Research Commons

https://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Investigating RubyGems Packages for Software Supply Chain Attack Susceptibility Through Socio-Technical Metadata

A Thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Engineering

at

The University of Waikato

by

Shilpa Nair

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato

2026

Abstract

Software supply chain attacks targeting open-source package ecosystems have increasingly become prevalent, where the compromise of a single third-party dependency can affect many downstream applications. Existing approaches to supply chain security are largely reactive, focusing on known vulnerabilities rather than identifying packages that are inherently more susceptible to attack.

This thesis proposes an attacker-centric risk profiling framework that investigates package-level susceptibility using socio-technical metadata, including dependency structure, ecosystem impact, and maintenance activity. A hierarchical composite scoring framework is developed, with metric weights derived using ROC-AUC analysis to produce an interpretable risk score.

The framework is evaluated on 79,000 RubyGems packages, including 380 known malicious instances. Results show that malicious packages are more likely to appear in higher-ranked regions of the score distribution, with consistent enrichment over a random baseline. It also shows that supply chain risk is distributed across a broader high-risk region rather than concentrated at the extreme tail. Quantitative evaluation and case study analysis further show that maintenance and impact-related signals offer the best discrimination between benign and malicious packages. The proposed approach therefore functions as a prioritisation tool, supporting effective allocation of security effort in large-scale ecosystems.

Acknowledgements

I would like to express my sincere gratitude to the following individual and institutions for their invaluable support and contributions to this thesis.

To my supervisor, Vimal Kumar, thank you for your expert guidance, helpful feedback, and constant support throughout my research project. Your mentorship has greatly shaped my work and my growth as a student researcher.

To the University of Waikato, for providing the resources, facilities, and opportunities necessary to pursue my research. The university's commitment to academic excellence and fostering a vibrant research community has been a constant source of motivation.

I am grateful to everyone who has contributed to this thesis. Your support and guidance have been invaluable to my academic and personal growth.

Contents

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Question 3
1.3 Research Approach 4
1.4 Contributions 5
1.5 Thesis Structure 5

2 Background **7**
2.1 Software Supply Chain 7
2.2 Software Supply Chain Security 9
2.2.1 Supply Chain Attack Vectors 9
2.3 Socio-Technical Perspective of Supply Chain Risk 10
2.4 Risk Signals and Indicators 10
2.5 RubyGems Ecosystem 11

3 Literature Review **12**
3.1 Software Supply Chain Security 13
3.1.1 Definition and Evolution of Software Supply Chain Security 13
3.1.2 Historical Incidents as Motivation 14
3.1.2.1 M.E.Doc (NotPetya, 2017) 14
3.1.2.2 CCleaner (Piriform/Avast, 2017) 15
3.1.2.3 event-stream (npm, 2018) 15
3.1.2.4 SolarWinds Orion (SolarWinds, 2020) 15
3.1.2.5 npm ua-parser-js (2021) 16
3.1.2.6 Dependency Confusion (2021, Alex Birsan) 16

3.1.2.7 XZ-Utils (Linux, 2024) 17
3.2 Open Source Package Ecosystems 18
3.2.1 Ecosystem Growth and Software Supply Chain Risks 18
3.2.2 Software Supply Chain Attack Vectors 20
3.2.2.1 Upstream Dependency Compromise (Transitive Dependency Poisoning): 20
3.2.2.2 Typosquatting and Dependency Confusion 21
3.2.2.3 Maintainer Account Takeover and Package Hijacking: 21
3.2.2.4 Malicious Package Injection (Direct Package Poisoning): 22
3.2.2.5 Build Pipeline and CI/CD Compromise: 22
3.2.2.6 Source Code Repository Compromise: 23
3.3 Signal based Risk Assessment 23
3.3.1 Weak-Link Signals Categories 23
3.3.2 Automated Security Metrics (Scorecards and Tools) 25
3.3.3 RubyGems Ecosystem Security Studies 26
3.4 Literature Gaps 28
3.4.1 Research Gaps 28

4 Methodology **30**
4.1 Research Design 30
4.1.1 Selection of RubyGems Ecosystem 30
4.1.2 Study Scope 31
4.1.3 Pre-Compromise Risk Profiling 32
4.2 Data Collection 33
4.2.1 Data Sources 33
4.2.2 Data Preprocessing 34
4.3 Socio-Technical Risk Indicators 35
4.3.1 Metric Definitions and Rationale 35
4.3.1.1 Dependency count 35
4.3.1.2 Dependency Depth 36
4.3.1.3 File Count 37
4.3.1.4 Directory depth 37

4.3.1.5 Lines of Code (LOC) 38
4.3.1.6 Download Count 38
4.3.1.7 Dependents count 39
4.3.1.8 Stars 40
4.3.1.9 Forks 41
4.3.1.10 Commit Gap 42
4.3.1.11 Issues count 43
4.3.1.12 PR(pull request) count 44
4.3.1.13 Contributors count 44
4.3.2 Metric Transformation Pipeline 45
4.4 Composite Risk Scoring framework 47
4.4.1 Hierarchical Aggregation Stages 49
4.4.2 Weighted aggregation for scoring 51
4.4.2.1 Understanding ROC-AUC 52
4.4.2.2 Applying ROC-AUC to Risk Scoring 53
4.4.2.3 Hierarchical Aggregation Scoring 53

5 Evaluation 58
5.1 Risk Distribution Analysis 58
5.1.1 Dataset Overview 58
5.1.2 Metrics Performance and ROC-AUC Based Weighting 59
5.1.2.1 Expected vs Observed Performance 61
5.1.3 Risk Distribution: Malicious vs Benign 63
5.1.3.1 Composite Score Distribution Overview 63
5.1.3.2 Composite Score Density Normalised Comparison . 64
5.1.3.3 Aspect Score Density Normalised Comparison . . . 65
5.2 Tail Concentration and Enrichment Analysis 68
5.2.1 Composite Enrichment Behaviour 69
5.2.2 Aspect Level Behaviour 71
5.2.3 Aspect Level Comparison Summary 74
5.3 Statistical Validation using the Kolmogorov Smirnov (KS) Test . . 75
5.3.1 Aspect-Level KS Separation Analysis 76
5.4 Summary 80

6 Case Study Analysis and Discussion **82**
6.1 Case Studies 83
6.1.1 bootstrap-sass 84
6.1.2 rest-client 86
6.1.3 strong_password 87
6.1.4 Cross-Case Analysis 89
6.1.5 Malicious Packages from Backstabbers dataset 90
6.1.5.1 activerecord-duplicate 90
6.1.5.2 capistrano-copy-subdir 91
6.1.5.3 aasm_history 92
6.2 Discussion 93
6.2.1 Metric Discriminatory Strength 93
6.2.2 Distribution of Risk Score 94
6.2.3 Role of Socio-Technical Signals 94
6.2.4 Composite Score Formulation and Design Choice 96
6.2.5 Practical Interpretation of Results 97
6.2.6 Comparison with Existing Approaches 99

7 Conclusions, Limitations and Future Work **100**
7.1 Conclusion 100
7.2 Limitations 102
7.3 Future Work 103

References **105**

Appendices **121**

Appendices **121**
A Risk Metrics and Directionality 123
B Notation Guide 124
C Metric-Level AUC Results 125
D Full Enrichment Analysis Across Ranking Thresholds 126

Chapter 1

Introduction

Modern software development relies heavily on open source package ecosystems, where reusable components are distributed through centralised registries such as npm, PyPI, RubyGems and many more. However, such reuse may further make the project rely on other direct or transitive dependencies, thus forming complex dependency relationships [47, 25]. The entire software supply chain lifecycle spans from upstream package creation and distribution through centralised registries, to downstream integration, deployment via CI/CD (Continuous Integration/Continuous Deployment software development practice integrates, tests and releases code changes automatically)pipelines, up to their ongoing maintenance. As this interconnected dependency chain grows, they introduce broader attack surfaces vulnerable to upstream software supply chain attacks.

In recent years, software supply chain attacks have emerged as a major cybersecurity threat. High profile incidents such as the SolarWinds compromise [54] and the XZ Utils backdoor incident [70] demonstrate how attackers can infiltrate trusted components and propagate malicious code to a wide range of downstream

consumers. Similarly, attacks targeting open source package registries, such as dependency confusion or maintainer account hijacking, have shown that adversaries increasingly exploit weaknesses in package ecosystem practices as well as their inherent trust in third-party components [6, 86].

The supply chain attacks highlight a fundamental challenge: open source packages are not only widely reused but also interconnected through complex dependency networks. A compromise in a single upstream package can affect thousands of downstream applications, amplifying the impact of an attack [97, 51]. The complex dependency network introduces socio-technical vulnerabilities, such as hidden compromise points, to inject malicious payloads or obtain unauthorised privileges through social engineering and ecosystem practices for pinning and updating dependencies. These challenges highlight the need for proactive approaches that can identify packages with higher susceptibility to supply chain attacks before a real compromise occurs, motivating the focus of this thesis.

1.1 Problem Statement

Despite growing attention to software supply chain security, existing approaches remain largely reactive. Most tools and frameworks focus on detecting known vulnerabilities, analysing past incidents, or evaluating project level security practices [65, 35, 77]. While effective for incident response, these methods do not address a critical unanswered question: *which packages are most likely to be targeted from an adversarial perspective?*

Current frameworks, including OpenSSF Scorecard [65] and metadata driven detection approaches [35], exhibit three key limitations. First, they rely on post

compromise signals rather than pre-compromise indicators observable during attacker reconnaissance. Second, they operate at project or repository granularity, overlooking individual packages as the true units of attack within dependency ecosystems. Third, existing scoring systems prioritise general software health rather than attacker driven factors such as discoverability, impact, and ease of compromise.

From an adversarial perspective, targeting decisions are likely to be influenced by a combination of factors, including package popularity, maintenance activity, and ecosystem security practices. However, these factors to date have not been integrated into a unified framework that captures susceptibility to supply chain attacks prior to compromise. Consequently, there remains a limited systematic capability for identifying and prioritising packages with higher susceptibility to supply chain attacks, constraining proactive security efforts in large scale open source ecosystems.

1.2 Research Question

This study addresses the following research question: *To what extent can socio-technical metadata signals be used to capture the susceptibility of open source packages to software supply chain attacks from an attacker-centric perspective?*

To support this investigation, the following supporting questions are considered:

- Which socio-technical signals provide the strongest discriminatory power for identifying susceptible packages?

- How can these signals be integrated into a composite risk scoring framework?
- Do higher-ranked packages exhibit statistically higher composite scores for known malicious instances?

1.3 Research Approach

To address the research question, we adopted a data driven and attacker centric approach to risk profiling. The study focuses on the RubyGems ecosystem, leveraging socio-technical metadata such as dependency structure, download statistics, and maintenance activity.

A set of publicly observable socio-technical signals is extracted and grouped into three aspects: complexity, impact, and maintenance. These signals are evaluated using ROC-AUC analysis against a dataset of known malicious packages, enabling the identification of features that contribute to distinguishing malicious packages.

The framework produces a composite risk score that ranks packages based on their relative susceptibility to supply chain attacks using metric level signals that are weighted based on their discriminatory strength and aggregated into an overall risk score. This approach prioritises interpretability and transparency, allowing the contribution of individual signals to be analysed.

The resulting framework is evaluated through distribution analysis, enrichment metrics, and statistical testing, providing insights into how highly susceptible packages are distributed across the ecosystem to be prioritised. The proposed approach is designed as a prioritisation framework rather than a predictive classifier, focusing on ranking packages based on their susceptibility to supply chain attacks rather than predicting specific attack events.

1.4 Contributions

This thesis makes the following contributions:

- **Attacker Centric Risk Framing:** Introduces a novel perspective for software supply chain security by modelling package susceptibility to supply chain attacks from an attacker reconnaissance viewpoint, rather than traditional vulnerability centric approaches.
- **Socio-Technical Signal Integration:** Proposes a hierarchical composite scoring framework that integrates behavioural, structural, and maintenance related signals into a unified risk framework.

Collectively, these contributions provide a data-driven and attacker-centric approach to software supply chain risk assessment at package granularity. The proposed framework is designed for ranking based prioritisation rather than predictive classification.

1.5 Thesis Structure

This thesis develops a scoring methodology that allows us to identify open source packages that makes an viable target for successful compromise from an adversary's perspective. Firstly, Chapter 2 introduces the background concepts of software supply chain security, open source ecosystems, and relevant terminology. Next, Chapter 3 reviews existing literature on supply chain attacks, risk modelling, and metadata based approaches, identifying key research gaps. Then, Chapter 4 presents the methodology, including data collection, signal extraction, and the

construction of the composite risk scoring framework. Next, Chapter 5 evaluates the proposed framework using statistical analysis, enrichment metrics, and case studies. Chapter 6 reports the case study analysis and presents a critical discussion of our research findings. The final Chapter 7 concludes the thesis and outlines the limitations as well as the future prospective work based on this research.

Chapter 2

Background

This chapter introduces the fundamental concepts of software supply chain security, open-source ecosystems, and risk profiling and some common terms that are used throughout this thesis.

2.1 Software Supply Chain

Software supply chain comprises various components involved in the software development lifecycle and their interconnected relationships. These components mainly refer to software artifacts, development tools and platforms, processes, CI/CD pipelines, distribution, and ongoing updates and maintenance, and most of all, the people involved in the process. It represents the relationships among these components at both technical (software development process) and social levels, such as stakeholder engagement and community interactions. In other words, we study it as a socio-technical ecosystem in this thesis, observing how different dependencies influence the security posture of open-source software.

When it comes to open source, it reflects the main motivation of the supply chain ecosystem, with a huge number of software developers depending on code reuse. In terms of the software supply chain, libraries and packages that serve as building blocks for other projects, forming the foundational layer of modern software composition through reuse, are called upstream dependencies. These dependencies can be further split into direct and transitive dependencies.

Direct dependencies are explicitly declared by developers in the project configuration files (package.json, Gemfile, requirements.txt). Transitive dependencies, on the other hand, are indirect dependencies that are automatically pulled in through direct dependencies. We can say that, even if there is sufficient scrutiny of direct dependencies, the same cannot be said for transitive dependencies, especially those that are very deep in the dependency network.

