

Análise Reduzida de Ataques de Cadeia de Suprimento no Ecosistem pip

Bruno S. França¹

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

b217787@dac.unicamp.br

Resumo. *Este artigo apresenta uma análise das cadeias de dependência dos pacotes no repositório PyPI, utilizando um grafo para expressar as relações de dependência entre pacotes, especialmente aqueles que dependem de versões mais recentes de outros pacotes. O estudo está inserido no contexto de ataques à cadeia de suprimento, onde vulnerabilidades em pacotes amplamente utilizados podem se propagar e afetar o ecossistema de software como um todo. A partir dessa análise, busca-se identificar como a estrutura de dependências pode influenciar a segurança do ecossistema.*

1. Introdução

Atualmente, os ataques à cadeia de suprimentos representam uma das principais fontes de risco para grandes empresas em todo o mundo. Com a crescente dependência de bibliotecas e pacotes distribuídos por terceiros, as organizações se tornam cada vez mais vulneráveis a ameaças que podem comprometer a integridade de seus produtos e serviços.

Um exemplo recente que ilustra essa ameaça é o caso da biblioteca Log4j, amplamente usada em sistemas de registro de eventos de aplicações Java. Em 2021, uma vulnerabilidade crítica conhecida como Log4Shell foi descoberta, permitindo que invasores executassem código remotamente em servidores que utilizavam a biblioteca, expondo sistemas ao risco de roubo de dados e controle remoto. O impacto foi global e levou à mobilização urgente de empresas e equipes de segurança para corrigir a falha, que foi explorada em larga escala e afetou setores como o financeiro, governamental e de infraestrutura crítica. Esse incidente destacou o alcance e a seriedade dos ataques à cadeia de suprimentos, especialmente quando envolve bibliotecas amplamente utilizadas.

Diversos estudos têm investigado os impactos de ataques à cadeia de suprimentos em ecossistemas de pacotes amplamente utilizados pela indústria. Em particular, este trabalho se baseia em dois desses estudos, PDgraph [Li et al. 2021] e DVgraph [Liu et al. 2022]. O primeiro oferece uma análise abrangente dos ataques, explorando dependências em repositórios do GitHub e pacotes disponíveis no Maven, enquanto o segundo foca exclusivamente no ecossistema NPM. Ambos modelam o problema utilizando grafos de dependência entre pacotes, o que permite uma visualização mais clara do impacto das vulnerabilidades em toda a cadeia de pacotes de cada ecossistema.

Inspirados por esses estudos, o objetivo deste trabalho é analisar os impactos das vulnerabilidades no ecossistema Python Package Index (PyPi) por meio de um grafo de dependências entre pacotes. O PyPi é uma das maiores fontes de pacotes de software para a linguagem Python, amplamente utilizado por desenvolvedores ao redor do mundo. O

uso disseminado do PyPi implica que diversas entidades podem ser afetadas por vulnerabilidades presentes diretamente nos pacotes do repositório ou, de forma indireta, em suas dependências, que também são hospedadas no PyPi.

Contúdo, devido à complexidade de modelar um grafo de dependências que considere todas as versões disponíveis de cada pacote, este estudo adota uma abordagem simplificada do problema: modelar o grafo de dependências levando em conta apenas as relações na versão latest dos pacotes presentes no PyPi. Essa escolha reduz tanto a complexidade quanto o tamanho do grafo, permitindo uma análise mais direta e eficiente dos impactos de vulnerabilidades nas versões mais atuais, também amplamente utilizadas e geralmente instaladas ao iniciar um novo projeto em Python. Embora essa simplificação limite o escopo da análise e não abranja o impacto das vulnerabilidades em todas as versões dos pacotes presentes no PyPi, ela ainda possibilita respostas a questões relevantes, como os riscos associados ao uso do comando "pip install nome do pacote", cujo geralmente instala as últimas versões dos pacotes e de suas dependências presentes no repositório de pacotes. Portanto essa abordagem ainda fornece uma visão prática e relevante sobre o cenário de vulnerabilidades no PyPi.

Este estudo buscará responder a perguntas como: Quantos pacotes do PyPi precisam ter vulnerabilidades para que a maior parte da rede seja afetada? Existem pacotes que são amplamente utilizados, e como uma vulnerabilidade deles pode afetar a rede? Quais características pacotes mais suscetíveis teriam no grafo de dependências? Entre os questões que serão desenvolvidas ao longo deste artigo.

2. Construção da Base de Dados

O repositório PyPi abriga atualmente cerca de 580 mil pacotes, e além de gerenciar a disponibilização desses pacotes, a autoridade responsável pelo PyPi oferece um conjunto de metadados associados a cada um deles. Esses metadados estão disponíveis por meio de diversos canais, facilitando o acesso e a utilização das informações contidas no repositório.

