

Graph Metrics for Improving Cybersecurity on Software Dependency Networks

by

Darren Z. Yao

S.B., Electrical Engineering and Computer Science and Mathematics
Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

© 2025 Darren Z. Yao. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Darren Z. Yao
Department of Electrical Engineering and Computer Science
May 9th, 2025

Certified by: Ranjan Pal
Research Scientist, Thesis Supervisor

Certified by: Michael D. Siegel
Principal Research Scientist, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair
Master of Engineering Thesis Committee

Graph Metrics for Improving Cybersecurity on Software Dependency Networks

by

Darren Z. Yao

Submitted to the Department of Electrical Engineering and Computer Science
on May 9th, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

ABSTRACT

Modern software ecosystems are deeply interconnected, allowing a vulnerability in a single component to propagate and affect many others. In this thesis, we model software ecosystems as directed graphs, and apply various graph-theoretic metrics to quantify security risk. We compare two deep learning frameworks (PyTorch and TensorFlow) with two traditional software frameworks (npm and PyPI), identifying critical properties of their dependency structures, which motivates several recommendations for improving software supply chain security.

Thesis supervisor: Ranjan Pal

Title: Research Scientist

Thesis supervisor: Michael D. Siegel

Title: Principal Research Scientist

Contents

<i>List of Figures</i>	7
<i>List of Tables</i>	9
1 Introduction	11
2 Background and Related Works	13
2.1 Software Dependencies	13
2.2 SBOMs and AIBOMs	14
3 Analysis of Deep Learning Libraries	17
3.1 Methodology and Data	17
3.2 Component Size	18
3.3 Centrality Product Metrics	22
3.4 Analysis of Top Packages	25
4 Analysis of Traditional Software Libraries	29
4.1 Methodology and Data	30
4.2 Component Size	30
4.3 SourceRank	34
4.4 Comparing Traditional Software and Deep Learning	36
5 Additional Proposed Metrics, Future Works, and Recommendations	39
5.1 SBOM recommendations	39
5.2 Conclusion	40
5.3 Future Works	41
5.3.1 PyTorch and TensorFlow	41
5.3.2 npm and PyPI	41
5.3.3 Weighted Graphs	42
5.3.4 Stochastic Graphs	43
<i>References</i>	45

List of Figures

3.1	Top Component Sizes, PyTorch	20
3.2	Top Component Sizes, PyTorch	21
3.3	Top Component Sizes, PyTorch	23
3.4	Top Component Sizes, TensorFlow	23
4.1	Top Component Sizes, npm and PyPI	31
4.2	Top SourceRank scores histogram	35
4.3	Top SourceRank scores by rank	35
4.4	Component size comparison between PyTorch/TensorFlow and npm/PyPI	36

List of Tables

3.1	Top 10 Modules by Component Size, PyTorch	19
3.2	Top 10 Modules by Component Size, TensorFlow	20
3.3	Top 10 Modules by Product Metric, PyTorch	25
3.4	Top 10 Modules by Product Metric, TensorFlow	25
4.1	Top 10 Modules by Component Size, npm	32
4.2	Top 10 Modules by Component Size, PyPI	33

Chapter 1

Introduction

Suppose an attacker gains access to a widely used open-source library, and is able to inject malicious code, which then propagates downstream to other software packages that use the open-source library as a dependency, eventually spreading widely across many pieces of software. One real-world example of this is the event-stream incident, where code had been introduced into a popular npm package in order to steal Bitcoin from user accounts. These attacks rely on the nature of software dependencies, as developers who use third-party packages treat them as black boxes without understanding them. As many open-source libraries are widely used across software development in every industry, protecting against such vulnerabilities is increasingly important. This is an ever growing concern as modern software development becomes highly compartmentalized into individual libraries and reliant on third-party software components, as the transitive nature of dependencies allow individual vulnerabilities to have wide-ranging security consequences. The proliferation of generative AI in recent years has added another layer to these concerns, as the vast datasets needed for training and the adaptation of pre-trained models expand the potential impact of security flaws. Therefore, it is now critical to understand and track dependencies in both traditional software and AI models.

The example above is one common type of attack on software systems. The other type

is an attack on a specific software component, causing a simultaneous failure of all devices using that component (one recent example of this is the CrowdStrike-related outages). In this thesis, we focus on only the first type of attack.

Accordingly, one goal of developers and cybersecurity experts is to design software that minimizes the chances of such attacks occurring. This largely consists of investigating software dependency graphs, in which nodes represent packages, and directed edges represent dependencies. In this thesis, we investigate various graph metrics on the dependency graphs of popular open-source libraries to find results that help with prioritization of developer efforts. The rest of the thesis will be structured as follows:

- Chapter 2 provides the necessary background on dependency graphs of software ecosystems, and SBOMs/AIBOMs that we will make recommendations about.
- Chapter 3 details the machine learning infrastructure libraries PyTorch and TensorFlow.
- Chapter 4 investigates traditional software ecosystems, exemplified by the JavaScript package manager npm and the Python package manager PyPI.
- Chapter 5 discusses recommendations, conclusions, and potential future works.

Chapter 2

Background and Related Works

We begin with a summary of related work, consisting of results about software dependencies and an introduction to SBOMs and AIBOMs.

2.1 Software Dependencies

Recent research has revealed significant security risks in software packages, especially those with many dependencies or black-boxed third party dependencies. Zimmerman et al found that in npm (Node Package Manager), dependencies on popular packages and important maintainer accounts constitute many of the vulnerabilities in the ecosystem, and therefore the number of points of attack are actually relatively small [1]. Similarly, Zahan et al provided additional insight into the weakest links in the npm supply chain, identifying criteria heavily correlated with security vulnerabilities, including unmaintained packages, installation scripts, and the number of maintainers and contributors [2].

Existing analyses on software supply chains tend to focus on user accounts, maintainers, and other properties of the software itself, rather than properties of the dependency graph. Graph theoretic methods for software dependencies can yield useful insights around the relative importance of modules and packages, but they have not been sufficiently well studied in the current research literature. Therefore, we aim to investigate various graph metrics, applying

a more theoretical approach to the practical problem of security on software dependencies.

2.2 SBOMs and AIBOMs

For large software systems, cataloging individual modules is critical, yet increasingly difficult. One method of tracking software components is known as the Software Bill of Materials (SBOM) [3], which provides a comprehensive list of components, libraries, and dependencies in a software system. As Bi et al explains, this allows for easy verification of individual components and convenient tracing of issues. [4] As security-critical software systems in government and defense related industries set stringent regulations for their software systems, SBOMs are increasingly becoming a compliance mechanism for these requirements [4] [5], as well as an industry standard in other critical sectors like healthcare and finance. SBOMs are particularly useful in understanding dependencies on third party packages, which are essentially black boxes in terms of understanding the internal implementations.

