MODELING RISK IN THE FRONT-END OF THE OSS DEBIAN SUPPLY-CHAIN USING MODELS OF NETWORK PROPAGATION

by

Sahithi Kasim

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Polytechnic Institute

West Lafayette, Indiana

June 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL STATEMENT OF COMMITTEE APPROVAL

To my parents, Dr. Kasim Jagadish Kumar and Kasim Bharathi, my sister Kasim Ananya, and my friends.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my committee members for their invaluable guidance, support, and encouragement throughout this research journey. Their insightful feedback and expertise were instrumental in shaping this thesis and in honing my research skills. Each of their contributions has significantly impacted the quality and depth of this work.

I am also profoundly grateful to all my professors at Purdue University. Their dedication to teaching and their willingness to share their vast knowledge have been crucial to my academic development. Through every class, they have inspired me to explore new ideas and concepts, pushing me to expand my understanding and capabilities. Their support has played a vital role in my growth as both a scholar and an individual.

Finally, heartfelt thanks to my family and friends; you have been my anchor throughout this challenging yet rewarding journey. Your patience, understanding, and continuous motivation have been invaluable. This accomplishment would not have been possible without each of you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Our research revolves around the evolving landscape of Open-Source Software (OSS) supply chains, emphasizing their critical role in contemporary software development while investigating the escalating security concerns associated with their integration. As OSS continues to shape the software ecosystem, our research acknowledges the paradigm shift in the software supply chain, highlighting its complexity and the associated security challenges. Focusing on Debian packages, we employ advanced network science methods to comprehensively assess the structural dynamics and vulnerabilities within the OSS supply chain. The study is motivated by the imperative to understand, model, and mitigate security risks from interconnected software components.

Our research questions delve into 1) identifying high-risk packages 2) comparing risk profiles between source and build stages and 3) predicting future vulnerabilities. Data collection involves collecting source code repositories, build-info information, and vulnerability data of Debian packages. Leveraging a multifaceted methodology, we perform the following things: graph construction, subsampling, metrics creation, explorative data analysis, and statistical investigations on the Debian package network. This statistical approach integrates the Wilcoxon test, Chi-Square test, and advanced network dynamics modeling with machine learning, to explore evolving trends and correlations between different stages of the OSS supply chain.

Our goals include providing actionable insights for industry practitioners, policymakers, and developers to enhance risk management in the OSS supply chain. The expected outcomes encompass an enriched understanding of vulnerability propagation, the identification of high-risk packages, and the comparison of network-based risk metrics against traditional software engineering measures. Ultimately, our research contributes to the ongoing discourse on securing open-source ecosystems, offering practical strategies for risk mitigation and fostering a safer and more resilient OSS supply chain.

1. INTRODUCTION

Open-source software (OSS) represents a paradigm shift in software development and distribution. Under OSS licensing, users can use, examine, modify, and distribute software and its source code without constraints (Corbly, 2014; St Laurent, 2008). OSS thrives on collaboration, with a community of developers and users working collectively to enhance software functionality. This ecosystem's open-source packages, easily accessible through platforms like GitHub and npm, serve as fundamental building blocks for contemporary software development.

1.1 The Complexity of the Software Supply Chain

However, this paradigm shift has brought about a revolution not only in innovation but also in the complexity of the software supply chain. It significantly influences four key phases of the OSS supply chain: source code development, building & packaging, re-configuration & re-packaging (e.g., containerizing), and deployment. In today's landscape, businesses of all sizes, from agile startups to industry behemoths such as Microsoft, IBM, Google, and RedHat, seamlessly integrate OSS products into their operations, thereby capitalizing on the benefits of open collaboration ("OSSRA Report", 2023).

To fully comprehend the significance of these transformations, it is essential to consider the historical context. The roots of OSS trace back to the collaborative endeavors of early computer programmers, which laid the foundation for the open-source ethos (Fitzgerald, 2006). Over time, movements like the Free Software Foundation and the Open Source Initiative have formalized the principles underpinning open source. These ideals have fostered innovation, resulting in software that underpins a substantial portion of our digital world. However, this historical backdrop underscores the potential and the responsibilities intrinsic to the OSS paradigm.

1.2 The Growing Security Concerns

Despite the benefits, integrating open-source software has introduced security challenges. Software supply chain attacks, exploiting vulnerabilities within software components, have emerged as a potent threat. Vulnerabilities, defined as weaknesses or flaws in software systems and artifacts, can be exploited by attackers to gain unauthorized access (Editor, 2022). Breaches, whether due to insider threats, negligence, or malicious external actors, can compromise data confidentiality, integrity, and availability (Jang-Jaccard & Nepal, 2014).

The interconnected nature of the OSS supply chain amplifies the impact of vulnerabilities. Recent reports, such as the Symantec Internet Security Threat Reports (Inc, 2023), highlight a significant increase in compromised software products. The SolarWinds Hack, a sophisticated supply chain attack executed by suspected Russian hackers, underscores the urgency of prioritizing supply chain security (Constantin, 2020; Jena, 2023). In response to these incidents, research, and modeling of security risks within OSS supply chains become imperative. The growing complexity and interconnectedness of these chains make them susceptible to evolving cyber threats, requiring proactive measures and continuous vigilance.

Current risk assessment methods often focus on package metadata (Stuckman & Purtilo, 2014) and dependency approaches that prioritize direct dependencies in their evaluation (Ponta et al., 2020). However, the build stage of the OSS supply chain, a critical phase, is not frequently addressed in existing dependency analyses. Vigilant management, continuous monitoring, and effective mitigation strategies for open-source software and packages are imperative (Duan et al., 2020; Elizalde Zapata et al., 2018). This comprehensive approach acknowledges the dynamic nature of risk within the OSS supply chain, emphasizing the importance of actively addressing and mitigating potential threats to ensure the security and integrity of software ecosystems.

1.3 Network View of OSS Supply Chain Dependencies

The primary goal of our research is to analyze and represent the structure of dependencies within the open-source software (OSS) supply chain. The accompanying figure 1.1 illustrates the network of dependencies in the OSS supply chain, highlighting the relationships between source packages and their dependencies.

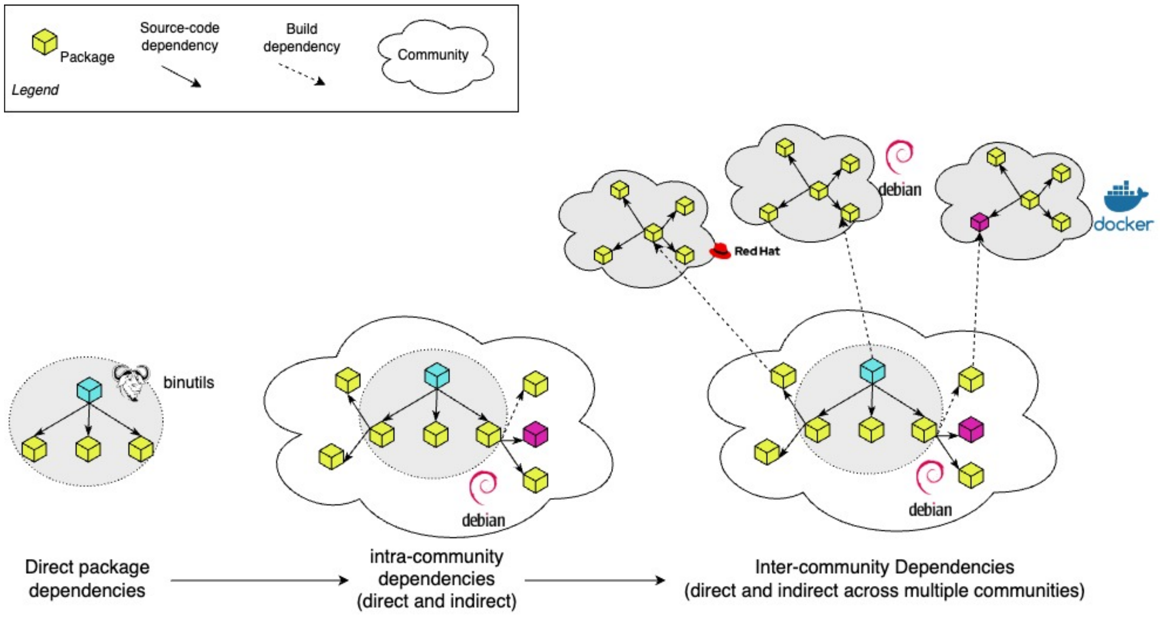
Package
Source-code dependency
Build dependency
Community
Legend
binutils
debian
Red Hat
docker
Direct package dependencies
intra-community dependencies (direct and indirect)
Inter-community Dependencies (direct and indirect across multiple communities)

Figure 1.1. OSS Supply Chain Ecosystem as Network of Interdependent Packages

The network includes source packages, which collect source code, configuration files, and necessary resources constituting a software application or component before it is compiled or built into its executable form. These source packages serve as the foundation for creating executable binaries or installable packages. Additionally, there are dependency packages, representing external libraries, frameworks, or modules required for a software project to build and run successfully.

This network encompasses intra-community dependencies within the same community and inter-community dependencies across different communities. It reveals various dependency types throughout the supply chain, such as source code dependencies (direct dependencies) and build dependencies (indirect dependencies). For example, a source package within the Debian ecosystem may directly depend on another package within Debian and further extend its dependency to involve another package within the Debian ecosystem. It could also depend on packages from external ecosystems, such as Docker. This network reveals hidden direct and indirect dependencies crucial for understanding supply-chain security risks (Brunswicker, 2022).

1.4 Theoretical Framework

In the theoretical framework of our research, we employ concepts from network science and software engineering to analyze and understand the structure of dependencies within the open-source software (OSS) supply chain. The following key terms and concepts form the foundation of our theoretical framework (figure 1.2):

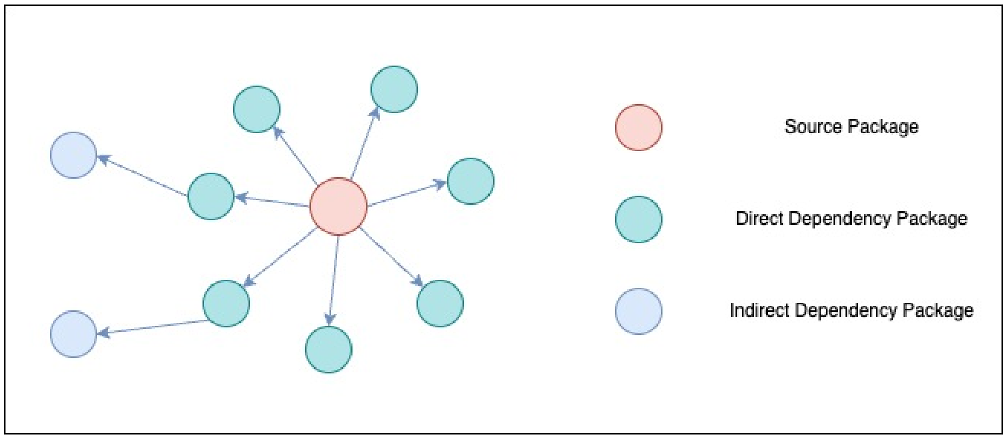

Figure 1.2. A Sample Network

- **Network:** A network is a collection of interconnected elements (nodes) and the relationships between them (edges). Networks model complex systems, allowing us to assess dependencies, vulnerabilities, and potential risks within interconnected elements (Bender, 2010).

- **Node:** A node represents an entity within the graph, such as a software package in our context. Nodes can have attributes or properties associated with them, and understanding these attributes is crucial for assessing their role and potential impact (Bender, 2010).

- **Edge:** An edge is a connection or link between two nodes in a graph, representing relationships or interactions between entities. Analyzing edges helps reveal dependencies and interactions contributing to a system's overall structure and functionality (Bender, 2010).

- **Direct Dependency:** This term characterizes a relationship where one software package explicitly relies on the features or resources of another (Saleh et al., 2018).
- **Indirect Dependency:** Refers to a scenario where a software package relies on another, not through a direct link but through one or more intermediary packages (Saleh et al., 2018).
- **Network Propagation:** The spread or transmission of effects, changes, vulnerabilities, or risks within a network (Saleh et al., 2018).
- **Risk (in Software Engineering):** Defined as the potential probability of exploiting vulnerabilities in an asset or group of assets, posing a specific threat to the organization (ISO, 2022).
- **Risk (in Network Science):** The degree of influence of a node in the network. Nodes with higher influence may play a critical role in the network's functionality, and risks associated with such nodes may have a more significant impact (Lawyer, 2015; Rogers, 2015).
- **Risk Profile/Propagation Costs:** Measures the risks associated with the influence/impact of changes of a package on other packages within the software supply chain network (MacCormack et al., 2006).
    - **Degree:** The number of connections a node has in the network, indicating its level of connectivity (Bang, 2008).
        * **In-degree:** Refers to the number of incoming connections or dependencies that a node has. Higher in-degree may indicate that a particular component is heavily depended upon by other components, making it critical (Bang, 2008).

    * **Out-degree:** Signifies the number of outgoing connections or dependencies from a particular node. A higher out-degree may suggest that a component relies on multiple other components, potentially increasing the surface area for vulnerabilities (Bang, 2008).

- **Coreness:** Quantifies the level of connectedness of a node to other nodes in the network (Lü et al., 2016).
    * **Core:** A subset of software packages or components that are highly interconnected with other packages. Cores play a central role in the overall functionality, and risks or vulnerabilities in core components may have a more significant impact on the system (Baldwin et al., 2014).
    * **Periphery:** Encompasses software packages or components that are less interconnected with other packages and may have fewer dependencies. They are less critical to the system's core functionality, but risks or vulnerabilities in the periphery may have a more localized impact (Baldwin et al., 2014).
    * **K-Core:** A subgraph where each node has at least k neighbors. It measures the connectedness and cohesion within the network. Nodes with higher core values have a higher level of centrality and influence, and risks associated with these high-core nodes may have a more significant impact on the overall stability and functionality of the network (Kong et al., 2019).
- **Path Length:** The number of intermediate steps or dependencies required for a specific change, vulnerability, or information to traverse from one software package to another in the network ("Paths, Diffusion, and Navigation", 2016).
- **Betweenness Centrality:** A metric that measures the extent to which a node serves as a bridge connecting other nodes in the network, indicating its importance in facilitating information flow (Borgatti & Everett, 2006; Brandes, 2001).
- **Attack Radius:** The size or influence of a specific component within the network, providing insights into its impact and reach (Corneil et al., 2003).

This theoretical framework enables us to assess and analyze the complex dependencies and risks within the OSS supply chain, providing a foundation for our research methodology and analysis.

1.5 Research Questions and Framework Implications

1.5.1 Research Questions Justification

To guide our investigation effectively, we have formulated three key research questions:

RQ1: *High-Risk Packages in Source and Build Stages* - What are the high-risk packages created in the source and build stages when considering OSS supply-chain risk from a network perspective?

This question is pivotal as it directs our focus to understand the specific stages where high-risk packages may have pronounced impacts on the overall supply chain security.

RQ2: *Correlating Risk Profiles in Source and Build Stages* - How does the risk profile in the source stage correlate with the risk profile in the build stage when assessing OSS supply-chain risk?

This question aims to establish connections between the risk profiles at different stages, providing insights into the consistency or divergence of risk propagation patterns.

RQ3: *Predicting Vulnerable Packages* - How well does a network-based risk profile of a package (source and build stage) predict a packages vulnerability?

This question seeks to evaluate the predictive power of network-based risk profiles in identifying vulnerable packages, thereby aiding in proactive risk management.

1.5.2 Implications of the Theoretical Framework

The theoretical framework we employ serves as a compass for our research, navigating the intricate landscape of OSS supply chain security. Not confined to observation, this framework empowers us to actively mitigate risks. By incorporating network theory, software engineering principles, and statistical modeling, we enhance the depth and scope of our investigation, allowing for a nuanced examination of the complex interplay of factors affecting OSS supply chain security.

Moreover, this framework lays the groundwork for the development of risk mitigation strategies. By identifying potential weak points within the network, we can formulate proactive security measures tailored to specific stages of the supply chain. This interdisciplinary approach enriches our research with diverse perspectives and methodologies, allowing for a comprehensive understanding of the complex dynamics influencing OSS supply chain security.

1.6 Contributions of Our Research

In the following sections, our research seeks to address significant gaps in understanding and provide actionable insights for industry practitioners and policymakers. We explore the practical scope of implementing enhanced supply chain security practices, benefiting organizations and users relying on OSS products. Our contributions extend to:

- Enhanced Risk Management: Our research empowers organizations to proactively manage risks in the OSS supply chain by identifying high-risk packages and understanding their propagation dynamics.
- Practical Security Strategies: We provide actionable strategies to address vulnerabilities at various supply chain stages, improving overall security.
- Policy and Industry Impact: Our findings have the potential to influence industry practices and policies, making OSS products safer for everyone.
- Holistic Understanding: We offer a holistic understanding of how vulnerabilities propagate within OSS supply chains, helping stakeholders make informed decisions.

In summary, our research expands knowledge and offers practical contributions impacting OSS supply chain security. Thorough analysis during the source and build stages provides insights into network propagation mechanisms influencing these risks, bridging existing knowledge gaps and deepening comprehension of vulnerabilities in OSS ecosystems.

2. OSS SUPPLY CHAIN

The Open Source Software (OSS) supply chain is a multifaceted and dynamic process comprising four essential phases: source code development, building & packaging, re-configuration & re-packaging, and deployment. These phases collectively contribute to the overall integrity, compliance, and security of software within the open-source ecosystem. This research paper aims to provide a comprehensive exploration of the OSS supply chain life cycle, offering insights into the processes that shape the development, distribution, and maintenance of open-source software.

Source Code Development: The initial phase of the OSS supply chain involves source code development, where contributors within the open source community collaborate to author and refine code. This collaborative approach fosters innovation, transparency, and shared knowledge, laying the groundwork for subsequent phases in the supply chain.

Building & Packaging: The "Building and packaging" phase plays a pivotal role in transforming source code into executable software packages. This process involves compiling the code and organizing it into distributable binary packages. Key stages within this phase, such as "Debianize" and "Build," mirror the Debian software lifecycle, ensuring adherence to rigorous standards.

