEs.

Analyzing Torchvision's ecosystem using the DIC we find that four of the dependencies for Torchvision have versions where researchers discovered vulnerabilities, for which CVE numbers were issued. These four packages, their CVE numbers, and the depth of the package in Torchvision's dependency tree are in Table A.2. The CVE data we display represents CVEs found in any previous versions of packages in the set of dependencies. The current version of Torchvision, v0.13.0, and the current compatible versions of packages in Torchvision's dependency tree have no CVEs available to date.

Table A.2. Torchvision Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	1
Pillow	CVE-2022-30595, CVE-2022-22815, CVE-2022-24303, CVE-2020-35653, CVE-2022-22817, CVE-2022-22816, CVE-2021-27922, CVE-2021-27921, CVE-2021-27923, CVE-2021-25290, CVE-2021-25289, CVE-2021-25291, CVE-2021-25292, CVE-2021-25293, CVE-2021-25287, CVE-2021-25288, CVE-2021-23437, CVE-2021-34552, CVE-2021-28676, CVE-2021-28675	1
requests	CVE-2021-21676, CVE-2021-21674, CVE-2021-21675, CVE-2021-29476, CVE-2018-18074, CVE-2015-2296, CVE-2014-1830, CVE-2014-1829	1
urllib3	CVE-2021-33503, CVE-2021-28363, CVE-2020-26137, CVE-2019-11324, CVE-2019-11236, CVE-2018-20060, CVE-2020-7212, CVE-2016-9015	2

A majority of the forty-four total CVEs in the packages of Torchvision's supply chain exist at a depth of 1. Eight CVEs for both the Numpy and Requests packages and twenty for Pillow. The final eight CVEs apply to urllib3, which resides at depth 2 in the Torchvision dependency tree. All of these same packages also exist in TensorFlow's supply chain. We find shared vulnerable dependencies in many of our chosen sample packages and analyze this further in Section 4.2. The following two graphs illustrate the relationship between CVE quantity and dependency depth.

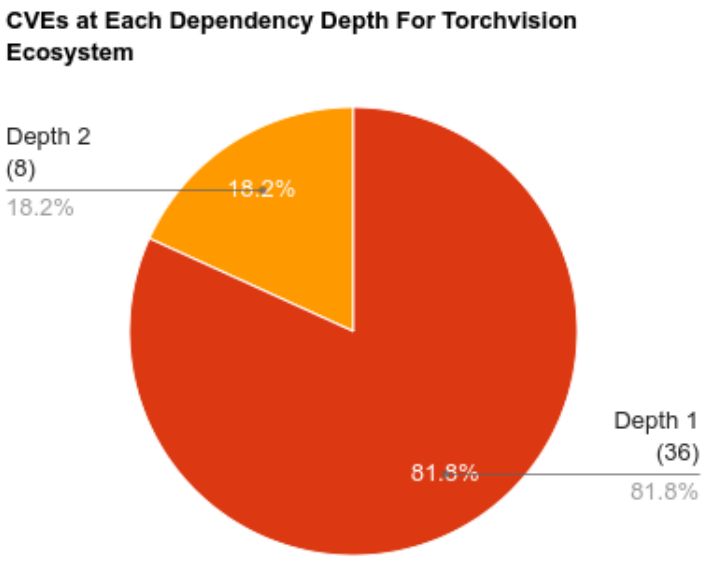

Figure A.5. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 44 total CVEs in the Torchvision Ecosystem. No CVEs exist at Depth 0 and thus it is not displayed.

Figure A.5 displays the quantity of CVEs found at each dependency depth in the Torchvision ecosystem. Here we see that, unlike TensorFlow, the vast majority of CVEs reside at a depth of one instead of at depth two or depth zero. In the case of Torchvision, because there are no further dependencies beyond depth 2, and possibly due to the Torch developers practice of vendoring large amounts of dependencies, we observed the entire ecosystem within the first three dependency levels.

A.3 SciPy

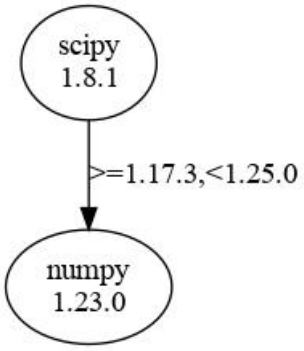

Figure A.6. This directed acyclic graph displays the packages in the SciPy ecosystem installed using pip. The package creators and maintainers intended it to rely on a minimal number of dependencies.