An analogous tool for artificial intelligence models is the AI Bill of Materials (AIBOM). An AIBOM extends the SBOM concept to include both pure software dependencies as well as datasets and the processes used in obtaining the datasets. By listing their software components and data sources for open auditing, AIBOMs help guard against data quality, privacy, and bias concerns. Tan et al conducted an exploration of deep learning supply chains focusing on the two common libraries TensorFlow and PyTorch. Their work reveals the dependency structure of these libraries, and how any individual vulnerability can have cascading downstream effects affecting numerous dependencies. [6]

This is particularly relevant as artificial intelligence services are increasingly used for automation across various fields such as finance, healthcare, and education. In the event of a security issue, such as a model being trained on proprietary data, an AIBOM would allow the organization to trace the extent to which their data was compromised.

Currently, SBOMs [7] [8] and AIBOMs contain only information about the software

components themselves and their immediate dependencies. Our proposed research on graph metrics will yield valuable insights on the importance of modules, and we propose augmenting SBOMs and AIBOMs with additional fields based on this information.

Chapter 3

Analysis of Deep Learning Libraries

Machine learning frameworks such as PyTorch and TensorFlow have become fundamental infrastructure for building all sorts of deep learning models. These frameworks provide tools for all steps of the machine learning pipeline, ranging from model construction, training, and testing, to convenient dataset operations and GPU acceleration capabilities [9].

The importance of PyTorch and TensorFlow to deep learning across both academia and industry makes them highly useful examples of software packages to analyze for cybersecurity concerns. In this chapter, we consider the underlying dependency graphs on their modules, extending the work done by Tan et al. [6]. In these graphs, we investigate the component size (number of transitive dependents) for each module, as well as two product metrics on upstream dependencies and downstream dependents, and analyze the purposes and functionalities of the most important modules in PyTorch and TensorFlow.

3.1 Methodology and Data

We first construct the dependency graphs of PyTorch and TensorFlow, using dependency edge list dataset provided by Tan et al [6]. We convert the edge list to an adjacency list, which we use for our remaining computations on the graph.

In our dataset, there are 136507 modules in PyTorch and 355662 modules in TensorFlow.

Due to limits on computing power, we can run roughly at most 10^9 operations in a reasonable amount of time. Therefore, the large size of the dataset imposes constraints on what algorithms we can use to compute the various metrics.

3.2 Component Size

The first metric we consider is the component size within the dependency graph, counting the number of (direct and transitive) downstream dependents for a module. This represents how many software modules an unchecked vulnerability can eventually propagate to. This is a well-known and often-studied metric for software systems; it is also known as indirect dependents, transitive dependents, etc.

Finding the component sizes for each node is a computationally difficult problem, and it is not possible to do asymptotically better than the naive solution of simply running a graph search from each node to find the number of reachable nodes in $O(N^2)$. However, motivated by a classical technique in competitive programming, we use binary representations of dependency information and bitmask operations for computation. This improves the constant factor significantly, and allows the algorithm to run in under half an hour on an M2 MacBook Pro.

In our algorithm, we number all the modules 0 through $N - 1$, where for each module m we store, by an N -bit integer, where the k th bit is 1 if k is a dependency of m , and 0 otherwise. For each module m , we simply take the bitwise OR of 2^m with all the values of m 's dependents, by iterating through the dependents in topological order, and then count the number of set bits in the final value corresponding to each module, as given by the following algorithm.

Algorithm 1: Component Size

Input : Graph $G = (V, E)$ (in adjacency list format)

Output: Map of each node to its component size

```
1  $map \leftarrow [N]$ : [empty list]
2  $T \leftarrow$  topological ordering of vertices in  $G$   $ans \leftarrow$  empty map
3 for vertices  $i = 0$  through  $N - 1$  do
4    $map[i] \leftarrow 2^n$ 
5 for vertex  $v$  in topological ordering  $T$  do
6   for neighbor  $w$  of  $v$  do
7      $map[w] \leftarrow map[w]$  OR  $map[v]$ 
8 for vertex  $v$  in topological ordering  $T$  do
9    $ans[v] \leftarrow bitcount(map[v])$ 
10 return  $ans$ 
```

In each package, the 10 modules with the largest component size (most transitive dependents) are as follows.

Index	PyTorch Module	Component Size
1	pytorch_pytorch	136506
2	lanpa_tensorboardX	11302
3	bamos_block	1952
4	IljaManakov_PyTorch-Trainer	1422
5	onnx_onnx-coreml	1034
6	albu_albumentations	875
7	uber_hypothesis-gufunc	760
8	dmlc_gluon-cv	530
9	pytorch_audio	514
10	EvilPsyCho_TaskBot	462

Table 3.1: Top 10 Modules by Component Size, PyTorch

Index	TensorFlow Module	Component Size
1	tensorflow_tensorflow	355661
2	lanpa_tensorboardX	11549
3	InzamamRahaman_UWIGoPiGo	7629
4	OuYanghaoyue_gym_contra	2903
5	tf-coreml_tf-coreml	2165
6	keras-team_keras-applications	1583
7	yesup_tensorflow-serving-client-python	1114
8	dmlc_tvm	636
9	RetroRabbit_skip_thoughts	607
10	vivek3141_ml	534

Table 3.2: Top 10 Modules by Component Size, TensorFlow

In PyTorch, only 419 modules have any downstream dependents, and in TensorFlow this number is 599. We can visualize the number of dependents as follows:

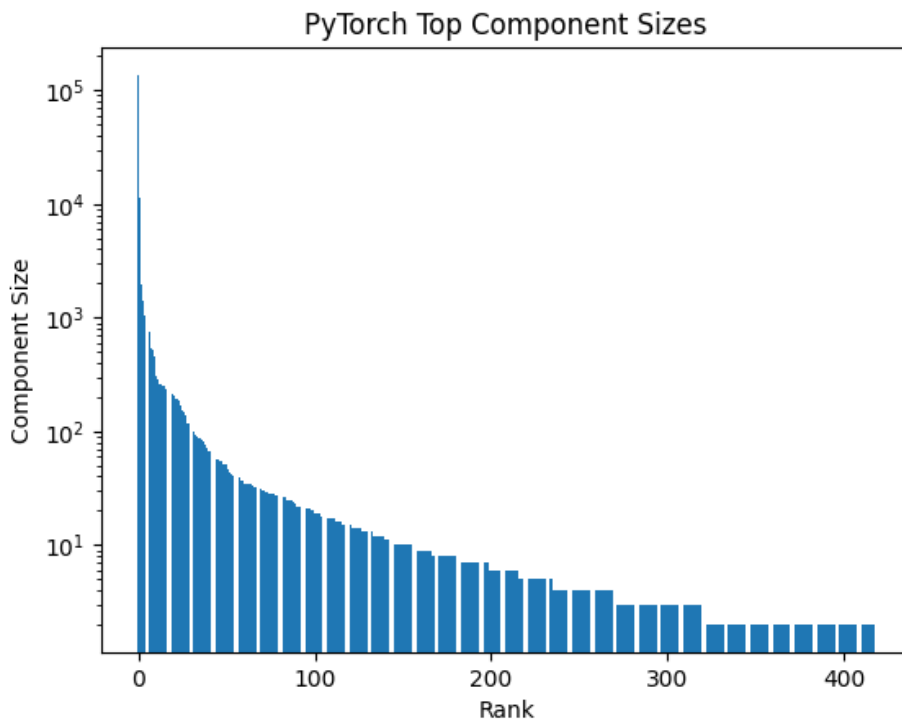


Figure 3.1: Top Component Sizes, PyTorch

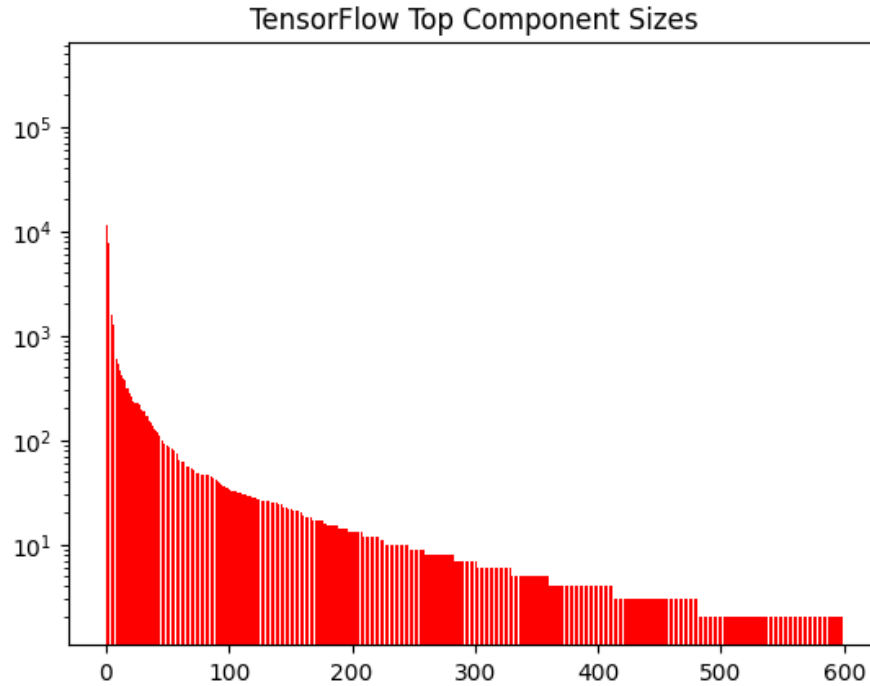


Figure 3.2: Top Component Sizes, PyTorch

We observe from these statistics that:

- Most modules don't have any dependents, and therefore aren't of concern. In PyTorch, roughly 0.31 percent of modules have any dependents, and in TensorFlow, the rate is 0.17 percent. Even though the large number of modules presents a significant surface, attackers will likely target only the highly interdependent infrastructure modules, which are the top few in each framework.
- Each of PyTorch and TensorFlow have a few hundred modules with dependents, but the top few modules have far more than the rest.
- Both PyTorch and TensorFlow have tensorboardX as the module with most dependents, other than the top-level modules themselves.

3.3 Centrality Product Metrics

A new metric we can consider for the spread of vulnerabilities in software systems is as follows: for each node, $(1 + \text{indegree})(1 + \text{outdegree})$, where we add 1 to each to count the module itself, and prevent root and leaf nodes from having a coefficient of zero.

This metric represents a module’s local contribution to security risk in the software dependency network by taking into account the number of modules immediately affecting it, and the number of modules it immediately affects. In context of the full supply chain, it considers both how likely a vulnerability is to affect a module, and the potential harm if it were compromised, providing a comprehensive measure of short-term security impact of the module.

We compute this product metric by storing the graph and the transpose graph, and multiplying the outdegrees of the node in the two graphs, as given by the following procedure:

Algorithm 2: Indegree-Outdegree Product

Input : Graph $G = (V, E)$ (in adjacency list format)

Output: Map of node to metric

```
1  $map \leftarrow [N] : [\text{empty list}]$ 
2  $G^T \leftarrow$  transpose graph of  $G$  (i.e.  $G$  with all edges reversed)
3 for  $node\ v\ in\ V$  do
4    $map[v] = (\text{outdegree of } v\ in\ G)(\text{outdegree of } v\ in\ G^T)$ 
5 return  $map$ 
```

Under this metric, we have the following graphs, where only modules with a coefficient of at least 10 are included.