Re-Configuration & Re-Packaging: The subsequent phase involves re-configuring and re-packaging software, often through containerization, to enhance portability and deployment flexibility. This stage addresses the evolving landscape of software distribution and deployment strategies, reflecting the adaptability inherent in the OSS supply chain.

Deployment: The final phase encompasses the deployment of software to end-users. Users install packages using package managers, initiating a feedback loop for bug identification and reporting. This feedback loop leads to issue tracking and resolution by developers, contributing to the continuous improvement of the software.

2.1 Debian Supply Chain

The Debian supply chain life cycle is a carefully orchestrated process as shown in figure 2.1 ensuring the acquisition, modification, and distribution of software following the stringent policies and standards set forth by the Debian project. Commencing with retrieving code from upstream open-source projects, the Debian life cycle comprises distinct stages, each serving a crucial role in guaranteeing the integrity, compliance, and security of the software propagated through Debian repositories (Torres-Arias, 2024).

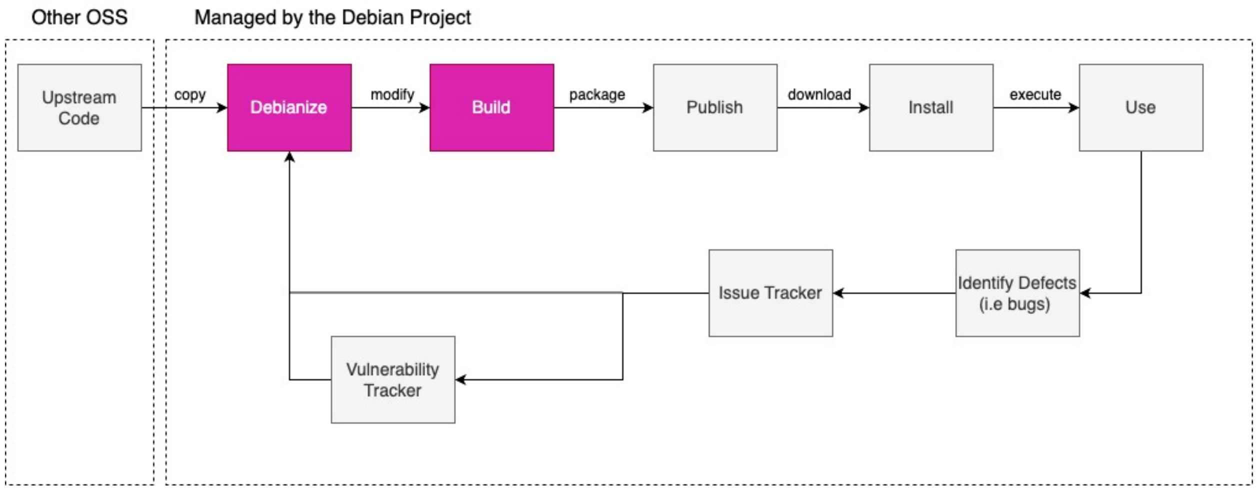

Figure 2.1. The Debian Supply Chain (Life Cycle)

Obtaining Upstream Code: Debian's software journey initiates with the acquisition of code from upstream sources, predominantly other open-source projects. This forms the foundational building block for subsequent stages in the Debian life cycle.

Debianize: The "Debianize" phase marks the first transformative step in the life cycle, wherein the obtained code is meticulously modified to align with Debian's packaging policies and standards. This process involves the addition of metadata and packaging scripts, ensuring uniformity and compliance with Debian's packaging guidelines. The Debianize stage establishes a crucial bridge between upstream code and the tailored packaging required for Debian distribution.

Build: Following Debianization, the code undergoes the "Build" phase, transforming it into binary packages ready for distribution. The build process is characterized by comprehensive buildinfo data, encompassing vital information such as architecture, dependencies, build path, environment variables, and checksums. This data ensures reproducible builds, and consistency across diverse systems, and serves as a critical resource for troubleshooting build issues.

Publish: After the Build phase, binary packages are published to the Debian repositories, forming a global network mirrored across servers worldwide, facilitating widespread access to Debian software.

Install and Use: Users can then install Debian packages using the package manager, which resolves dependencies and downloads the necessary packages. The installed software is subsequently utilized by end-users for their intended purposes.

Identify Bugs and Issue Tracking: In the event of bugs being identified during use, users can report them to Debian's bug tracking system. Developers employ the issue tracker to monitor and resolve reported bugs and other issues, contributing to Debian software's continuous improvement and refinement.

Vulnerability Tracking: Debian maintains a dedicated vulnerability tracker, a vital component in tracking and resolving security issues in software packages. This proactive approach underscores Debian's commitment to ensuring the security of its software throughout its life cycle.

The Debianization process is iterative, restarting if upstream code undergoes changes or issues are identified and rectified. This iterative approach ensures the continuous enhancement and alignment of Debianized code with evolving standards, reflecting Debian's dedication to delivering high-quality, secure, and well-maintained software to its user base.

3. RELATED WORK

The growth of open-source software (OSS) has been remarkable, driven by factors like cost-effectiveness, flexibility, and collaboration (Ukachi et al., 2014). However, this collaborative nature also introduces security vulnerabilities and risks necessitating exploration. This literature review delves into the complexities of OSS vulnerabilities, their unique attributes, and their broader implications within the development landscape. The aim is to provide a comprehensive overview of current research on OSS, its vulnerabilities, risk metrics, and network structures. By examining existing knowledge, we identify research themes, gaps, and areas needing further investigation. This review sets the stage for our subsequent analysis, contributing to a deeper understanding of challenges posed by OSS vulnerabilities within the software supply chain.

3.1 Open Source Software Ecosystems and Supply Chains

A comprehensive categorization of existing Open Source Software (OSS) research by Aksulu and Wade, 2010 focused on adoption, community, governance, performance, and quality. While insightful, their work primarily addresses the advantages and disadvantages of OSS, neglecting an in-depth exploration of open-source licenses and their implications. This paper is related to our research as it provides a comprehensive overview of OSS research, which forms the foundation for understanding the ecosystem.

Transitioning to examining package dependency networks, Kikas et al., 2017 emphasized the critical role of these networks for software developers. However, their analysis, centered on metrics like degree distribution and clustering coefficient, overlooked crucial elements such as code quality and software architecture, which we plan to integrate into our risk modeling methodology.

3.2 Vulnerabilities Analysis in OSS

The Debsources dataset, spanning two decades of free and open-source software (FOSS) packages, offers a valuable contribution to the FOSS research community by Zacchiroli, 2015.

Focusing on vulnerabilities in the Debian package dependency prompts the extension of the analysis to diverse OSS ecosystems for a comprehensive understanding of vulnerabilities in the software supply chain.

The interconnected nature of the npm ecosystem, identifying security threats, was highlighted by M. Zimmermann et al., 2019. Our research aims to extend this analysis to multiple OSS ecosystems, providing a holistic view of security threats and their implications in the OSS supply chain. Additionally, research by Temizkan et al., 2017 emphasized software diversity as a mitigation strategy, but our approach broadens the scope to consider other security threats beyond software-based vulnerabilities.

Addressing the behavior of software vendors in response to vulnerabilities, Arora et al., 2010 demonstrated the impact of vulnerability disclosures on patch release times. However, there is a need to consider the effectiveness of mitigation strategies alongside response times. Furthermore, the study by Mitra and Ransbotham, 2015 explored the relationship between information disclosure and security attacks, contributing to our understanding of root causes and mitigation strategies.

The presence of malicious packages in PyPI was scrutinized by Liang et al., 2021, offering user protection suggestions. Yet, further research requires a comprehensive strategy for preventing and mitigating supply chain attacks in OSS ecosystems. Pashchenko et al., 2020 explored the security implications of dependency management, primarily focusing on direct dependencies. We argue for a broader perspective encompassing indirect dependencies to grasp the complete landscape of vulnerabilities in OSS.

While Alexopoulos et al., 2019 introduced a system for automating vulnerability analysis in Debian GNU/Linux, our future research should address social and organizational factors contributing to vulnerabilities in open-source software. Vasilakis et al., 2021 explored a novel approach to eliminating vulnerabilities in software supply chains, aligning with our focus on addressing supply chain vulnerabilities.

In the realm of JavaScript-based OSS, Zahan et al., 2022 identified and analyzed vulnerable functions within the npm supply chain, contributing to our broader analysis of vulnerability types. Sejfia and Schäfer, 2022 introduced a practical approach to detecting malicious packages in the npm registry, aligning with our objective to explore ways to detect vulnerabilities and threats in OSS supply chains.

Focusing on dependency smells in JavaScript projects, Jafari et al., 2022 enhance our understanding of indirect dependencies' issues. Reid et al., 2022 investigated orphan vulnerabilities resulting from code reuse, aligning with our research interests in understanding the longevity of vulnerabilities in the supply chain. Kluban et al., 2022 proposed a method for measuring vulnerable JavaScript functions, providing insights into the prevalence and impact of specific vulnerabilities in the supply chain.

3.3 Risk Metrics and Network Structural Analysis

A screening test for detecting vulnerabilities in FOSS components by Dashevskyi et al., 2019, aligns with our focus on efficient vulnerability detection. Xue et al., 2013 emphasized risk mitigation strategies in supply chain digitization, aligning with our research objectives. Sabbagh and Kowalski, 2015 presented a socio-technical framework for threat modeling, emphasizing a comprehensive approach to vulnerabilities in software supply chains.

Research from an attacker's perspective advocated by Mahmood et al., 2010 aids our understanding of potential exploits and vulnerabilities in OSS supply chains. Lin et al., 2023 offered insights into vulnerability management in Linux distributions, supporting our investigation into vulnerability management practices in the context of OSS supply chains.

Champion and Hill, 2021 introduce a novel risk metric based on bug resolution time, highlighting the importance of aligning volunteer efforts with software demand. This concept resonates with Schueller and Wachs, 2024 which emphasizes the need to consider both social and technical factors in assessing systemic risks. The interconnected nature of these risks further highlights the method for analyzing vulnerability and developmental risks in OSS ecosystems, which visualizes these factors as a directed graph.

Network analysis is crucial for understanding vulnerabilities and risks in Open Source Software (OSS) ecosystems. MacCormack and Sturtevant, 2016 highlighted the importance of modularity in OSS, emphasizing how system architecture influences vulnerability propagation. This concept aligns with Brunswicker and Mukherjee, 2023, who introduced core-based and central-based structures, further illustrating how modularity affects vulnerability spread.

In identifying vulnerable nodes, Rogers, 2015 used the cavity method, inspiring our methodology to identify high-risk packages within Debian's source repositories. Lawyer, 2015 introduced the "expected force" (ExF) metric, guiding our evaluation of package risk and vulnerability within Debian's network. Additionally, T. Zimmermann and Nagappan, 2008 focused on defect prediction through network analysis, providing insights into vulnerability propagation in interconnected systems.

3.3.1 Explaining Network-Based Risk Metrics

The table 3.1 gives a clear view of the algorithms used and justifications for calculating each metric.

Table 3.1. Risk metrics, algorithms used, and their justification for assessing risks within a software supply chain network.

Risk Metric	**Algorithm**	**Justification**
Degree (Bang, 2008)	Count of edges (Bang, 2008)	Indicates the level of connectivity of a node. Higher degrees indicate greater connectivity and potential impact.
Coreness Value (Baldwin et al., 2014)	K-core decomposition (Seidman, 1983)	K-core decomposition helps identify subgraphs with strong internal connectivity, highlighting nodes with higher centrality and influence.
Path Length (“Paths, Diffusion, and Navigation”, 2016)	Depth-First Search (DFS) (Putri et al., 2011)	DFS is effective for exploring all possible paths from a node, making it suitable for measuring the number of steps required for changes or vulnerabilities to propagate.
Betweenness Centrality (Brandes, 2001)	Brandes Algorithm (Brandes, 2001)	Brandes algorithm measures the number of shortest paths that pass through a node, effectively indicating its role in connecting different parts of the network.
Attack Radius (Corneil et al., 2003)	Breadth-First Search (BFS) (Holdsworth, 1999)	BFS explores all nodes at the present "depth" level before moving on to nodes at the next depth level, making it ideal for measuring the reach or influence of a component within the network.

MacCormack et al., 2006 proposed propagation cost as a metric to quantify the degree of coupling in software designs, indicating how changes to one element can affect others. This metric, derived from the visibility matrix in Design Structure Matrices, provides insights into the structural complexity of software systems, aiding in decision-making for software development and maintenance. This relates to our research by providing a framework to analyze the impact of changes on software packages, aligning with my focus on software supply chain security and risk evaluation methodologies.

Baldwin et al., 2014 developed a method for detecting core-periphery structures in networks, crucial for identifying structural vulnerabilities in OSS supply chains. This aligns with our understanding of how vulnerabilities can be concealed within network structures, emphasizing the importance of considering network topology in vulnerability analysis. For our study, we want to use algorithms such as In-degree, K-Core Decomposition, DFS, Betweenness centrality, and BFS.

Identifying Critical Packages with In-Degree

As defined by Bang, 2008, the in-degree of a node in a directed graph represents the number of incoming edges pointing towards it. In software supply chains, in-degree signifies the number of packages that rely on a particular package. Packages with high in-degree become more critical from a risk perspective as any vulnerability within them can cascade effects on many dependent packages.

Justification: A high in-degree indicates a package with many dependent packages. Any vulnerability or issue within such a package can have cascading effects, impacting a large number of dependent packages in the supply chain. Packages with high in-degree become more critical from a risk perspective, requiring closer scrutiny and potentially stricter security measures.

Application: Imagine a core library with a surprisingly high in-degree value within the dependency graph. This information highlights the library's critical role and the extensive reach of its potential impact. Security teams can prioritize a thorough review of the library's code and implement stricter monitoring practices to ensure its security and stability.

Identifying Bridge Nodes with Betweenness Centrality

As described by Brandes, 2001, the betweenness centrality algorithm 1 measures how often a node acts as a bridge between other sub-networks/nodes in the graph. Within the context of software supply chain risk assessment, betweenness centrality helps identify bridge nodes. These bridge nodes essentially control the flow of information or dependencies between various components. A compromise or disruption within such bridge nodes could have a significant impact on the overall functionality and security of the software system.

Justification: Nodes with high betweenness centrality act as bridges, connecting different parts of the dependency graph. These bridge nodes essentially control the flow of information or dependencies between various components. A compromise or disruption within such bridge nodes could have a significant impact on the overall functionality and security of the software system.

Application: By calculating betweenness centrality for packages in the dependency graph, we can identify bridges that connect different parts of the software supply chain. Security teams can then prioritize these bridge nodes for more in-depth security analysis and potentially implement stricter controls to mitigate the risks associated with a potential compromise or disruption within these critical bridge nodes.

Algorithm 1 Betweenness Centrality (Brandes Algorithm)

```
Require: Directed or undirected graph $G = (V, E)$
Ensure: Betweenness centrality scores for all nodes in $V$
 1: Initialize $n \leftarrow$ V (number of nodes in $G$)
 2: Initialize betweenness $\leftarrow [0] * n$ (array to store centrality scores)
 3: for each node $source \in V$ do
 4:     (shortest_paths, _) $\leftarrow$ SingleSourceShortestPaths($G$, $source$) {Function definition needed}
 5:     AccumulateBetweenness($G$, shortest_paths, $source$, betweenness)
 6: end for
 7: for i $\in \{0, 1, \ldots, n-1\}$ do
 8:     betweenness[i] $\leftarrow \frac{\text{betweenness[i]}}{(n-1)(n-2)}$ {Normalize}
 9: end for
10: return betweenness
```

Identifying the Longest Path with Depth-First Search (DFS)

Pioneered in the work by Putri et al., 2011, DFS algorithm 2 is a graph traversal algorithm that prioritizes depth over breadth. It explores a path as far as possible until reaching a dead end. In software supply chain analysis, DFS can be applied to identify the longest path of dependencies within the dependency graph. This path highlights the sequence of packages that could potentially pose the greatest risk to a specific package, aiding in prioritizing security reviews.

Justification: The strength of DFS lies in its ability to prioritize depth over breadth. This characteristic aligns perfectly with the task of identifying the longest dependency chain. By exploring a path as far as possible, DFS efficiently identifies the sequence of dependencies that could pose the greatest risk to the security or stability of a specific package.

Application: Consider a vulnerability discovered in a core library within the supply chain. By applying DFS to the dependency graph, we can identify the longest path leading to the vulnerable library. This path highlights the critical dependencies that could be exploited if the vulnerability is compromised. Focusing on securing or mitigating vulnerabilities along this longest path becomes a priority in such a scenario.

Unveiling Core Components with K-Core Decomposition

Introduced by Seidman, 1983, the k-core decomposition algorithm 3 is a graph analysis technique that identifies densely interconnected subgraphs. It achieves this by iteratively removing nodes with degrees below a specified threshold. In software supply chain risk assessment, k-core decomposition plays a crucial role in uncovering core components or packages that are highly interconnected and essential for the overall functioning of the software system. Identifying these core components allows security teams to prioritize them for enhanced security measures.

Justification: The k-core analysis reveals densely connected subgraphs, essentially highlighting core components or packages that are highly interconnected and essential for the overall functioning of the software system. A compromise within these core components could have a significant cascading effect, disrupting the entire system.

Algorithm 2 Longest Path in a Directed Graph (DFS)

```
Require: Directed graph $G = (V, E)$
Ensure: Length of the longest path in $G$
 1: Initialize visited[$v$] $\leftarrow$ False for all $v \in V$
 2: Initialize path_length[$v$] $\leftarrow 0$ for all $v \in V$
 3: Initialize max_length $\leftarrow 0$
 4: DFS Function: DFS($v$)
 5: Set visited[$v$] $\leftarrow$ True
 6:
 7: for each $u$ such that $(v, u) \in E$ do
 8:     if visited[$u$] = False then
 9:         Call DFS($u$)
10:     end if
11:     Update path_length[$v$] $\leftarrow$ max(path_length[$v$], 1 + path_length[$u$])
12:     Update max_length $\leftarrow$ max(max_length, path_length[$v$])
13: end for
14: for each vertex $v \in V$ do
15:     if visited[$v$] = False then
16:         Call DFS($v$)
17:     end if
18: end for
19: return max_length
```

Application: Applying k-core decomposition to the dependency graph can expose these core components. Security teams can then prioritize these core components for enhanced security measures, such as stricter vulnerability scanning, continuous monitoring, and potentially implementing additional security controls to mitigate potential risks.