Em especial, os mantenedores do PyPi fornecem uma API que permite acessar alguns desses metadados diretamente. A API pode ser acessada por meio do seguinte endpoint:

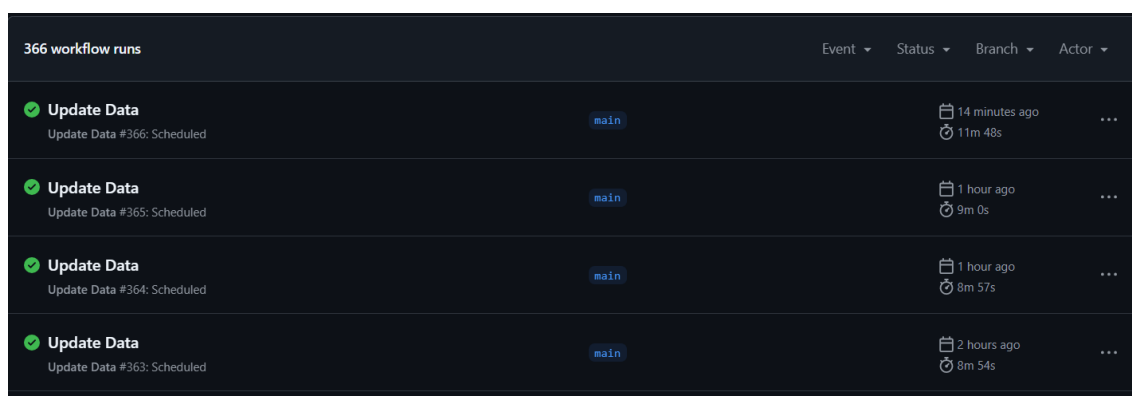
`https://pypi.org/pypi/<nome_do_pacote>/<versão_do_pacote>/json`

Esse endpoint retorna um arquivo JSON contendo informações detalhadas sobre o pacote solicitado, como a versão, dependências, descrição, autor, vulnerabilidades já descobertas em cada versão do pacote, entre outros dados. A utilização dessa API foi, portanto, fundamental para a construção da base de dados que sustenta o grafo de dependências. Em essência, ela contém todas as informações necessárias para montar o grafo de dependências que representa a cadeia de dependências dos pacotes presentes no PyPi. Além disso, os dados sobre vulnerabilidades dos pacotes, que são objetos de interesse deste trabalho, também estão incluídos, facilitando então a construção do grafo. [?]

Contudo, o PyPi não oferece uma forma de baixar todos os metadados dos pacotes de uma só vez, ou seja, não há um único arquivo ou URL que permita baixar

o conjunto completo de dados. Portanto, é necessário realizar múltiplos requests ao endpoint, variando o nome dos pacotes no URL de cada solicitação. Ao realizar essas requisições de maneira ingênua, foi atingido um limite de requests diários disponibilizado pela plataforma, que é algo em torno de 100 mil requests por dia. Logo para realizar a coleta de toda a base de dados foi necessário procurar uma maneira mais sofisticada de baixar esses dados.

Felizmente, diversas pessoas já demonstraram interesse em baixar todos os metadados do PyPi, e uma solução eficaz para atender a essa demanda está presente no repositório do GitHub `pypi-data/pypi-json-data` [json Data 2024]. Nesse repositório, foi criado um processo automatizado, semelhante a um bot, utilizando GitHub Actions para baixar periodicamente as informações dos pacotes em lotes de requisições. Ou seja, a cada determinado período de tempo, um job é disparado para baixar uma quantidade pré-definida de pacotes. Essa solução contorna o limite de requisições do PyPi, oferecendo uma abordagem eficiente e contínua, que mantém os dados dos pacotes sempre atualizados. Esta estratégia foi reproduzida no repositório deste trabalho, que fornece cerca de 20 GB de informações organizadas em arquivos JSON, contendo todos os metadados das versões de pacotes disponíveis no PyPi, de maneira estruturada em um diretório de arquivos.



The screenshot shows the GitHub Actions interface with a header indicating '366 workflow runs'. Below the header, there are four rows, each representing a workflow run. Each row starts with a green checkmark icon and the text 'Update Data'. Below this, it says 'Update Data #366: Scheduled', 'Update Data #365: Scheduled', 'Update Data #364: Scheduled', and 'Update Data #363: Scheduled' respectively. To the right of the job name, there is a blue pill-shaped button with the text 'main'. Further right, there are two clock icons: the first shows the time since the job was triggered (e.g., '14 minutes ago') and the second shows the duration of the job (e.g., '11m 48s'). On the far right of each row, there are three vertical dots indicating more options.