Those applications and systems that depend on upstream dependencies and inherit both functionality and security posture through dependency relationships are called downstream consumers in the software supply chain. To improve visibility into the software supply chain architecture, developers rely on representations such as dependency graphs that map all direct and transitive package interconnections within a project. Another method is through an SBOM (Software Bill of Materials), which is a comprehensive inventory of all software components, dependencies, and metadata in a software package, expressed in SPDX or CycloneDX formats. These representations provide structured visibility into the composition of software systems and the relationships between their components. This foundational understanding of the software supply chain establishes the structural and relational context necessary for analysing how dependencies are formed, managed, and propagated within modern software ecosystems. The following section builds

on this foundation by examining security considerations and attack vectors within the software supply chain.

2.2 Software Supply Chain Security

Following the description of software supply chain components, let us understand the software supply chain security aspects. To begin with, a supply chain attack involves the compromise of a trusted dependency, which propagates the breach through dependency chains to downstream applications and users. Hence, the attack surface of the software supply chain includes all potential entry points that threat actors can target across the software development lifecycle, including source code, build tools, CI/CD pipelines, maintainers, and distribution channels.

Since this thesis focuses on an attacker-centric perspective, we aim to capture the adversary reconnaissance phase, the first stage of the kill chain, where attackers observe public metadata (popularity, activity, ownership) to identify vulnerable packages and high-value targets with ease of compromise. This yields pre-compromise signals such as downloads, reviews, maintenance gaps, codebase dependencies in package registries and code-hosting platforms, visible during reconnaissance, that help attackers prioritise the easiest targets for supply chain compromise.

2.2.1 Supply Chain Attack Vectors

The above mentioned security aspect can be further understood by examining the attack techniques through which adversaries exploit the software supply chain. The most prominent one is a dependency-based attack method targeting upstream

package dependencies, leveraging transitive complexity to amplify compromise impact across downstream consumers. These mainly include dependency confusion, typosquatting, and malicious package injection, along with other techniques such as maintainer account compromise, exploitation of abandoned or weakly maintained projects, and build pipeline manipulation (e.g., CI/CD injection), which collectively exploit both technical vulnerabilities and trust relationships within the ecosystem.

2.3 Socio-Technical Perspective of Supply Chain Risk

Software supply chain risk is both technical and social. Technical factors include code complexity, dependency structure, and build processes. Social factors include maintainer activity, contributor behaviour, and community practices.

For example, a package maintained by a single inactive developer may be more vulnerable to compromise. Similarly, weak review practices or low community engagement can increase risk.

A socio-technical perspective considers both dimensions together. This provides a more complete understanding of risk than analysing code alone.

2.4 Risk Signals and Indicators

As stated earlier, this thesis focuses on pre-compromise signals, also referred to as weak-link signals, which indicate a package's ease of compromise during the reconnaissance and target selection phases from a threat actor's perspective. These

signals represent socio-technical vulnerability risk indicators examined in this research, which integrate social factors (maintainers, community interactions) with technical factors (source code, build tooling, CI/CD pipelines). Structural signals capture risk through codebase and dependency metrics, such as package count and dependency tree depth. Impact signals reflect the potential scope of breach impact. Finally, maintenance signals represent project sustainability indicators, tracking update frequency, security patches, and community activity, thereby signalling active maintenance.

2.5 RubyGems Ecosystem

Our research study is based on the Ruby programming ecosystem because it is most widely used for web applications, thus providing threat actors with ample attractive opportunities and incentives, such as access to production systems, exposure of sensitive user data, potential for financial gain, and opportunities for large-scale service disruption. RubyGems, a package manager, supports the distribution, installation, and management of Ruby source code and libraries, popularly known as gems, through a centralised public registry hosting platform. The ecosystem is characterised by extensive code reuse, a rich dependency structure, and strong interconnections between packages, which collectively increase exposure to supply chain risks and make it a relevant context for analysing pre-compromise signals.

This chapter provides the foundational concepts for analysing software supply chain risk, including key components, security considerations, and attacker-centric perspectives. The next chapter reviews existing research and identifies gaps that motivate the proposed approach.

Chapter 3

Literature Review

As introduced in Chapter 1, the existing software supply chain security approaches is part of the larger context of modern software development and open source. In addition to continuous attacks on open source code registries, the prevalent supply chain attacks have shown that attackers are increasingly targeting dependencies and package registries, modifying these benign third party components to act as entry points for large scale compromise. This chapter reviews existing literature in software supply chain attacks, open source package ecosystems, and risk profiling frameworks based on package data, with emphasis placed on the RubyGems ecosystem. The objective is to ascertain key concepts, existing frameworks, and methodological approaches in which the proposed risk profiling method in this thesis is designed.

3.1 Software Supply Chain Security

3.1.1 Definition and Evolution of Software Supply Chain Security

A software supply chain attack is typically defined as an incident in which an adversary injects malicious code or configuration into a legitimate software component, causing downstream consumers to install or execute it unknowingly. Over the past decade, such attacks have evolved from relatively simple dependency substitution schemes into more complex, multi stage campaigns that exploit modern development infrastructure, including CI/CD pipelines, compromised maintainer accounts, and registry level weaknesses.

The paper by Thompson[85] proposed the first fundamental concept of the supply chain type of attacks by showing how a trusted piece of code, e.g., the compiler, could be compromised so that malicious code is inserted into the binary code, but the source code and the source code of the compiler itself are not altered. Cox et al.[21] discuss the evolution of the concept of software reuse, which has shifted from the ideal concept to the complex interdependent ecosystem we now know, where the rapid adoption of open source software, the speed of deployment, and the lack of vetting have increased the attack surface and enabled more sophisticated attacks.

Ladisa et al.[45] provide a high level, general overview of the open source supply chain type of attacks, showing how attackers utilise the privileged positions of the people involved, e.g., the people who manage the code, the people who manage the infrastructure, and how the infrastructure itself is attacked at the registry level.

Ladisa et al.[47] also propose a systematic and language-independent categorisation of the 107 unique types of attacks, which are divided into eight categories, including attacks against the people involved in the dependencies, attacks against the infrastructure, attacks against the people involved in the CI/CD pipelines, and attacks against the people involved in the package registries.

Taken together, these studies suggest that the current software supply chain attacks have evolved from the basic dependency level exploitation to the entire software development lifecycle process, including initial development/integration, continuous integration/continuous deployment, and deployment/maintenance phases.

3.1.2 Historical Incidents as Motivation

Several software supply chain attacks, both in commercial and open source software domains, have had significant and lasting impacts on software practice and research. These software attacks serve as significant motivational cases for this work. They describe in detail how adversaries leverage software updates, dependency graphs, and software automation. In chronological order, some notable software attacks include:

3.1.2.1 M.E.Doc (NotPetya, 2017)

M.E.Doc is a tax and accounting software company based in Ukraine. Their update infrastructure was compromised, which enabled attackers to spread NotPetya malware worldwide. NotPetya is a destructive malware disguised as ransomware that caused significant business disruption and data destruction at some of the world's largest organisations. This is an example of how an attack on a single vendor's

update infrastructure could have significant cascading consequences throughout the software supply chain. [30, 22]

3.1.2.2 CCleaner (Piriform/Avast, 2017)

CCleaner is a popular system utility that was compromised by attackers. They used this infrastructure to spread malware in the form of a legitimate looking update that was digitally signed [79]. This attack affected more than two million users worldwide. The attack had two stages. The first stage was designed to be stealthy, gathering system information and identifying high value targets. The second stage was more targeted attacks on selected organisations[74]. The attack's impact came from the widespread distribution of the attack via a trusted software update. The attack's primary purpose was stealthy access to selected organisations.

3.1.2.3 event-stream (npm, 2018)

event-stream is a Node.js package for working with data streams that was handed over to a new maintainer, who added a malicious dependency aimed at a particular implementation of a Bitcoin wallet[34]. The compromised package was downloaded 8 million times before discovery, highlighting how a piece of open source code can be leveraged for a targeted supply chain attack to drain money to a malicious actor[3].

3.1.2.4 SolarWinds Orion (SolarWinds, 2020)

Orion, a widely used IT network monitoring tool, was compromised. The attackers inserted the SUNBURST backdoor in the software updates that were sent to

around 18,000 organisations across the globe[23, 80]. The adversary's goal in this scenario was to gain unauthorised access to the organisation's IT infrastructure.

3.1.2.5 npm ua-parser-js (2021)

In October 2021, the account of the package maintainer of the popular JavaScript library ua-parser-js was compromised on npm, allowing the attacker to release three versions (0.7.29, 0.8.0, and 1.0.0) of the library containing malware, including a cryptominer and credential exfiltration code[86]. Although the malicious package had a short exposure period, given the package's widespread adoption, with millions of weekly downloads, it posed a significant risk to downstream users and systems. This is another example of the potential impact of the compromise of a high reach package, which can lead to the introduction of malware into the development environment.

3.1.2.6 Dependency Confusion (2021, Alex Birsan)

A software supply chain attack in which the threat actors published malicious packages in package registries with the same name with a higher version number than the internally developed software packages used by the organisation. These malicious packages were then inadvertently used in the build process of other downstream users instead of the intended internal packages. Alex Birsan, a security researcher, simulated this scenario and proved the efficacy of this attack, where the malicious packages were used by over 35 companies whose build process inadvertently used the attackers' packages instead of the intended internal packages. This scenario proves that attackers can inadvertently inject malicious code into the build process of the software supply chain. The attackers' goal in this

scenario was to prove the efficacy of the Dependency Confusion attack scenario. The Dependency Confusion attack scenario was reported in the research papers published by Sonatype in 2021[75] and Birsan in 2021[6].

3.1.2.7 XZ-Utils (Linux, 2024)

In early 2024, malicious code was intentionally injected into the widely used open source data compression program xz-utils, in its liblzma library in version 5.6.0 and 5.6.1, via a compromised maintainer's account[70]. This backdoor, designated as CVE-2024-3094 with a CVSS rating of 10, had the potential, under certain build and runtime conditions, to allow an attacker with a certain private key to remotely execute arbitrary code via OpenSSH by modifying authentication mechanisms. This attack, which exploited xz-utils's widespread presence in Linux systems, was discovered and announced in March 2024 by researcher Andres Freund and subsequently fixed by reverting to safe versions. This case demonstrates how this core open source software, which is an integral part of Linux systems, can also function as an insidious supply chain attack vector with significant implications for trust and system integrity[81].

Table 3.1: Key Supply Chain Incidents: Attack Vectors and Impacts

Year	**Incident**	**Vector**	**Impact**	**Risk Relevance**
2017	M.E.Doc	Update infrastructure	Global NotPetya	CI/CD pipeline risks
2018	event-stream	Maintainer takeover	8M downloads	Transitive poisoning
2021	ua-parser-js	Account compromise	Millions downloads	high reach packages
2021	Dependency Confusion	Registry confusion	35+ companies	Naming attacks
2020	SolarWinds	Build pipeline	18K orgs compromised	Signed releases
2024	XZ Utils	Social engineering	CVE-10.0	Maintainer targeting

The attacks described above are historically recorded and clearly depict the trend that, with the increasing level of automation, interdependence, and openness in the ecosystem, the supply chain represents an ever more attractive target for attacks. As recent research indicates, the increasing reliance on unvetted dependencies, account security, and the validation of updates enables attackers to leverage a combination of technical vulnerabilities, automation focused techniques, and social engineering based techniques[21, 45, 47]. From this perspective, the thesis aims to discuss risk profiling techniques that can help mitigate the aforementioned risks at the early stages of the supply chain lifecycle.

3.2 Open Source Package Ecosystems

3.2.1 Ecosystem Growth and Software Supply Chain Risks

Research from 2024-2026 [77, 72] shows an increasing emphasis on the importance of pre-compromise risk detection in open source software ecosystems. Industry-scale measurements indicate that the rate of growth in malicious packages in software registries is rising significantly. To illustrate, Sonatype 2024 report[77] found more than 704,000 unique pieces of malware in open source software packages since 2019, which is an increase of 156% year over year, suggesting an accelerating trend in supply chain risks. The total number of unique pieces of malware in open source packages surpassed 1.2 million in 2025, indicating the industrialisation of open source malware attacks[78].

ReversingLabs 2025 report[72] states that there was a 73% rise in the number of malicious open source package detections in 2025, in addition to the rising threat

of self-propagating or registry based worm malware, indicating the rising level of sophistication of attackers in the open source ecosystem. Previous trend analyses indicated that the growth in supply chain risks rose by more than 1300% between 2020 and 2023[72].

To complement these industry scale measurements, recent studies show that package level metadata and dynamic behavioural signals are effective in distinguishing malicious from benign packages in open source software ecosystems[35, 60]. Halder et al.[35] performed a large scale empirical analysis across multiple package registries, demonstrating that metadata features such as dependency counts, versioning patterns, and maintainer activity can reliably flag high risk packages prior to exploitation. In parallel, Nguyen et al.[60] applied dynamic analysis to characterise the lifecycle and operational behaviours of malicious packages, revealing distinctive execution patterns and propagation strategies. Collectively, these findings motivate the use of socio-technical and behavioural characteristics for predictive risk modelling in open source ecosystems.

Based on the results of these studies, it is clear that the threat landscape of the software supply chain is expanding in terms of scale and sophistication. The evolution of the threat landscape of the software supply chain motivates the importance of moving from reactive risk detection to proactive risk modelling based on socio-technical characteristics of packages in software registries[45]. The results of the studies also indicate that the evolution of ecosystem security research follows a three phase trajectory consisting of measuring the scale of the problem, mapping the attacks, and predicting pre-compromise targets[24, 45]. The evolution of ecosystem security research motivates the risk profiling of RubyGems based on hierarchical socio-technical characteristics validated against the Backstabber

dataset[61].