Assessing Attack Radius with Breadth First Search (BFS)

As described by Holdsworth, 1999, BFS algorithm 4 explores all immediate neighbors of a node before moving on to the next level. When analyzing software supply chain risk, BFS helps determine the attack radius of a package. The attack radius represents the immediate dependencies or interactions that can directly impact the security or integrity of the package. By applying BFS, we can efficiently identify packages directly dependent on a potentially vulnerable library, allowing for focused risk mitigation efforts.

Algorithm 3 K-Core Decomposition

```
Require: Graph $G = (V, E)$
Ensure: K-core subgraph $G_k \subseteq G$
 1: Initialize $G_k \leftarrow G$
 2: while True do
 3:   Initialize $changed \leftarrow$ false
 4:   for each $v \in V(G_k)$ do
 5:     $deg(v) \leftarrow \deg_G(v)$ {Update degree in $G_k$}
 6:     if $deg(v) < k$ then
 7:       $G_k \leftarrow G_k \setminus \{v\}$
 8:       for each $u \in N_G(v)$ do
 9:         $deg(u) \leftarrow deg(u) - 1$ {Update neighbor degrees}
10:       end for
11:       $changed \leftarrow$ true
12:     end if
13:   end for
14:   if $\neg changed$ then
15:     break
16:   end if
17: end while
18: return $G_k$
```

Justification: BFS is ideal for identifying the closest potential threats. BFS efficiently identifies the packages directly dependent on a potentially vulnerable library or component by prioritizing the exploration of a node's immediate neighbors. Analyzing the security posture of these immediate dependencies becomes crucial in understanding the potential impact on the package under consideration.

Application: Imagine a newly discovered vulnerability in a widely used package. By applying BFS to the dependency graph centered on this vulnerable package, we can identify the immediate dependencies that directly rely on it. This information allows us to prioritize efforts toward assessing and mitigating the vulnerability's impact on these dependent packages.

The studies discussed collectively provide a comprehensive framework for understanding vulnerability propagation and structural vulnerabilities within open-source software (OSS) ecosystems. They shed light on crucial aspects of OSS security analysis by emphasizing

Algorithm 4 Radius using Breadth-First Search (BFS)

```
Require: Directed graph $G = (V, E)$ and source vertex $s \in V$
Ensure: Radius (maximum distance) from $s$ to any reachable node in $G$
 1: Initialize distance[$v$] $\leftarrow \infty$ for all $v \in V$
 2: distance[$s$] $\leftarrow 0$
 3: Initialize an empty queue $Q$
 4: Enqueue $s$ to $Q$
 5: while $Q$ is not empty do
 6:    Dequeue a vertex $u$ from $Q$
 7:    for each neighbor $v$ of $u$ in $G$ do
 8:       if distance[$v$] $= \infty$ then
 9:          distance[$v$] $\leftarrow$ distance[$u$] $+ 1$
10:          Enqueue $v$ to $Q$
11:       end if
12:    end for
13: end while
14: return $\max_{v \in V}$(distance[$v$])
```

modularity, vulnerability identification, and network topology. Specifically, they underscore the interconnectedness of components within the OSS supply chain ecosystem and the potential for vulnerabilities to propagate across this network. Additionally, they highlight the importance of identifying and mitigating structural vulnerabilities that may be concealed within the network architecture. Overall, these insights inform our research by providing a robust foundation for investigating risk metrics and vulnerability assessment methodologies within OSS supply chain ecosystems.

4. METHODOLOGY

Our research methodology encompasses a systematic approach aimed at gaining comprehensive insights into the intricate dynamics of the Open-Source Software (OSS) supply chain. The key phases of the methodology are shown in figure 4.1:

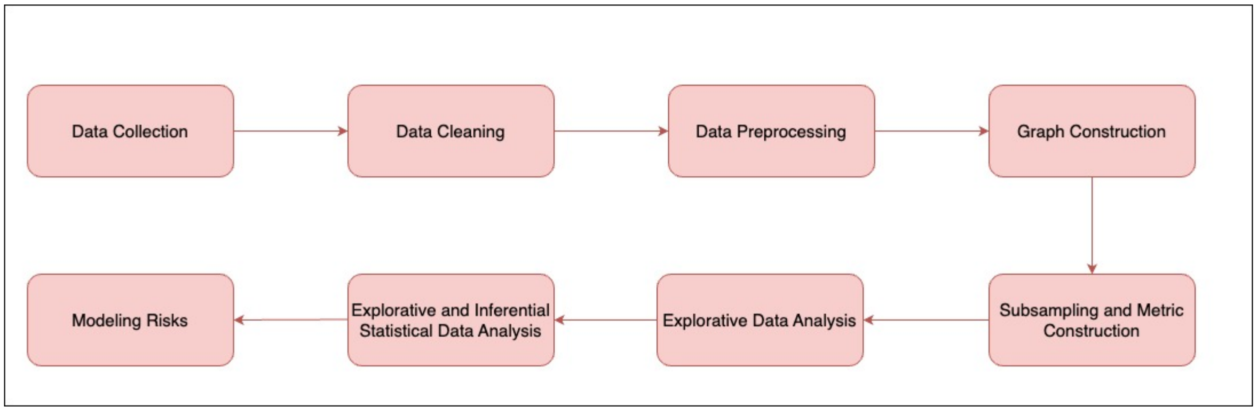
Data Collection
Data Cleaning
Data Preprocessing
Graph Construction
Subsampling and Metric Construction
Explorative Data Analysis
Explorative and Inferential Statistical Data Analysis
Modeling Risks

Figure 4.1. Overview of Methodology

4.1 Data Collection

We have collected various datasets that help us model the structural properties of representative supply chains, such as upstream source, package popularity, build provenance, and maintainers of Debian packages. Its vital to consider the structural properties of the overall network.

Our research methodology is underpinned by advanced network science methods, providing a comprehensive framework for assessing risks within the Open-Source Software (OSS) supply chain. The core of our approach revolves around a detailed examination of the intricate connections and dependencies between individual software components. By employing advanced network science techniques, we gain valuable insights into the structural dynamics of the OSS supply chain, enabling a thorough evaluation of the risks associated with each software package.

Our study is centered on Debian packages, representing diverse software components within the broader OSS supply chain. Debian packages are chosen for their widespread use and significance in the open-source community. Our selection process involves carefully curating a representative sample of these packages to ensure a nuanced analysis. The diversity of our chosen sample enhances the applicability of our findings to the broader ecosystem.

The collection of pertinent data is a pivotal aspect of our research. We employ a meticulous data collection process to obtain a comprehensive understanding of the selected Debian packages. This involves retrieving the source code from Debian Salsa, a process facilitated by a customized script tailored for accuracy and completeness. Our data collection extends beyond source code, encompassing information about the build process, package popularity, and maintenance details. Additionally, we incorporate data related to any security vulnerabilities associated with these packages. The comprehensive nature of our data collection process ensures a holistic view of each software component.

Ensuring the validity and reliability of our collected data is of paramount importance. To achieve this, we implement rigorous checks and measures. A crucial element of our validation process involves cross-checking checksums across our dataset, mitigating the risk of data corruption or inaccuracies. Redundancy checks further confirm the authenticity of the collected data, providing an additional layer of validation. The systematic organization of our data enhances its reliability, facilitating robust and trustworthy analyses. These meticulous procedures collectively contribute to the integrity of our research findings.

4.1.1 Debian Source & QA Sites

In our data collection methodology, the process revolves around systematically acquiring source code repositories from Debian Salsa (Sprint, 2023). This operation is streamlined through a customized Python script, necessitating configuration with the researcher's Salsa username and an authentication token. The script initiates data collection by retrieving the names of all unique packages from the 'Buildinfo Packages' table within the database.

To uphold integrity, we implement a redundancy check involving cross-referencing packages within the Salsa platform. This is crucial, considering a single package may have multiple associated repositories due to forks or other variations. To address this complexity, we query ’tracker.debian’, a pivotal step in identifying the precise URL to the authoritative source of the repository on Salsa.

Upon successful identification, the script triggers the cloning process using the Git Version Control System (VCS). It is imperative to note that this process can be resource-intensive and storage-hungry. Strategic measures have been embedded to optimize efficiency. Initially, the script performs a full cloning of the repository, focusing on the identification and exclusive re-cloning of the primary branch, typically named ’main’ or ’master’ (varies at times). This targeted approach minimizes unnecessary overhead.

Further efficiency enhancements include limited depth cloning, restricting commits to those within the specified timeframe. The first clone identifies the necessary commits within the timeframe, and the subsequent clone is executed with the set depth, effectively trimming the repository size to essential historical data. For large repositories exceeding 200MB, a thoughtful strategy is employed. As we clone a repository, we record the hash of its head commit. When encountering another repository with an identical head commit hash, a symbolic link is created to the matching repository, thereby mitigating space consumption.

This strategic sequence, encompassing complete cloning, branch selection, and depth adjustments, ensures an efficient data collection process. It strikes a delicate balance between acquiring comprehensive source code repositories and judiciously managing resources, thereby amplifying the overall effectiveness of our research efforts.

4.1.2 BUILDINFO Files

Debian Build Information is pivotal in meticulously collecting essential data related to the build environment and process. Its primary aim is to ensure reproducible builds, guaranteeing consistent results across diverse systems. The gathered data encompasses vital details such as build architecture, dependencies, build path, environment variables, and checksums of generated files (Zanardi, 2022).

The collection process is orchestrated by the debian/rules script, which oversees the build and captures pertinent environmental data. Additionally, the dpkg-gen buildinfo tool generates .buildinfo files, serving as repositories for these critical build details. The utility of this collected data is multifaceted, enabling accurate reproduction of builds, tracking and troubleshooting build issues, identifying discrepancies in build environments, and validating the correctness of builds across various systems. Debian Build Information stands as a vital resource, fostering build consistency, enhancing debugging practices, and validating the accuracy of the Debian package-building process ("ReproducibleBuilds - Debian Wiki", 2022; "SimplePackagingTutorial", 2022).

Our dedicated data collection effort spans from January 1, 2017, to December 30, 2022. During this period, we acquired and meticulously organized the buildinfos into five distinct tables for ease of evaluation and data access. These tables, namely the source table, buildinfo table, binary table, dependency table, and output table, are interconnected, providing a comprehensive understanding of the dependencies associated with each package in a detailed and structured manner. This approach ensures a holistic view of the build information, enabling effective analysis and management of Debian packages.

4.1.3 Vulnerability Database

The Debian Security Tracker stands as a pivotal system meticulously crafted for the comprehensive collection of data regarding security vulnerabilities impacting Debian packages (Allombert, 2022). This repository holds indispensable information, encompassing CVE identifiers, severity levels, distribution status tags, generated reports, and DSA advisories. Its primary objective revolves around furnishing the Debian Security Team with an efficient mechanism for monitoring and addressing security vulnerabilities. Beyond providing up-to-date information to users, the tracker plays a crucial role in evaluating the impact of vulnerabilities and facilitates collaborative editing of vulnerability data. This system adeptly manages data through a Git repository and text files.

The significance of the tracker extends to generating detailed reports on vulnerable packages for various Debian releases, spanning stable, testing, sid, and old stable distributions. The data, conveniently available in JSON format, enhances users' capabilities to stay well-informed and effectively prioritize security fixes.

To uphold the accuracy and timeliness of the data, a daily cronjob is implemented for downloading JSON-formatted files from the web. This daily update regimen ensures the vulnerability table remains current, providing the most precise and recent information. Furthermore, our data enrichment practices incorporate the published date and last modified date from the CVE page, enhancing our vulnerability data tracking. Additionally, we establish a robust connection between vulnerability data and our source data, creating a cohesive framework that enhances understanding and facilitates more effective management of overall package data.

4.2 Graph Construction

The initial phase involves constructing a directed graph using data, wherein source/main packages form connections with dependencies and binaries. These connections mirror the dependencies of source packages on these components for functionality (Figure 4.2). Our dataset was utilized to establish connectivity within packages, resulting in the creation of a directed network (Figure 4.3).

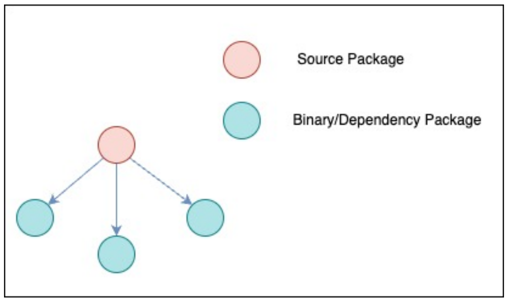

Figure 4.2. Source and Dependency Package Connectivity in OSS Supply Chain

4.3 Subsampling and Metrics Construction

In our analytical approach, we initially attempted to leverage community algorithms (Aynaud, 2022) to partition the network into subgraphs, but the results did not align with our expectations. Recognizing the sparse nature of the graph, we shifted our strategy to leverage weakly connected components for subgroup identification. These components rep-

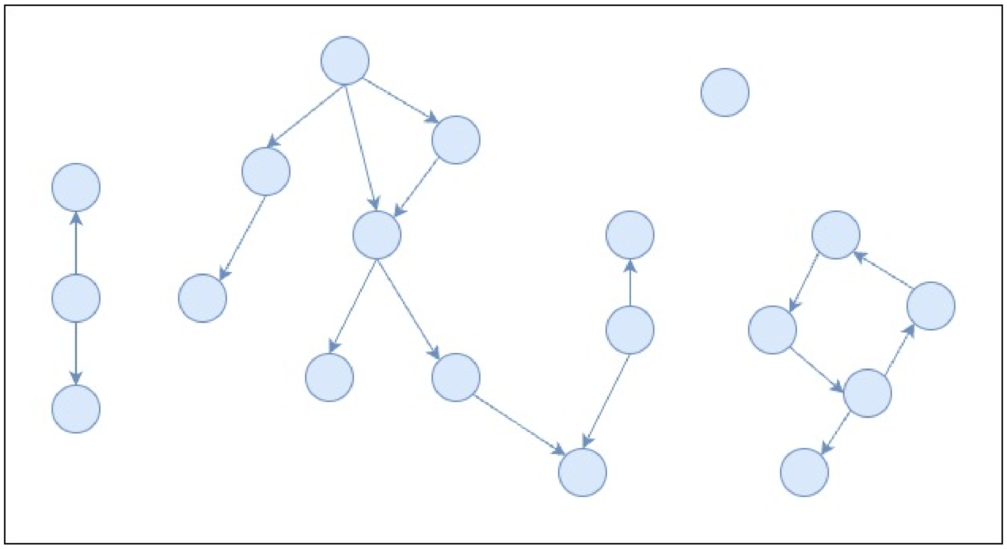

Figure 4.3. A sample directed network in Supply Chain

resent groups of nodes connected either directly or indirectly, regardless of edge directionality (Bendali et al., 2016). Our analytical focus on individual packages within these subgroups involves calculating in-degree, coreness value, path lengths, betweenness centrality, and attack radius metrics. These metrics offer a comprehensive understanding of interconnectedness, dependencies, and influence within the OSS supply chain network.

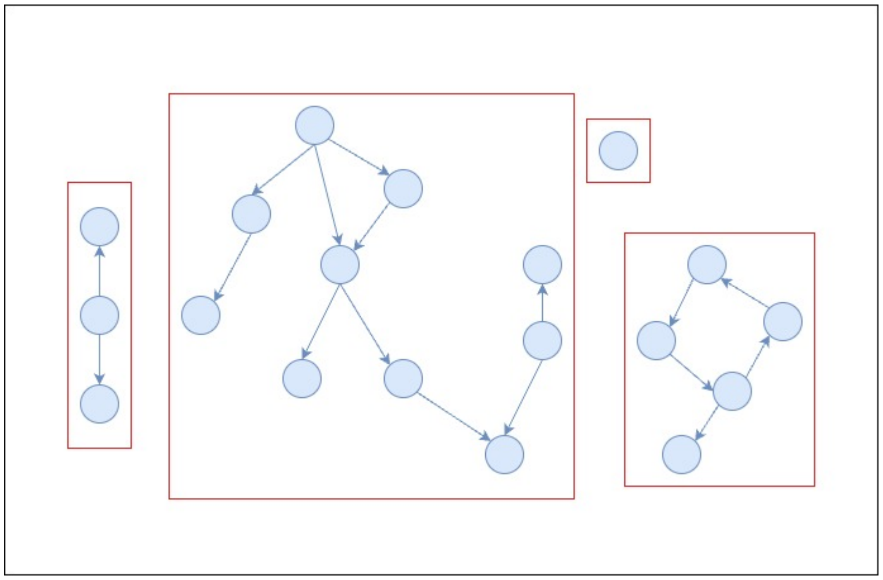

Figure 4.4. Weakly Connected Components in the graph

4.4 Explorative Data Analysis

Our statistical analysis adopts a nuanced approach, employing measures like mean and standard deviation to unveil central tendencies and data variability. Outlier analysis scrutinizes packages deviating significantly from the norm, and the distribution of each metric is thoroughly explored to decipher overarching patterns (ISO, 2022). These insights on the dynamic and evolving trends of the risk landscape within the source code and build stages of the OSS supply chain, systematically address our primary research question of identifying high-risk packages.

4.5 Explorative and Inferential Statistics Data Analysis

Our explorative descriptive analysis carefully scrutinizes disparities in risk profiles between the source and build graphs, focusing on key metrics including In-degree, Coreness, Path Length, Betweenness Centrality, and Attack Radius. We delve into inferential statistics to gain deeper insights, employing methods such as T-test and ANOVA. The T-test evaluates the significance of mean differences, and ANOVA assesses variations among groups.

This Explorative and Inferential Statistics Data Analysis plays a pivotal role in addressing our second research question concerning the correlation of risks within the supply chain.

4.6 Modeling Network-Based Risk Profiles for Predicting Vulnerable Packages

In our analysis, we built the regression models by considering the risk metrics as independent variables and the vulnerability of the package as the dependent variable. We initially employed logistic regression (Jr et al., 2013) and decision trees (Polat & Güne, 2009) to predict the vulnerability of software packages. Despite achieving high accuracy, these models exhibited low precision in identifying vulnerable packages. To address this issue, we applied the Synthetic Minority Over-sampling Technique (SMOTE) (Wang et al., 2006) to balance the dataset, which improved the performance of these models. However, the precision for predicting vulnerable packages remained suboptimal.