Event	Status	Branch	Actor
Update Data	Update Data #366: Scheduled	main	14 minutes ago 11m 48s
Update Data	Update Data #365: Scheduled	main	1 hour ago 9m 0s
Update Data	Update Data #364: Scheduled	main	1 hour ago 8m 57s
Update Data	Update Data #363: Scheduled	main	2 hours ago 8m 54s

Figure 1. Interface do GitHub Actions após 4 jobs disparados para baixar metadados de releases de 2 mil pacotes a cada 15 minutos

Essa coleção de dados abrange informações que vão além das necessidades deste projeto, podendo até auxiliar na construção de um grafo que representasse o versionamento dos pacotes. Contudo, dado o escopo deste trabalho, esses dados foram filtrados para utilizar apenas aqueles de interesse para a construção do grafo, que leva em consideração apenas as versões latest dos pacotes.

3. Definição e Construção do Grafo

A partir do dataset coletado, foi construído um grafo direcionado $G = (V, E)$, onde V representa o conjunto de pacotes presentes no repositório PyPi e E o conjunto de arestas direcionadas, que representam as dependências entre esses pacotes. Formalmente, cada nó $v \in V$ corresponde a um pacote específico no PyPi, e cada aresta direcionada $e = (u, v) \in E$ indica uma relação de dependência em que o pacote u depende do pacote v , exprimindo portanto a relação de u depende de v .

Em termos de orientação, as arestas direcionadas têm o seguinte significado:

- Se existe uma aresta $e = (u, v)$, então u possui uma dependência direta de v , ou seja, para o correto funcionamento do pacote u , o pacote v também deve ser instalado.
- As arestas de entrada em um nó v representam todos os pacotes que dependem diretamente de v .
- As arestas de saída a partir de um nó v indicam os pacotes dos quais v depende diretamente.

Para a construção deste grafo, foram consideradas apenas as dependências das versões latest de cada pacote, ignorando restrições de versionamento. Isso significa que, ao processar as dependências, qualquer entrada que incluísse símbolos de versionamento (como $>$, $<$, $=$, ou $!$) foi desconsiderada, e apenas as dependências indicadas pelo nome dos pacotes, sem restrições de versão, foram incluídas.

Por exemplo, ao processar as dependências do pacote `requests`, entradas como `urllib3 >= 1.21.1` ou `chardet != 3.0.2` foram ignoradas, enquanto uma dependência sem versão, como `idna`, foi mantida no grafo. Essa abordagem simplifica o modelo, focando nas relações diretas das versões mais recentes de cada pacote disponível no repositório. Conforme mencionado anteriormente, essa escolha visa reduzir a complexidade da análise e reflete o cenário mais comum de instalação de pacotes no PyPi. Ademais, ela facilita a construção do grafo, evitando a necessidade de lidar com a sintaxe de versionamento específica do PyPi.

Outro ponto levado em consideração durante a criação das arestas do grafo, foram as versões da linguagem Python suportadas por cada pacote. Isso é necessário porque, no momento da instalação, o PyPi verifica a compatibilidade entre as versões do Python suportada por cada dependência. Essa verificação de compatibilidade é realizada com base no campo `requires_python` dos metadados de cada pacote, que indicam quais versões da linguagem são compatíveis com o pacote específico.

Ao construir arestas apenas entre pacotes que possuem versões do Python compatíveis, evita-se relações de dependência inviáveis no grafo. Essa restrição é especialmente importante para tentar garantir que o grafo represente corretamente as dependências que poderiam ocorrer em um ambiente real, refletindo o comportamento do sistema PyPi durante a instalação. Dessa forma, os ciclos e caminhos de dependência representados no grafo levam em conta as restrições práticas que influenciam o processo de instalação.

Ademais, é importante destacar que, para o cálculo de determinadas métricas e a realização de algumas análises, foi utilizado o grafo transposto do grafo originalmente construído. No grafo transposto, as direções de todas as arestas são invertidas em relação ao grafo inicial. Dessa forma, enquanto o grafo original representa a relação de "dependência direta", o grafo transposto representa a relação inversa, ou seja, "usado por" — indicando quais pacotes são usados por outros. Esse grafo transposto permite identificar e analisar facilmente a influência, pois torna mais direto avaliar quais pacotes têm maior importância no ecossistema em função do número de dependentes diretos e indiretos que possuem por exemplo.

Para validar de forma simplória a construção do grafo, utilizou-se a API do `Libraries.io`, uma plataforma que monitora pacotes de software e suas dependências em

repositórios como PyPI, npm e Maven [Libraries.io 2024]. O Libraries.io fornece dados detalhados sobre pacotes e quantos outros dependem diretamente de um pacote específico. Essa contagem de pacotes dependentes foi utilizada como métrica para verificar se a estrutura do grafo estava sendo construída corretamente, ajudando a identificar possíveis discrepâncias ou anomalias muito grandes nas relações de dependência entre pacotes.

Para isso, foi realizada uma amostragem aleatória de pacotes do grafo gerado localmente. Para cada pacote selecionado, a quantidade de pacotes que dependem dele foi obtida do Libraries.io. Em seguida, essa contagem foi comparada com a quantidade de pacotes dependentes calculada a partir do grafo construído.