Table A.3. SciPy Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
scipy	CVE-2013-4251	0
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	1

The CVE data for SciPy shows that it has nine total CVEs in its ecosystem. One CVE applies to SciPy itself, and eight apply to its dependency, NumPy. As there are no further dependencies, we examine the entirety of the ecosystem we see and its CVE data within two dependency levels.

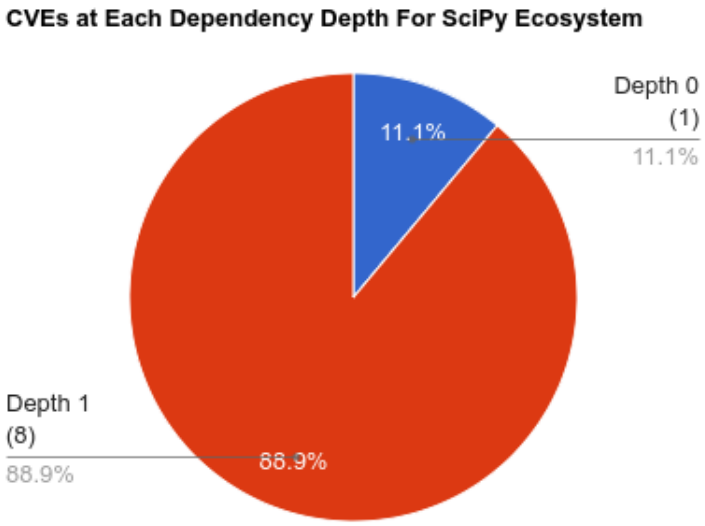

Figure A.7. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 9 total CVEs in the SciPy Ecosystem.

A.4 Scikit-learn

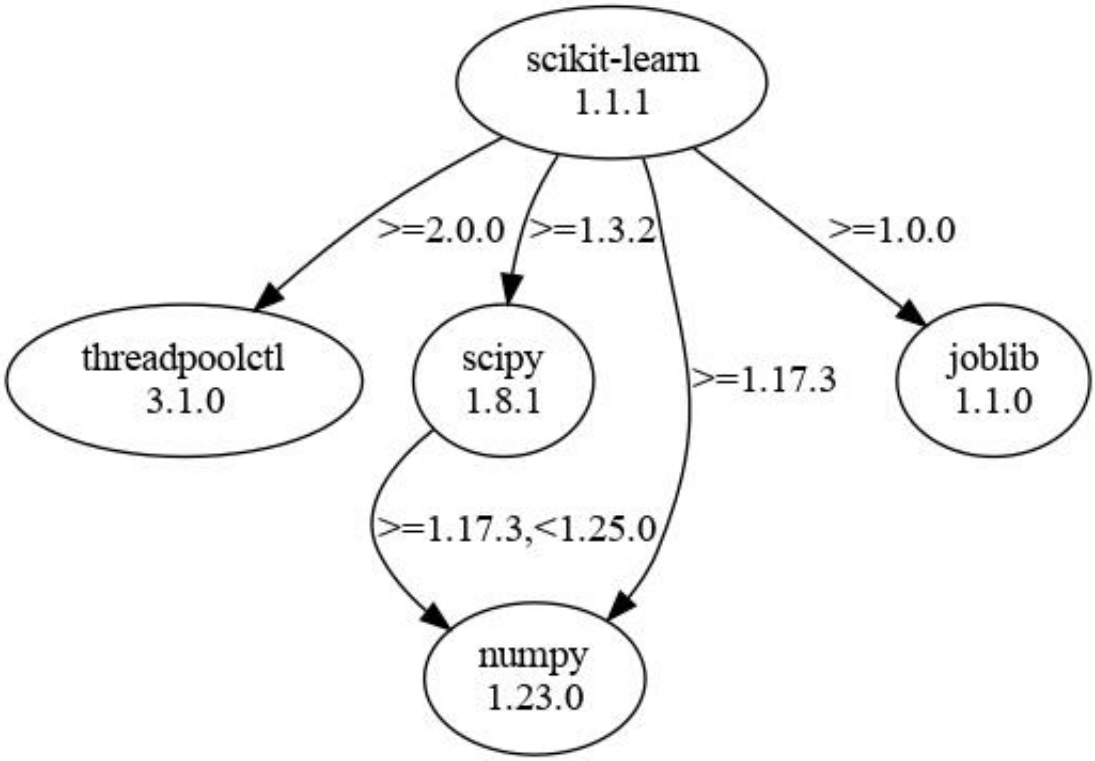

Figure A.8. This directed acyclic graph displays the packages in the Scikit-learn ecosystem installed using pip. The package creators and maintainers intended it to rely on a minimal number of dependencies.