To further enhance the precision of our models, we explored more advanced techniques, including BorderlineSMOTE (Han et al., 2005) for dataset balancing and k-fold cross-validation (Anthony & Holden, 1998) for model evaluation. Subsequently, we developed Random Forests and gradient-boosting models, which showed significant improvements in both accuracy and precision.

The Random Forests (Breiman, 2001), Gradient Boosting (Friedman, 2002) and the XG-Boost models (Chen & Guestrin, 2016), which build models sequentially to correct errors, both achieved an accuracy of approximately 98% and significantly improved the precision for predicting vulnerable packages. These results indicate that these advanced ML algorithms are effective in identifying vulnerabilities in software packages, thus enhancing software security.

Our analysis demonstrates the importance of employing advanced ML techniques and dataset-balancing methods for predicting vulnerable packages in software ecosystems. These models offer valuable insights for software developers and security analysts to identify and mitigate potential security risks in software systems.

5. RESULTS

5.1 Explorative Data Analysis

In this section, we apply network-based risk metrics to source and build dependencies, considering both direct and indirect dependencies, to assess the risk profile of packages to answer our RQ1.

5.1.1 Source Phase Analysis

Our exploration into the source phase of the Open-Source Software (OSS) supply chain has provided insightful results. The constructed graph, comprising 330,410 nodes and 2,055,521 edges, forms the foundational basis for our analysis. While the overall graph exhibits neither strong nor weak connectivity, a more detailed examination of subgraphs has uncovered significant patterns.

Table 5.1. Statistics of Risk Metrics for Source Phase Data

	In-Degree	Betweenness Centrality	Path Length	Coreness Value	Attack Radius
min	0.00	0.00	0.00	0.00	0.00
max	63 017.00	2.87×10^{-6}	6.00	69.00	6.00
mean	28.43	6.63×10^{-11}	0.72	8.45	0.71
median	2.0	0.0	1.0	5.0	1.0
std	404.28	1.25×10^{-8}	0.80	9.27	0.79

The table 5.1 provides a summary of the risk profiles of packages in the source phase. If we focus on two critical metrics, namely attack radius and path length, which are indicative of the network's structure and vulnerability to attacks. The median of attack radius and path length valued "1" suggests a relatively shallow network, with most packages having direct dependencies within one hop. However, the wide standard deviation implies a significant variability in the number of hops required to reach dependent packages, with some vulnerabilities potentially propagating up to six hops through indirect dependencies. Package connectivity reveals a network characterized by both extensive interconnectivity and significant variations in dependency levels. While the high average in-degree indicates a generally well-connected network, the vast standard deviation underscores the presence of outlier packages with an unusually high number of dependencies. These "superstar" packages pose a considerable risk, given their potential for disproportionate impact if compromised. Centrality metrics - betweenness centrality and coreness value, tell about the importance of individual packages within the network. The low median and mean values of betweenness centrality suggest that only a few packages play crucial roles as bottlenecks, influencing the flow of information within the network. Similarly, the coreness value metric indicates a moderately centralized network, with certain packages being more critical due to their central position and potential impact on network stability.

The in-degree distribution (Figure 5.1) demonstrates that a substantial portion (75%) of nodes possess a relatively low number of incoming dependencies lying in the range from 0 to 10. This suggests a decentralized structure where most components rely on a limited number of upstream projects.

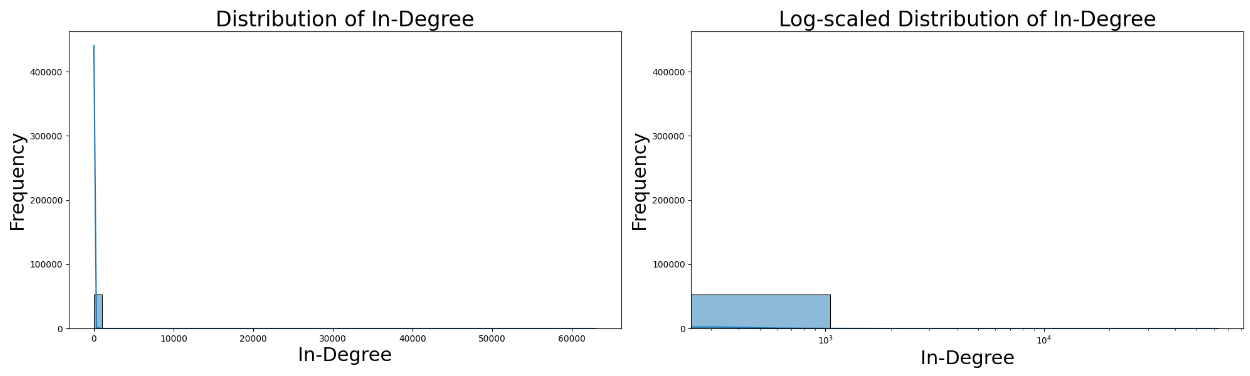

Figure 5.1. Distribution of Indegree in Source Stage

Similarly, the Betweenness Centrality distribution (Figure 5.2) indicates that a significant number of nodes do not lie on critical paths between other components. This implies that a disruption in these nodes might have a lesser impact on the overall functionality of the network. The Path Length distribution (Figure 5.3) reinforces this notion, revealing that most nodes reside in close proximity within the network, potentially facilitating efficient information flow.

The Coreness Value distribution (Figure 5.4) strengthens the core-periphery structure observed earlier, with most nodes concentrated at the periphery (Coreness Value of 0) or

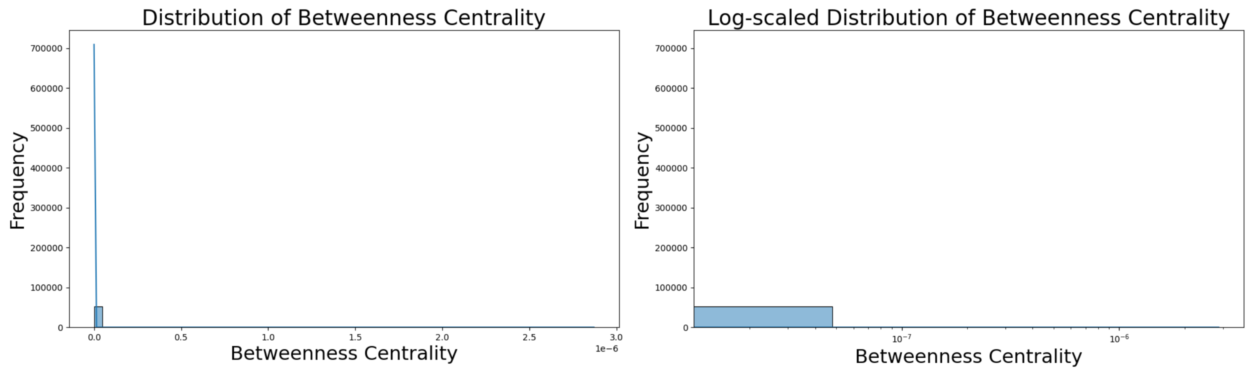

Figure 5.2. Distribution of Betweenness Centrality in Source Stage

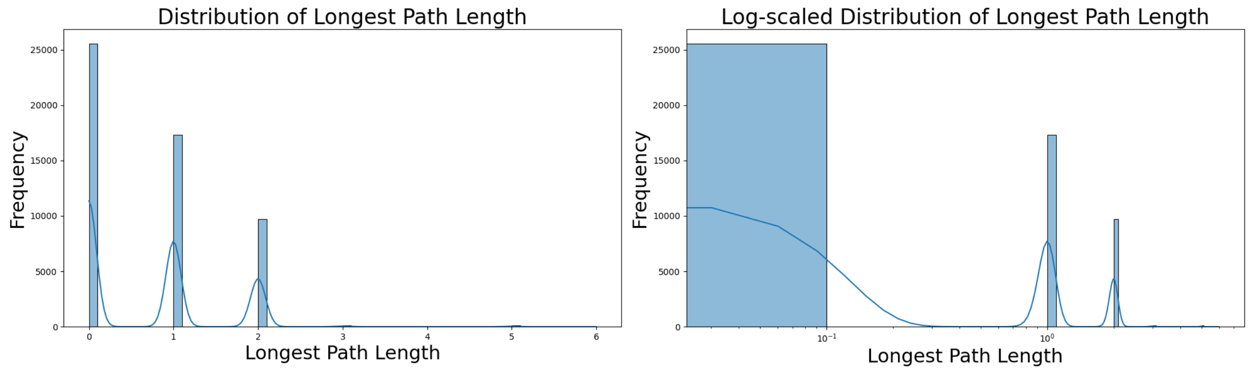

Figure 5.3. Distribution of Path Length in Source Stage

within the core (Coreness Value of 11-69). This highlights the existence of a well-defined central group with high interconnectedness.

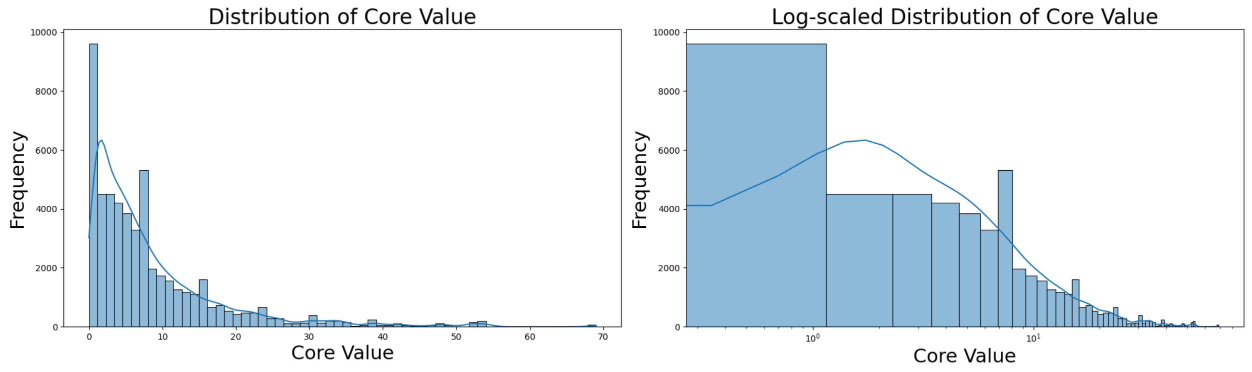

Figure 5.4. Distribution of Coreness Value in Source Stage

Finally, the Attack Radius distribution (Figure 5.5) underscores a low susceptibility to targeted attacks for the majority of nodes. This suggests that a localized attack might be contained within a specific area, potentially limiting its overall impact. However, the data's lack of adherence to traditional distributions like normal or Poisson warrants further investigation.

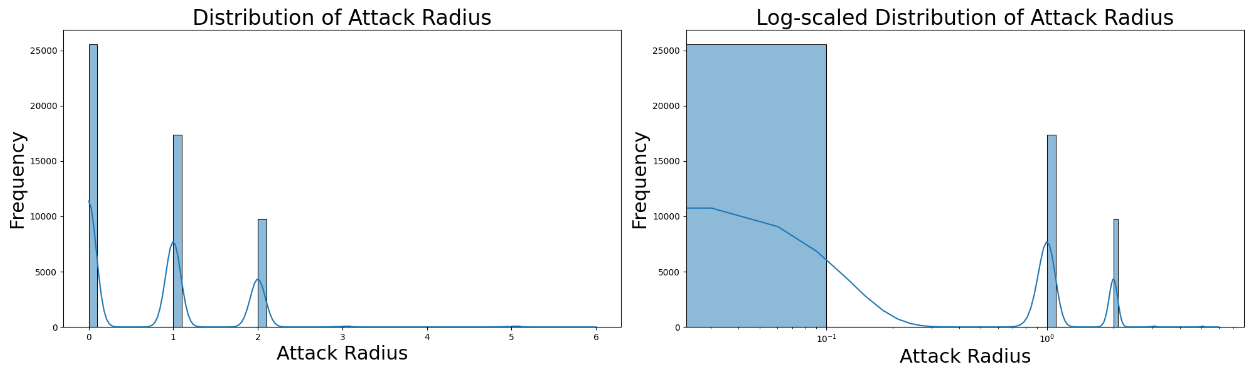

Figure 5.5. Distribution of Attack Radius in Source Stage

The analysis of the metrics in the source phase, as depicted in Figure 5.6, highlights potential vulnerabilities. The attack radius and path length exhibit a very strong positive correlation, indicating that packages with longer dependency chains tend to have higher attack radii and cover greater distances in the network. Conversely, there is a weak negative correlation between attack radius and in-degree, suggesting that packages at the end of dependency chains (higher attack radius) have fewer incoming dependencies.

Additionally, in-degree shows weak positive correlations with both betweenness centrality and coreness value, implying that packages with many incoming dependencies might slightly influence information flow and are more likely to be part of densely connected communities. The remaining ones exhibit either very weak or no correlation between them. This analysis suggests that packages with higher in-degree are less directly reachable but might still play crucial roles within the network, making them important targets for vulnerability mitigation efforts.

The analysis shown in figure 5.7 of network properties reveals significant differences between vulnerable and non-vulnerable packages, revealing distinct patterns that can inform vulnerability assessment and mitigation strategies. Vulnerable packages, are labeled

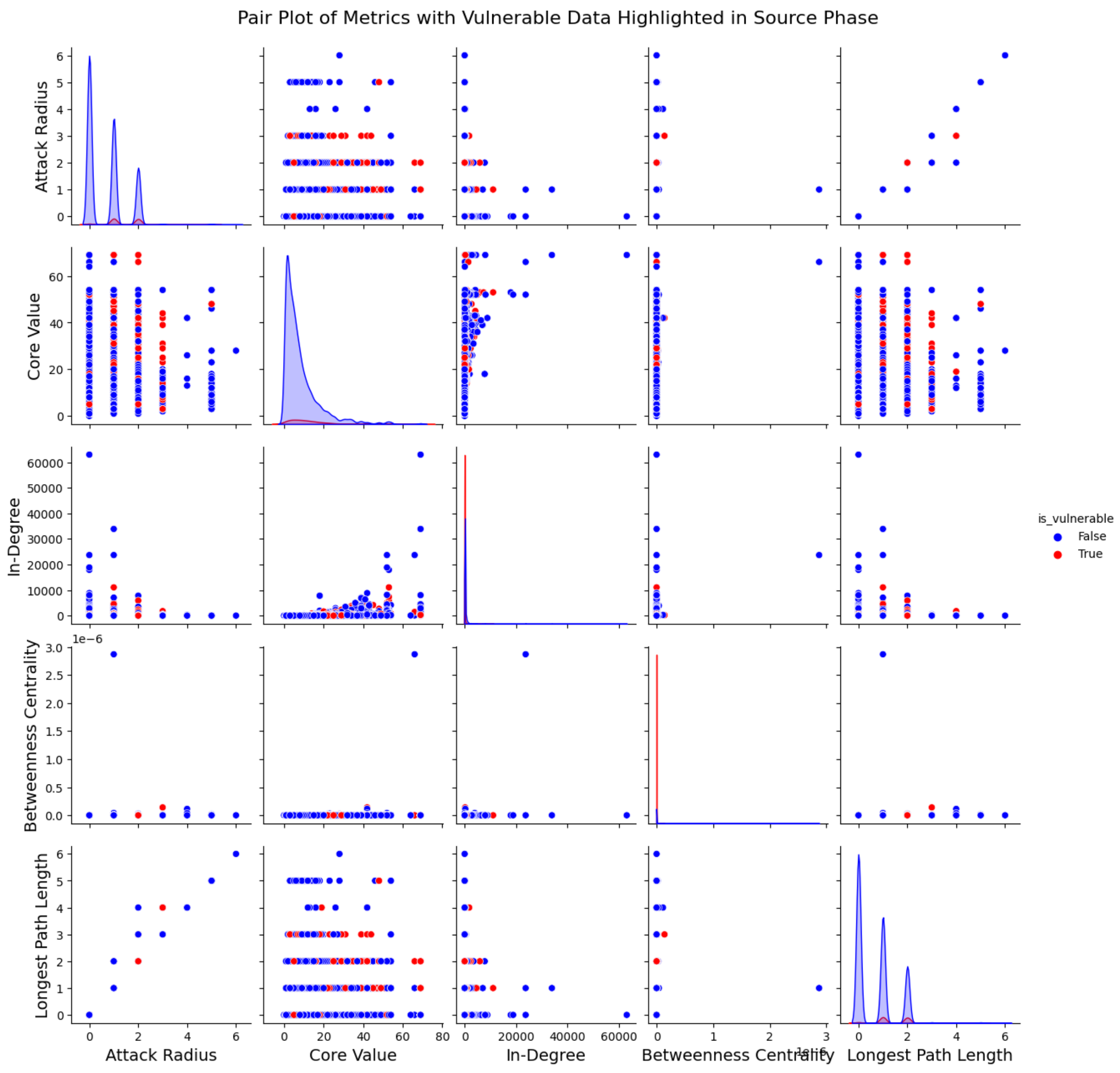

Figure 5.6. Pair plot of Data in Source Stage

as "True," and non-vulnerable ones as "False". Vulnerable packages exhibit larger attack radii and path lengths, indicating their potential to impact a larger portion of the network compared to non-vulnerable packages. They also tend to have higher in-degree values, suggesting they are depended on by more packages, making them challenging to update or remove without affecting other parts of the system. Furthermore, vulnerable packages have

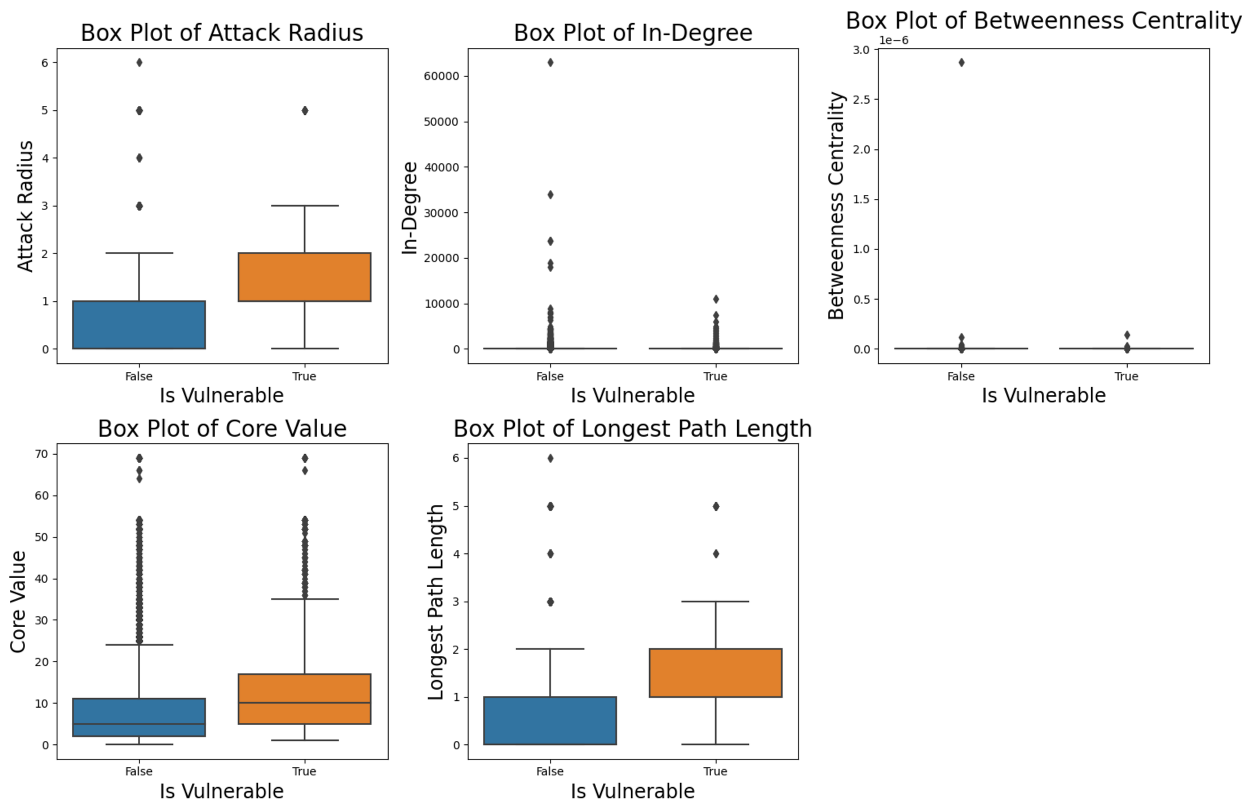

Figure 5.7. Comparison of medians between the vulnerable and non-vulnerable data of risk metric

higher betweenness centrality and coreness values, indicating their critical position in the network's communication pathways and overall structure. This emphasizes the importance of addressing vulnerabilities in these packages promptly to reduce the risk of network disruptions. Overall, prioritizing the mitigation of vulnerabilities in packages with higher risk metrics can significantly enhance the network's security and resilience.