Para calcular o erro, utilizou-se a mediana dos erros absolutos de cada pacote, que corresponde à diferença entre o valor calculado no grafo e o valor obtido do Libraries.io. Essa abordagem foi escolhida para minimizar o impacto de outliers, como pacotes com um número muito elevado de dependências, que poderiam distorcer a análise. A mediana do erro foi então empregada como um indicador da precisão do grafo: uma mediana baixa sugeriria uma boa correspondência entre os dados do grafo e os dados do Libraries.io, enquanto uma mediana alta poderia sinalizar falhas ou inconsistências na construção do grafo. Ao realizar uma amostragem simples de 200 pacotes, obteve-se uma mediana do erro de zero. No entanto, como a amostra não é grande o suficiente, esse resultado deve ser interpretado com cautela. Vale ressaltar que o grafo construído considera apenas as relações latest, enquanto o Libraries.io pode levar em conta o versionamento dos pacotes, o que pode resultar em diferenças nos dados comparados. Portanto, embora o método tenha sido útil para identificar discrepâncias significativas, ele não fornece uma avaliação detalhada da qualidade do grafo. Para uma análise mais precisa e robusta, seria necessário adotar métodos mais sofisticados.

Por fim, afim de facilitar posteriormente a análise do grafo de dependências, ele foi instanciado em Python com o uso do framework networkx. Essa biblioteca é uma poderosa ferramenta para a criação, manipulação e estudo de estruturas de grafos e redes complexas. Ela oferece uma variedade de facilidades para construir grafos direcionados ou não direcionados, adicionar e remover nós e arestas, bem como realizar operações avançadas como buscas em grafos, cálculo de centralidade, detecção de comunidades, entre outras funcionalidades.

4. Metricas para a Análise do grafo

Diante do grafo de dependências criado, entende-se que, caso algum nó no subgrafo de dependências de um pacote x seja vulnerável, o pacote x torna-se indiretamente suscetível a essa vulnerabilidade. Isso ocorre porque as dependências de x podem propagar falhas ou vulnerabilidades para o próprio pacote, afetando seu funcionamento e segurança. Assim, a vulnerabilidade de pacotes que x depende podem comprometer também o pacote. A figura 2 exemplifica um grafo de dependências de pacotes. Nela, os nós vulneráveis, que são marcados em vermelho (D e E), indicam que qualquer pacote que dependa diretamente ou indiretamente desses pacotes pode ser suscetível à mesma vulnerabilidade. Por exemplo, o pacote A depende de B e C, ambos dependendo de D, que é vulnerável, logo, A também se torna vulnerável indiretamente devido à cadeia de dependências.

Diante dessa postulação, diversos algoritmos foram utilizados para entender e simular os impactos de possíveis vulnerabilidades em pacotes específicos sobre o grafo de

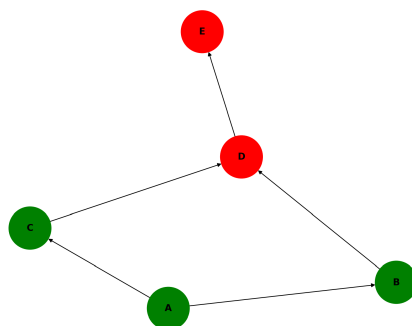


Figure 2. Exemplo de Grafo de Dependências

dependências criado, que, em tese, exemplificaria o impacto de vulnerabilidades sobre a rede de pacotes interconectados pela relação latest no ecossistema PyPi. Vale ressaltar que todos os algoritmos foram executados em um grafo no qual as componentes triviais, que não possuem arestas, foram removidas. A seguir, são descritos alguns desses algoritmos:

In-degree e Out-degree Centrality: Ambas as métricas são fundamentais para a análise de grafos de dependências e ajudam a identificar pacotes críticos dentro do ecossistema. A **In-degree Centrality** quantifica a importância de um nó com base no número de arestas direcionadas a ele, ou seja, no número de pacotes que dependem diretamente daquele pacote. Pacotes com um alto in-degree são considerados críticos, pois uma falha ou vulnerabilidade nesses pacotes pode afetar muitos outros pacotes que deles dependem. Isso é particularmente relevante quando pacotes que dependem diretamente de outros geralmente utilizam mais funcionalidades desses pacotes externos, tornando-se mais expostos caso esses pacotes possuam vulnerabilidades. Já a **Out-degree Centrality** quantifica a importância de um nó com base no número de arestas que saem dele, ou seja, no número de pacotes dos quais ele depende. Pacotes com um alto out-degree dependem de muitos outros pacotes, portanto, estão mais expostos a vulnerabilidades exteriores.