3.2.2 Software Supply Chain Attack Vectors

The software supply chain attack utilises direct or indirect dependency chains, which allow for a single attack to have a significant impact on a large number of dependent projects and organisations, while maintaining a high level of operational obscurity. Furthermore, most prevalent attack vectors align with MITRE's ATT&CK framework for "Supply Chain Compromise" tactic and technique T1195[56] and OWASP's A03:2025 category "Software Supply Chain Failures"[66]. They are also systematically detailed in the "Risk Explorer Attack Tree for Software Supply Chains" interactive visualization[46], which maps high level attacker goals to 107 specific techniques identified across 94 real world incidents. The attack vectors discussed below specifically relate to open source package registry based supply chains, such as RubyGems, npm, and PyPI, in which attackers utilise technical dependencies in addition to socio-technical trust relationships to achieve maximum impact with minimal detection risk.

3.2.2.1 Upstream Dependency Compromise (Transitive Dependency Poisoning):

In this attack class, adversaries compromise or gain control of an upstream dependency that is transitively included in numerous downstream projects. Since most transitive dependencies are often pulled in automatically without rigorous auditing, malicious payloads can propagate widely before detection. An example of such is the event-stream incident [3], in which a widely used JavaScript

library was covertly modified to include a credential stealing payload targeting cryptocurrency wallets. This vector highlights the supply chain risk introduced by implicit trust in third-party dependencies and the amplification effect of transitive inclusion.

3.2.2.2 Typosquatting and Dependency Confusion

These attacks exploit ambiguities in package naming conventions and dependency resolution mechanisms across major registries. In typosquatting, adversaries publish packages with names closely resembling popular libraries (e.g., `request` → `requests`), preying on common developer typos, autocomplete errors, or imprecise version specifications during installation. Dependency confusion attacks are more sophisticated, where adversaries publish malicious packages to public registries using the same names as internal or private organisational dependencies. They rely on developers' errors in explicitly pinned dependency versions or misconfigured build systems in lockfiles like `Gemfile.lock` or `package-lock.json`, causing these malicious public packages to be pulled instead of legitimate intended ones. This results in the execution of attacker modified code during the build process. Prior research has demonstrated both the feasibility and scalability of these attacks across npm, PyPI, and RubyGems ecosystems[6], highlighting persistent weaknesses in namespace management and package resolution policies.

3.2.2.3 Maintainer Account Takeover and Package Hijacking:

Adversaries may gain unauthorised access to maintainer accounts through phishing, credential reuse, or token compromise, enabling them to publish malicious updates to otherwise legitimate packages. Additionally, unmaintained or "orphaned"

packages may be claimed by attackers and repurposed for malicious distribution. Recent reports [31] document campaigns involving hijacked maintainer accounts, where attackers introduced obfuscated data exfiltration payloads and leveraged registry ownership recovery mechanisms to retain control. This vector underscores the critical role of maintainer identity and access control in supply chain security.

3.2.2.4 Malicious Package Injection (Direct Package Poisoning):

In this vector, adversaries directly publish seemingly legitimate packages embedded with malicious functionality to public registries. These packages often masquerade as useful utilities or dependency alternatives and execute payloads during installation via lifecycle hooks (e.g., postinstall, setup.py). Large scale campaigns, such as MUT-8964[71], have demonstrated coordinated injection of malicious packages across npm and PyPI, delivering info stealerstyle malware and leveraging trusted hosting platforms (e.g., GitHub) for payload distribution.

3.2.2.5 Build Pipeline and CI/CD Compromise:

Rather than modifying source code directly, adversaries may target build infrastructure and continuous integration/continuous deployment (CI/CD) systems to inject malicious artifacts during the build process. This includes compromising CI runners, exfiltrating secrets (e.g., API tokens, signing keys), or introducing covert build steps that alter outputs. The SolarWinds supply chain attack [80], where attackers inserted backdoored code into signed software releases via a compromised build environment. Such attacks are particularly difficult to detect, as resulting artifacts often appear authentic and correctly signed.

3.2.2.6 Source Code Repository Compromise:

This vector involves direct manipulation of version controlled source repositories (e.g., GitHub, GitLab). Adversaries may inject malicious commits, hijack release branches, or abuse repository permissions to alter code before packaging and distribution. In some cases, social engineering or credential compromise enables attackers to bypass code review processes and introduce backdoors into widely used projects. These modifications may persist undetected until downstream anomalies are observed, highlighting weaknesses in repository governance and code integrity verification.

The above attack vectors mostly succeeded because dependencies were rarely scrutinised until high profile incidents like event-stream[34] and xz-utils[70] forced ecosystem wide re-evaluation. As such, research directions focused on proactive signal based risk assessment that examines package metadata to detect and prevent compromises [37, 29, 52, 91].

3.3 Signal based Risk Assessment

3.3.1 Weak-Link Signals Categories

open source package ecosystems expose a large amount of metadata that can be used to identify potential security risks for future attack probability. These metadata, often called weak-link signals[93], can be studied and observed to reveal patterns that indicate high susceptibility to compromise. For example, Zahan et al.[93] identified several indicators, such as install scripts, inactive maintenance, and outdated maintainer email domains. The developer surveys in their study

confirm the importance of these signals, as they shift the focus from known vulnerabilities to ecosystem level early warning signs. Prior studies systematically categorise these signals across three dimensions:

Behavioural signals: Behavioural risk signals can be based on empirical studies of real world attacks, which provide us with useful context. The dataset introduced in Backstabber's Knife Collection [61] highlights how malicious packages behave in practice across ecosystems, for instance, in npm, PyPI, and RubyGems. By comparing these malicious instances with benign packages, it becomes possible to observe patterns that highlight markers of compromise. This study further supports the idea that risk can be investigated as observable package characteristics rather than relying only on post incident analysis.

Structural Signals: The most important aspects of software supply chain risk link signals come from the structure of dependency networks. Research papers by Decan et al. [25] and Zimmermann et al. [97] show that package ecosystems form complex, highly connected graphs where risk can propagate through transitive dependencies. Similarly, a study by Lauinger et al. [48] highlights how implicit trust in third-party libraries can introduce hidden risks. These studies suggest that factors such as dependency depth, centrality, and transitive exposure are meaningful indicators of potential weakness, even if the package itself doesn't contain malicious code.

Maintenance Signals: In addition to structural properties, maintenance related signals also play a key role, which are discussed in studies by Pashchenko et al. [67] (Maven ecosystem) and Zerouali et al. [94] (npm ecosystem), which show outdated package dependencies as risk signals. On the other hand, an article by Slinger Jansen[39] provides ecosystem level health metrics such as maintainer

activity and update frequency, which can help identify weak link indicators. More recent work by Liu et al. [52] further frames these characteristics as "smells" that indicate potential risk rather than confirmed compromise.

3.3.2 Automated Security Metrics (Scorecards and Tools)

Automated scorecards and risk assessment tools generate composite health or risk scores based on diverse signals, including repository practices, community activity, external threat data, and dependency relationships. These scorecard frameworks provide a systematic quantitative evaluation of supply chain security posture across open source ecosystems.

OpenSSF Scorecard provided us the foundational reference, where the health score of the packages was computed, ranging from 0 to 10 across 18 automated checks, including branch protection (requiring code review), binary artifacts (release integrity), fuzzing coverage and others. The low scores in OpenSSF scorecards warrant remediation, although it focuses on repository practises rather than transitive package risks[65].

Similarly, the OpenSCRM (open source Supply Chain Risk Management) framework's RiskStack engine uses known threat signatures across more than 15 data sources, like GitHub activity and CVE feeds, to generate package level risk rankings[64]. Another framework, Bitergia Risk Radar, which uses SBOM (Software Bill of Materials) data for dependency scoring, incorporates community signals such as maintainer bus factor and activity velocity to address sustainability risks absent in other tools [7].

In addition to open source tools, commercial platforms advance this signal

based methodology. SecurityScorecard provides A-F vendor ratings derived from external signals such as patching latency and exposed credentials [73]. BitSight generates 250-900 security ratings, emphasising endpoint exposure and patching cadence[8, 9]. Black Duck delivers comprehensive OSS risk scoring across vulnerability, license, and operational risk metrics from continuous repository scanning [10, 11].

The above mentioned existing scorecards aggregate risk at project or vendor levels using health based scoring (higher = better), target repository hygiene, emphasising development and security best practices. They include limited maintenance signals in their methodology, but overlook adversary reconnaissance perspective as seen pre-compromise within software supply chain ecosystems, which makes a package susceptible to discovery and targeting.

3.3.3 RubyGems Ecosystem Security Studies

The RubyGems ecosystem, as a central repository for Ruby libraries, has been the subject of various empirical and socio-technical studies that examined the evolution of the system, its dependencies, and security risks in the context of software supply chain security. Although compared to more popularly analysed ecosystems such as npm, PyPi and Maven, the security focused research on RubyGems remains relatively limited. However, existing work offers valuable insights into the factors contributing to package level risk and ecosystem vulnerability.

The initial socio-technical studies on the RubyGems system by Constantinou and Mens [20] reveal that reduced commit rates and declining contributor interest are strong predictors of maintainer abandonment in the RubyGems ecosystem.

From a security perspective, this could indicate a socio-technical vulnerability that may lead to account takeover or ease of compromise from adversaries' perspective, as inactive maintainers may fail to detect malicious activity.

Furthermore, Decan et al.[25] state dependency based risk from RubyGems' deeply nested transitive dependencies. The complex dependency chains amplify the upstream attack surface, propagating vulnerabilities and malicious insertions across downstream dependent projects.

In a similar manner, empirical analyses of real world supply chain attacks illustrate the attack vector prevalent in RubyGems. The Backstabber's dataset [61] documents multiple attack vectors observed in package ecosystems, especially typosquatting, malicious code injection, and account compromise, in RubyGems packages that maybe leveraged in software supply chain attacks. These attacks use ecosystem characteristics such as package naming conventions and the implicit trust of maintainers, facilitating adversaries to compromise packages that downstream developers inadvertently adopt.

RubyGems specific security studies remain sparse in the context of risk signals, and much of the broader understanding of supply chain threats is derived from cross ecosystem analyses. For example, [28] demonstrates that software package managers are broadly susceptible to similar type large scale supply chain attacks, which are ecosystem agnostic. Given the architectural similarities among package registries, these findings suggest that the risks observed in other ecosystems are also applicable to RubyGems.

Existing literature identifies three key RubyGems risks: (i) maintainer inactivity, (ii) complex dependency networks, and (iii) supply chain attacks exploiting trust. However, prior work focuses on ecosystem level patterns and known attacks,

with little systematic analysis of individual package susceptibility using observable metadata. This gap motivates our package level risk scoring approach.

3.4 Literature Gaps

3.4.1 Research Gaps

Existing research addresses software supply chain attack categorisation[47, 46], industry reports documenting malicious package proliferation in open source registries[78, 77, 75], malicious package analysis[61], package sustainability metrics[20], dependency network structures[25], and project level scorecards[65].

Despite this comprehensive foundation, five critical research gaps persist:

1. Reactive versus proactive approaches, where current methods focus on post compromise detection rather than pre-attack risk identification.

2. Project level versus package level assessment, where existing tools evaluate repositories but overlook individual package vulnerabilities within dependency graphs.

3. Isolated versus integrated signal analysis, where risk indicators are examined separately rather than combined into unified framework.

4. General ecosystem versus RubyGems specific analysis. RubyGems hosts over 180,000 packages with rich socio-technical metadata, has the activesupport library with 1.2 billion downloads cascading to 500,000 Rails apps. However, the supply chain security research is largely focused towards npm/pypi ecosystems.

5. Security hygiene versus adversary perception approach. Current frameworks measure compliance but not adversary target selection criteria.

This thesis addresses these limitations through a RubyGems-specific package level risk scoring methodology that integrates behavioural, structural, and maintenance signals, using the Backstabber's Knife Collection dataset as training data and validating scores against real-world attacks as case studies in the RubyGems ecosystem to enable proactive prioritisation of high impact attack targets.

Chapter 4

Methodology

In this chapter, we discuss our approach to build a risk profile based on socio-technical metadata for RubyGems packages. The profile reflects the susceptibility of an open source package being targeted for supply chain attacks owing to the ease of compromise from an adversary's perspective.

4.1 Research Design

We start by discussing the selection of RubyGems, the scope of our work and the pre-compromise risk profile.

4.1.1 Selection of RubyGems Ecosystem

RubyGems [1], the package manager for the Ruby programming language, hosts over 180,000 packages and is enriched with socio-technical metadata such as download statistics, commit history, dependency structures, and maintainer activity, which

[1] `https://rubygems.org`

aligns with our software supply chain risk profiling. It was selected over other ecosystems due to three key advantages:

- RubyGems has a manageable corpus size (180k+ packages) with socio-technical metadata at a statistically meaningful scale.
- RubyGems lacks supply chain security research studies prevalent in npm/PyPI ecosystems.
- RubyGems has high impact software like Rails, which is the core of web development applications that power applications such as, but not limited to, Shopify, GitHub, Airbnb, and Basecamp (enterprise SaaS). Rails further depends on core utility libraries such as activesupport (1.2B downloads), which in turn relies on bundler (3.3B downloads), actioncable (5.4B downloads) and others. Thus, a single compromise(even a typosquatting) could cascade to every Rails application worldwide, amplifying risk to billions of dependents.

4.1.2 Study Scope

The study examines a snapshot of the RubyGems ecosystem captured between September and December 2025. Data collection began with an initial list of 187,181 packages from the RubyGems official registry, for which we then extracted metadata on dependency metrics, usage and popularity metrics, codebase structure metrics, maintenance metrics, and repository statistics metrics.

After gathering the data, we processed it further to ensure that each package had a minimum of 80% valid metadata coverage. An 80% metadata completeness

threshold was selected as a practical trade-off to balance data quality and dataset coverage. This involved the removal of null values, the validation of GitHub repository links, the exclusion of forked packages, and private repositories. As a result, the primary data set was reduced to 79,000 packages (42.2% retention), which we determined to be of sufficient magnitude and quality for our subsequent analysis.