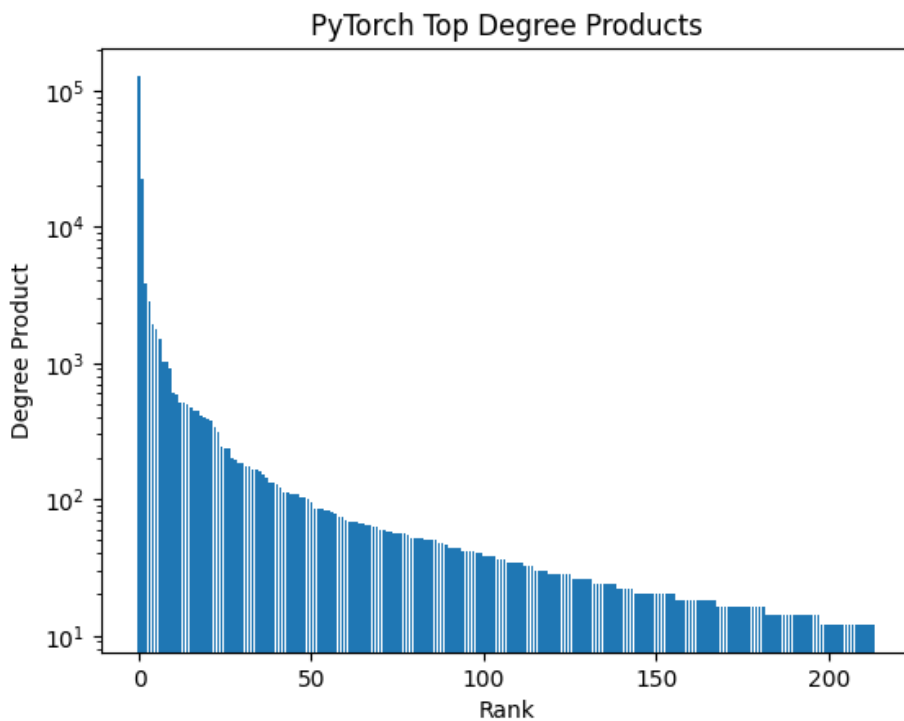


Figure 3.3: Top Component Sizes, PyTorch

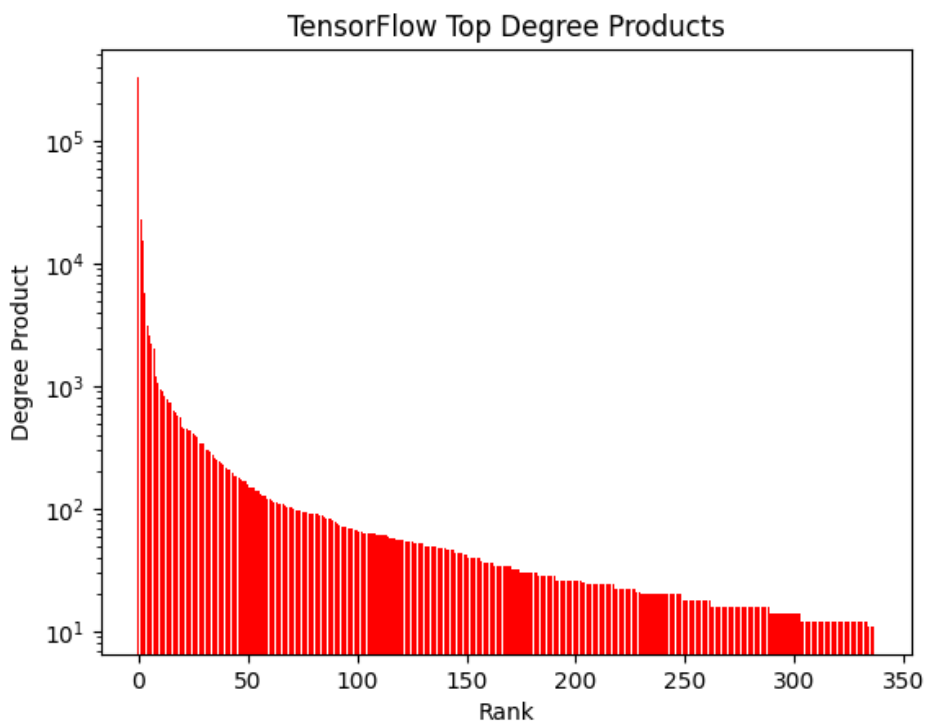


Figure 3.4: Top Component Sizes, TensorFlow

The ten most important modules in both PyTorch and TensorFlow according to this product metric are exactly the same as according to the component size metric, and the coefficients are almost proportional to the component sizes as well. This seems to suggest that modules rarely have both large numbers of immediate upstream dependents and immediate downstream dependents. Furthermore, modules usually import at most a few other modules, but may be imported by many immediate downstream modules.

We can also consider the product of component size in the original dependency graph with the component size in the transpose graph. In context of the full supply chain, it considers both how many potential vulnerabilities could spread to a certain module, and also how many downstream modules could be affected if it were compromised, measuring long-term security impact if issues fail to be resolved quickly.

We compute this metric by running the component size algorithm for both the original graph and the transpose graph, and then multiplying the results node-wise.

Algorithm 3: Component Size Product

Input : Graph $G = (V, E)$ (in adjacency list format)

Output: Map of node to component size product metric

```

1  $map \leftarrow [N] : [\text{empty list}]$ 
2  $G^T \leftarrow$  transpose graph of  $G$  (i.e.  $G$  with all edges reversed)
3  $C \leftarrow \text{ComponentSize}(G)$ 
4  $C^T \leftarrow \text{ComponentSize}(G^T)$ 
5 for node  $v$  in  $V$  do
6    $map[v] = C[v] \cdot C^T[v]$ 
7 return  $map$ 

```

Under this metric, these are the top ten most important modules:

Index	PyTorch Module	Coefficient
1	pytorch_pytorch	136506
2	lanpa_tensorboardX	22604
3	bamos_block	3904
4	IljaManakov_PyTorch-Trainer	2844
5	onnx_onnx-coreml	2068
6	albu_albumentations	1750
7	uber_hypothesis-gufunc	1520
8	dmlc_gluon-cv	1060
9	pytorch_audio	1028
10	EvilPsyCHo_TaskBot	924

Table 3.3: Top 10 Modules by Product Metric, PyTorch

Index	TensorFlow Module	Coefficient
1	tensorflow_tensorflow	355661
2	lanpa_tensorboardX	23098
3	InzamamRahaman_UWIGoPiGo	15258
4	OuYanghaoyue_gym_contra	5806
5	tf-coreml_tf-coreml	4330
6	onnx_onnx-coreml	3876
7	keras-team_keras-applications	3166
8	yesup_tensorflow-serving-client-python	2228
9	dmlc_tvm	1272
10	RetroRabbit_skip_thoughts	1214

Table 3.4: Top 10 Modules by Product Metric, TensorFlow

We note that the top ranking modules under this metric are very similar to those under the component size metric. In fact, for PyTorch, the top ten modules are exactly the same, and in TensorFlow there is only one difference (vivek3141_ml vs onnx_onnx-coreml).

3.4 Analysis of Top Packages

Excluding the overarching top-level packages *pytorch_pytorch* and *tensorflow_tensorflow*, we examine the next five packages from each library:

- `lanpa_tensorboardX`: Model training progress visualizer and model parameter logging system
- `bamos_block`: Block matrix computations
- `IljaManakov_PyTorch-Trainer`: For training models and logging results
- `onnx_onnx-coreml`: ML framework by Apple for optimizing hardware usage
- `albu_albumentations`: Image augmentation
- `uber_hypothesis-gufunc`: Generation of test data for numpy and related packages
- `InzamamRahaman_UWIGoPiGo`: For Raspberry Pi
- `OuYanghaoyue_gym_contra`: Python emulator environment for Nintendo Entertainment System
- `tf-coreml_tf-coreml`: Converter for neural network models to coreml, similar to `onnx-coreml`

Result 3A: The majority of important modules in PyTorch and TensorFlow are important infrastructure for training and testing of machine learning models, but a few are for rather niche uses such as NES console emulation.

Investigating the actual structure of the PyTorch and TensorFlow dependency graphs, we find that they highly resemble trees, where the second-level modules depend only on the top-level module, and the subtrees defined by the second-level nodes are largely disjoint from each other. Furthermore, the vast majority of nodes have indegree at most one, or outdegree at most one. When taken together, these observations explain the rapid exponential decay in component size and coefficient that we see in both graphs.