PageRank: Adaptado do algoritmo utilizado pelo Google, o PageRank mede a importância de um pacote no grafo de dependências com base no número e na qualidade das dependências que o referenciam. Ao contrário da *in-degree*, que conta apenas os pacotes que dependem diretamente de um pacote, o PageRank considera também a importância dos pacotes que fazem referência ao pacote em questão. Assim, pacotes referenciados por dependências críticas ou amplamente utilizadas têm um PageRank mais alto, mesmo que seu número de dependentes diretos seja menor. Isso destaca pacotes essenciais no ecossistema, cujas vulnerabilidades podem se propagar rapidamente entre outros pacotes.

Closeness Centrality: A métrica de closeness centrality avalia a proximidade de um pacote em relação a todos os outros pacotes da rede. Quanto menor a distância média entre um pacote e os demais pacotes, maior a sua closeness centrality. Pacotes com alta proximidade estão em posições estratégicas dentro da rede, pois têm acesso através de um caminho menor a outros pacotes, ou seja, os pacotes que dependem indiretamente dele estão mais próximos do mesmo e por consequência mais suscetíveis a possíveis vulnerabilidades que possam existir neste pacote.

5. Análises do Grafo e Simulações

As métricas propostas para a análise do grafo foram calculadas em uma versão modificada do grafo original, da qual foram removidas componentes triviais, pois estas não possuem arestas e, portanto, não contribuem para a análise das interdependências entre pacotes. O grafo resultante possui 181.225 nós e 569.458 arestas, conforme ilustrado na Figura 3, gerada com a ferramenta Cosmograph [Cosmograph 2024]. A área central em roxo da figura abrange cerca de 150 mil pacotes, enquanto aproximadamente 30 mil pacotes adicionais distribuem-se entre os que se conectam diretamente ao núcleo central e os que compõem o círculo mais externo, caracterizado por múltiplas pequenas componentes desconectadas. Essa estrutura revela uma organização predominantemente centralizada, onde muitos pacotes estão densamente interligados, com numerosas dependências, enquanto as regiões periféricas contêm pacotes com menor conectividade. Esse padrão sugere uma hierarquia, na qual pacotes situados no núcleo central exercem um papel mais crítico dentro da rede de dependências.

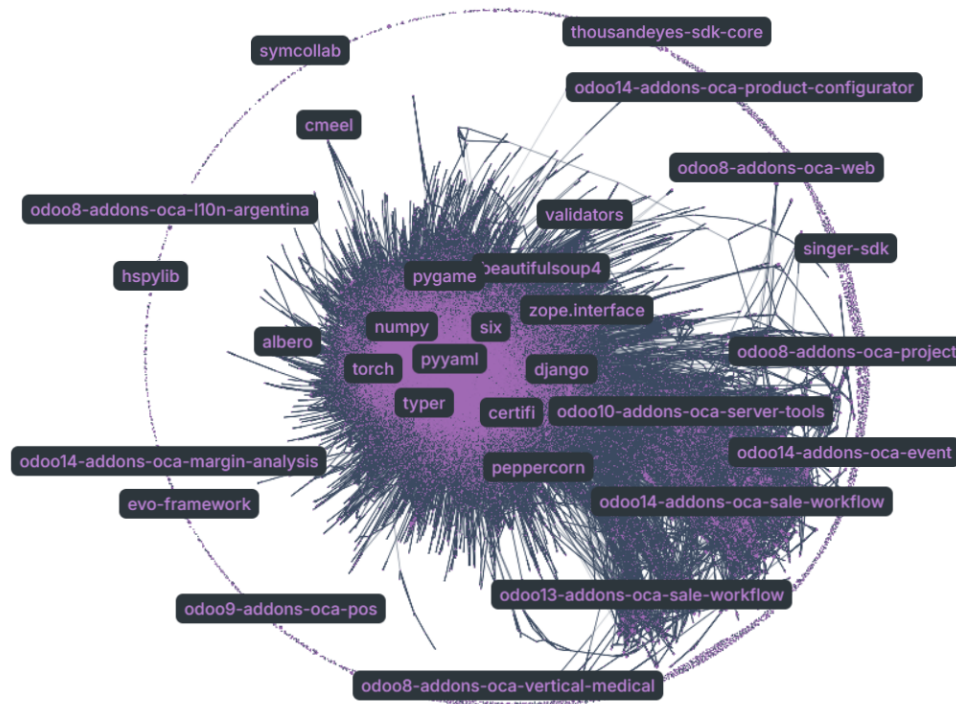


Figure 3. Grafo direcionado de dependências criado a partir do repositório PyPi. Componentes triviais foram removidos do grafo para os propósitos de análise. Este grafo contém 181.225 nós e 569.458 arestas. As labels no grafo representam os nomes de alguns pacotes presentes em determinadas regiões do grafo.

É interessante notar que essa configuração reflete a realidade do ecossistema de pacotes: pacotes amplamente conhecidos e frequentemente utilizados tendem a estar contidos na região central, o que reforça a ideia de uma hierarquia baseada em popularidade e interdependência. Na figura, é possível identificar pacotes como torch, numpy, pygame, e django entre os elementos do núcleo central. Esses pacotes possuem muitas são amplamente empregados por outros pacotes, o que justifica sua posição central no grafo e

sugere sua importância na rede de dependências do PyPi.