The analysis shown in figure 5.8 of network metrics in the source phase for popular (High Inst) and non-popular (Non-High Inst) packages reveals that popular packages tend to be more central within the network. We have considered the top 1% of the popularity metric values as popular packages. This is evident from their higher coreness values, indicating their central role and greater connectivity. In contrast, metrics such as attack radius, in-

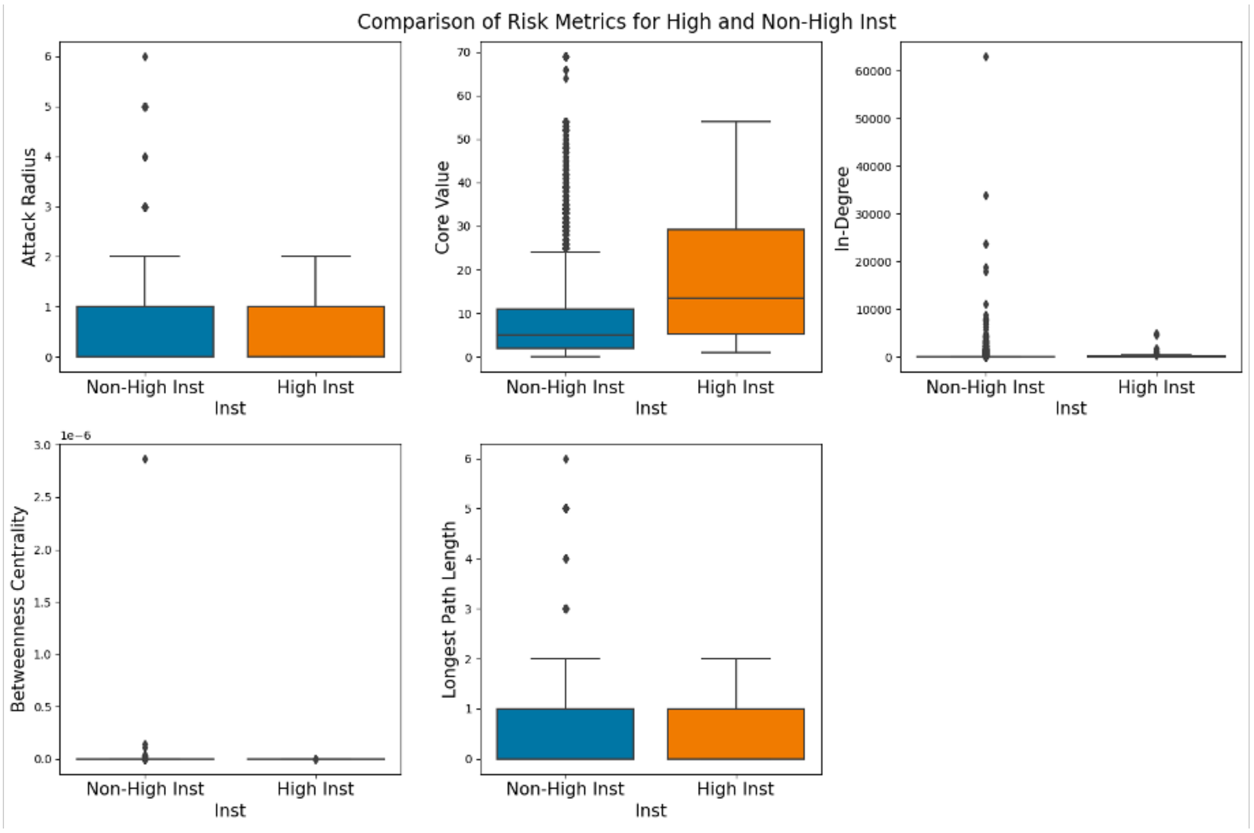

Figure 5.8. Risk Profile of High Popular Package in Source

degree, betweenness centrality, and path length show no significant differences between the two groups.

Table 5.2. Number of Past Vulnerability Packages found at Different Thresholds of Outliers for each Risk Metric in Source

Risk Metric	**# Outliers (5%)**	**# Outliers (10%)**	**# Outliers (15%)**	**# Outliers (20%)**
Attack Radius	0	0	0	31
In-degree	108	213	329	451
Betweenness Centrality	42	42	42	42
Coreness value	7	88	148	255
Path Length	0	0	0	35

Table 5.2 presents the number of outlier vulnerabilities identified using different thresholds for risk metrics within the source code. The analysis reveals distinct patterns for each metric. Attack Radius shows no outliers until the 20% threshold, where 31 outliers are found,

indicating a lower prevalence of vulnerabilities with a large impact radius. In-degree exhibits a steady increase in outliers with increasing thresholds, suggesting a significant portion of vulnerabilities reside in functions or modules with a high number of incoming dependencies. Betweenness Centrality shows no variation in the number of outliers across any threshold, indicating a consistent identification of critical vulnerabilities. Coreness value in the source code has outliers identified at lower thresholds, suggesting a more prominent presence of vulnerabilities in core functionalities or central modules. Path Length shows no outliers until the 20% threshold, indicating that vulnerabilities in the source code may also reside in functions or modules with shorter dependency paths.

5.1.2 Build Phase Analysis

Transitioning to the build phase, our analysis encountered a more intricate and expansive graph, comprising 16,071,669 nodes and 598,926,877 edges. Similar to the source phase, conducting a holistic examination of the entire graph for connectivity proved challenging. Consequently, we concentrated on subgraph analysis to derive meaningful insights.

The table 5.3 focuses on descriptive analysis of risk metrics during the build phase, revealing a mix of isolated and highly interconnected packages. The median values of betweenness centrality and coreness value suggest that many packages are isolated, while the high mean values indicate significant connectivity among the rest. The median values of attack radius and path length indicate that half of the packages have only direct dependencies, but the wide standard deviation of 7.164 for attack radius suggests some vulnerabilities could reach packages up to 73 hops away, posing a risk of widespread impact. The high average in-degree of 54.688 reflects a complex and interconnected network, making vulnerability isolation challenging. While the mean betweenness centrality suggests a balanced influence distribution, the wide standard deviation implies some packages could act as critical bottlenecks if compromised. The low mean coreness value indicates a decentralized network, but the wide standard deviation suggests varying degrees of centrality among packages. The analysis underscores the need for robust security measures to mitigate the risk of widespread vulnerability exploitation in the build phase network.

Table 5.3. Statistics of Risk Metrics for Build Phase Data

	In-Degree	Betweenness Centrality	Path Length	Coreness alue	Attack Radius
min	0.00	0.00	0.00	0.00	0.00
max	121 474.00	8.16×10^{-6}	619.00	382.00	73.00
median	1.0	0.0	0.0	0.0	0.0
mean	54.69	5.61×10^{-10}	15.74	31.34	2.37
std	1163.06	3.13×10^{-8}	86.70	80.94	7.16

The in-degree distribution (Figure 5.9) shows that the overwhelming majority of nodes boast a low in-degree, indicating they receive information from a limited number of sources. This observation suggests potential bottlenecks in the build process. A build task with a low in-degree might represent a critical dependency for many downstream tasks, making it a vulnerable point if prone to delays or failures.

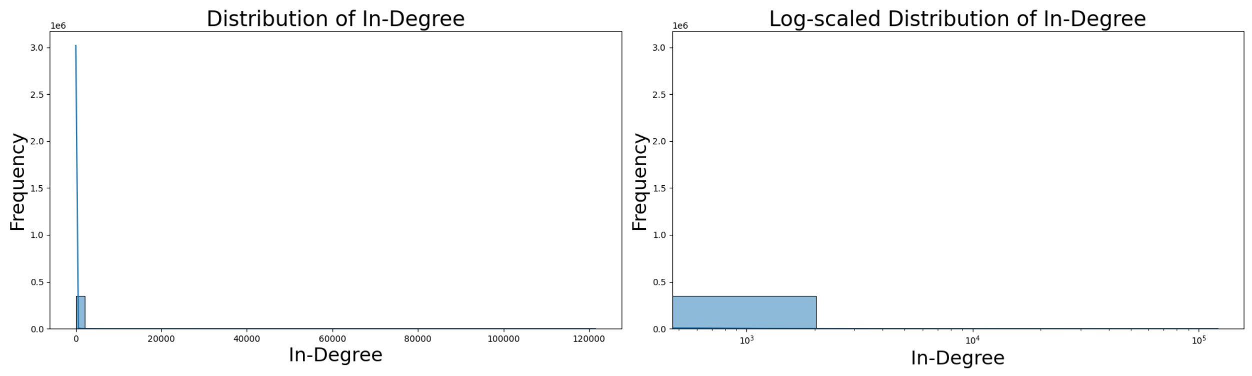

Figure 5.9. Distribution of In-Degree in Build Stage

Similarly, the Betweenness Centrality distribution (Figure 5.10) indicates that most nodes exhibit a centrality value of 0, implying they have minimal influence in connecting different parts of the network. While not inherently negative, this finding suggests a potentially decentralized build process, where information might flow through multiple paths, making it resilient to single-point failures.

The Path Length distribution (Figure 5.11) shows that many packages have very low indirect dependencies connections but few packages take the dominance paths with a length

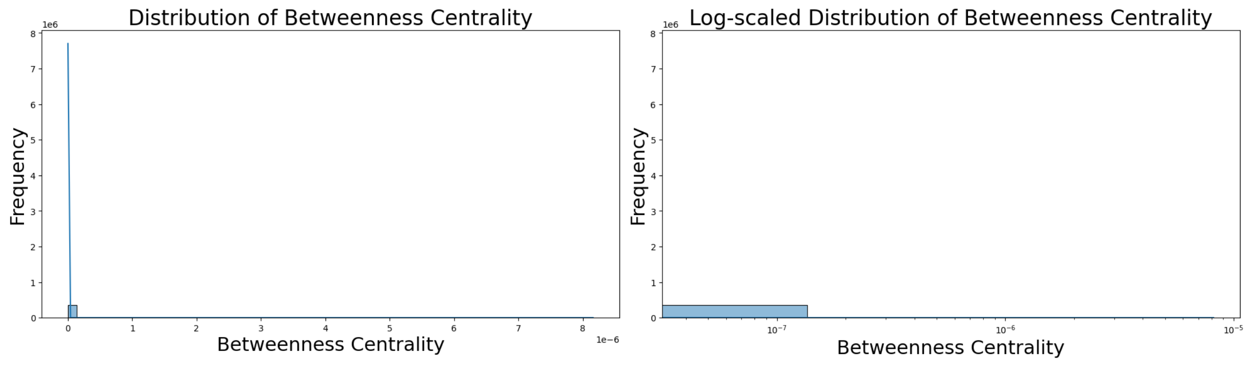

Figure 5.10. Distribution of Betweenness Centrality in Build Stage

of 619. Exploring these paths might reveal "hidden" dependencies or inefficiencies within the build process.

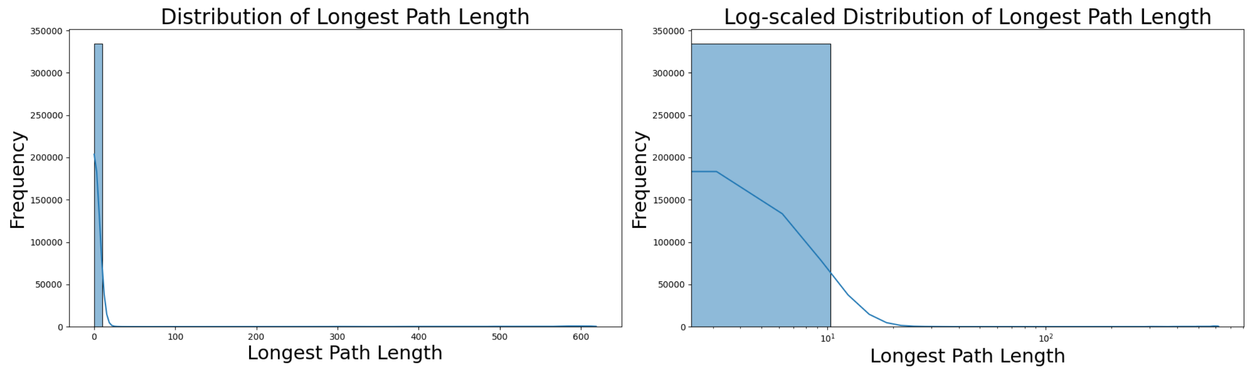

Figure 5.11. Distribution of Path Length in Build Stage

The Coreness alue Distribution (Figure 5.12) is similar to betweenness centrality, with a large portion of nodes residing on the periphery with a coreness value of 0. This reinforces the notion of a potentially decentralized build structure.

The Attack Radius Distribution (Figure 5.13) shows that most nodes possess an attack radius of 0, signifying a limited propagation of issues or vulnerabilities. While positive from a security standpoint, its crucial to analyze the behavior of nodes with a higher attack radius to identify potential weaknesses that could disrupt larger portions of the build process.

In the build phase analysis as shown in figure 5.14, a noticeable trend is observed in the distribution of vulnerable data points concerning the In-Degree and Coreness Value

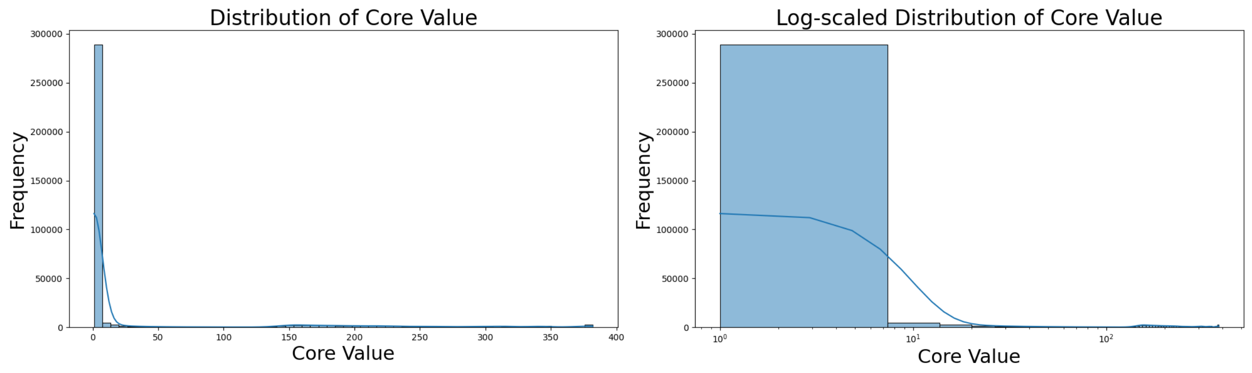

Figure 5.12. Distribution of Coreness Value in Build Stage

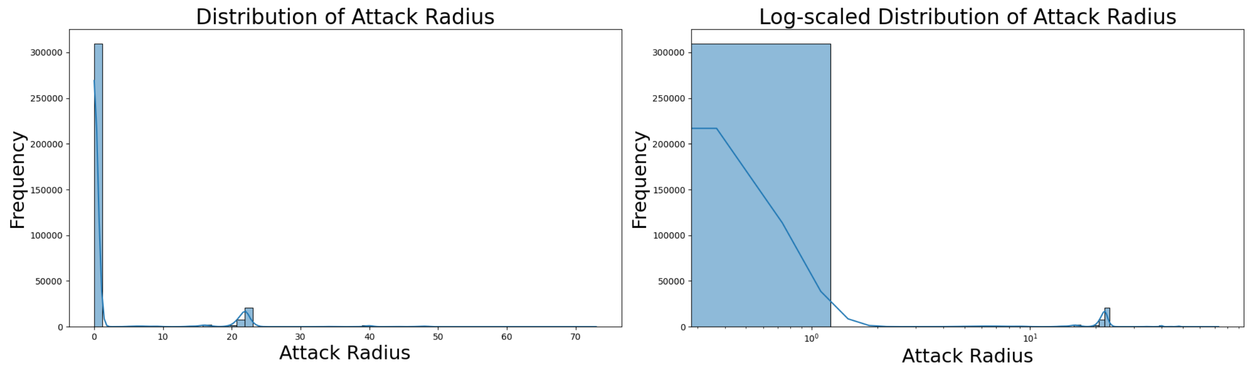

Figure 5.13. Distribution of Attack Radius in Build Stage

metrics. Vulnerable points tend to cluster towards the higher end of the In-Degree metric, suggesting that vulnerable data often resides in nodes with a high number of incoming edges. The correlation between attack radius and path length is moderate, indicating a less direct relationship compared to the source phase. This suggests a more complex network structure in the build phase. A strong positive correlation between attack radius and coreness value reveals that packages at the end of dependency chains are likely central within tightly knit communities, unlike in the source phase where this correlation was weak. Additionally, there is a moderate positive correlation between in-degree and betweenness centrality, indicating that packages with more incoming dependencies play a significant role in information flow in the build phase. The weak negative correlation between attack radius and in-degree suggests that packages at the end of dependency chains have fewer incoming dependencies, similar