This indicates the supply chains of PyTorch and TensorFlow are not very interdependent, especially when compared to traditional software packages such as npm, where a large set of important infrastructure is widely used across a large fraction of modules in the ecosystem.

The resulting structural and security implication is that risks that occur in the second layer or below have their effects largely confined to only the relevant subtree. Therefore, developers who are concerned about security implications of using deep learning libraries should focus their efforts on the largest second-layer modules, as most other modules have relatively few dependents.

Result 3B: The PyTorch and TensorFlow dependency graphs are highly tree-like, and most modules exhibit a relatively low level of interdependence. Under all the metrics we considered, the large second-level modules are the most critical and should be the primary focus in regards to security.

Chapter 4

Analysis of Traditional Software Libraries

In traditional software development, package managers are tools that handle installation and maintenance of software modules, including ensuring that dependencies are properly satisfied throughout the process. These package managers are responsible for very large open-source libraries which are relied upon for essentially every field of software development.

Last chapter, we focused on PyTorch and TensorFlow, two critical libraries for deep learning infrastructure. Analyzing several graph-theoretic metrics of importance on software modules, we find that across all metrics, the importance coefficients exhibit a rapid decay, indicating that most security risk is concentrated in the first few modules.

In this chapter, we turn our attention to npm (Node Package Manager), which manages JavaScript software across all aspects of front-end and back-end web development, as well as PyPI (Python Package Index), which manages Python software for a wide range of applications. We investigate similar metrics to those we used for the deep learning libraries, as well as ones specific to traditional software. This allows us to compare risks between the two realms of deep learning and traditional software, and give recommendations to developers for improving security.

4.1 Methodology and Data

As npm has over 3 million packages, and PyPI has over 600,000 packages, we are unable to directly construct the dependency graph, as querying dependencies or performing any sort of scraping at that scale requires more computational power than we have access to. Instead, we use data queried from libraries.io [10], without loss of generality; their API supports retrieving extensive information about individual packages. As the API imposes a rate limit for querying such information of 50 packages per minute, we retrieve only roughly the top 500 packages in each library under each metric. In this chapter, we consider the component size of each package in the graph, which gives us a metric of the long-term consequences of compromised software packages, within the entire library or ecosystem. For npm packages, we also examine SourceRank [11], a comprehensive software quality metric that considers various good programming practices as well as the scale to which a package is used and integrated into the broader software ecosystem.

4.2 Component Size

We investigate the top 500 component sizes over all packages in npm and PyPI, and observe the following:

Result 4A: Traditional software modules exhibit a much slower decay in component sizes than the deep learning libraries, indicating a much higher level of interdependence between modules. As a result, vulnerabilities, particularly in critical infrastructure packages, can easily spread to large swaths of the software ecosystem, making any potential security issues much more costly.

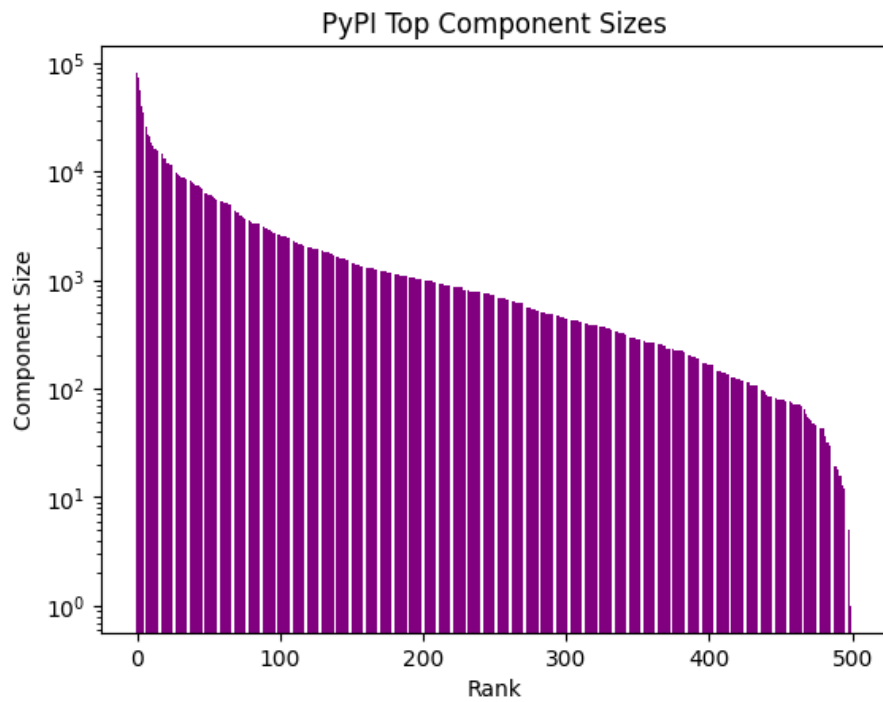
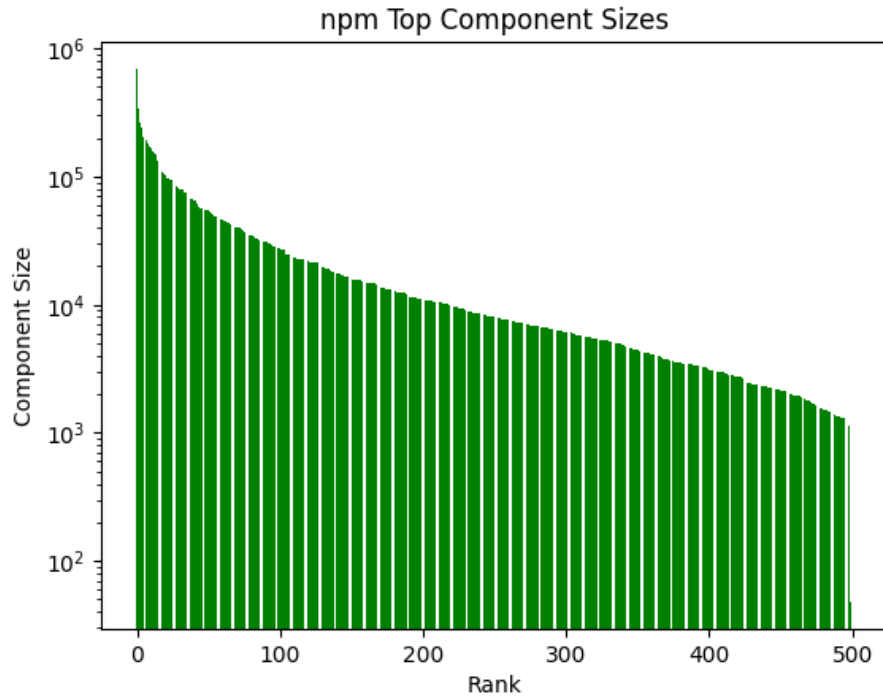