Já os resultados apresentados na tabela 1 fornecem uma visão dos pacotes mais influentes dentro do ecossistema PyPI, com base na métrica de PageRank. A análise dos dados revela alguns padrões interessantes sobre a distribuição das dependências e a importância dos pacotes selecionados na rede.

Primeiramente, ao comparar as métricas de PageRank, grau de entrada e grau de saída, observamos que, em geral, pacotes com maiores valores de PageRank possuem também altos graus de entrada, como é o caso de requests, numpy, pandas e matplotlib. Essa correlação sugere que pacotes com alta influência na rede de dependências tendem a ser aqueles que são amplamente utilizados por outros pacotes. O PageRank reflete não só a quantidade de dependências que um pacote possui, mas também a importância dessas dependências no grafo. Ou seja, pacotes com altos valores de PageRank são provavelmente dependências de outros pacotes igualmente influentes, o que amplifica seu impacto na rede. Isso explica o motivo pelo qual, por exemplo, o pacote requests está acima do numpy, mesmo que o numpy impacte diretamente mais pacotes.

Table 1. Dados de dependências dos Top-10 pacotes ordenados pelo Page Rank do grafo de dependências PyPI

Nome	Page Rank	G de saída	G de Entrada	P. Alcançados	% Rede
requests	0.0375	0	29296	37089	20.47%
numpy	0.0331	0	33228	41704	23.01%
pandas	0.0159	0	21291	23521	12.98%
matplotlib	0.0088	0	13847	18454	10.18%
typing-extensions	0.0081	0	1695	14789	8.16%
scipy	0.0080	0	12871	15696	8.66%
pyyaml	0.0080	0	8774	13467	7.43%
click	0.0073	0	7548	10278	5.67%
setuptools	0.0070	0	4292	10877	6.00%
beautifulsoup4	0.0063	1	6077	10085	5.56%

Além disso, a diferença nos graus de saída (número de pacotes dos quais o pacote depende) para os graus de entradas é extremamente expressiva. Os pacotes disponibilizados na tabela basicamente não possuem dependências de outros Pacotes do PyPi. Isso pode sugerir que são pacotes muito estáveis e autossuficientes, sendo essenciais para muitos outros pacotes, mas não necessitando de muitas bibliotecas externas para funcionar.

Ao analisar a métrica % Rede, observamos que pacotes como numpy (23.01%) e requests (20.47%) têm um grande impacto no ecossistema PyPI. O numpy é fundamental para computação científica e matemática, sendo amplamente utilizado em projetos que envolvem análise de dados e aprendizado de máquina. Já o requests facilita a interação com APIs e a comunicação HTTP, o que o torna indispensável para muitos projetos. A ampla adoção desses pacotes em diversos tipos de projetos é o principal fator para seu alto grau de impacto na rede de dependências PyPI.

Diante da influência significativa dos projetos listados na Tabela 1, surge a seguinte questão: qual seria o impacto de vulnerabilidades nesse subconjunto de pacotes sobre toda a rede? Em outras palavras, quantos pacotes desse grupo central precisariam

apresentar vulnerabilidades para que a rede como um todo (ou grande parte dela) fosse afetada?

Ao analisar a Figura 4, conseguimos simular uma resposta para a pergunta proposta. Essa figura apresenta o impacto dos Top-k pacotes de acordo com diferentes métricas de centralidade sobre o grafo de dependências, com k variando de 50 a 1000 pacotes. É notável que, ao considerar os 1000 primeiros pacotes segundo a métrica de PageRank — representando aproximadamente 0,17% dos nós do grafo original — mais de 80% da rede é impactada. Além disso, é importante ressaltar que, ao simular o impacto na rede dos 10 primeiros pacotes presentes na Tabela 1, cerca de 50% da rede já é impactada, um noção gráfica deste último levantamento pode ser encontrado na figura 5

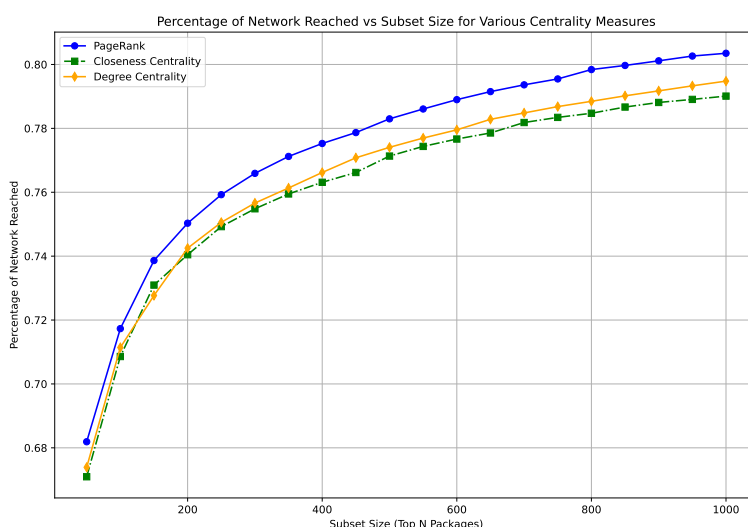


Figure 4. Impacto dos Top-k pacotes na rede de acordo com diferentes métricas de centralidade. Neste gráfico, considerou-se as métricas de In-Degree Centrality, PageRank e Closeness Centrality. O eixo y mostra a porcentagem da rede de pacotes impactados, enquanto o eixo x apresenta os Top-k pacotes segundo cada medida.