For the analysis of malicious packages, we utilised the Backstabbers Knife Collection [61] that lists known malicious packages used in real world attacks on open source software supply chains across multiple ecosystems. The Backstabbers dataset identifies 804 malicious packages across RubyGems ecosystem, of which 380 correspond to our primary dataset of 79,000, identified through systematic name matching and manual verification.

4.1.3 Pre-Compromise Risk Profiling

Current research [38, 27, 42] in this field aligns with established frameworks, guidelines, secure tooling, and post incident response strategies provided by NIST[2], S2C2F[3], and SLSA[4]. These approaches primarily focus on defining best practices for the prevention, detection, and mitigation of software supply chain attacks. However, these studies highlight that such frameworks are largely defence oriented and do not explicitly focus on attacker driven target selection or pre-compromise susceptibility within package ecosystems [62, 38].

In contrast, recent attack patterns highlight a shift where attackers increasingly

[2]https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-supply-chain-security-guidance

[3]https://github.com/openssf/s2c2f

[4]https://slsa.dev

target humans in the software development process through social engineering and related tactics [91]. This suggests that the adversaries now seek socio-technical vulnerabilities rather than just code level vulnerabilities to exploit. Accordingly, our work veers towards addressing the pre-compromise phase of a supply chain attack. We profile open source packages for high susceptibility to being chosen as a target during the reconnaissance stage of the kill chain. In particular, we aim to investigate for indicators that could help in assessing the susceptibility risk for RubyGems, leveraging socio-technical metadata from 79,000 packages to study adversary reconnaissance signals (downloads, maintenance decay, dependency centrality) and identify potential targets for compromise.

4.2 Data Collection

4.2.1 Data Sources

We started data collection for our study by getting a snapshot of RubyGems on September 15, 2025, from the RubyGems.org API, which yielded an extensive 187,181 unique package names and their versions. This was then used to extract publicly available metadata on metrics, including dependencies, dependents, download counts, and contributor counts, all pulled during the September–November 2025 timeline to ensure consistency. Next, we queried the GitHub API on packages with valid repository URLs (89,889 packages, or 48% of the set) for additional metric data to enrich our socio-technical signal analysis and pulled the following:

- Codebase structure (file counts, file size, LOC, directory depth).
- Maintainer activity (commits, recent issues/PRs over 6 months, release fre-

quency).

- Repository stats (stars, forks).

The Backstabber's Knife Collection [61] provided 380 confirmed malicious RubyGems within our sample as calibration labels.

Pre-compromise perspective: During raw data collection, we excluded vulnerability databases (RubySec, OSV, NVD), incident reports, and post compromise signals. This preserved only RubyGems registry and GitHub metadata dependencies, usage stats, maintenance signals, and repository metrics, at crawl time, maintaining the clean pre-compromise state as observed by an adversary during the reconnaissance phase of target selection. Malicious labels (Backstabber's dataset, security blogs, yanked packages) were applied post transformation only, ensuring our risk framework analysis on socio-technical signals available before any attack occurs.

4.2.2 Data Preprocessing

From the initial 187,181 unique packages:

- **57.7%** (108,000 packages) passed the initial 80% completeness check.
- **48.0%** (89,889 packages) had valid GitHub URLs for next stage data collection.
- **42.2%** (79,000 packages) passed the second stage of 80% completeness recheck.

This resulted in a final corpus size of 79,000 packages for our framework implementation.

4.3 Socio-Technical Risk Indicators

In this section, we discuss risk indicators which serve as signals targeting opportunities for adversaries during reconnaissance. These risk signals represent the socio-technical aspects of open source ecosystems which integrate the social aspects (developers, contributors and interactions) with the technical factors (software development and its infrastructure) [53].

4.3.1 Metric Definitions and Rationale

The metric selection for our framework was primarily motivated by CHAOSS metrics[5] [33]. In addition to that, we also introduced a few novel risk metrics deemed important for our susceptibility scoring framework. These metrics serve as quantifiable socio-technical risk indicators, which we define and justify as follows:

4.3.1.1 Dependency count

This metric captures the total number of direct and transitive dependencies for a RubyGems package. Zimmermann et al. (2019)[97] show each direct dependency averages 80 transitive ones, creating broad attack surfaces that adversaries exploit through poisoning and dependency confusion.

Despite standard mitigations like dependency lockfiles (package-lock.json, Gemfile.lock), auditing tools (npm audit, bundle audit), and SBOM generation (CycloneDX, SPDX), high counts remain highly exploitable. This can be seen from incidents, such as event-stream, where [34] a malicious maintainer injected a bitcoin-stealing flatmap-stream, infecting 2M+ projects via transitive paths.

[5] `https://chaoss.community/kbtopic/all-metrics/`

Therefore, dependency count represents the attack surface and increases the likelihood that compromise propagates through transitive dependencies[51]. As such, the packages with high dependency count are attractive targets during adversarial reconnaissance, and are thus incorporated into the framework as a key signal for modelling susceptibility to supply chain attacks.

4.3.1.2 Dependency Depth

Another important metric is dependency depth which measures the maximum nesting level of a RubyGems package's transitive dependency tree. As per the dataset, some packages have reached transitive dependency depth of 10. Unlike dependency count, depth captures the network chain length of the dependencies. It represents invisible attack paths with unmonitored blind spots where adversaries can hide their obfuscated malware code, evading shallow scanners and audits[43].

Despite dependency tree visualisation and depth-limited policies (OpenSSF best practices), deep trees persist with unexposed risk. Codecov Bash Uploader (2021)[17] hid in deep transitive layers, compromising 1 million plus CI pipelines. The chalk/debug (2025) incident [36] illustrates how even shallow exploits can scale through dependency amplification mechanisms. As stated by Pashchenko et al.[68], deep chains create persistent blind spots where compromise hides from standard tools. Thus, deeper dependency chains increase structural blind spots and enable stealthier compromise paths, making such packages more attractive to attackers and making it a relevant factor in our scoring framework.

4.3.1.3 File Count

From a structural perspective, this metric reflects the total number of files in a RubyGems package's GitHub repository. Unlike dependency metrics (external networks), it measures the internal codebase size. A high number of files can overwhelm human review and also make it easier for an adversary to hide malware among legitimate files, such as config files, tests, and documents. The current SCA tools detect known malware signatures but miss novel code in bloated repositories. Although maintainers review the core functionality lib files, they ignore the boilerplate utils such as test utilities, string helpers or translations.

Since large codebases create ideal hiding spots for malware, maintainers and downstream developers rarely audit every file thoroughly. This increases the likelihood that malicious code remains hidden within large repositories, making such packages more susceptible to compromise and therefore a meaningful signal for modelling susceptibility to supply chain attacks.

4.3.1.4 Directory depth

This metric measures the folder nesting level in a RubyGems package's GitHub repository (e.g., depth 5 level: lib/utils/helpers/formatters/utils.rb). Unlike file count (total volume), it captures structural complexity, creating navigation blind spots for reviewers, just like in the case of transitive dependencies.

As with file count, deep directory trees signal reviewers' oversight that adversaries exploit to bury malware in obscure paths that maintainers dismiss as infrastructure/helper utilities (e.g., spec/support/fixtures/test helpers/obscure patch.rb)

This creates navigation blind spots that reduce code visibility and allow at-

tackers to conceal malicious logic, making such packages more susceptible to compromise and thus an important signal in the framework.

4.3.1.5 Lines of Code (LOC)

Lines of Code enumerates the source code lines across .rb files in a RubyGems package's GitHub repository. Similar to file count and directory depth, high LOC overwhelms reviewer capacity, enabling adversaries to conceal malware within legitimate bulk that evades detection.

Shin et al. [76] and Moshtari et al. [57] demonstrate that code based metrics particularly code churn, code complexity, and code size can be used to identify having high probability of containing vulnerabilities prior to release. Their findings show that such vulnerability prone files can be prioritised, although they do not achieve complete coverage and are best suited for risk ranking rather than precise vulnerability detection. This reflects the likelihood of malicious code to be embedded within large volumes of legitimate code, making such packages more susceptible to compromise and a relevant signal for modelling susceptibility to supply chain attacks.

4.3.1.6 Download Count

Download count measures the number of times a package has been fetched from its respective package registry, reflecting its adoption and usage scale across the ecosystem. From an adversary's perspective, high download count packages are attractive targets for supply chain attacks as the compromise of a popular package propogates the compromise effortlessly to reach millions of downstream users and projects.

Historical attacks further illustrate this prioritisation. For example, the RubyGems Fastlane plugin attack [87, 83] in 2025 involved typosquatting based malicious packages targeting widely used CI/CD tooling to exfiltrate sensitive data.

Cross ecosystem cases further illustrate the pattern, with npm's event-stream (millions of downloads) drew a takeover attack [3], chalk/debug (2025)[36], which spread through maintainer compromise and package propagation mechanisms, and xz-utils [70] targeted for its Linux ubiquity despite modest metrics. Ohm et al. (2020) [61] analysed 174 malicious OSS packages across npm/PyPI/RubyGems, noting that attackers favour popular packages for their "attractive" propagation potential. At the same time, Duan et al. (NDSS 2021)[28] did a qualitative assessment of security features of package managers and found that popular packages with high download are prime targets for malicious compromises. This consistent pattern indicates that packages with higher download volumes are more likely to be prioritised by attackers seeking to maximise downstream impact. Therefore, download count serves as a strong indicator of attacker target selection, and is included as a key signal for modelling susceptibility to supply chain attacks.

4.3.1.7 Dependents count

Another metric that represents the usage of a packages is dependent count. This represents number of downstream packages that list it as a direct dependency. It measures the reverse dependency reach through the ecosystem graph, and hence is an important factor for targeted attacks from the threat actors' perspective.

The Solarwinds Orion attack (2020) [54][80] compromised a core network management service with over 18,000 enterprise customers including, high value government and corporate networks. The threat actors exploited Orion's key role

in downstream IT infrastructures, demonstrating how a strategic compromise through precise dependent targeting can cause far reaching impact.

Another example highlighting the importance of the dependents metric is the left-pad incident [90], where a disgruntled maintainer removed their JavaScript package, *left-pad*, from the npm registry, causing widespread crashes across thousands of dependent projects. Furthermore, the studies by Abdelkareem et al. [1] and Chowdhury et al. [16] state how dependency on trivial packages can impact an ecosystem, which increases the potential impact of compromise across the ecosystem, thus making it an attractive target for adversaries. Therefore, an important signal for modelling susceptibility to supply chain attacks.

Prior studies by Abdelkareem et al. [1] and Chowdhury et al. [16] demonstrate that even trivial packages are critical within dependency networks which amplify the after effects of a successful compromise significantly. Thus, a highly depended upon packages are attractive targets for adversaries. Consequently, the number of dependents serves as a key signal for modelling susceptibility to software supply chain compromises.

4.3.1.8 Stars

The purpose of GitHub stars is to allow community stakeholders to mark the repositories or topics of interest, which they can reference later, along with related topics. It is also used to show appreciation to the repository maintainer for their work [32]. Unlike downloads and dependent count that signify the usage of package, stars represent popularity.

In Borges et al.[12], they discuss the growth of popularity of the GitHub repositories while predicting project performance based on stars and in the same context

the study by Kula et al.[44]and Mujahid et al.[58] indicate how these stars become one of the key factors for developers to choose a dependency raising the chance of adoption in future. Thus, in our risk profiling framework, high star counts point towards potential future targets for adversaries, who could social engineer maintainers to enable subsequent compromises. This increases visibility of future adoption potential, making such packages more likely to be targeted by attackers and is therefore incorporated into the framework as a useful signal for modelling susceptibility to supply chain breaches.

4.3.1.9 Forks

Forks are copy of a repository sharing the same code and visibility settings with the parent/original repository.

The paper by Jiang et al.[40] states that the main reason for forking is to make changes such as bug fixes or adding additional features. Some of these forks also diverge into independent projects or serve as an experimental environment before being merged into the parent repository. In both scenarios, forks expand adversaries' opportunities for exploitation.

For example, Cao et al.[13] reference the Electrum wallet fork attack (2018)[84], in which adversaries created typosquatted forks (nicehashe) of the popular *spesmilo/electrum* Bitcoin wallet and tricked users into downloading compromised versions to steal their bitcoins.

In case of a divergent fork, the threat actors first raise its popularity before injecting it with malicious code, or they simply rely on the lack of code review when its being pulled as a dependency. This expands opportunities for malicious modification thus, making forks a relevant signal for modelling susceptibility to

supply chain attacks.

4.3.1.10 Commit Gap

The commit gap metric captures the time gap between a package's last commit and our static reference date of 11 November 2025. In other words, it indicates the last recorded development activity in the package repository. Several studies [18, 19, 4] on GitHub project maintenance show that inactivity and abandonment arise from factors such as maintainer fatigue, small contributor bases, and project discontinuance, and that unmaintained projects are more likely to remain unpatched, error prone and at risk. An empirical study by Ait et al.[2] links project survival with long term inactivity and low commit/PR activity to higher abandonment likelihood, which leads to inactivity driven supply chain risk.

From an adversary perspective, the gap signifies a lack of security patches, vulnerability disclosure follows-up and security reviews. Attackers therefore preferentially target such packages by promoting malicious forks, exploiting unpatched weaknesses that persist. Another scenarios is where threat actors target maintainer, and gain privileges via social engineering, as experienced during XZ-Utils[70] and the event-stream incidents[34].

Larger commit gaps reflects overlooked maintenance and delayed security updates making such packages more attractive targets for attackers. Therefore, commit gap serves as a direct indicator of maintenance related socio-technical vulnerability and is included as a key signal for modeling our compromise susceptibility profiling in supply chain attacks.

4.3.1.11 Issues count

Additional metric that reflect maintenance activity is the issues count. This metric measures how many new issues were opened on a repository in the six months before our cut-off date of 11-November-2025 to indicate ongoing community usage and active reporting of bugs or feature requests. Issues count is one of the factors that reflects community activity influencing the project's life cycle [50].