Table A.4. Scikit-learn Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
scikit-learn	CVE-2020-28975, CVE-2020-13092	0
scipy	CVE-2013-4251	1
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	2

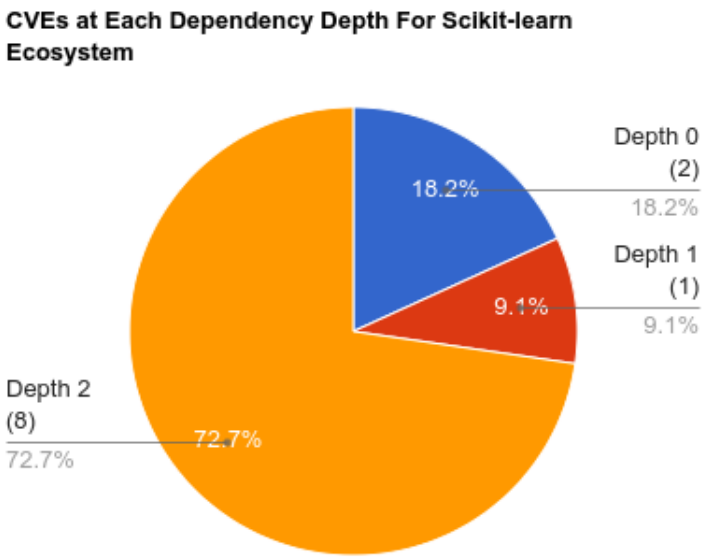

Figure A.9. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 11 total CVEs in the Scikit-learn Ecosystem.

In Scikit-learn's ecosystem, we found CVEs for the Scikit-learn, SciPy, and NumPy packages. Due to Scikit-learn being the parent package, the dependency depth data looks similar to SciPy's with the exception of two CVEs for Scikit-learn at depth 0. Overall, all CVEs can be observed within three dependency depths as there are no deeper dependencies. In summary, the CVE-dependency depth data for Scikit-learn incorporates the data collected for SciPy and includes CVEs for Scikit-learn itself.