Ao analisar as outras métricas da Figura 4, também podemos observar a influência dos pacotes de maior grau sobre o grafo, de modo que os 1000 primeiros nós, classificados de acordo com essa métrica, apresentam um impacto próximo ao observado pelo PageRank. Isso era esperado, pois quanto maior o grau de entrada dos vértices, mais pacotes dependem deles.

Além disso, apesar da métrica de closeness centrality apresentar um número menor de pacotes impactados, é importante ressaltar que essa métrica indica a proximidade dos pacotes dentro da rede. Ou seja, pacotes que estão mais próximos de uma dependência vulnerável tendem a utilizar mais funções dessa dependência. Assim, maior é a probabilidade de utilizarem componentes vulneráveis desses pacotes.

Por fim, quando pensamos em pacotes mais suscetíveis a vulnerabilidades, geralmente estamos nos referindo àqueles que dependem de outros pacotes de forma extensiva dentro da rede. Isso ocorre porque, ao ter muitas dependências, esses pacotes se tornam

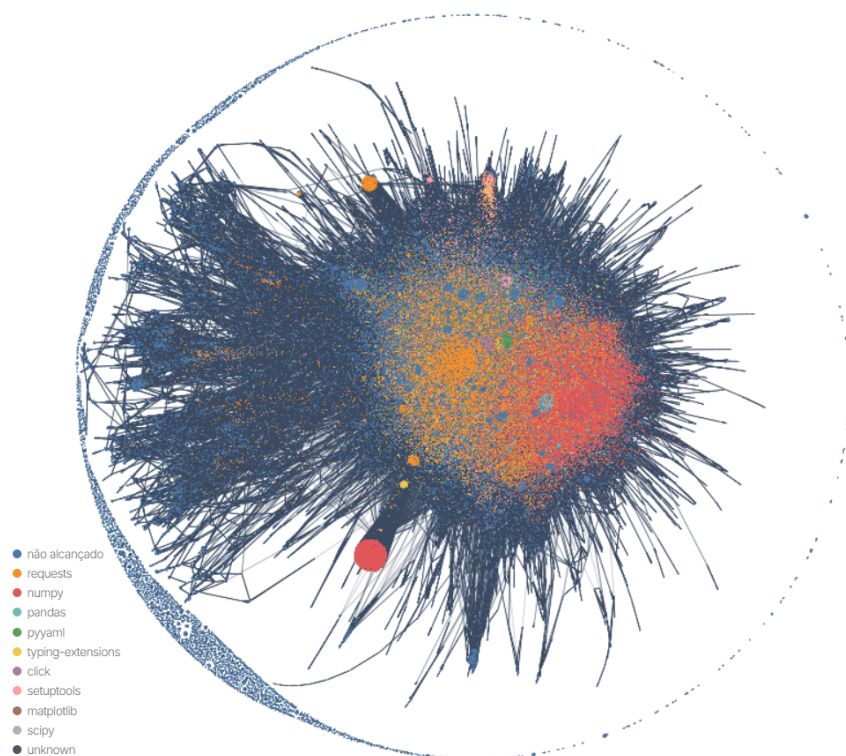


Figure 5. Grafo direcionado de dependências construído a partir do repositório PyPI, com o propósito de ilustrar o impacto dos 10 pacotes mais influentes, classificados pelo PageRank, sobre a rede. Componentes triviais foram removidos para fins de análise. Este grafo contém 181.225 nós, dos quais cerca de 50% são impactados pelo top-10 pacotes, e 569.458 arestas. As cores indicam o alcance dos pacotes dentro do subconjunto top-10 do PageRank.

mais expostos a vulnerabilidades presentes em pacotes de terceiros. Quando um pacote é atualizado com uma correção de segurança, outros pacotes que dependem dele também podem ser afetados.

Para entender como esses pacotes se comportam dentro do grafo criado, podemos usar as informações da Tabela 2, nas quais esses pacotes se caracterizam por uma combinação de um grande número de graus de saída e uma grande profundidade em sua árvore de dependências. Como exemplo, podemos observar os pacotes `brasil.gov.portal` e `imio.events.policy`. Para comparação, o grau de saída médio de cada nó é de cerca de 3. Podemos, então, alegar que pessoas que confiam no pacote `superpilot`, por exemplo, correm bem mais riscos do que pessoas que confiam em pacotes como `numpy` ou `requests`, os quais não apresentam muitas dependências.