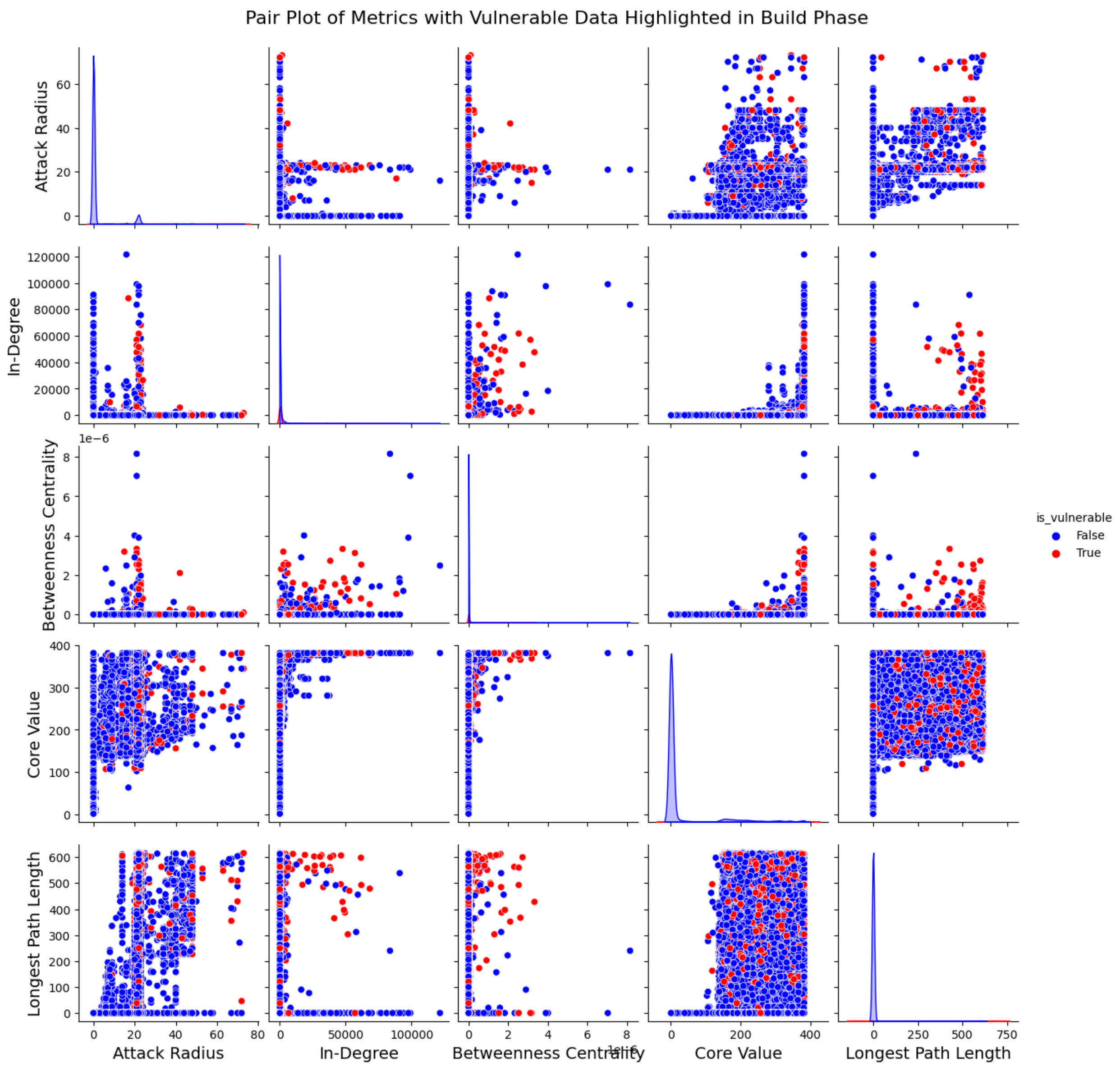

Figure 5.14. Pair plot of Data in Build Stage

to the source phase. The remaining ones exhibit either very weak or no correlation between them. Overall, the build phase network shows a more complex structure.

The box plots shown in figure 5.15 further illustrate the distribution of the vulnerable package. Vulnerable packages exhibit notably higher attack radii compared to non-vulnerable ones, indicating a wider potential impact radius for vulnerabilities. In terms of

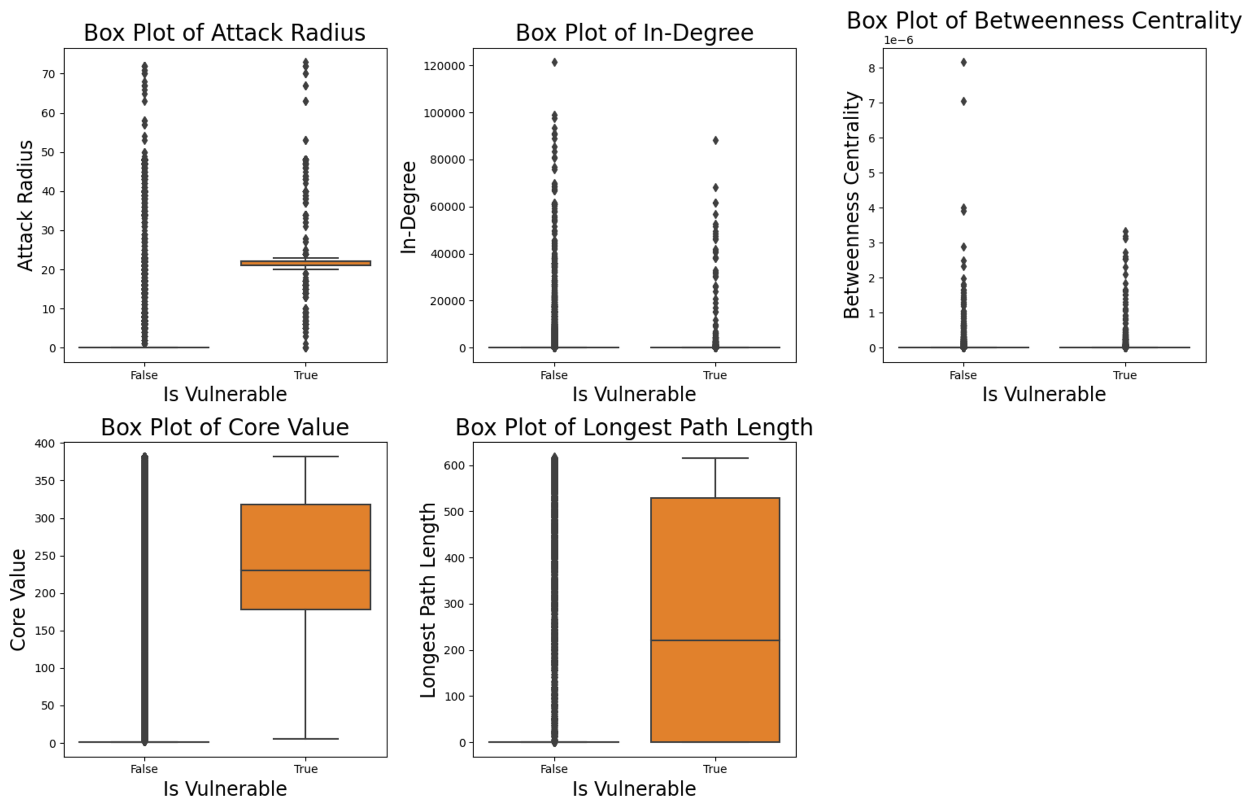

Figure 5.15. Comparison of medians between the vulnerable and non-vulnerable data of risk metric

in-degree and betweenness centrality, vulnerable and non-vulnerable packages show similar distributions, with most packages having low values but with outliers in both categories, suggesting that in-degree and betweenness centrality alone may not be a strong indicator of vulnerability. Coreness value, is significantly higher for vulnerable packages, indicating that they are more central to the network structure. This suggests that vulnerabilities in these packages could have a more widespread impact. Additionally, vulnerable packages tend to have longer path lengths, indicating they are part of longer dependency chains within the network, potentially making them more susceptible to cascading failures. Overall, focusing on metrics such as attack radius, coreness value, and longest path length could be more effective in identifying and prioritizing vulnerable packages within a network.

Table 5.4. Number of Past Vulnerability Packages found at Different Thresholds of Outliers for each Risk Metric in Build

Risk Metric	**# Outliers (5%)**	**# Outliers (10%)**	**# Outliers (15%)**	**# Outliers (20%)**
Attack Radius	10	98	3466	3466
In-degree	397	741	3369	3369
Betweenness Centrality	282	282	282	282
Coreness Value	0	0	3466	3471
Path Length	1925	1925	1925	1925

Table 5.4 summarizes the number of past vulnerability packages (a total of 3539 vulnerable packages) identified as outliers at different thresholds for each risk metric in the build phase. Attack Radius and In-Degree show increased outliers with higher thresholds, indicating vulnerabilities in packages with wider attack radii and high incoming connections. Betweenness Centrality has consistent outliers, indicating stable identification of critical vulnerabilities. Coreness Value has no outliers at lower thresholds but spikes at higher thresholds, suggesting vulnerability concentration in central network positions. Path Length has consistent outliers, implying vulnerabilities in packages with long dependency paths are evenly distributed. Critical nodes with high connectivity or incoming dependencies should be scrutinized for stricter security measures, given their potential vulnerability.

These findings suggest the potential existence of critical nodes or packages within the network. These critical nodes, due to their high connectivity or large number of incoming dependencies, might warrant closer scrutiny and potentially stricter security and maintenance practices. The presence of vulnerable packages within these outlier groups highlights the importance of these metrics in pinpointing potential weak spots within the OSS supply chain.

Our analysis shown in figure 5.16 revealed another interesting finding regarding the relationship between package popularity metrics (`inst` and the risk profiles. The build phase, reveals that high-`inst` or popular packages are more influential and central within the network. They exhibit higher connectivity (in-degree), greater reach (attack radius), stronger sub-graph membership (K-core index), and longer propagation paths (path length). These characteristics suggest that popular packages are critical nodes in the network, and their security and stability are crucial for maintaining network resilience.

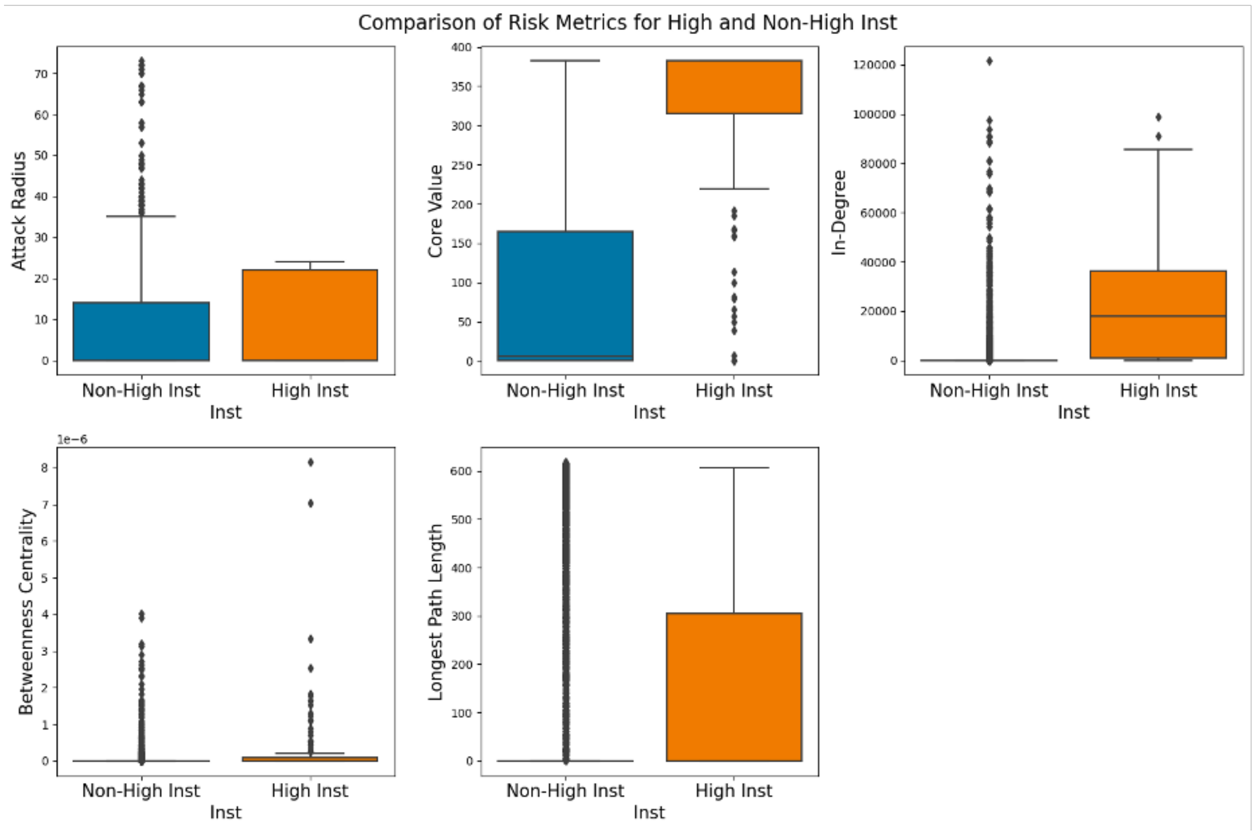

Figure 5.16. Risk Profile of High Popular Package in Build

5.2 Explorative and Inferential Statistics Data Analysis

Table 5.5. Number of Packages with Higher Risk Metrics Values in the Source Stage Compared to the Build Stage

Risk Metric	Number of Packages (Source_value - Build_value > 0)
In-Degree	10,685
Coreness Value	7,506
Betweenness Centrality	186
Attack Radius	1
Path Length	0

Our comparative analysis of risk profiles between the source and build stages in the Open Source Software (OSS) supply chain reveals insights into the effectiveness of risk reduction measures. With this analysis, we aim to answer our RQ2.

Table 5.5 presents the number of packages with higher risk metric values in the source stage compared to the build stage. The source phase exhibits packages with higher in-degree and coreness values, and some with higher betweenness centrality compared to the build phase. This indicates that some of the packages in the source phase rely more on dependencies, potentially increasing vulnerability and suggesting tighter-knit communities. In contrast, the build phase shows similar attack radius and path length metrics, suggesting comparable vulnerability reach and network depth. However, the lower in-degree, betweenness centrality, and coreness value metrics in the build phase, can suggest that the software developers are trying to reduce interconnectedness ie to reduce dependencies to lower the vulnerability risk of the packages.

The effectiveness of these measures in enhancing the security and resilience of OSS ecosystems is evident in the build stage's reduced risk profile for some packages.

Our initial consideration of parametric tests like t-tests and ANOVA was well-suited for data with normal distributions. However, due to the skewed nature of our data on network properties and vulnerability, non-parametric tests provided a more appropriate analysis method.

The Chi-Square test (McHugh, 2013) assessed deviations from expected data distributions, while the Wilcoxon signed-rank test (Rosner et al., 2006) compared network metrics across the two phases, revealing significant connections. Table 5.6 summarizes the statistical test results:

This analysis in table 5.6 delves into the correlation between risk metrics for packages in the source and build phases, revealing insightful patterns in their behavior. Attack Radius demonstrates a strong positive correlation, indicating that packages with higher risk in the source phase tend to maintain that risk level in the build phase. Path Length and Coreness Value show weaker positive correlations, suggesting some consistency in risk profiles across phases. In contrast, In-Degree and Betweenness Centrality exhibit minimal to no correlation, implying these metrics may undergo more significant changes between phases.

The high chi-square and Wilcoxon test values suggest significant deviations from expected values by chance and significant differences in metric distributions between the source and build phases, respectively. The low p-values further support these findings, indicating a lack of association between the variables. This analysis underscores the importance of considering the continuity of risk analysis between phases for effective vulnerability management. It highlights that the behavior of packages, in terms of risk metrics, varies significantly, indicating that relying on a single phase to determine the nature of a package may not be sufficient for comprehensive risk assessment.

Table 5.6. Statistical Test Results

Metric	Correlation	Chi-Square Test (χ^2, p)	Wilcoxon Test (Statistic, p-value)
Attack Radius	0.808	χ^2 = 46100.40, p=0.0	Statistic: 4.5, p=0.0
Path Length	0.358	χ^2 = 11397.524, p=0.0	Statistic: 175360051.0, p=0.0
Coreness Value	0.223	χ^2 = 57050.29, p=0.0	Statistic: 42717711.5, p=0.0
In-Degree	0.116	χ^2 = 16582236.99, p=0.0	Statistic: 117737555.5, p=0.0
Betweenness Centrality	0.018	χ^2 = 2408643.50, p=0.0	Statistic: 59930.0, p=1.0e-247

5.3 Modeling Network-Based Risk Profiles for Predicting Vulnerable Packages

Our research aims to assess the predictive power of network-based risk metrics, particularly focusing on hidden dependencies, in determining potential future package vulnerabilities (RQ3). The performance metrics of various machine learning models in predicting a vulnerable package on source data are summarized in Table 5.7. Logistic Regression, Decision Tree, and Random Forest models were initially evaluated by applying SMOTE and without cross-validation. Logistic Regression achieved a decent accuracy of 0.75 but had a very low precision of 0.13, indicating a high number of false positives. The Decision Tree model showed a lower accuracy of 0.65 with both precision and F1 Score being quite low, suggesting poor overall performance. The Random Forest model, before cross-validation, had an accuracy of 0.73 but also suffered from low precision and F1 Score.