According to Coelho et al.[18] findings, low activity across issues and related signals is characteristic of unmaintained projects, which in our context are interpreted as a lack of community engagement and potential abandonment. This is further supported by Li et al.'s OSSARA framework[49], which demonstrates how low issue activity identify abandonment and exposes downstream consumers to persistent vulnerabilities. But vulnerabilities are not the only exploitable attack vector. Jfrog security research team reported a detailed supply chain attack technique that was caught in September 2024. The technique known as *Revival Hijack* ,[69] re-registered deleted packages (22,000) in PyPI with same name and malicious payload. CI/CD pipelines and auto-update pulled these compromised version, subsequently leading to hundreds of thousands of malicious package downloads.

The low issues indicates reduced community scrutiny and likelihood of undetected vulnerabilities and abandonment, making such packages an attractive opportunity for adversaries to compromise and thus an important signal in the subsequent methodology.

4.3.1.12 PR(pull request) count

The pull request count tally new PRs opened over the six months before our cut off date (11-November-2025). It is a count based metric that represents the level of activity by the maintainers. A high PR count usually means more external developers are involved with the project, while a low or zero PR count can suggest reduced contributor interest, weaker collaboration, or limited maintenance activity.

Though both issue count and PR count reflect community activity, issues mainly capture discussion, bug reports, and feature requests, while PRs, on the other hand, are direct attempts to change the codebase and thus closer to actual code maintenance and review work. In this sense, a low PR count is better described as a proxy for reduced project health, which in turn can raise security risk.

Studies by Coelho et al.[18], Jiang et al.[41] and Zhang et al.[95] on pull requests show that PRs are representatives of review activity, project context, and contributor relationships, which makes them a useful signal of development and maintenance dynamics.

The reduced review activity from a attacker's viewpoint is associated with weak maintenance, and hence are often exploited in supply chain breaches. Therefore it makes a meaningful signal for our profiling framework.

4.3.1.13 Contributors count

The number of contributors reflects the extent of community participation and gives an overall view of maintenance capacity, project activity, and support. A low contributor count indicates limited interest or a small and concentrated trusted

base, which can make the project easier to observe, influence, and gradually breach it. In risk terms, fewer contributors can also mean fewer independent reviewers and less resilience against adversary reconnaissance or social engineering. Contributor count can therefore serve as a useful signal for profiling a package's community level scrutiny and the reduced diversity in audits increases the likelihood of successful compromise. The contributor count hence becomes an important signal for building our profiling methodology.

Prior studies [15, 5] on open source project health treat contributor count as a sustainability signal, because it reflects how broadly work is shared, how resilient the project is to contributor loss, and whether the project has enough community support to continue over time. This can be treated as an opportunity for adversaries to use social engineering and build trust gradually over time through benign contributions, as seen in the XZ Utils case[81].

A consolidated summary of all risk metrics, their directionality, and adversary rationale is provided in Appendix A.

4.3.2 Metric Transformation Pipeline

After selecting metrics based on their relevance to an attacker's perception and respective data collection, the next step was to transform these raw values to meaningful numbers that can be effectively integrated into the risk scoring framework. The metric transformation pipeline addresses three statistical challenges: skewness, scale disparity, and mixed directionality.

- **Stage 1: Skewness correction - Log Transformation**

  The metrics, such as downloads, file counts, and stars, show an extreme

right skewness where a few of them have extremely high values, while the remaining tend to be near zero or vice versa. To compress these outliers, we used a log transformation to linearise relationships and prevent skewed values from dominating the scoring. The +1 ensures non-negative inputs.

$$\textbf{Log Transformation} \quad x' = \log(x+1) \quad where \quad x = metric_{value} \tag{4.1}$$

- **Stage 2: Normalisation - Uniform Quantile Transformation**

  This maps all metrics to a common scale regardless of their units (counts, days, size). We chose uniform QuantileTransformer(QT) because repository metrics have extreme outliers, and QT ranks data by percentile instead of raw value magnitudes, thus keeping the relative order.

  $$\textbf{QT Transformation} \quad x'' = QT(x') \quad where \quad QT : R \rightarrow [0,1] \tag{4.2}$$

  This approach aligns with the objective of ranking based risk prioritisation, where relative ordering of packages is more important than absolute value.

- **Stage 3: Directionality alignment**

  The metrics currently have mixed directionality. This stage adjusts them all to the "high score = high risk" convention to establish consistent risk signalling across all 13 metrics. The normalised values for those where lower values indicate higher risk(Issues, Contributors, PRs) are inverted as follows:

Directionality Transformation:

$$T(x) = \begin{cases} QT(\log(x+1)), & \text{for high-value high-risk metrics} \\ 1 - QT(\log(x+1)), & \text{for low-value high-risk metrics} \end{cases} \tag{4.3}$$

This three stage standardisation pipeline (log → quantile → directionality unification) establishes a consistent high score = high risk uniform scale framework across all 13 metrics (Table A.1 in Appendix A). Note that this transformed dataset compresses large numeric ranges into smaller values resulting in relatively low absolute magnitude. However the proposed framework has ranking based evaluation, therefore the absolute scale of the metrics does not affect the analysis. In the next section the scoring framework is explained in detail.

4.4 Composite Risk Scoring framework

A composite risk score in our framework is a single aggregate mean value which indicates risk susceptibility score of an package. The aggregation is hierarchical in nature. This hierarchal structure allows us to gradually build the significance of all metrics into a final score which cannot be achieved through simple averaging. This also helps in interpretation, as the composite score can be traced to its decisive factor(s). For example, we can analyse whether a score of 0.7 is influenced by poor maintenance, high impact, or complex dependencies of the packages. Hierarchical approach also complements grouped signals and account for dependence within groups. It separates variation within the group and between groups, allowing lower level and higher level effects to be modelled simultaneously[92, 89].

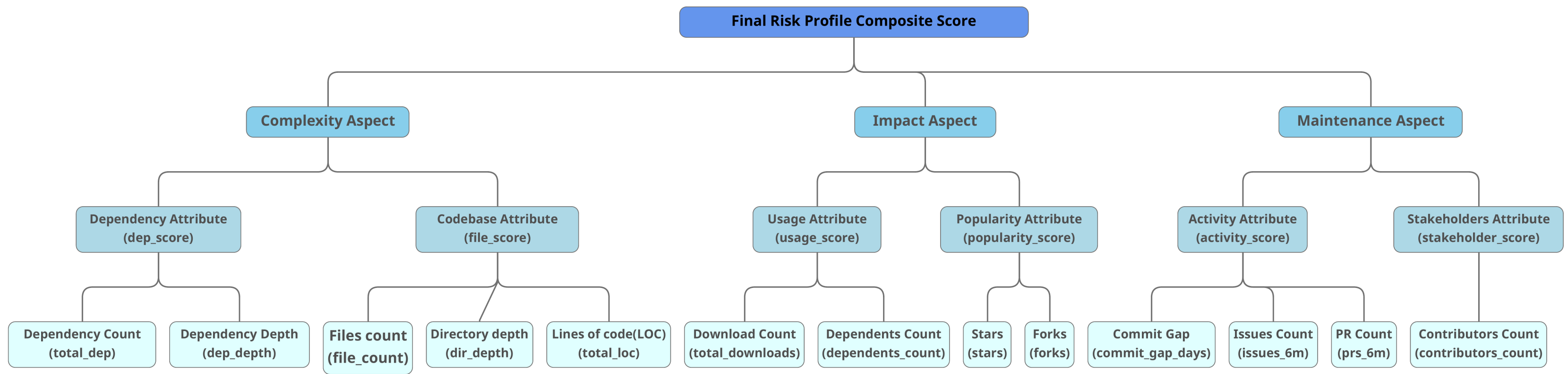


Figure 4.1: Hierarchical structure of the composite risk score.

4.4.1 Hierarchical Aggregation Stages

As stated earlier, the hierarchy of our composite score follows a bottom-up architecture, building scores progressively from metrics to the final risk profile. This hierarchy is shown in Figure 4.1. At each level, we analyse distributions across metrics, attributes, and aspects within the RubyGems ecosystem. This scoring does not represent actual risk; rather, it provides an indicative signal of why certain packages in the RubyGems dataset may attract adversarial interest for supply chain compromise. To implement the structure in Figure 4.1, aggregation is implemented across four stages:

Stage 1: Leaf Metrics Leaf metrics from Section 3.3.1 serve as initial input for scoring. Each metric undergoes normalisation, QT-transformation, and directionality adjustment described in Section 3.3.2. Metrics are then analysed to identify those best able to separate malicious from benign packages based on our tuning dataset, retaining only effective ones for Stage 2 attribute aggregation.

Stage 2: Attribute Score Computation

Attributes aggregate the metrics retained from using data driven weights as explained in Section 4.4.2.2. The weights reflect each metric's ability to distinguish malicious from benign packages. Weights for these metrics are determined by evaluating individual metric performance on the Backstabber's dataset within each specific attribute. Those showing stronger separation between malicious and benign cases receive higher normalised weights. This prioritises the most discriminatory signals during aggregation, forming robust composite attributes for subsequent analysis.

Each attribute captures key package characteristics influencing adversary in-

terest:

- **Dependency Attribute**: total_dep, dep_depth
  It indicates the supply chain exposure.
- **Codebase Attribute**: file_count, dir_depth, total_loc
  It indicates the code size and complexity.
- **Usage Attribute**: total_downloads, dependents_count
  It indicates the proxies adoption and attack surface.
- **Popularity Attribute**: stars, forks
  Ii indicates the community engagement.
- **Activity Attribute**: commit_gap_days, issues_6m, prs_6m
  It indicates the ongoing development activities.
- **Stakeholders Attribute**: contributors_count
  It indicates the maintenance capacity.

Stage 3: Aspect Formation Aspects combine related attributes into higher level categories reflecting broader package characteristics:

- **Complexity**: Dependency, Codebase
  It represents the combination of both technical and supply chain related complexity.
- **Impact**: Usage, Popularity
  It represents the potential impact or "blast radius" of a compromise.

- **Maintenance**: Activity, Stakeholders

  It represents the ongoing maintenance patterns and related trust indicators.

Stage 4: Composite Risk Profile The final score integrates the complexity, impact, and maintenance aspects into a single risk profile value. This top level indicator highlights which packages may be more likely targets within the RubyGems ecosystem.

- **Composite Risk Score**: Complexity, Impact, Maintenance

  Integrates all risk dimensions into one single attack susceptibility indicator.

This hierarchical approach builds interpretable risk signals from raw metrics in a stepwise manner. Each stage provides a different perspective on package characteristics that are relevant to the adversary during target reconnaissance, and the final composite score can be used to help prioritise supply chain security decisions.

4.4.2 Weighted aggregation for scoring

Generally, in multivariate scoring methodologies with heterogeneous factors, weightage are assigned to allow discriminative, strong features to strengthen the results while suppressing the noise from the weak ones. Similarly, for our diverse metrics framework, equal weightage does not justify their distinct significance in their abilities to distinguish highly susceptible packages. In this subsection we thus explain our weight allocation approach. To identify appropriate weights for each metric we use ROC-AUC(Receiver Operating Characteristic - Area Under Curve) graphs. The rest of this subsection describes our approach.

4.4.2.1 Understanding ROC-AUC

The use of ROC-AUC(Receiver Operating Characteristic - Area Under Curve)as a weighting mechanism is motivated by its ability to quantify the discriminatory power of individual metrics in separating malicious from benign packages. [63]. Unlike threshold dependent measures such as accuracy or precision, ROC-AUC evaluates the ranking quality of a metric across all possible decision thresholds, making it particularly suitable for a prioritisation based framework.

An AUC value of 0.5 represents random ranking, indicating no discriminatory capability. Therefore, we define the strength of each metric as the extent to which its AUC exceeds this baseline (Equation 4.6). This formulation ensures that only metrics with meaningful separation ability contribute positively to the framework, while metrics with no discriminatory signal are excluded.

The metric strengths s_k are normalised within each attribute to derive weights (Equation 4.7), ensuring that weights reflect the relative discriminatory contribution of each metric within the attribute. The resulting strength values are normalised within each aspect to produce relative weights, allowing the framework to emphasise metrics that provide stronger evidence of susceptibility to supply chain attacks while maintaining interpretability across heterogeneous signal types.

This approach supports our framework's aim to identify packages with high risk susceptibility. Specifically, using socio-technical signals from publicly available metadata indicates ease of compromise or potential compromise success from the threat actor's perspective. In our scoring methodology, these signals should exhibit high scores.

4.4.2.2 Applying ROC-AUC to Risk Scoring

We used, (a) RubyGems ecosystem metadata (primary), and (b) 380 malicious gems from the Backstabbers dataset as ground truth label to test metric level discriminatory power.

ROC curves test each metric's ability to distinguish malicious packages from benign ones by showing if malicious packages consistently receive higher scores. AUC quantifies this and represents the probability that a randomly selected malicious package scores higher than a randomly selected benign package on that metric. AUC scores with values above 0.5 demonstrate positive discriminatory power. Table 4.1 presents the AUC interpretation scale.

Table 4.1: AUC Score Interpretation [63]

AUC Range	**Performance**	**Description**
1.0	Perfect	Perfect classification across all thresholds
0.9–1.0	Excellent	Strong discrimination, minimal errors
0.8–0.9	Good	Reliable class separation, room for improvement
0.7–0.8	Acceptable	Moderate discrimination, may need tuning
0.5	Random	No better than chance (coin flip)
<0.5	Poor	Worse than random.

4.4.2.3 Hierarchical Aggregation Scoring

This section formalises the four stage hierarchical aggregation framework used to compute the composite risk score. The framework follows a bottom up structure,

aggregating metrics, attributes, aspects and finally a composite score.

Stage 1–4: Hierarchical Aggregation (See Appendix B for complete notation.)

Preliminaries and Notation

- Let $M = \{1, \dots, m\}$ denote the set of all leaf level metrics, $A = \{1, \dots, P\}$ the set of attributes, and $S = \{1, 2, 3\}$ the set of aspects (Complexity, Impact, Maintenance).
- Each attribute $p \in A$ is associated with a subset of metrics: $M_p \subseteq M$, where $\bigcup_{p \in A} M_p = M$. Each aspect $s \in S$ is associated with a subset of attributes: $A_s \subseteq A$, where $\bigcup_{s \in S} A_s = A$.
- Ground truth labels from the Backstabbers dataset: $y \in \{0, 1\}$ where $y = 1$ indicates malicious packages and $y = 0$ benign packages.