6. Vulnerabilidades Reais

O grafo construído mantém, entre seus atributos, as vulnerabilidades que afetavam os pacotes no momento da criação do dataset. Algumas dessas vulnerabilidades podem ser vistas na Tabela 3. Com base nela, podemos concluir que, dado o baixo número de nós contaminados, as versões atuais dos pacotes do PyPI que dependem da versão mais recente de outros pacotes estão relativamente protegidas. Para assegurar que esses números de dependências estejam próximos da realidade, utilizou-se novamente o Libraries.IO,

Table 2. Informações do top-10 pacotes com maior grau de saída do grafo. Esse são possivelmente os pacotes mais suscetíveis a vulnerabilidades da rede

Node	N de dependências	G de Saída	G de Entrada	Profundidade
superpilot	368	359	0	4
cpskin.demo	278	5	0	27
brasil.gov.portal	277	71	1	27
brasil.gov.temas	277	4	1	28
cpskin.policy	273	39	1	26
ideabox.policy	253	27	0	26
design.plone.iocittadino	238	4	0	28
imio.directory.policy	237	15	0	23
imio.events.policy	234	15	0	24
design.plone.ioprenoto	234	4	0	28

cujos valores também estão presente na tabela. Nota-se que o Libraries.IO reporta mais pacotes dependentes daqueles que possuem vulnerabilidade isso é esperado, dado que o site possivelmente olha para questões de versionamentos dos pacotes. Mesmo diante dessa um número mais elevado de pacotes de dependências, pode-se supor que esse subconjunto de pacotes vulneráveis não é capaz de atingir a rede pacote total do PyPi que possivelmente supera os 180 mil nós analisados neste trabalho, ou seja, essas vulnerabilidades ao menos são bem contidas.

Table 3. Impacto de vulnerabilidades reais sobre o ecossistema de pacotes

Pacote	Pacotes Dependentes	Libraries.IO	CVE ID
python-jose	170	536	CVE-2024-33664
django-rest-framework-simplejwt	37	159	CVE-2024-22513
flair	19	87	CVE-2024-10073
xhtml2pdf	14	73	CVE-2024-25885
sentry	7	83	CVE-2024-41656
text-generation	6	24	CVE-2024-3924
dtale	2	28	CVE-2024-3408
pyassimp	1	5	CVE-2024-48423
mezzanine	1	13	CVE-2024-25170
ryu	1	11	CVE-2024-28732
confidant	0	0	CVE-2024-45793
case-utils	0	1	CVE-2024-22194
chuanhuchatgpt	0	0	CVE-2024-6035
apache-submarine	0	1	CVE-2024-36264
ait-core	0	2	CVE-2024-35059
frigate	0	0	CVE-2024-32874
temporal	0	0	CVE-2024-0936
lollms	0	1	CVE-2024-3121

7. Conclusão

Diante dos resultados apresentados, podemos concluir que o grafo construído representa, de forma aproximada, a realidade dos pacotes do ecossistema PyPI que dependem de outros em suas versões mais recentes. Observa-se que um conjunto de apenas 10 bibliotecas é capaz de afetar mais de 50% dos pacotes analisados, o que corresponderia a cerca de 90 mil pacotes em um total aproximado de 580 mil pacotes presentes no PyPI, ou seja, aproximadamente 18% do total. Vale ressaltar que esse número pode estar subestimado, devido às restrições adotadas na construção do grafo e ao fato de que o versionamento dos pacotes não foi considerado.

No entanto, esses 10 pacotes críticos, cuja influência foi destacada na análise através do uso do algoritmo PageRank, são mantidos por equipes altamente qualificadas e recebem suporte de grandes empresas, como é o caso do Numpy. Sendo assim, a probabilidade de surgirem vulnerabilidades nesses pacotes é menor, embora o impacto de uma eventual vulnerabilidade possa ser significativo, dado o número de dependências que eles concentram.

Além disso, observou-se que as vulnerabilidades atualmente presentes no grafo de dependências e, conseqüentemente, no PyPI, têm um impacto limitado sobre o subconjunto do ecossistema analisado neste trabalho. Dessa forma, pode-se afirmar que, para este conjunto de pacotes, o PyPI não apresenta grandes riscos aos seus usuários.

References

- Cosmograph (2024). How to use cosmograph. Accessed: 2024-11-11.
- json Data, P. (2024). Pypi data repositories. Accessed: 2024-11-11.
- Li, Q., Song, J., Tan, D., Wang, H., and Liu, J. (2021). Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173.
- Libraries.io (2024). Libraries.io. Accessed: 2024-11-11.
- Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., and Peng, X. (2022). Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 672–684, New York, NY, USA. Association for Computing Machinery.