Figure 4.1: Top Component Sizes, npm and PyPI

The top 10 npm modules in terms of component size are as follows:

Index	npm module	Component Size
1	<code>eslint</code>	693429
2	<code>mocha</code>	343320
3	<code>webpack</code>	259080
4	<code>chai</code>	241242
5	<code>eslint-plugin-import</code>	205756
6	<code>rimraf</code>	205343
7	<code>babel-loader</code>	203838
8	<code>axios</code>	190036
9	<code>postcss</code>	179943
10	<code>rollup</code>	173462

Table 4.1: Top 10 Modules by Component Size, npm

These modules serve the following purposes:

- `eslint`: Most commonly used JavaScript Linter
- `mocha`: Asynchronous JavaScript test framework
- `webpack`: Bundling files for browser use
- `chai`: Assertion library for testing
- `eslint-plugin-import`: Plugins and more customizability for eslint
- `rimraf`: Support for file deletion, Windows equivalent of `rm rf` command
- `babel-loader`: Transpiling JavaScript files using webpack
- `axios`: HTTP client
- `postcss`: Automation for routine CSS operations
- `rollup`: Module bundler to combine small pieces of code into large applications

We see that all of these modules are widely used across JavaScript web development, and observe a much higher level of interdependence between modules as compared to the deep learning frameworks.

The top 10 PyPI modules in terms of component size are as follows:

Index	npm module	Component Size
1	<code>numpy</code>	81327
2	<code>requests</code>	73742
3	<code>pandas</code>	55240
4	<code>pytest</code>	39372
5	<code>matplotlib</code>	35022
6	<code>scipy</code>	30979
7	<code>pyyaml</code>	27506
8	<code>click</code>	26105
9	<code>tqdm</code>	21690
10	<code>pydantic</code>	21329

Table 4.2: Top 10 Modules by Component Size, PyPI

They are used for the following purposes:

- `numpy`: Multidimensional arrays and numerical computations
- `requests`: Sending HTTP requests
- `pandas`: Data science and data analysis, particularly on numerical tables and time series models
- `pytest`: Software testing framework
- `matplotlib`: Graphing and data visualization
- `scipy`: Scientific computation algorithms for problems like optimization, statistics, integrals, differential equations
- `pyyaml`: YAML parser
- `click`: Command-line interface framework
- `tqdm`: Progress bar for loops and iterable tasks
- `pydantic`: Data validation and formatting checker

All of the above modules are commonly used, and in fact most Python developers are likely familiar with the majority of them, so it is unsurprising that these are the most depended-upon modules in the Python index, and similarly we see much more interdependence than the deep learning frameworks.

If a module like `eslint` or `requests` were to be compromised, it would affect a significant fraction of all software; the former has tens of millions of weekly downloads [12], and the latter is essentially required for any Python software using an HTTP request, which is most web development.

Result 4B: Across both npm and PyPI, the top modules across npm and PyPI are likely to be used in most software across the ecosystems, and any of them being compromised would have wide-ranging effects.

4.3 SourceRank

Another metric used by libraries.io for quality of a package is SourceRank, which was inspired by the PageRank algorithm used for Google Search. SourceRank scores consist of a list of metrics, including the presence of readme and license documentation, proper versioning, regular maintenance, and number of dependent projects and repositories on a logarithmic scale. Under SourceRank’s metrics, the highest scores corresponding to the highest quality packages are the large packages that are widely used across much of the software ecosystem and are regularly maintained and well documented. On the other hand, low scores would be assigned to packages that are obscure or poorly maintained.

Observing the top 500 packages in npm, we have the following distribution of SourceRank scores, with mean and median around 25, and with a sharp drop-off below 20.