A.5 Pandas

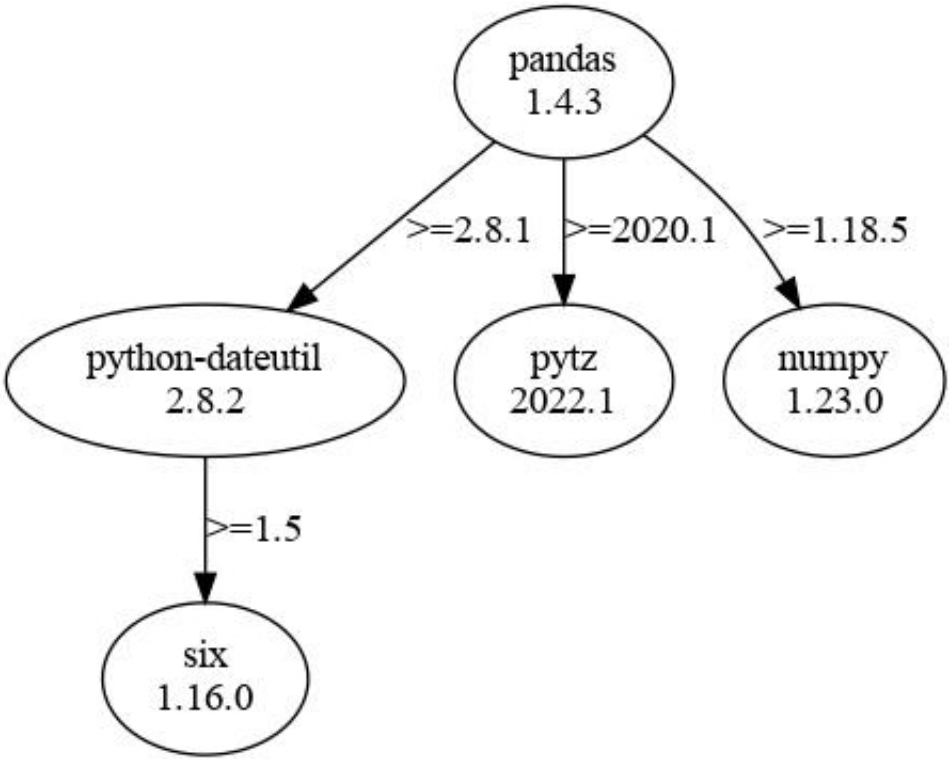

Figure A.10. This directed acyclic graph displays the packages in the Pandas ecosystem installed using pip.

Table A.5. Pandas Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
pandas	CVE-2020-13091	0
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	1

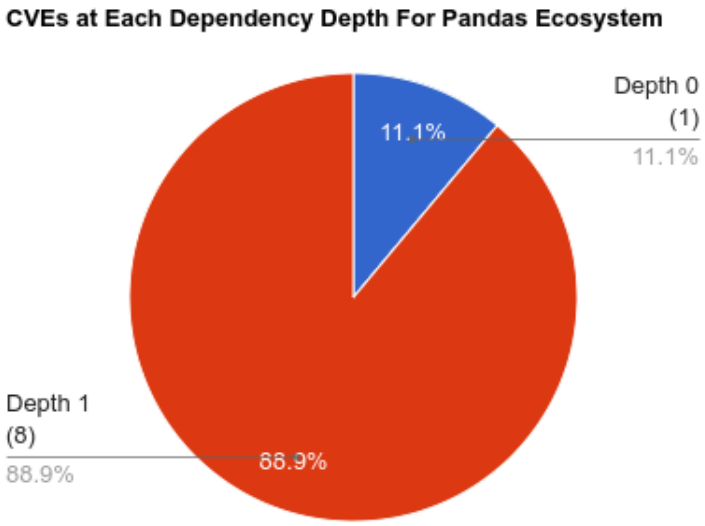

Figure A.11. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 9 total CVEs in the Pandas Ecosystem.

Observing the depth of CVEs for the Pandas ecosystem, we find that the NumPy package, a direct dependency, is the only dependency that has CVEs associated. One CVE existed for the Pandas package itself, which resides at depth zero as it is the parent package. Overall, we only needed to evaluate the first two dependency levels of Pandas to find all of the CVEs for the ecosystem.

A.6 Matplotlib

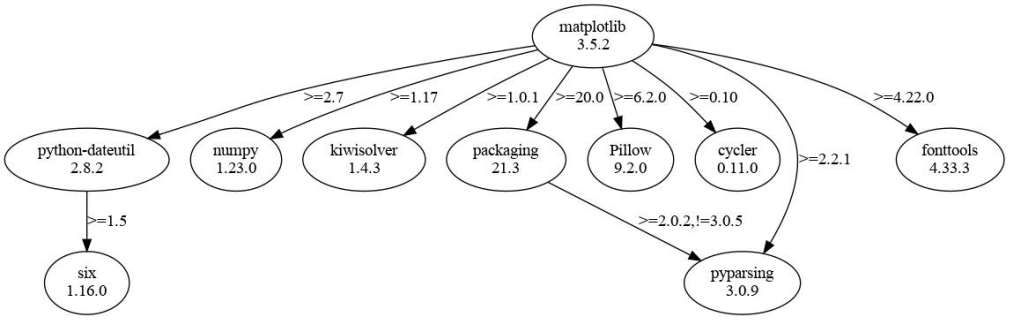

Figure A.12. This directed acyclic graph displays the packages in the Matplotlib ecosystem installed using pip.

Table A.6. Matplotlib Ecosystem CVEs for All Versions of Dependencies and Corresponding Depths.