Stage 1: Metric Strength Computation

Using the Backstabbers dataset, we computed AUC for each leaf metric X_k against the 'is_malicious' target:

$$\text{AUC}_k = P(X_k(\text{malicious}) > X_k(\text{benign})) \tag{4.4}$$

where X_k is the k-th leaf metric ($k \in \{1, \dots, m\}$).

The following equation measures strength of each metric's discriminatory power for separating malicious from benign packages:

$$s_k = \max(\text{AUC}_k - 0.5, 0) \quad \forall k \in \{1, \dots, m\} \tag{4.5}$$

Metrics that achieve AUC > 0.5 yield positive strength values and form the filtered set $M^* = \{k \in M \mid s_k > 0\}$, which proceeds to the aggregation stage. Metrics with AUC ≤ 0.5, including both random performance (AUC $= 0.5$) and inverse discrimination (AUC < 0.5) are assigned zero strength and therefore receive zero weight. This approach avoids introducing noise from non-discriminatory signals while maintaining a fully data driven weighting scheme. It also eliminates the need for arbitrary feature selection thresholds, since metric inclusion is determined purely by empirical performance on the tuning dataset.

Although metrics with AUC < 0.5 could theoretically be incorporated by inverting their scores (i.e., treating them as negatively correlated discriminators), this approach was not adopted in the current study. Inverted signals may be unstable under class imbalance, sensitive to sample size, and dependent on ecosystem specific characteristics. Moreover, a metric exhibiting inverse discrimination in one dataset may not generalise consistently across other ecosystems such as npm or PyPI. For these reasons, we conservatively exclude such metrics in the present framework.

Exploring the integration of negatively weighted or inverted metrics remains an avenue for future work, particularly in larger datasets or cross ecosystem studies where such signals may demonstrate stable and transferable discriminatory value after appropriate calibration.

Stage 2: Attribute Score Computation

For each attribute $p \in A$, weights are computed over all metrics within the attribute:

$$w_{pk} = \frac{s_k}{\sum_{j \in M_p} s_j} \tag{4.6}$$

The next step is to compute the attribute score as the weighted mean of normalised metric values within the attribute:

$$A_p = \sum_{k \in M_p^*} w_{pk} \cdot X_k \tag{4.7}$$

In cases where all metrics within an attribute have zero strength, the attribute contributes no discriminatory signal and is effectively neutral in the aggregation.

Stage 3: Aspect Score Aggregation

The step following attribute score computation is aspect score computation. For each aspect $s \in S$:

$$S_s = \frac{1}{|A_s|} \sum_{p \in A_s} A_p \tag{4.8}$$

where:

$$S_{\text{Complexity}} = \frac{A_{\text{Dependency}} + A_{\text{Codebase}}}{2}$$

$$S_{\text{Impact}} = \frac{A_{\text{Usage}} + A_{\text{Popularity}}}{2}$$

$$S_{\text{Maintenance}} = \frac{A_{\text{Activity}} + A_{\text{Stakeholders}}}{2}$$

Stage 4: Final Composite Score

The composite score is the mean of aspect scores:

$$C = \frac{1}{|S|} \sum_{s \in S} S_s \tag{4.9}$$

or equivalently:

$$C = \frac{S_{\text{Complexity}} + S_{\text{Impact}} + S_{\text{Maintenance}}}{3}$$

This yields a single composite score that captures the overall risk profile of the package. It is important to note that the proposed approach is not designed as a predictive classification framework. Instead, it is formulated as a ranking based prioritisation framework that aims to order packages based on their relative susceptibility to supply chain attacks. This distinction is intentional, as the objective of this work is not to predict specific attack events, but to identify packages that are more likely to be attractive targets from an adversarial perspective.

Consequently, traditional classification metrics such as accuracy, precision, and recall are not directly applicable. Instead, evaluation focuses on ranking oriented measures, including enrichment analysis and distribution based metrics, which better reflect the effectiveness of prioritising high susceptibility packages within a large ecosystem.

Chapter 5

Evaluation

In this chapter, we apply the scoring framework to the RubyGems dataset and investigate it empirically. We evaluate the risk distribution, followed by enrichment performance in the top percentiles, and conclude with statistical validation.

5.1 Risk Distribution Analysis

In this section, we examine the distribution of composite risk scores across the RubyGems dataset to evaluate how susceptibility scores are represented within the ecosystem. It further investigates whether packages associated with our malicious labels are positioned within higher risk regions of the score distribution.

5.1.1 Dataset Overview

The primary evaluation dataset comprises 79,000 packages collected from the RubyGems ecosystem after preprocessing and data cleaning. Among these, 78,620 packages (99.52%) are benign, while 380 packages (0.48%) are labelled as mali-

cious, resulting in a class imbalance of approximately 207:1 as shown in Table 5.1. The 207:1 imbalance reflects the skewed proportions of non-malicious packages to a small set of known malicious packages. This provides a realistic evaluation setting that mirrors real-world software supply chain conditions, where compromised packages are few relative to the total ecosystem.

The Backstabbers Knife Collection[1] identifies packages that are historically associated with software supply chain security incidents. The final dataset of 380 malicious RubyGems is used as an indicator of high susceptibility to software supply chain compromise during adversary reconnaissance. It is important to note that this dataset may not capture all historical or undiscovered malicious packages. Our goal here is not to detect undiscovered malicious packages, but to relatively prioritise known ones through this framework.

Table 5.1: Dataset Composition After Preprocessing (n = 79,000)

Category	**Count**	**Percentage**
Total Packages	79,000	100.00%
Benign	78,620	99.52%
Malicious	**380**	**0.48%**

5.1.2 Metrics Performance and ROC-AUC Based Weighting

In Section 4.4.2.1, we discussed the use of ROC-AUC in our framework. In this section, we present how this approach shaped metric performance in separating malicious from benign packages.

[1] `https://dasfreak.github.io/Backstabbers-Knife-Collection/`

We first evaluate the standalone performance of each metric. This was determined by comparing the metric scores against the ground truth labels and evaluating the true positive and false positive rates across varying thresholds. The area under the resulting ROC curve was used as a measure of each metric's discriminatory power (Equation 4.4).

The next stage was to use the metric's AUC score and compute strength scores, as explained in Section 4.4.2.2, using Equation(4.6) to quantify their relative contribution. Metrics with $AUC \leq 0.5$ were excluded from contributing to the final score by assigning them zero weight (Section 4.4.2.3 - Stage 2). This makes certain that only positively discriminative signals contribute towards the aggregation process while excluding non-discriminatory ones. Upon computing the strengths, they were normalised within each attribute category to derive relative weights for hierarchical aggregation, as described in Section 4.4.2.2 using Equation 4.7.

As shown in Table 5.2, 7 out of 13 metrics achieved AUC values above the random baseline of 0.5. Among these, commit_gap_days (AUC = 0.604) and total_downloads (AUC = 0.595) demonstrated the strongest discriminatory power, highlighting the importance of maintenance inactivity and package impact in identifying high risk components. A complete breakdown of metric level ROC-AUC scores, derived strength values, and normalised weights is provided in Appendix C. Note that metric values appear numerically small due to prior log transformation and normalisation (Section 4.3.2) but AUC based evaluation solely depend on ranking order rather than the absolute magnitude.

Table 5.2: Metric Discriminatory Power and AUC Based Weights ($n = 79{,}000$)

Aspect	**Metric**	**AUC**	**Strength**	**Weight**	**Status**
Complexity	total_dep	0.545	0.045	0.530	Included
	dep_depth	0.540	0.040	0.470	Included
Complexity	file_count	0.517	0.017	0.333	Included
	dir_depth	0.530	0.030	0.585	Included
	total_loc	0.504	0.004	0.082	Excluded
Impact	total_downloads	0.595	0.095	1.000	Included
	dependents_count	0.488	0.000	0.000	Excluded
Impact	stars	0.481	0.000	0.000	Excluded
Impact	forks	0.520	0.020	1.000	Included
Maintenance	commit_gap_days	0.604	0.104	1.000	Included
	issues_6m	0.500	0.000	0.000	Excluded
	prs_6m	0.500	0.000	0.000	Excluded
Maintenance	contributors_count	0.500	0.000	0.000	Excluded

5.1.2.1 Expected vs Observed Performance

Metric selection was guided by three expectations derived from prior literature: (i) popularity related metrics should reflect impact incentive for attacker, (ii) maintenance decay should reflect ease of compromise, and (iii) dependency related complexity should reflect structural exposure.

The observed results largely support these expectations. commit_gap_days and total_downloads emerge as the strongest metrics, indicating that adversaries are more likely to target packages that maximise the package compromise impact while exhibiting weaker maintenance conditions. Complexity related metrics such as total_dep and dep_depth show moderate discriminatory power, suggesting that structural risk plays a secondary role in the RubyGems ecosystem.

An unexpected finding is the contribution of forks, which shows modest dis-

criminatory power despite limited theoretical emphasis in prior discussion. Forks as a popularity attribute in the impact aspect reflects active experimentation and alternatives to original projects[96]. Prior work by Cao et al. also shows how developers actively search for and select among alternative forks based on visibility, activity, and perceived quality [14], which presents the threat actors with an attractive opportunity for compromise and hence the discriminatory power.

In contrast, metrics such as issues_6m, prs_6m, and contributors_count remain non-discriminative in the final framework. This could be because they capture community activity in our framework, which may not be a significantly beneficial factor from the attacker's standpoint.

Overall, these results provide empirical support for the weighting scheme used in the composite risk score and motivate the subsequent analysis of how malicious packages are distributed across the ranked score spectrum.

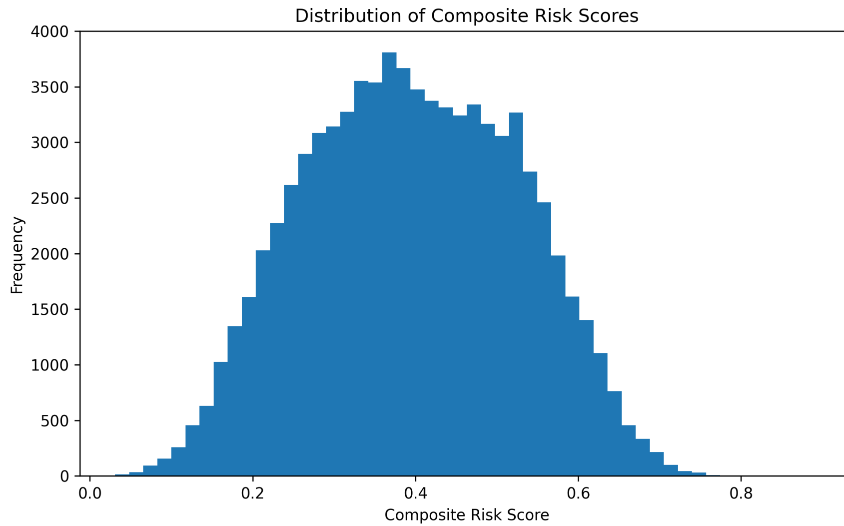


Figure 5.1: Distribution of composite risk scores across all packages.

5.1.3 Risk Distribution: Malicious vs Benign

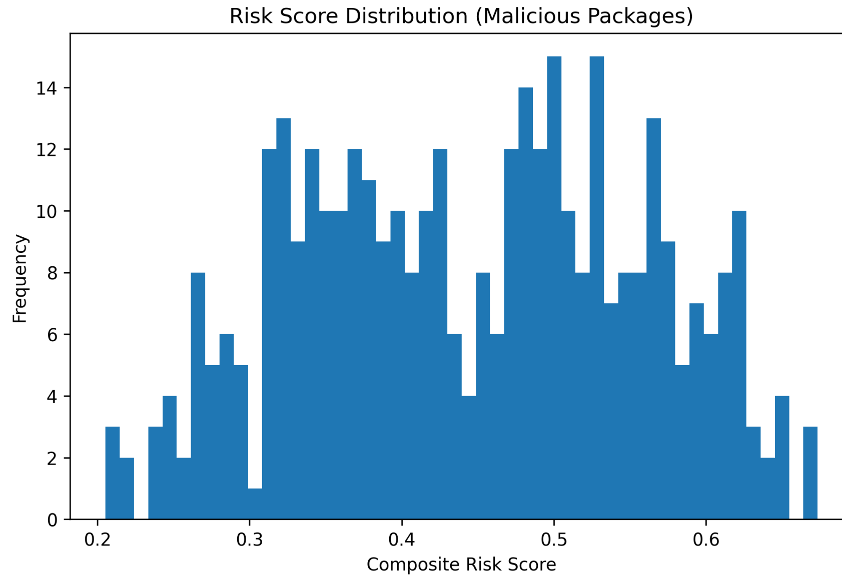


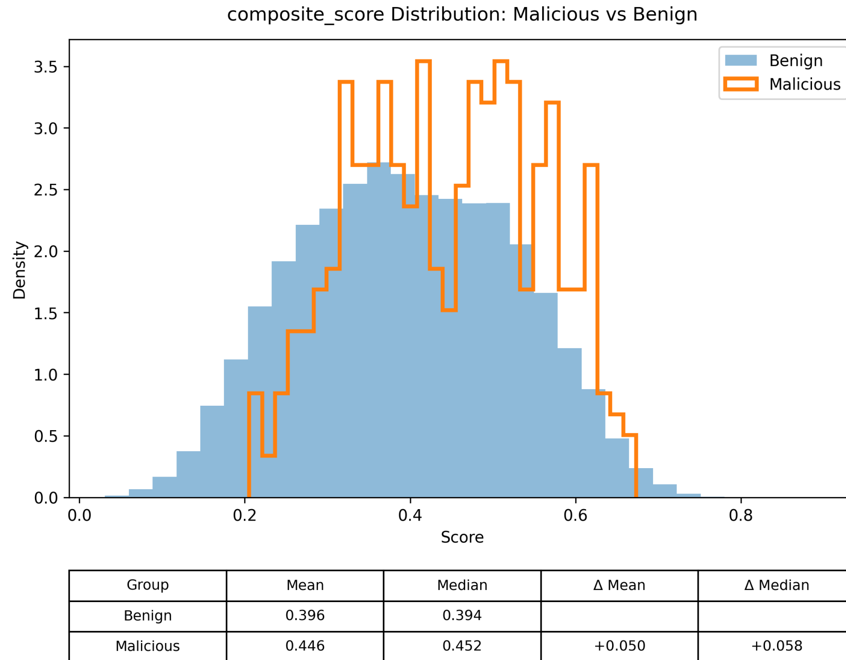
Figure 5.2: Distribution of Malicious risk scores across all packages.