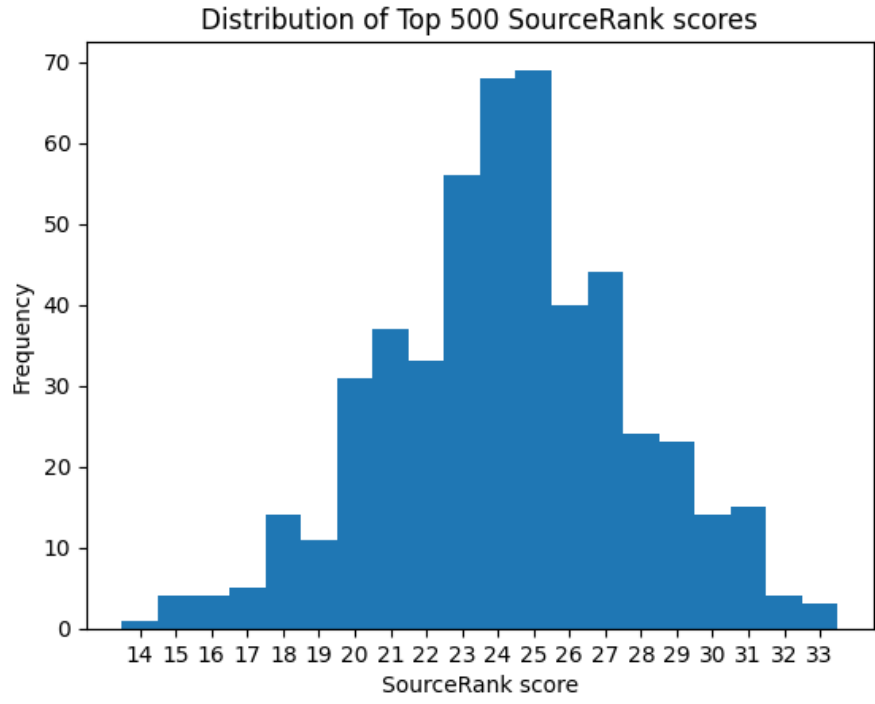


Figure 4.2: Top SourceRank scores histogram

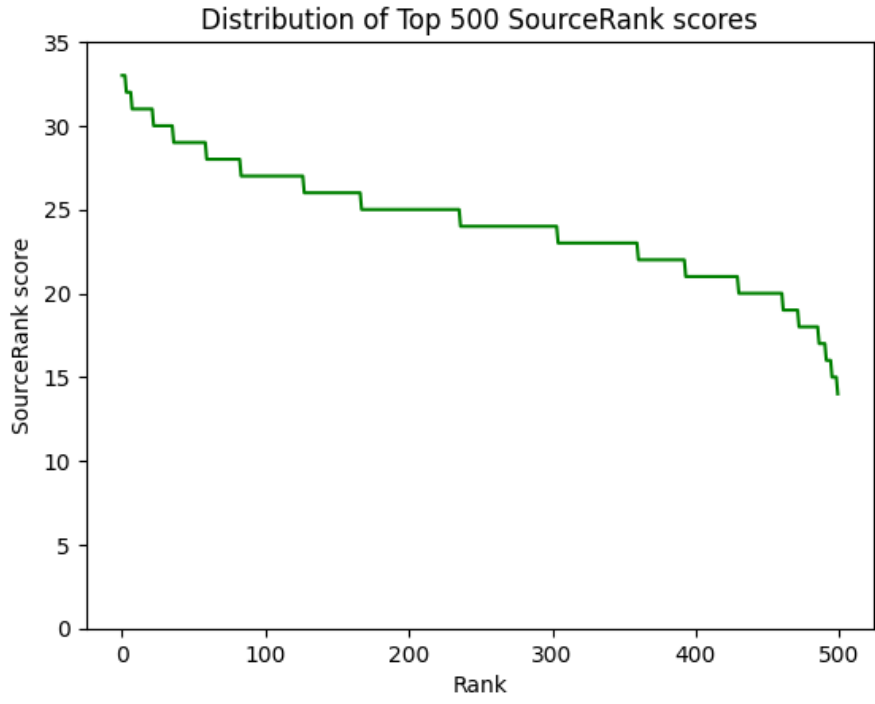


Figure 4.3: Top SourceRank scores by rank

Out of the allocations in the SourceRank criteria, there are 9 points from good codebase practices, and the remaining points come from the logarithms of dependents, contributors, github stars, and so forth; this suggests that package size seems to drop off significantly after roughly the first 400 packages. The exponential decrease in these metrics package size also corresponds to a decrease in security risk, and as a result, developers concerned about security should focus their efforts on these leading packages.

Result 4C: Most of the security risk in the npm ecosystem is concentrated within roughly the largest 400 packages.

4.4 Comparing Traditional Software and Deep Learning

Comparing the relative normalized component sizes of the four packages we have examined in PyTorch, TensorFlow, npm, and PyPI, we have the following graph:

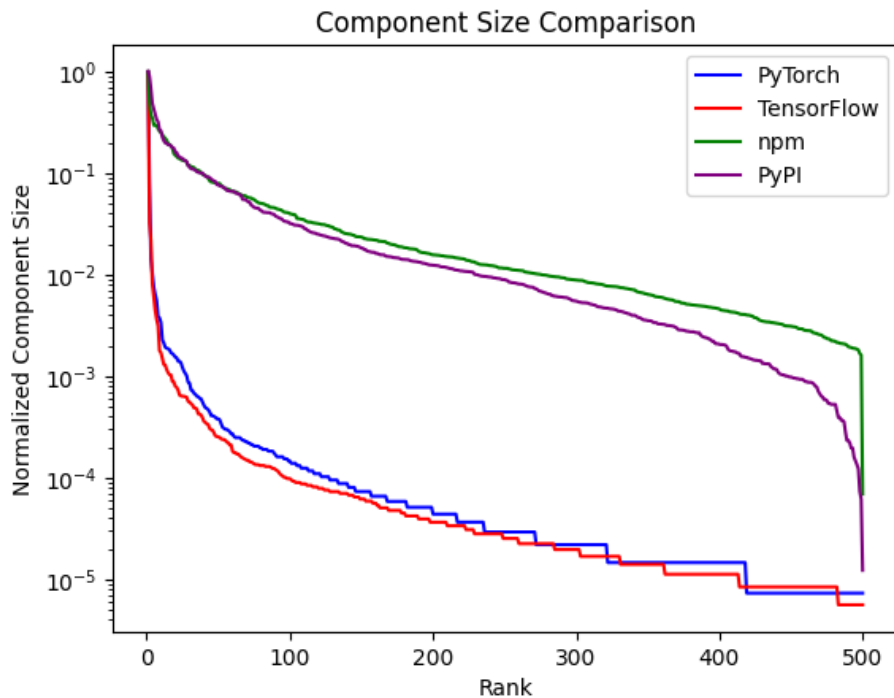


Figure 4.4: Component size comparison between PyTorch/TensorFlow and npm/PyPI

In our analysis, the deep learning software packages exhibit very different graph-theoretic properties than the traditional software packages, which motivates different security recommendations.

PyTorch and TensorFlow exhibit a treelike supply chain structure with a single overarching root node and relatively few critical nodes in the second layer. Most modules have limited reach in their direct dependents, not affecting much outside their immediate subtree. Therefore, for these deep learning modules, the best approach would be to proactively secure the most commonly used modules that present the highest risk.

Contrastingly, the networks of the npm and PyPI ecosystems are far more interdependent. The top component sizes belong to fundamental infrastructure packages used across the entirety of their software ecosystems, and it is not uncommon for projects to import many of the top packages. As a result, vulnerabilities can propagate much further and more quickly, across many layers and packages. For such frameworks in traditional software, reactive monitoring with tracking (for instance, through use of SBOMs) would be of the most benefit, motivating the recommendations we give in the next chapter.

Chapter 5

Additional Proposed Metrics, Future Works, and Recommendations

5.1 SBOM recommendations

Currently, SBOMs and AIBOMs contain mostly information about individual software components and their local dependencies, and ignore how they fit into the software supply chain as a whole. We propose augmenting SBOMs and AIBOMs with a field representing the global security risk posed by a potentially compromised module. This could be one of a few coefficients: unweighted or weighted component size, number of direct dependencies and dependents, or many others. We specifically recommend the use of two metrics: component size (number of total transitive dependents) and SourceRank score for SBOMs.

Since component size is defined as the number of modules that rely in some way on a certain module, it measures the potential propagation of vulnerabilities across the software ecosystem, and serves as a bound on the potential harm caused by such vulnerabilities. On the other hand, SourceRank’s synthesis of many criteria serves as an aggregate metric of each module’s quality, where many factors within the categories of documentation, maintenance frequency, and popularity, all contribute to the overall score. Since open-source projects

require good coding practices and maintenance standards to be safely and effectively used, SourceRank achieves a more accurate view of security risk, making it particularly useful to include in SBOMs.

For datasets in AIBOMs, we recommend a composite scoring method similar to SourceRank, as many of SourceRank’s scoring categories are still relevant to AI components. However, datasets often only have one version and don’t need to be regularly updated or maintained in the same manner that software does. Also, datasets usually aren’t part of a package manager, and the number of contributors isn’t as important, so all of these categories should be removed. Instead, with datasets, we should award points for meeting standards for data quality and integrity (by satisfying checksums or similar), passing a bias evaluation to ensure fairness, and compliance with privacy regulations (if relevant).