Package	CVEs	Depth
numpy	CVE-2021-41496, CVE-2021-34141, CVE-2021-41495, CVE-2021-33430, CVE-2017-12852, CVE-2019-6446, CVE-2014-1859, CVE-2014-1858	1
Pillow	CVE-2022-30595, CVE-2022-22815, CVE-2022-24303, CVE-2020-35653, CVE-2022-22817, CVE-2022-22816, CVE-2021-27922, CVE-2021-27921, CVE-2021-27923, CVE-2021-25290, CVE-2021-25289, CVE-2021-25291, CVE-2021-25292, CVE-2021-25293, CVE-2021-25287, CVE-2021-25288, CVE-2021-23437, CVE-2021-34552, CVE-2021-28676, CVE-2021-28675	1

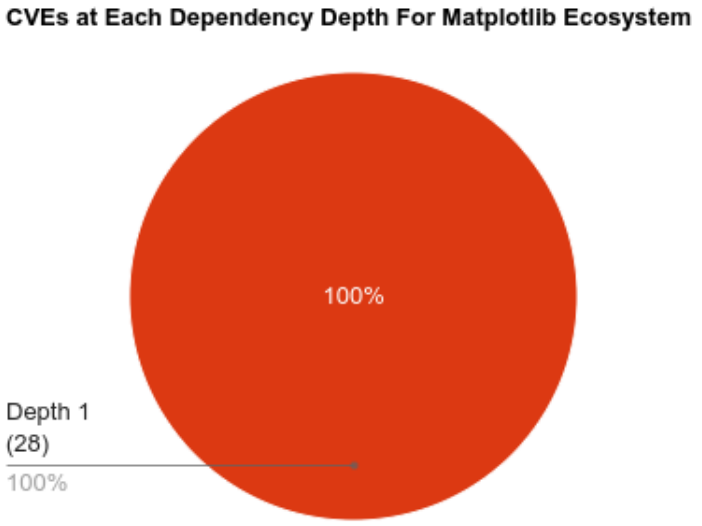

Figure A.13. This pie chart shows the quantity in parentheses and percentages of CVEs at each depth out of the 28 total CVEs in the Matplotlib Ecosystem.

Matplotlib's ecosystem is the only one in our sample with only its direct dependencies having associated CVE entries. In the rest of our sample, either a transitive dependency or the parent package would have CVEs associated with them.

List of References

[1] U.S. Government Accountability Office, "SolarWinds cyberattack demands significant federal and private-sector response (infographic)," Apr. 22, 2021 [Online]. Available: https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic

[2] npm, "npm blog archive: kik, left-pad, and npm," Mar. 23, 2016 [Online]. Available: https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm

[3] A. Koçulu, "I've just liberated my modules — Medium," Mar. 22, 2016 [Online]. Available: https://web.archive.org/web/20160625033227/https://medium.com/@azerbike/i-ve-just-liberated-my-modules-9045c06be67c

[4] K. Collins, "How one programmer broke the internet by deleting a tiny piece of code," Mar. 27, 2016 [Online]. Available: https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/

[5] C. Williams, "How one developer just broke node, babel and thousands of projects in 11 lines of JavaScript," Mar. 23, 2016 [Online]. Available: https://www.theregister.com/2016/03/23/npm_left_pad_chaos/

[6] JAIC, "Joint Artificial Intelligence Center," Sep. 15, 2021 [Online]. Available: https: //dodcio.defense.gov/About-DoD-CIO/Organization/jaic/

[7] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc, 2019.

[8] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*. ACM Press, 2008, p. 565. Available: http: // portal.acm.org/citation.cfm?doid=1455770.1455841

[9] 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 *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, 2021. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[10] M. Thoma, "Dependency vendoring," Jan. 3, 2021 [Online]. Available: https: //medium.com/plain-and-simple/dependency-vendoring-dd765be75655@

[11] W. Kennedy, “Manage dependencies with GODEP,” Oct. 26, 2013 [Online]. Available: https://www.ardanlabs.com/blog/2013/10/manage-dependencies-with-godep.html

[12] C. Wanstrath, “Vendor everything,” Mar. 27, 2007 [Online]. Available: http://errtheblog.com/posts/50-vendor-everything

[13] Go Developers, “Go modules reference - The Go programming language,” Accessed June 8, 2022 [Online]. Available: https://go.dev/ref/mod

[14] P. Gedam, *Vendoring: A Command Line Tool, to Simplify Vendoring Pure Python Dependencies*, 2021 [Online].