Significant improvements were observed when cross-validation was applied. The Random Forest model's performance metrics notably improved, achieving an accuracy of 0.85 and, a precision of 0.80 making it the best-performing model. The Gradient Boost and XG-

Table 5.7. Different Model Values in Predicting Vulnerable Packages on Source Data

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression (Before CV)	0.75	0.13	0.67	0.21
Decision Tree (Before CV)	0.65	0.09	0.65	0.15
Random Forest (Before CV)	0.73	0.09	0.50	0.16
Random Forest (After CV)	0.85	0.80	0.95	0.87
Gradient Boost (After CV)	0.80	0.75	0.91	0.82
XGBoost (After CV)	0.82	0.77	0.94	0.84

Boost models also performed well after cross-validation. These results indicate that applying cross-validation significantly enhances model performance by improving generalizability and robustness, as reflected in the improved precision, recall, and F1 scores.

The feature importance analysis of the Random Forest model showed in table 5.8, the best-performing model after cross-validation, reveals insights into the factors influencing the prediction of vulnerable packages. Among the features, Coreness Value emerges as the most crucial, with an importance score of 0.362, indicating its significant impact on the model's predictions. In-degree follows closely with a score of 0.254, suggesting its strong influence on the target variable. Attack Radius and Path Length also play important roles, with importance scores of 0.199 and 0.185, respectively. However, Betweenness Centrality shows minimal importance (0.00004), suggesting it has little effect on the model's predictive power.

The table 5.9 provides insights into the performance and feature importance of machine learning models in predicting vulnerable packages on build data. Initially, Logistic Regression, Decision Tree, and Random Forest models were evaluated without cross-validation. Logistic Regression achieved the highest accuracy of 0.91, indicating its ability to correctly classify non-vulnerable and vulnerable packages. However, its low precision of 0.09 suggests a high rate of false positives. The Decision Tree model, with an accuracy of 0.96, exhibited similar challenges with low precision and recall, indicating limitations in correctly identifying vulnerable packages. The Random Forest model, despite its accuracy of 0.97, also faced challenges with low precision and recall.

Table 5.8. Feature Importances of Random Forest Model After Applying Cross-Validation in Source

Model	Feature Importance
Attack Radius	0.199411
In-Degree	0.253842
Betweenness Centrality	0.000038
Coreness Value	0.361796
Path Length	0.184912

Table 5.9. Different Model Values in Predicting Vulnerable Packages on Build Data

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression (Before CV)	0.91	0.09	0.97	0.17
Decision Tree (Before CV)	0.96	0.11	0.37	0.17
Random Forest (Before CV)	0.97	0.12	0.38	0.18
Random Forest (After CV)	0.98	0.97	0.99	0.98
Gradient Boost (After CV)	0.95	0.92	1.00	0.96
XGBoost (After CV)	0.96	0.93	1.00	0.96

Table 5.10. Feature Importances of Random Forest Model After Applying Cross-Validation in Build

Model	Feature Importance
Attack Radius	0.419480
In-Degree	0.119270
Betweenness Centrality	0.000268
Coreness Value	0.372007
Path Length	0.088974

After applying cross-validation, substantial performance improvements were observed across all models. The Random Forest model notably excelled, achieving an accuracy of 0.98, a precision of 0.97, and a recall of 0.99, resulting in an impressive F1 Score of 0.98. This suggests that the model effectively identifies vulnerable packages while minimizing false positives. The Gradient Boost and XGBoost models also demonstrated improved performance after cross-validation, with high accuracy, precision, recall, and F1 scores.

In terms of feature importance 5.10, the Random Forest model identified Coreness Value as the most critical feature, followed by Attack Radius and In-Degree. Path Length and Betweenness Centrality also contributed to the model's predictions, albeit to a lesser extent. These findings highlight the significance of certain network metrics in determining the vulnerability of software packages, underscoring the importance of incorporating these metrics into vulnerability prediction models for improved software security practices.

Table 5.11. Different Model Values in Predicting Vulnerable Packages on Combined Data

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression (Before CV)	0.64	0.12	0.84	0.21
Decision Tree (Before CV)	0.87	0.17	0.34	0.23
Random Forest (Before CV)	0.88	0.22	0.39	0.28
Random Forest (After CV)	0.93	0.91	0.95	0.93
Gradient Boost (After CV)	0.82	0.80	0.85	0.82
XGBoost (After CV)	0.87	0.84	0.90	0.87

Table 5.12. Feature Importances of Random Forest Model After Applying Cross-Validation in Total Data

Feature	Importance
Attack Radius_source	0.041116
In-Degree_source	0.048251
Betweenness Centrality_source	0.000014
Coreness Value_source	0.104563
Path Length_source	0.055784
Attack Radius_build	0.173814
In-Degree_build	0.070409
Betweenness Centrality_build	0.001511
Coreness Value_build	0.303393
Path Length_build	0.201144

The performance metrics of various machine learning models in predicting vulnerable packages on combined data are summarized in Table 5.11. Logistic Regression, Decision Tree, and Random Forest models were initially evaluated without cross-validation. Logistic Regression achieved an accuracy of 0.64, with a precision of 0.12, indicating a high number of false positives, despite its high recall of 0.84. The Decision Tree model performed better in terms of accuracy (0.87) but still exhibited low precision (0.17) and a moderate F1 Score (0.23), indicating it struggled to balance precision and recall. The Random Forest model, before cross-validation, showed an accuracy of 0.88 but also faced challenges with lower precision (0.22) and recall (0.39), leading to an F1 Score of 0.28.

Significant improvements were observed when cross-validation was applied. The Random Forest models performance metrics improved substantially, achieving an accuracy of 0.93, precision of 0.91, recall of 0.95, and an F1 Score of 0.93, making it the best-performing model. The Gradient Boost model, after cross-validation, also showed good performance with an accuracy of 0.82 and balanced precision (0.80) and recall (0.85), resulting in an F1 Score of 0.82. XGBoost, after cross-validation, achieved high performance across all metrics with an accuracy of 0.87, precision of 0.84, recall of 0.90, and an F1 Score of 0.87. These results indicate that applying cross-validation significantly enhances model performance by improving generalizability and robustness.

The feature importance using Mean Decrease in Impurity (MDI) (Breiman et al., 2017) analysis of the Random Forest model, as shown in Table 5.12 and illustrated in the plot 5.17, provides valuable insights into the factors influencing the prediction of vulnerable packages. Coreness Value_build emerged as the most crucial feature, with an importance score of 0.303, indicating its significant impact on the models predictions. This is followed by Path Length_build (0.201) and Attack Radius_build (0.174), underscoring the critical role of build environment metrics in determining package vulnerability. Among the source data features, Coreness Value_source (0.105) and Path Length_source (0.056) are notable, albeit less influential than their build counterparts. Attack Radius_source and In-Degree_source show moderate importance, while Betweenness Centrality_source (0.000014) and Betweenness Centrality_build display minimal impact, suggesting their limited predictive power. The error bars indicate variability in importance scores, with features like Attack Radius_build showing higher variability, suggesting their influence may fluctuate across different data subsets.

The plot 5.18 further complements this analysis by providing a different perspective on feature impact using permutation importances (Breiman, 2001). In this analysis, Path Length_source has the highest mean accuracy decrease when permuted, indicating its sub-

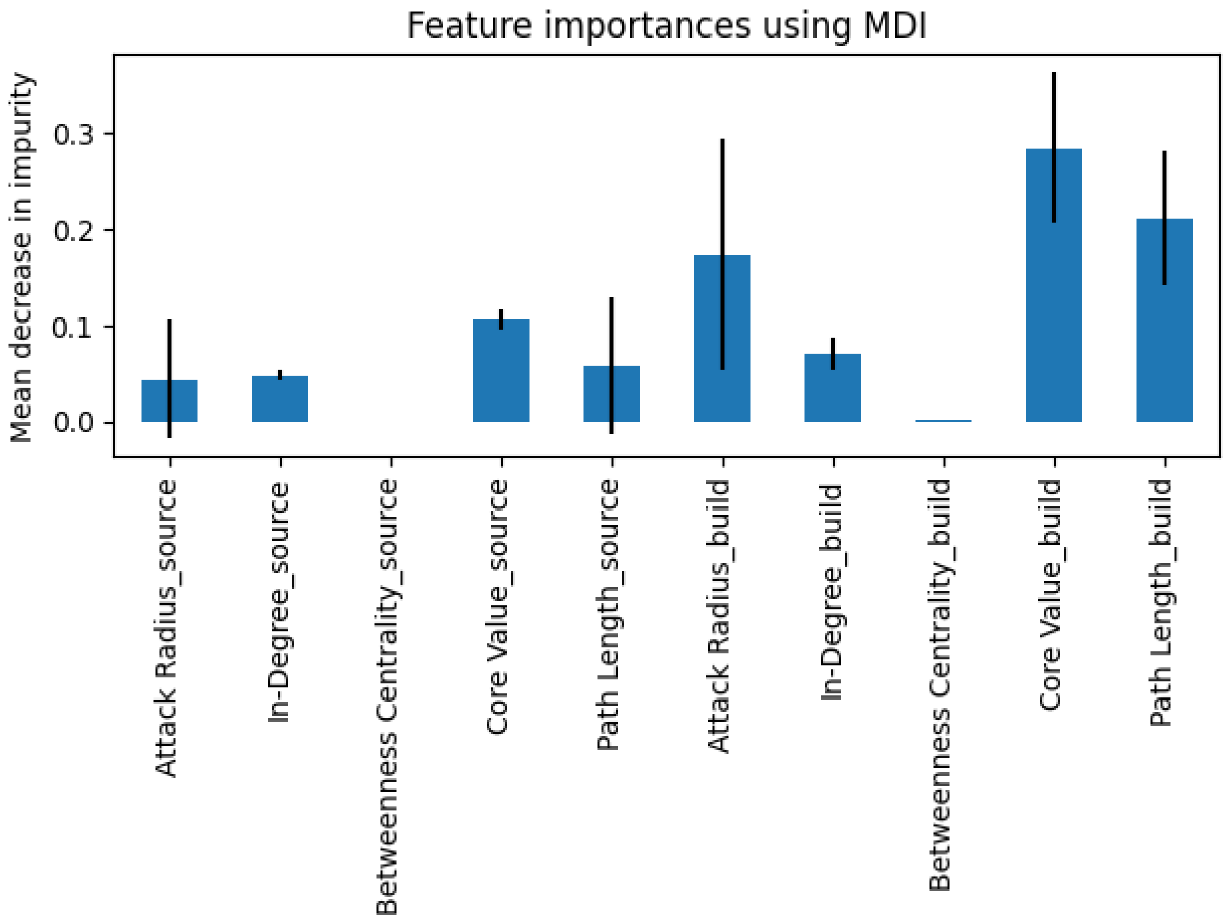

Figure 5.17. Feature Importances Using Mean Decrease in Impurity (MDI)

stantial influence on model accuracy. Conversely, Attack Radius_build shows a negative mean accuracy decrease, suggesting that its permutation might improve model performance or that it introduces noise.

Overall, the combined analyses highlight that build-related metrics, particularly Coreness Value_build and Path Length_build, are paramount in predicting package vulnerabilities. Additionally, Path Length_source plays a crucial role as per the permutation analysis. Traditional methods often neglect the `build` stage and the intricate relationships between packages within a software ecosystem.

By incorporating `"hidden"` dependencies information – direct and indirect dependencies traced through coreness value and path length of packages – we gain a more holistic

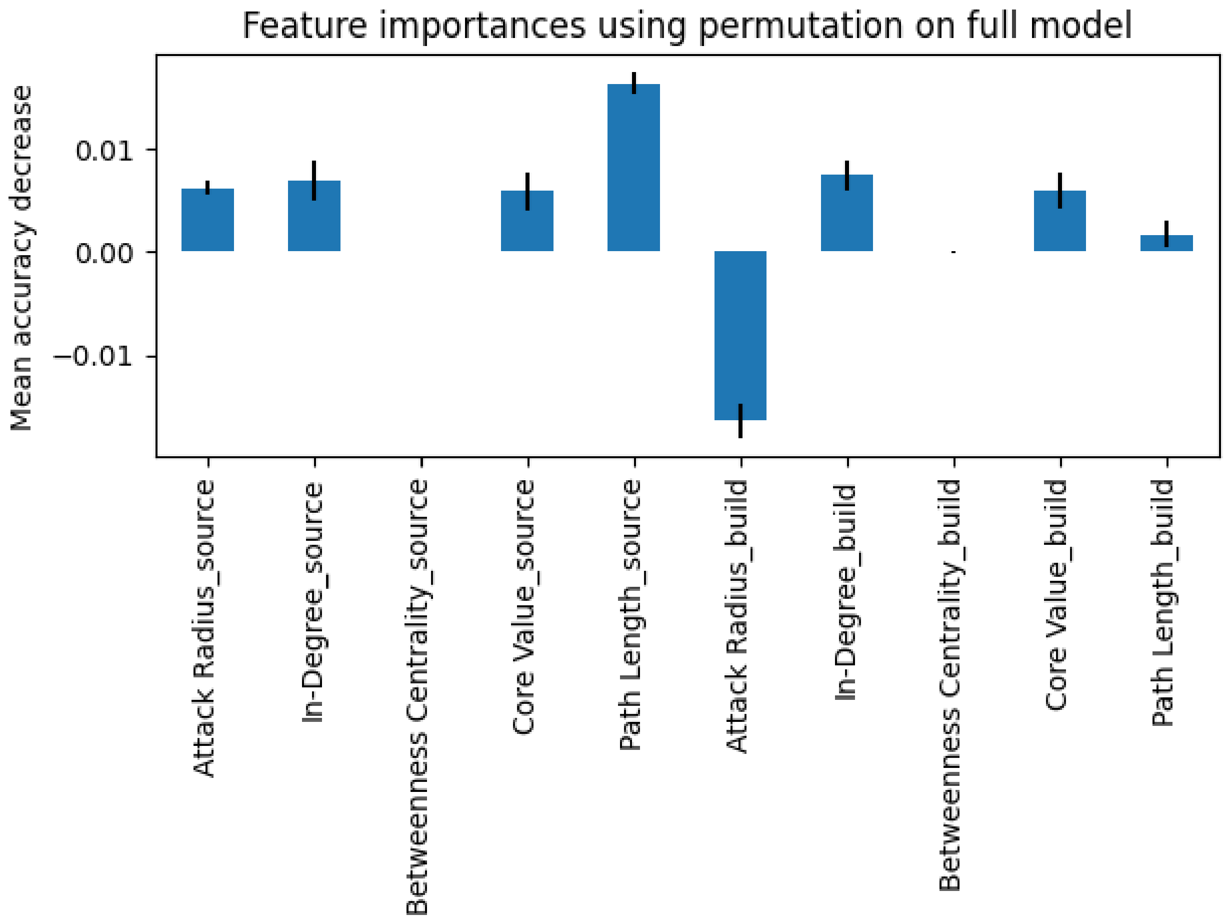

Figure 5.18. Feature Importances Using Permutation Importance

understanding of a package's vulnerability based on its position and complexity within the network. This approach highlights how these hidden dependencies can significantly impact a package's security profile, providing a nuanced perspective that traditional models might miss.

These findings emphasize the necessity of focusing on these metrics to enhance software security assessments. By leveraging impurity-based and permutation-based methods, we achieve a more comprehensive understanding of feature impacts, offering a robust framework for predicting and mitigating vulnerabilities within software packages. This multifaceted approach ensures a thorough evaluation of potential risks, ultimately contributing to more secure software development practices.

6. LIMITATIONS

While our research contributes valuable insights into the security dynamics of Open-Source Software (OSS) supply chains, it is essential to recognize and address the inherent limitations that shape the scope and generalizability of our findings.

Our study focuses exclusively on Debian packages. While chosen for their representativeness, this specific focus may not fully capture the diverse landscape of all OSS ecosystems. Extrapolating our results to other ecosystems should be approached with caution, considering potential structural variations and distinctive risk profiles that may exist.

The comprehensiveness of our data collection process relies on information available through Debian Salsa and Buildinfos. This introduces potential constraints on the breadth of vulnerability assessment. Moreover, our reliance on publicly disclosed vulnerabilities may introduce bias, given the inherent issues of underreporting or delayed disclosures in such databases.

The chosen time frame for data collection, spanning from January 1, 2017, to December 30, 2022, imposes temporal constraints on our study. This limitation raises the possibility of overlooking emerging trends or shifts in the landscape of the OSS supply chain beyond the specified period, emphasizing the need for cautious interpretation.

While our proposed risk assessment methodology demonstrates robustness, scalability challenges may emerge when applied to larger datasets. As OSS ecosystems continually evolve, adapting the methodology to accommodate the increasing complexity and size of datasets becomes imperative for sustained efficacy in risk assessment.

External factors add another layer of unpredictability to our research landscape. Unforeseen events may exert influence on the dynamics of risk within the OSS supply chain. Adapting our research to account for these external influences and ensuring the resilience of our findings in the face of uncertainties becomes imperative.

Addressing these challenges necessitates a thoughtful and adaptive approach. We acknowledge the constraints and complexities inherent in our research journey and actively work towards mitigating their impact. Flexibility in our methodologies, robust data management strategies, and a nuanced understanding of risk models contribute to overcoming these challenges.

The overarching goal is to ensure the robustness and relevance of our research findings despite these challenges. By acknowledging and addressing these limitations, we strive to enhance the credibility and applicability of our research, contributing valuable insights to the understanding of risk dynamics in the OSS supply chain. This acknowledgment serves as a commitment to the continuous improvement of our research methodology and its responsiveness to the evolving nature of the OSS landscape.

7. FUTURE WORK

Our research has significantly contributed to unraveling the security dynamics of Open-Source Software (OSS) supply chains. As we look ahead, our commitment to addressing challenges and expanding the depth of our contributions remains steadfast, laying the groundwork for future endeavors.