5.1.3.1 Composite Score Distribution Overview

The distribution of the composite risk score across all 79,000 packages provides an initial indication of how susceptibility risk is distributed within the RubyGems ecosystem. The composite score ranges from 0.087 to 0.910, with a near symmetric distribution (skewness = 0.09), suggesting that the scoring framework does not produce extreme clustering but instead assigns scores progressively across the risk spectrum shown in Figure 5.1.

The distribution of composite risk scores for malicious packages is shown in Figure 5.2. It can be seen that the malicious packages are distributed across a broad range of scores rather than concentrated within a narrow region. This

distribution is consistent with the fact that our malicious labels do not cover all possible attack vectors, which can lead to variation in how malicious packages are represented within the scoring framework.



Group	Mean	Median	Δ Mean	Δ Median
Benign	0.396	0.394		
Malicious	0.446	0.452	+0.050	+0.058

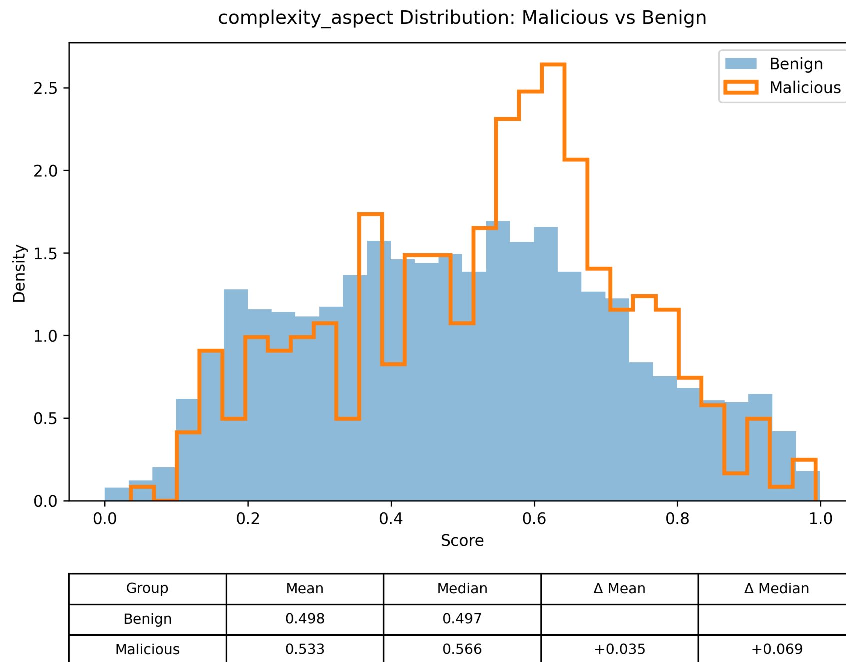
Figure 5.3: Density normalised distribution of composite scores for malicious and benign packages, showing moderate separation with overlapping regions.

5.1.3.2 Composite Score Density Normalised Comparison

Due to the significant class imbalance between benign (78,620) and malicious (380) packages, we use density normalised histograms instead of raw frequencies, to ensure comparison on a common scale.

The comparison in Figure 5.3, shows that the distribution of malicious packages is shifted towards higher composite scores relative to benign packages. Although

it can also be seen that there is substantial overlap between the two distributions. The mean composite score for malicious packages (0.446) is 12.62% higher than that of benign packages (0.396), with a corresponding 14.72% shift observed in median values (0.452 vs 0.394). This results in a mean difference of +0.050 and a median difference of +0.058, indicating a consistent upward shift in the distribution of malicious packages.



Group	Mean	Median	Δ Mean	Δ Median
Benign	0.498	0.497		
Malicious	0.533	0.566	+0.035	+0.069

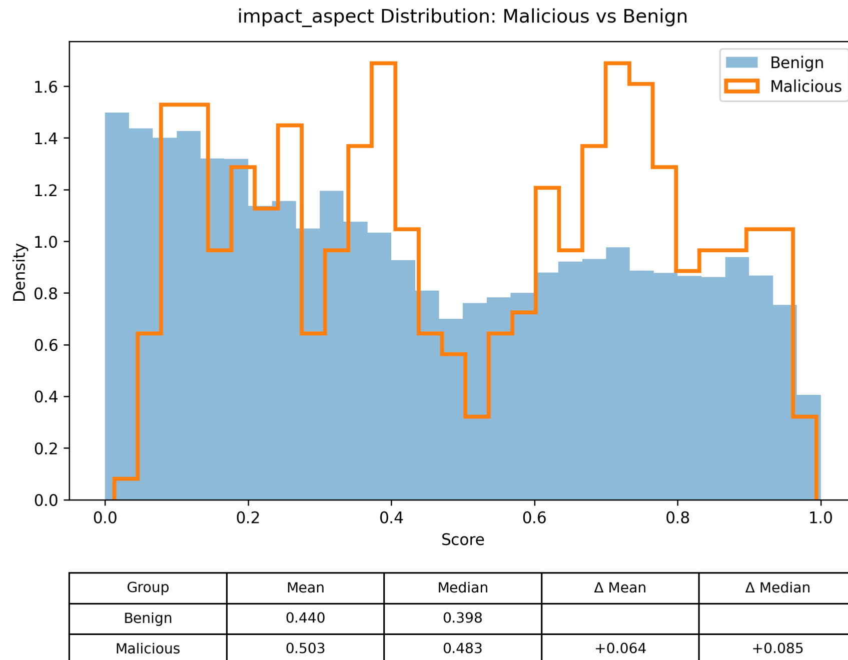
Figure 5.4: Density normalised distribution of complexity scores for malicious and benign packages.

5.1.3.3 Aspect Score Density Normalised Comparison

We further examine the individual aspect-level scores for their ability to differentiate malicious and benign packages. Similar to the composite analysis, density

normalised histograms were used to account for class imbalance and ensure comparability.

Figure 5.4 shows the comparison of bengin vs malicious packages in terms of complexity scores. As shown in the figure, the complexity scores for malicious packages exhibit a slight upward shift relative to benign packages, with considerable overlap between the two distributions. The mean complexity score for malicious packages (0.533) is higher than that of benign packages (0.498), with a corresponding increase in median values (0.566 vs 0.497). This results in a mean difference of +0.035 and a median difference of +0.069, indicating a modest separation. Fig-

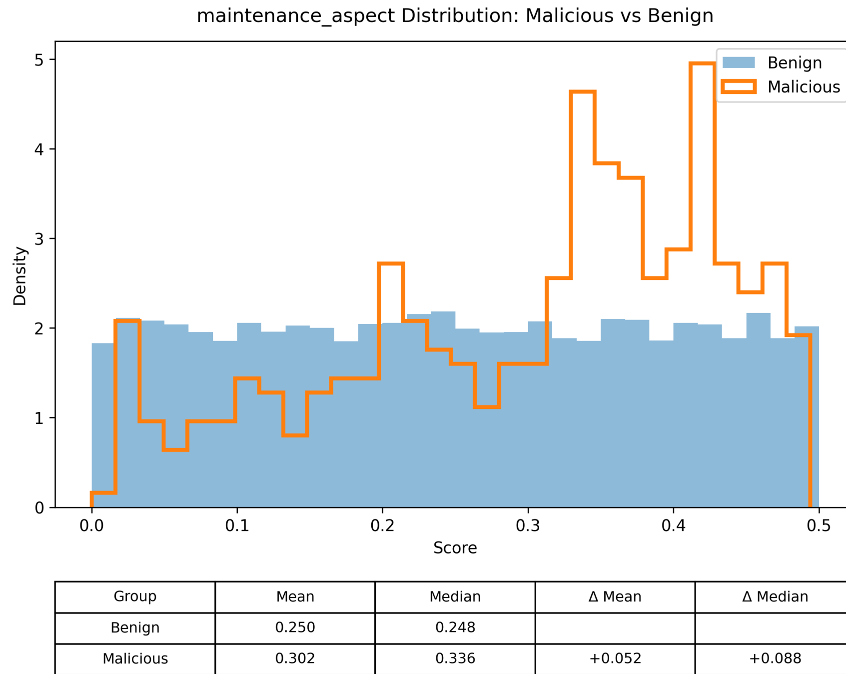


Group	Mean	Median	Δ Mean	Δ Median
Benign	0.440	0.398		
Malicious	0.503	0.483	+0.064	+0.085

Figure 5.5: Density normalised distribution of impact scores for malicious and benign packages.

ure 5.5 shows a more noticeable shift in the distribution for the impact aspect.

The mean score for malicious packages (0.503) exceeds that of benign packages (0.440), with median values of 0.483 and 0.398, respectively. This corresponds to a mean difference of +0.064 and a median difference of +0.085, indicating a stronger separation compared to complexity.



Group	Mean	Median	Δ Mean	Δ Median
Benign	0.250	0.248		
Malicious	0.302	0.336	+0.052	+0.088

Figure 5.6: Density normalised distribution of maintenance scores for malicious and benign packages.

Figure 5.6 shows the comparison of bengin vs malicious packages in terms of Maintenance scores. The maintenance aspect exhibits the most pronounced shift among the three aspects, as shown in Figure 5.6. The mean maintenance score for malicious packages (0.302) is higher than that of benign packages (0.250) showing a 20.8% difference. A larger difference of 35.48% is observed in median values (0.336 vs 0.248). This yields a mean difference of +0.052 and a median difference

of +0.088, indicating the strongest separation at the aspect level.

Overall, the comparison of aspect-level score distributions shows different degrees of separation between malicious and benign packages for each of teh aspects. The complexity aspect distribution indicates only a modest shift with considerable overlap, while the impact aspect exhibits a clearer upward shift for malicious packages. The maintenance aspect reflects the strongest separation with the largest median difference among the three. These results indicate that maintenance related signals provide the most distinct separation at the distribution level, followed by impact, with complexity offering comparatively weaker discrimination. In other words, this shows maintenance aspect emerging as the distinctive aspect where high scores may signal susceptibility to a supply-chain attack.

5.2 Tail Concentration and Enrichment Analysis

To evaluate the prioritisation capability of the proposed scoring framework, a top percentile concentration analysis was conducted. Packages were ranked in descending order based on their composite risk score, and the concentration of malicious packages within top-ranked segments was examined relative to a random baseline.

To interpret the observed concentration of malicious packages, a random baseline is used as a reference. Given that 380 out of 79,000 packages are labelled as malicious, the expected number of malicious packages in a subset of size k is given by:

$$Expected_baseline(E_k) = (380/79{,}000) \times k. \tag{5.1}$$

This baseline represents random ranking with no capacity to distinguish between benign and malicious package scores. Observed counts of malicious packages in top percentile segments are therefore compared against this expectation to assess whether the framework effectively prioritises packages with higher susceptibility to supply chain attacks.

The enrichment measures how many malicious packages are identified within a selected top-ranked subset compared to what would be expected under random selection. It is defined as the ratio of the observed number of malicious packages to the expected random count, where values greater than 1 indicate better-than-random prioritisation.

$$\text{Enrichment} = \frac{M_k}{E_k}, \quad E_k = \left(\frac{M}{N}\right) k \tag{5.2}$$

where M_k = malicious in top-k, M = total malicious, and N = total packages.

Similarly, capture (%) represents the proportion of all malicious packages in the dataset that are included within the selected subset. It reflects how much of the total malicious population is successfully retrieved by the framework within the inspected portion of the ranking.

$$\text{Capture (\%)} = \frac{M_k}{M} \times 100 \tag{5.3}$$

5.2.1 Composite Enrichment Behaviour

Table 5.3 summarises the enrichment and captures performance for the top 1%, 5%, and 10% of ranked packages. These thresholds were selected to reflect practical analysis scenarios, where only a small proportion of packages can be examined in

Table 5.3: Enrichment and capture rates across top ranked segments of the composite score

Top % Inspected	**Score Range**	**Packages**	**Expected (Random)**	**Actual Malicious**	**Enrichment (×)**	**Capture (%)**
1%	0.6680–0.8953	790	3.8	1	0.26	0.26
5%	0.6073–0.8953	3950	19.0	30	1.58	7.89
10%	0.5679–0.8953	7900	38.0	60	1.58	15.79

detail due to scale. A full threshold wise enrichment breakdown from 1% to 100% of the ranked package list is provided in Appendix D.

The results show that the framework achieves consistent enrichment within the upper ranking range. At the top 10%, the framework captures 60 malicious packages compared to an expected 38 under the random baseline, corresponding to an enrichment of 1.58× and a capture rate of 15.79%. A similar level of enrichment is observed at the top 5%, where 30 malicious packages are captured compared to an expected 19, again corresponding to 1.58× enrichment. We can see that composite score provides improved prioritisation over random selection across practical inspection ranges, as also illustrated in Figure 5.7. In contrast, the top 1% segment does not exhibit enrichment (0.26×), capturing only a single malicious package compared to an expected 3.8 which indicates that malicious packages are not concentrated exclusively at the extreme high end of the score distribution.

Overall, this demonstrates that the proposed scoring framework provides some prioritisation capability. While the enrichment levels are moderate, they consistently exceed random selection and therefore the composite score can be a helpful first step in identification of high risk packages. Figure 5.7, shows the concentration

curve over the entire percentile range.