[15] The pip developers, “Vendoring policy - Pip documentation v22.2.dev0,” Ac-cessed June 7, 2022 [Online]. Available: https://pip.pypa.io/en/latest/development/vendoring-policy/#rationale

[16] D. E. Mann and S. M. Christey, “Towards a common enumeration of vulnerabilities,” in *2nd Workshop on Research with Security Vulnerability Databases*, 1999. Available: https://www.cve.org/Resources/General/Towards-a-Common-Enumeration-of-Vulnerabilities.pdf

[17] Sonatype, “2021 state of the software supply chain,” Sonatype, Tech. Rep., 2021. Available: https://www.sonatype.com/hubfs/Q3%202021-State%20of%20the%20Software%20Supply%20Chain-Report/SSSC-Report-2021_0913_PM_2.pdf?hsLang=en-us

[18] M. Alfadel, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in Python packages,” in *2021 IEEE International Conference on Software Analy-sis, Evolution and Reengineering (SANER)*, 2021, pp. 446–457.

[19] Cybersecurity and Infrastructure Security Agency, “Defending against software supply chain attacks,” Cybersecurity and Infrastructure Security Agency, Tech. Rep., 2021 [Online]. Available: https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf

[20] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A taxonomy of computer program security flaws,” *ACM Comput. Surv.*, vol. 26, no. 3, pp. 211–254, 1994. Available: https://dl.acm.org/doi/10.1145/185403.185412

[21] P. Karger and R. Schell, “Multics security evaluation: Vulnerability analysis,” Information Systems Technology Application Office Deputy for Command and Management Systems Electronic Systems Division (AFSC), Hanscom AFB, Bedford, MA 01730, Tech. Rep. ESD-TR-74-193, 1974. Available: http://ieeexplore.ieee.org/document/1176286/@

[22] P. A. Myers, "Subversion: The neglected aspect of computer security," M.S. thesis, Dept. of Comp. Sci., NPS, Monterey, CA, USA, 1980. Available: https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/myer80.pdf

[23] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984. Available: https://dl.acm.org/doi/10.1145/358198.358210

[24] Cybersecurity and Infrastructure Agency, "Apache log4j vulnerability guidance | CISA," Apr. 8, 2022 [Online]. Available: https://www.cisa.gov/uscert/apache-log4j-vulnerability-guidance

[25] M. Čarnogurský, "Attacks on package managers," Bachelor's thesis, Faculty of Informatics, Masaryk University, 2019 [Online]. Available: https://is.muni.cz/th/y41ft/thesis_final_electronic.pdf

[26] ESLint, "ESLint - Pluggable JavaScript linter," Accessed May 13, 2022 [Online]. Available: https://eslint.org/

[27] H. Zhu, "Postmortem for malicious packages published on july 12th, 2018," July 12, 2018 [Online]. Available: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes

[28] C. McConnell, "Urgent: Security: New maintainer is probably malicious · issue #1263 · greatsuspender/thegreatsuspender," Nov. 3, 2020 [Online]. Available: https://github.com/greatsuspender/thegreatsuspender/issues/1263

[29] R. Lakshmanan, "Warning — Hugely popular 'the great suspender' chrome extension contains malware," Feb. 6, 2021 [Online]. Available: https://thehackernews.com/2021/02/warning-hugely-popular-great-suspender.html

[30] J. Cox, "Open source maintainer sabotages code to wipe Russian, Belarusian computers," Mar. 18, 2022 [Online]. Available: https://www.vice.com/en/article/dypeek/open-source-sabotage-node-ipc-wipe-russia-belraus-computers

[31] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110653, 2021. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121220301199

[32] F. Besson, T. Blanc, C. Fournet, and A. Gordon, "From stack inspection to access control: A security analysis for libraries," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, 2004, pp. 61–75.

[33] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2021, pp. 1–11. Available: https://dl.acm.org/doi/10.1145/3475716.3475769

[34] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112. Available: http://ieeexplore.ieee.org/document/7962360/

[35] M. Zimmermann, C.-A. Staicu, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," *Proceedings of the 28th USENIX Security Symposium*, 2019.

[36] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Aspect Security Inc.*, pp. 1–26, 2012.