To overcome limitations and extend the scope of our research, we prioritize enhanced data collection. While our current focus centers on the source and build phases, a comprehensive understanding of the risk landscape mandates broadening data collection to encompass all supply chain stages. Including multiple OSS ecosystems will enrich insights and enhance the generalizability of our findings.

Recognizing the importance of temporal analyses, we plan to employ sophisticated time series modeling techniques. Leveraging classical time-series analysis methods, such as autoregressive integrated moving average (ARIMA) models and exponential smoothing, will be integral (Roumani et al., 2015). ARIMA models capture dependencies within temporal data, enhancing predictive capabilities for vulnerabilities and risks. Complementarily, exponential smoothing discerns variations in vulnerability propagation over time, contributing to a nuanced understanding of temporal dynamics.

To overcome temporal constraints, we propose integrating continuous monitoring and real-time analysis. This enhancement facilitates dynamic risk assessments, ensuring prompt identification of emerging threats. Making risk models accessible to industry practitioners and policymakers is a key goal. Developing mechanisms for real-time risk monitoring is a crucial avenue, contributing to proactive risk management and adaptability to evolving OSS supply chain dynamics.

Future research will delve deeper into the human factor in supply chain security. In-depth analyses of developer behaviors, collaboration patterns, and organizational structures will contribute to a more comprehensive risk assessment. Recognizing the intricate interplay of these human-centric factors is imperative for identifying vulnerabilities arising from social and organizational dynamics. These efforts collectively aim to overcome current limitations and propel the field toward a more comprehensive, adaptive, and actionable risk management framework.

Strategic advancements in data collection, real-time analysis, and human-centric factors not only enhance academic rigor but also hold practical significance. Aligning our research with the evolving needs and challenges faced by stakeholders in the OSS supply chain landscape ensures its applicability and impact.

These endeavors collectively aim to overcome current limitations and propel the field toward a more comprehensive, adaptive, and actionable risk management framework. Such strategic advancements not only enhance the academic rigor of our study but also hold practical significance, aligning our research with the evolving needs and challenges faced by stakeholders in the OSS supply chain landscape.

8. CONCLUSION

In conclusion, this study underscores the critical role of network analysis and machine learning in enhancing vulnerability management within the open-source software (OSS) supply chain. The research reveals that each risk profile is distinct and behaves differently at various phases of the software development lifecycle, necessitating a focused approach for each.

By emphasizing packages with high coreness values and longer path lengths, organizations can better prioritize their risk management efforts. This targeted focus enables a more effective allocation of resources toward the most vulnerable components, thereby enhancing overall security.

The findings of this study are poised to significantly influence the industry's approach to managing vulnerabilities in OSS supply chains. Organizations can leverage network analysis and machine learning models to proactively identify and mitigate vulnerabilities in their OSS dependencies. This proactive stance not only strengthens the security and resilience of their software development processes but also fosters a more robust defense against potential threats.

The research provides practical guidance for prioritizing risk management efforts, suggesting a focus on packages with high coreness values, longer path lengths, and high popularity. By doing so, companies can develop targeted security strategies that address vulnerabilities more effectively. This approach not only improves individual software security but also contributes to a more secure and robust software ecosystem.

Moreover, the study has the potential to shape industry practices and policies related to OSS supply chain risk management. By demonstrating the effectiveness of network analysis and machine learning, the study encourages broader adoption of these approaches, leading to industry-wide improvements in security practices and ultimately fostering more secure and resilient OSS ecosystems.

The research introduces the concept of "hidden dependency," using coreness and path length metrics to enhance the understanding of vulnerability spread in interconnected OSS networks. This provides a comprehensive view of how vulnerabilities propagate, offering valuable insights for risk assessment.

Furthermore, the study validates the importance of established network metrics such as core value and path length across different development stages. This validation provides actionable insights for developing effective risk assessment methodologies tailored to the OSS environment.

Lastly, the research showcases the potential of machine learning models in predicting vulnerabilities within OSS supply chains. This highlights the power of machine learning as a tool for enhancing security practices in the OSS community.

Overall, this research makes significant contributions to understanding and managing vulnerabilities in OSS supply chains. It provides practical tools and insights for improving security practices in the industry, advancing our collective ability to safeguard OSS ecosystems.

REFERENCES

Aksulu, A., & Wade, M. (2010). A Comprehensive Review and Synthesis of Open Source Research [Num Pages: 81 Place: Atlanta, United States Publisher: Association for Information Systems Section: Special Issue]. *Journal of the Association for Information Systems*, *11*(11), 576–656.

Alexopoulos, N., Egert, R., Grube, T., & Mühlhäuser, M. (2019). Poster: Towards Automated Quantitative Analysis and Forecasting of Vulnerability Discoveries in Debian GNU/Linux. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2677–2679. https://doi.org/10.1145/3319535.3363285

Allombert, B. (2022). Debian Popularity Contest.

Anthony, M., & Holden, S. B. (1998). Cross-validation for binary classification by real-valued functions: Theoretical analysis. *Proceedings of the eleventh annual conference on Computational learning theory*, 218–229. https://doi.org/10.1145/279943.279987

Arora, A., Krishnan, R., Telang, R., & Yang, Y. (2010). An Empirical Analysis of Software Vendors' Patch Release Behavior: Impact of Vulnerability Disclosure. *Information Systems Research*, *21*(1), 115–132. https://doi.org/10.1287/isre.1080.0226

Aynaud, T. (2022). Community detection for NetworkX Documentation.

Baldwin, C., MacCormack, A., & Rusnak, J. (2014). Hidden structure: Using network methods to map system architecture. *Research Policy*, *43*(8), 1381–1397. https://doi.org/10.1016/j.respol.2014.05.004

Bang, J. (2008). *Digraphs: Theory, algorithms and applications*. Springer Science & Business Media.

Bendali, F., Mailfert, J., & Mameri, D. (2016). The minimum weakly connected independent set problem: Polyhedral results and BranchandCut. *Discrete Optimization*, *22*, 87–110. https://doi.org/10.1016/j.disopt.2016.04.006

Bender, E. A. (2010). *Lists, decisions and graphs*. S. Gill Williamson.

Borgatti, S. P., & Everett, M. G. (2006). A Graph-theoretic perspective on centrality. *Social Networks*, *28*(4), 466–484. https://doi.org/10.1016/j.socnet.2005.11.005

Brandes, U. (2001). A faster algorithm for betweenness centrality*. *The Journal of Mathematical Sociology*, *25*(2), 163–177. https://doi.org/10.1080/0022250X.2001.9990249

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*(1), 5–32. https://doi.org/10.1023/A:1010933404324

Breiman, L., Friedman, J., Olshen, R. A., & Stone, C. J. (2017, October). *Classification and Regression Trees*. Chapman; Hall/CRC. https://doi.org/10.1201/9781315139470

Brunswicker, S. (2022). DASS: Increasing Collective Accountability in OSS Supply-Chain Networks with Algorithmic and Peer-based Transparency.

Brunswicker, S., & Mukherjee, S. (2023). The Microstructure of Modularity in Design: A Design Motif View. *Industrial Corporate Change*.

Champion, K., & Hill, B. M. (2021). Underproduction: An Approach for Measuring Risk in Open Source Software [arXiv: 2103.00352]. *CoRR*, *abs/2103.00352*.

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System [arXiv:1603.02754 [cs]]. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. https://doi.org/10.1145/2939672.2939785

Constantin, L. (2020). SolarWinds attack explained: And why it was so hard to detect.

Corbly, J. E. (2014). The Free Software Alternative: Freeware, Open Source Software, and Libraries. *Information Technology and Libraries*, *33*(3), 65. https://doi.org/10.6017/ital.v33i3.5105

Corneil, D. G., Dragan, F. F., & Kohler, E. (2003). On the Power of BFS to Determine a Graphs Diameter.

Dashevskyi, S., Brucker, A. D., & Massacci, F. (2019). A Screening Test for Disclosed Vulnerabilities in FOSS Components [Conference Name: IEEE Transactions on Software Engineering]. *IEEE Transactions on Software Engineering*, *45*(10), 945–966. https://doi.org/10.1109/TSE.2018.2816033

Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., & Lee, W. (2020, December). Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages [arXiv:2002.01139 [cs]].

Editor, C. C. (2022). Vulnerability - Glossary

CSRC.

Elizalde Zapata, R., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., & Ihara, A. (2018). Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages [ISSN: 2576-3148]. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 559–563. https://doi.org/10.1109/ICSME.2018.00067

Fitzgerald, B. (2006). The Transformation of Open Source Software [Publisher: MIS Quarterly]. *MIS Quarterly*, *30*(3), 587–598. https://doi.org/10.2307/25148740

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, *38*(4), 367–378. https://doi.org/10.1016/S0167-9473(01)00065-2

Han, H., Wang, W.-Y., & Mao, B.-H. (2005). Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning. In D.-S. Huang, X.-P. Zhang, & G.-B. Huang (Eds.), *Advances in Intelligent Computing* (pp. 878–887). Springer. https://doi.org/10.1007/11538059_91

Holdsworth, J. (1999). The Nature of Breadth-First Search.

Inc, S. (2023). Sonatype's 2021 Software Supply Chain Report.

ISO. (2022). *ISO/IEC 27005: Information Technology, Security Techniques, Information Security Risk Management.*

Jafari, A. J., Costa, D. E., Abdalkareem, R., Shihab, E., & Tsantalis, N. (2022). Dependency Smells in JavaScript Projects [Conference Name: IEEE Transactions on Software Engineering]. *IEEE Transactions on Software Engineering*, *48*(10), 3790–3807. https://doi.org/10.1109/TSE.2021.3106247

Jang-Jaccard, J., & Nepal, S. (2014). A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences*, *80*(5), 973–993. https://doi.org/10.1016/j.jcss.2014.02.005

Jena, B. K. (2023). SolarWinds Attack & Details You Need To Know About It `Simplilearn`.

Jr, D. W. H., Lemeshow, S., & Sturdivant, R. X. (2013, February). *Applied Logistic Regression* [Google-Books-ID: bRoxQBIZRd4C]. John Wiley & Sons.

Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017). Structure and Evolution of Package Dependency Networks. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 102–112. https://doi.org/10.1109/MSR.2017.55

Kluban, M., Mannan, M., & Youssef, A. (2022). On Measuring Vulnerable JavaScript Functions in the Wild. *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 917–930. https://doi.org/10.1145/3488932.3497769

Kong, Y.-X., Shi, G.-Y., Wu, R.-J., & Zhang, Y.-C. (2019). K-core: Theories and applications. *Physics Reports*, *832*, 1–32. https://doi.org/10.1016/j.physrep.2019.10.004

Lawyer, G. (2015). Understanding the influence of all nodes in a network [Number: 1 Publisher: Nature Publishing Group]. *Scientific Reports*, *5*(1), 8665. https://doi.org/10.1038/srep08665

Liang, G., Zhou, X., Wang, Q., Du, Y., & Huang, C. (2021). Malicious Packages Lurking in User-Friendly Python Package Index [ISSN: 2324-9013]. *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 606–613. https://doi.org/10.1109/TrustCom53373.2021.00091

Lin, J., Zhang, H., Adams, B., & Hassan, A. E. (2023). Vulnerability management in Linux distributions. *Empirical Software Engineering*, *28*(2), 47. https://doi.org/10.1007/s10664-022-10267-7

Lü, L., Zhou, T., Zhang, Q.-M., & Stanley, H. E. (2016). The H-index of a network node and its relation to degree and coreness. *Nature Communications*, *7*(1), 10168. https://doi.org/10.1038/ncomms10168

MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, *52*(7), 1015–1030. https://doi.org/10.1287/mnsc.1060.0552

MacCormack, A., & Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, *120*, 170–182. https://doi.org/10.1016/j.jss.2016.06.007

Mahmood, M. A., Siponen, M., Straub, D., Rao, H. R., & Raghu, T. S. (2010). Moving Toward Black Hat Research in Information Systems Security: An Editorial Introduction to the Special Issue [Publisher: MIS Quarterly]. *MIS Quarterly*, *34*(3), 431–433. https://doi.org/10.2307/25750685

McHugh, M. L. (2013). The Chi-square test of independence. *Biochemia Medica*, *23*(2), 143–149. https://doi.org/10.11613/BM.2013.018

Mitra, S., & Ransbotham, S. (2015). Information Disclosure and the Diffusion of Information Security Attacks. *Information Systems Research*, *26*(3), 565–584. https://doi.org/10.1287/isre.2015.0587

Jena, B. K. (2023). OSSRA Report.

Pashchenko, I., Vu, D.-L., & Massacci, F. (2020). A Qualitative Study of Dependency Management and Its Security Implications. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 1513–1531. https://doi.org/10.1145/3372297.3417232

Jena, B. K. (2016, January). Paths, Diffusion, and Navigation. In A. Fornito, A. Zalesky, & E. T. Bullmore (Eds.), *Fundamentals of Brain Network Analysis* (pp. 207–255). Academic Press. https://doi.org/10.1016/B978-0-12-407908-3.00007-8

Polat, K., & Güne, S. (2009). A novel hybrid intelligent method based on C4.5 decision tree classifier and one-against-all approach for multi-class classification problems. *Expert Systems with Applications*, *36*(2), 1587–1592. https://doi.org/10.1016/j.eswa.2007.11.051

Ponta, S. E., Plate, H., & Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, *25*(5), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

Putri, S. E., Tulus, T., & Napitupulu, N. (2011). Implementation and Analysis of Depth-First Search (DFS) Algorithm for Finding The Longest Path. *Simplilearn.com*. https://doi.org/10.13140/2.1.2878.2721

Reid, D., Jahanshahi, M., & Mockus, A. (2022). The extent of orphan vulnerabilities from code reuse in open source software. *Proceedings of the 44th International Conference on Software Engineering*, 2104–2115. https://doi.org/10.1145/3510003.3510216

Jena, B. K. (2022). ReproducibleBuilds - Debian Wiki.

Rogers, T. (2015). Assessing node risk and vulnerability in epidemics on networks [Publisher: EDP Sciences, IOP Publishing and Società Italiana di Fisica]. *Europhysics Letters*, *109*(2), 28005. https://doi.org/10.1209/0295-5075/109/28005

Rosner, B., Glynn, R. J., & Lee, M.-L. T. (2006). The Wilcoxon signed rank test for paired comparisons of clustered data. *Biometrics*, *62*(1), 185–192. https://doi.org/10.1111/j.1541-0420.2005.00389.x

Roumani, Y., Nwankpa, J. K., & Roumani, Y. F. (2015). Time series modeling of vulnerabilities. *Computers & Security*, *51*, 32–40. https://doi.org/10.1016/j.cose.2015.03.003

Sabbagh, B. A., & Kowalski, S. (2015). A Socio-technical Framework for Threat Modeling a Software Supply Chain [Conference Name: IEEE Security & Privacy]. *IEEE Security & Privacy*, *13*(4), 30–39. https://doi.org/10.1109/MSP.2015.72

Saleh, M., Esa, Y., & Mohamed, A. (2018). Applications of Complex Network Analysis in Electric Power Systems. *Energies*, *11*(6), 1381. https://doi.org/10.3390/en11061381

Schueller, W., & Wachs, J. (2024). Modeling interconnected social and technical risks in open source software ecosystems [Publisher: SAGE Publications]. *Collective Intelligence*, *3*(1), 26339137241231912. https://doi.org/10.1177/26339137241231912

Seidman, S. B. (1983). Network structure and minimum degree. *Social Networks*, *5*(3), 269–287. https://doi.org/10.1016/0378-8733(83)90028-X

Sejfia, A., & Schäfer, M. (2022). Practical Automated Detection of Malicious npm Packages [arXiv:2202.13953 [cs]]. *Proceedings of the 44th International Conference on Software Engineering*, 1681–1692. https://doi.org/10.1145/3510003.3510104

Jena, B. K. (2022). SimplePackagingTutorial.

Sprint, A. (2023, November). Explore projects ů GitLab.

St Laurent, A. M. (2008). *Understanding Open Source and Free Software Licensing* [OCLC: 609841443]. O'Reilly Media, Inc.

Stuckman, J., & Purtilo, J. (2014). Mining Security Vulnerabilities from Linux Distribution Metadata. *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, 323–328. https://doi.org/10.1109/ISSREW.2014.101

Temizkan, O., Park, S., & Saydam, C. (2017). Software Diversity for Improved Network Security: Optimal Distribution of Software-Based Shared Vulnerabilities. *Information Systems Research*, *28*(4), 828–849. https://doi.org/10.1287/isre.2017.0722

Torres-Arias, S. (2024). DebTrace: A Framework and Techniques to Trace Artifacts in the Debian Software Supply Chain.

Ukachi, N. B., Nwachukwu, V. N., & Onuoha, U. D. (2014). Library Automation and Use of Open Source Software to Maximize Library Effectivenss. *Library of Progress-Library Science, Information Technology & Computer*, *34*(2), 97–111. https://doi.org/10.5958/2320-317X.2014.00002.6

Vasilakis, N., Benetopoulos, A., Handa, S., Schoen, A., Shen, J., & Rinard, M. C. (2021). Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 1755–1770. https://doi.org/10.1145/3460120.3484736

Wang, J., Xu, M., Wang, H., & Zhang, J. (2006). Classification of Imbalanced Data by Using the SMOTE Algorithm and Locally Linear Embedding [ISSN: 2164-523X]. *2006 8th international Conference on Signal Processing*, *3*. https://doi.org/10.1109/ICOSP.2006.345752

Xue, L., Zhang, C., Ling, H., & Zhao, X. (2013). Risk Mitigation in Supply Chain Digitization: System Modularity and Information Technology Governance. *Journal of Management Information Systems*, *30*(1), 325–352. https://doi.org/10.2753/MIS0742-1222300110

Zacchiroli, S. (2015). The Debsources Dataset: Two Decades of Debian Source Code Metadata. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 466–469. https://doi.org/10.1109/MSR.2015.65

Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., & Williams, L. (2022). What are weak links in the npm supply chain? *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 331–340. https://doi.org/10.1145/3510457.3513044

Zanardi, E. (2022). Index of /ftp-master.debian.org/buildinfo.

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *Simplilearn.com*, 995–1010.

Zimmermann, T., & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. *Proceedings of the 30th international conference on Software engineering*, 531–540. https://doi.org/10.1145/1368088.1368161