5.2 Conclusion

With the rapid growth of open-source software ranging from traditional Python and JavaScript applications to deep learning models, and the increasing interdependence of software modules, a security vulnerability in a single component could have cascading implications across many other modules. In this thesis, we analyze graph-theoretic metrics, considering component size (number of transitive dependencies) and product metrics for the deep learning infrastructure libraries of PyTorch and TensorFlow, as well as component size and SourceRank score for the traditional software libraries of npm and PyPI.

We observe that in PyTorch and TensorFlow, coefficients across all metrics decay exponentially in the first ten modules, indicating that a handful of modules are responsible for most of the security risk, and they should be the focus of developer attention. On the other hand, in npm and PyPI, risk is spread out across several hundred nodes, so it is best to reactively monitor and respond to issues as they arise. We also recommend populating SBOMs and AIBOMs with additional information about quality and importance of each component.

5.3 Future Works

5.3.1 PyTorch and TensorFlow

Other metrics we considered investigating were mostly various forms of centrality [13]. However, computing centrality is computationally expensive. A typical algorithm for eigenvector centrality requires $O(V^3)$ time to compute, closeness centrality requires $O(VE + V^2)$ time, and betweenness centrality requires $O(VE)$ time, all of which also have high constant factors. As the size of the graph increases, which will happen as software ecosystems grow, evaluating such metrics is infeasible given reasonable limits on computing power.

A logical extension of this research would be into machine learning *models* rather than just *infrastructure software packages* for machine learning. In the supply chain graph of models, there are two different types of nodes: models and datasets, where models can depend on other models, and on datasets. This topic was investigated by [14], who also found a relatively treelike dependency structure among Hugging Face models [14]. However, most models' use of training datasets was not documented; in fact less than 10 percent of models declared any training data at all. As a result, most edges between models and datasets in a potential supply chain graph would be missing, making it impossible to apply our metrics to Hugging Face [14].

5.3.2 npm and PyPI

For npm and PyPI, we would like to investigate the same product metrics we considered in chapter 3: direct dependents product and component size product. However, due to computational limitations we mentioned earlier, we are unable to explicitly construct the entire dependency graph. Furthermore, as the number of direct dependents is not retrievable, and querying for the entire list of dependents is disabled in the API, we cannot compute the product metrics.

We would also like to analyze past incidents of security vulnerabilities within the npm ecosystem. One possible solution would be to retrieve the list of dependents for every package involved in a known vulnerability, and all their downstream dependents, and then consider only those subgraphs in the analysis.

The packages npm and PyPI share many similarities in the results we found, but JavaScript and Python are not necessarily representative of libraries in other programming languages. A more comprehensive analysis could include, for instance, RubyGems for Ruby, Maven Central for Java, and crates.io for Rust. Each of these has their own API for retrieving dependency information, but sometimes capabilities are limited due to being very resource-intensive.

5.3.3 Weighted Graphs

So far, in the graph metrics we considered for dependency graphs, all nodes and edges were unweighted. However, this may not be a fully accurate representation of security risk, given a couple factors. Firstly, modules themselves are of different sizes. An often quoted saying is that there exists one exploitable bug per thousand lines of code; a module with a very large codebase has a much larger attack surface. Secondly, not all dependencies are equally important. Some dependencies rely heavily on core functionalities, while others may only call a few utility functions.

Therefore, we propose weighting the nodes and edges. Similarly to SourceRank’s subcriteria, nodes are weighted using the logarithm of the codebase size of the module. This allows module size and therefore security importance to be considered, without having large modules completely dominate the weighting coefficient. For edges, we weight using the logarithm of the usage frequency, i.e. for module A depending on module B, the logarithm of the number of times module A calls a function from module B.

The weighted component size for a module is then the sum of the weights of all the module nodes that depend on it, as well as all the intermediate dependency edges.

As weighted nodes and edges no longer allow for the bitmask optimization, running a full

BFS from each node requires $O(N^2)$ with a far worse constant factor, which would necessitate more computing power.

5.3.4 Stochastic Graphs

In this thesis, we used an unweighted graph representation of software dependency networks. A standard generalization to weighted graphs would still assume constant edge weights with no uncertainty. However, in reality, package dependencies vary in importance over time, especially as modules expand and are updated and refactored regularly, so we would like a way to update edge weights.

We can then construct such a graph using stochastic edge weights. Soh introduced in [15] Variational Bayesian Centrality, a framework that treats centralities and the principal eigenvalue as latent variables, where node centralities are updated using incident edges as data. He also extended the model using Gaussian processes to take into account a function mapping node attributes (e.g. node weights) to centrality coefficients. This also requires fewer data points: where a regular eigenvector centrality problem has computational cost $O(n^3)$ in the number of vertices, Soh's method requires only $O(m^3)$ where m is the size of a small subset of the vertices used in estimating an objective [15]. Applying this method to our graphs, we hope to not only be able to compute centrality coefficients, but also model the changes in the software dependency network over time.

References

- [1] M. Zimmermann et al. “Small World with High Risks: A Study of Security Threats in the npm Ecosystem”. In: *USENIX Conference on Security Symposium* (2019).
- [2] N. Zahan et al. “What are Weak Links in the npm Supply Chain?” In: *IEEE/ACM ICSE* (2022).
- [3] GitLab. *The ultimate guide to SBOMs*. <https://about.gitlab.com/blog/2022/10/25/the-ultimate-guide-to-sboms/>.
- [4] T. Bi et al. “On the Way to SBOMs: Investigating Design Issues and Solutions in Practice”. In: *ACM Transactions on Software Engineering and Methodology* (2023).
- [5] T. Stalnakier et al. “BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems”. In: *IEEE/ACM ICSE* (2024).
- [6] X. Tan et al. “An Exploratory Study of Deep Learning Supply Chain”. In: *IEEE/ACM ICSE* (2022).
- [7] CycloneDX. *Authoritative Guide to SBOM*. https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf.
- [8] SPDX. *The System Package Data Exchange*. <https://spdx.dev/>.
- [9] ToopLoox. *PyTorch vs. TensorFlow - a detailed comparison*. <https://tooploox.com/pytorch-vs-tensorflow-a-detailed-comparison>.
- [10] *Libraries.io*. <https://libraries.io/>.
- [11] *Libraries.io*. “SourceRank”. <https://libraries.io/pypi/pandas/sourcerank>.
- [12] *npm-compare*. <https://npm-compare.com/eslint/>.
- [13] F. Bloch et al. *Centrality Measures in Networks*. Social Choice and Welfare, 2021.
- [14] T. Stalnakier et al. “The ML Supply Chain in the Era of Software 2.0: Lessons Learned from Hugging Face”. In: *arXiv preprint, arXiv:2502.04484v1* (2025).
- [15] H. Soh et al. “Probabilistic Network Metrics: Variational Bayesian Network Centrality”. In: *Unknown Journal* (2015).