[37] P. P. W. Pathirathna, V. A. I. Ayesha, W. A. T. Imihira, W. M. J. C. Wasala, N. Kodagoda, and E. A. T. D. Edirisinghe, "Security testing as a service with docker containerization," in *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, 2017, pp. 1–7.

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

[39] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "SV-AF — A security vulnerability analysis framework," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 219–229.

[40] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," arXiv, Tech. Rep. arXiv:2005.09535, 2020. Available: http://arxiv.org/abs/2005.09535

[41] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "TensorSCONE: A secure TensorFlow framework using intel SGX," arXiv, Tech. Rep. arXiv:1902.04413, 2019. Available: http://arxiv.org/abs/1902.04413

[42] NIST, "Software supply chain security guidance under executive order (EO) 14028 section 4e," Washington, D.C., 2022 [Online]. Available: https://www.nist.gov/system/files/documents/2022/02/04/software-supply-chain-security-guidance-under-EO-14028-section-4e.pdf

[43] M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (SSDF) version 1.1: (draft): Recommendations for mitigating the risk of software vulnerabilities," National Institute of Standards and Technology, Tech. Rep. NIST 800-218, 2022 [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf

[44] E. Camper, "What is the NIST SP 800-218 (draft) "Secure Software Development Framework" and why should we (as an org selling software to the USG) care?" Nov. 10, 2021 [Online]. Available: https://www.pivotpointsecurity.com/blog/what-is-the-nist-sp-800-218-draft-secure-software-development-framework-and-why-should-we-as-an-org-selling-software-to-the-usg-care/

[45] National Institute of Standards and Technology, "Framework for improving critical infrastructure cybersecurity, version 1.1," National Institute of Standards and Technology, Tech. Rep. NIST CSWP 04162018, 2018 [Online]. Available: http://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf

[46] Reproducible Builds, "Reproducible builds — A set of software development practices that create an independently-verifiable path from source to binary code." Accessed May. 2, 2022 [Online]. Available: https://reproducible-builds.org/

[47] Docker, "What is a container? | docker," Nov. 11, 2021 [Online]. Available: https://www.docker.com/resources/what-container/

[48] NTIA Multistakeholder Process on Software Component Transparency Framing Working Group, "Framing software component transparency: Establishing a common software bill of materials (SBOM)," National Telecommunications and Information Administration, Tech. Rep. Second Edition, 2021 [Online]. Available: https://www.ntia.gov/files/ntia/publications/ntia_sbom_framing_2nd_edition_20211021.pdf

[49] NTIA Multistakeholder Process on Software Component Transparency, "SBOM at a glance," National Telecommunications and Information Administration, Tech. Rep., 2021 [Online]. Available: https://www.ntia.gov/files/ntia/publications/sbom_at_a_glance_apr2021.pdf

[50] NTIA Multistakeholder Process on Software Component Transparency Use Cases and State of Practice Working Group, "Roles and benefits for SBOM across the supply chain," National Telecommunications and Information Administration, Tech. Rep., 2019 [Online]. Available: https://www.ntia.gov/files/ntia/publications/ntia_sbom_use_cases_roles_benefits-nov2019.pdf

[51] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin,

S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. Available: http://agilemanifesto.org/

[52] Atlassian, "What is DevOps?" Accessed July. 2, 2023 [Online]. Available: https://www.atlassian.com/devops

[53] GSA Office of the CTO, "Understanding the differences between Agile & DevSecOps - from a business perspective," Accessed May 4, 2017 [Online]. Available: https://tech.gsa.gov/guides/understanding_differences_agile_devsecops/

[54] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A multivocal literature review," in *International Conference on Software Process Improvement and Capability Determination*, 2017, pp. 17–29.

[55] R. Graham, "Software bill of materials (SBoM) - Does it work for DevSecOps?" Jan. 14, 2019 [Online]. Available: https://cybersecurity.att.com/blogs/security-essentials/software-bill-of-materials-sbom-does-it-work-for-devsecops

[56] V. Naik, *pipdeptree*, 2022 [Online]. Available: https://github.com/naiquevin/pipdeptree

[57] National Vulnerability Database, "API vulnerabilities," Accessed May 18, 2022 [Online]. Available: https://nvd.nist.gov/developers/vulnerabilities

[58] Google Cloud Platform, *google-auth: Google Authentication Library*, Accessed June 17, 2022 [Online]. Available: https://github.com/googleapis/google-auth-library-python

[59] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow, Large-scale machine learning on heterogeneous systems*, 2015 [Online]. Available: https://github.com/tensorflow/tensorflow

[60] Tensorflow, "Case studies and mentions," Accessed May 14, 2022 [Online]. Available: https://www.tensorflow.org/about/case-studies

[61] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016, vol. 16,

pp. 265–283. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[62] J. Brownlee, "PyTorch tutorial: How to develop deep learning models with Python," Mar. 22, 2020 [Online]. Available: https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/

[63] A. Paszke, S. Gross, S. Chintala, and G. Chanan, *pytorch/pytorch*, 2022 [Online]. Available: https://github.com/pytorch/pytorch

[64] PyTorch Contributors, "PyTorch," Accessed May 18, 2022 [Online]. Available: https://www.pytorch.org

[65] NVIDIA, "CUDA zone - Library of resources," July. 18, 2017 [Online]. Available: https://developer.nvidia.com/cuda-zone

[66] T. A. Caswell, M. Droettboom, A. Lee, E. S. De Andrade, T. Hoffmann, J. Klymak, J. Hunter, E. Firing, D. Stansby, N. Varoquaux, J. H. Nielsen, B. Root, R. May, P. Elson, J. K. Seppänen, D. Dale, J.-J. Lee, D. McDougall, A. Straw, P. Hobson, Hannah, C. Gohlke, A. F. Vincent, T. S. Yu, E. Ma, S. Silvester, C. Moad, N. Kniazev, E. Ernest, and P. Ivanov, *matplotlib/matplotlib: REL: v3.5.2*, 2022 [Online]. Available: https://zenodo.org/record/6513224

[67] P. Mahto, "Matplotlib for machine learning," Oct. 16, 2020 [Online]. Available: https://medium.com/mlpoint/matplotlib-for-machine-learning-6b5fcd4fbbc7

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. Available: http://jmlr.org/papers/v12/pedregosa11a.html

[69] Pandas Team, "Pandas - Python data analysis library," Accessed July 3, 2022 [Online]. Available: https://pandas.pydata.org/about/

[70] SciPy Organization. Introduction — SciPy v1.8.1 Manual. Accessed July. 3, 2022 [Online]. Available: https://docs.scipy.org/doc/scipy/tutorial/general.html

[71] SciPy Organization, *scipy: SciPy: Scientific Library for Python*, Accessed July 3, 2022 [Online]. Available: https://www.scipy.org

[72] B. A. Cheikes, D. Waltermire, and K. Scarfone, "Common platform enumeration :: naming specification version 2.3," National Institute of Standards and Technology, Tech. Rep. NIST IR 7695, 2011 [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7695.pdf

[73] Torch Contributors, "Torchvision — Torchvision 0.13 documentation," Accessed July 7, 2022 [Online]. Available: https://pytorch.org/vision/stable/index.html

[74] G. Niemeyer, *python-dateutil: Extensions to the Standard Python Datetime Module*, Accessed July 12, 2022 [Online]. Available: https://github.com/dateutil/dateutil

[75] S. Bishop, *pytz: World Timezone Definitions, Modern and Historical*, Accessed July 12, 2022 [Online]. Available: http://pythonhosted.org/pytz

[76] Google Developers, *FlatBuffers*, 2022 [Online]. Available: https://github.com/google/flatbuffers

[77] MITRE, "NVD - CVE-2020-35864," Dec. 31, 2020 [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-3586

[78] Github Inc., "Using the dependency submission API," Accessed July 19, 2022 [Online]. Available: https://ghdocs-prod.azurewebsites.net/_next/data/MAZIHvJs8lNbl34zWZRHd/en/free-pro-team@latest/code-security/supply-chain-security/understanding-your-software-supply-chain/using-the-dependency-submission-api.json?versionId=free-pro-team%40latest&productId=code-security&restPage=supply-chain-security&restPage=understanding-your-software-supply-chain&restPage=using-the-dependency-submission-api

[79] Torchaudio Contributors, "Torchaudio documentation — Torchaudio 0.12.0 documentation," Accessed July 7, 2022 [Online]. Available: https://pytorch.org/audio/stable/index.html@

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California