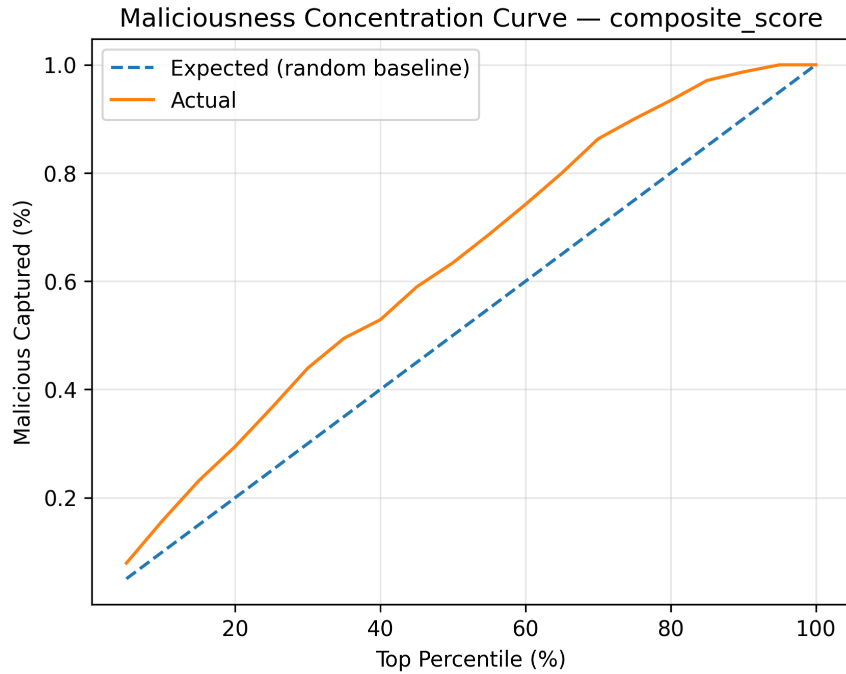


Figure 5.7: Actual versus expected cumulative malicious capture across ranked packages for the composite score. The dashed line represents the random baseline, while the solid line shows the observed cumulative capture produced by the scoring framework.

5.2.2 Aspect Level Behaviour

To examine aspect-level ranking behaviour, the same concentration analysis (using Equations 5.2 and 5.3) was applied separately to the complexity, impact, and maintenance aspect scores. In each case, packages were ranked in descending order according to the aspect score, and cumulative malicious capture was compared against the expected random baseline.

Enrichment and capture values were then computed to assess how effectively each aspect prioritised malicious packages across ranked inspection ranges.

Complexity Aspect As shown in Figure 5.8, the complexity aspect exhibits the weakest concentration behaviour among the three aspects. At the top 5%, enrichment is 0.63×, remaining below random at the top 10% (0.74×). However, the curve gradually rises above the random baseline at broader inspection ranges, indicating that the contribution of complexity is more diffuse and becomes more visible beyond the highest-ranked segments.

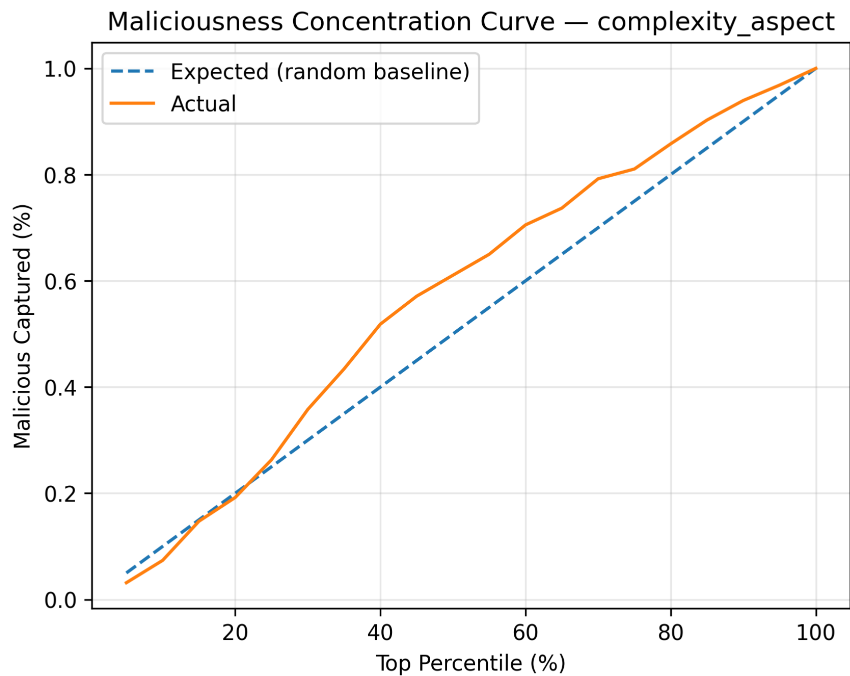


Figure 5.8: Actual versus expected cumulative malicious capture across ranked packages for the complexity aspect.

Impact Aspect The impact aspect performs better than complexity and shows a clearer deviation above the random baseline, as illustrated in Figure 5.9. At the top 10%, enrichment reaches 1.11×, and the cumulative capture curve remains consistently above baseline as inspection increases. This indicates that impact-related signals provide moderate but stable prioritisation performance.

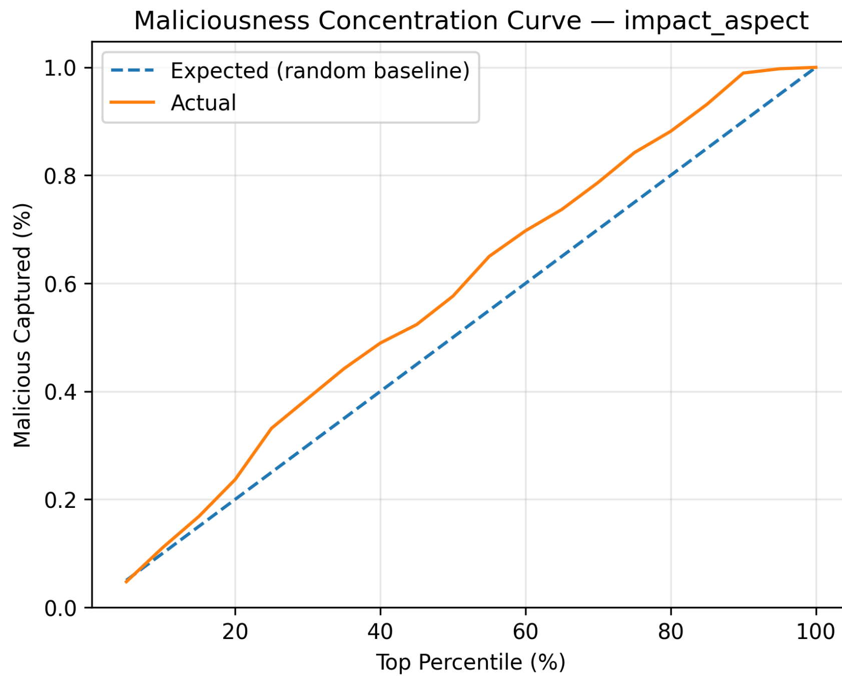


Figure 5.9: Actual versus expected cumulative malicious capture across ranked packages for the impact aspect.

Maintenance Aspect Among the three aspects, the maintenance aspect shows the strongest concentration behaviour, as shown in Figure 5.10. Although the top 5% begins below random, enrichment rises above baseline at broader inspection ranges and becomes the strongest aspect-level signal, reaching 1.38× at 20% and

1.46× at 30%. The cumulative capture curve remains above the random baseline throughout much of the ranked range, indicating the strongest prioritisation performance among the three aspects.

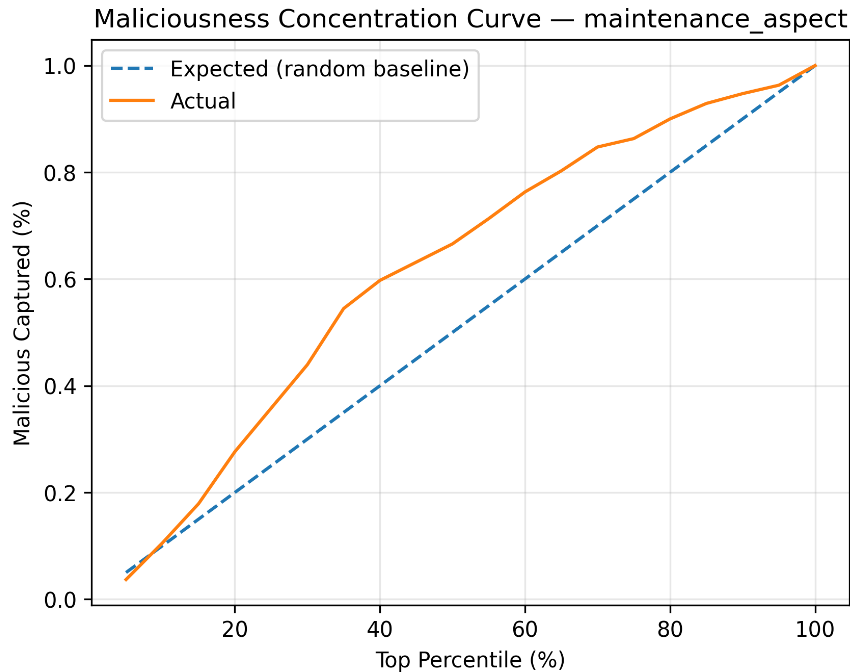


Figure 5.10: Actual versus expected cumulative malicious capture across ranked packages for the maintenance aspect.

5.2.3 Aspect Level Comparison Summary

The aspect-level results show clear differences in prioritisation behaviour across the three components. As summarised in Table 5.4 and illustrated in Figures 5.8, 5.9, and 5.10, the maintenance aspect exhibits the strongest concentration of malicious packages, particularly beyond the top 20% inspection threshold. The impact aspect provides moderate but consistent enrichment across ranked segments, while

Table 5.4: Comparative enrichment and capture rates across aspects at representative inspection thresholds (10%, 20%, and 30%)

Aspect	**Top %**	**Expected (Random)**	**Actual Malicious**	**Enrichment (×)**	**Capture (%)**
Complexity	10%	38	28	0.74	7.37
Complexity	20%	76	73	0.96	19.21
Complexity	30%	114	136	1.19	35.79
Impact	10%	38	42	1.11	11.05
Impact	20%	76	90	1.18	23.68
Impact	30%	114	147	1.29	38.68
Maintenance	10%	38	40	1.05	10.53
Maintenance	20%	76	105	1.38	27.63
Maintenance	30%	114	167	1.46	43.95

the complexity aspect shows the weakest early-stage performance and only exceeds the random baseline at broader inspection ranges.

Taken together, these results indicate that maintenance contributes the strongest aspect-level prioritisation signal, followed by impact, with complexity providing comparatively weaker discriminatory behaviour. This supports the results obtained from the distribution analysis.

5.3 Statistical Validation using the Kolmogorov Smirnov (KS) Test

To further quantify and validate the separation between the two distributions, we conducted Kolmogorov–Smirnov (KS) statistical tests. This test is a non-parametric statistical method that measures the maximum vertical distance between the empirical cumulative distribution functions (ECDFs) of two samples,

providing a quantitative measure of separation between two distributions[59].

The KS statistic ranges from 0 to 1 and measures the maximum difference between the cumulative distributions of two groups [55]. A value close to 0 indicates that the two distributions are very similar, while a value closer to 1 indicates greater separation between them. In practice, lower values suggest minimal separation, moderate values indicate partial separation, and higher values reflect stronger discriminatory ability.

$$KS = \sup_x |F_{\text{malicious}}(x) - F_{\text{benign}}(x)| \tag{5.4}$$

The test on the composite score resulted in a KS statistic of 0.175 between the two distributions. indicating a moderate level of separation between the two distributions (Figure 5.11). This result is consistent with the distributional shift observed in Section 5.1.3.2.

5.3.1 Aspect-Level KS Separation Analysis

Table 5.5: KS statistics for aspect-level separation between benign and malicious packages

Aspect	**KS Statistic**	**Figure**
Maintenance	0.210	Figure 5.14
Impact	0.107	Figure 5.13
Complexity	0.125	Figure 5.12

The KS test was further applied at the aspect level to evaluate the separation between malicious and benign packages across two distributions(Table 5.5).

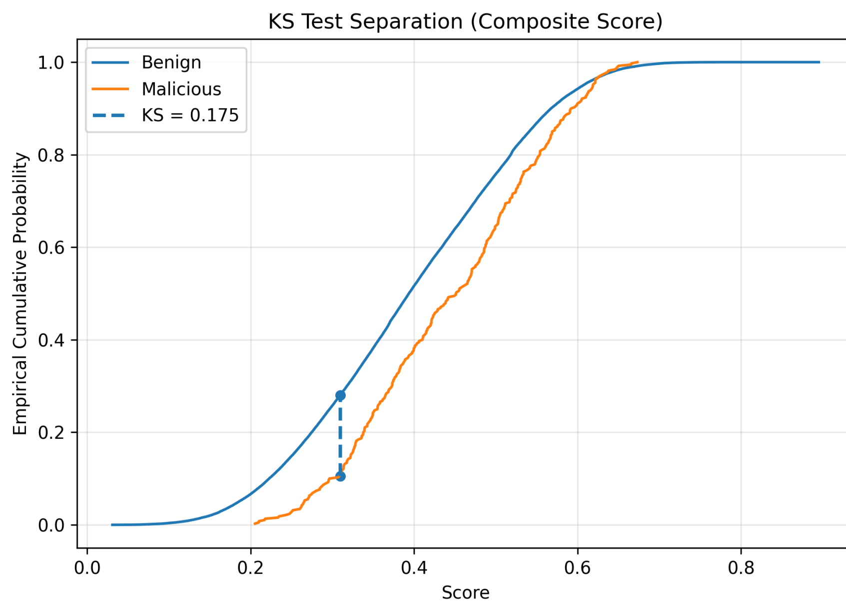


Figure 5.11: Empirical cumulative distribution functions (ECDF) of composite scores for malicious and benign packages. The maximum vertical separation corresponds to the KS statistic.

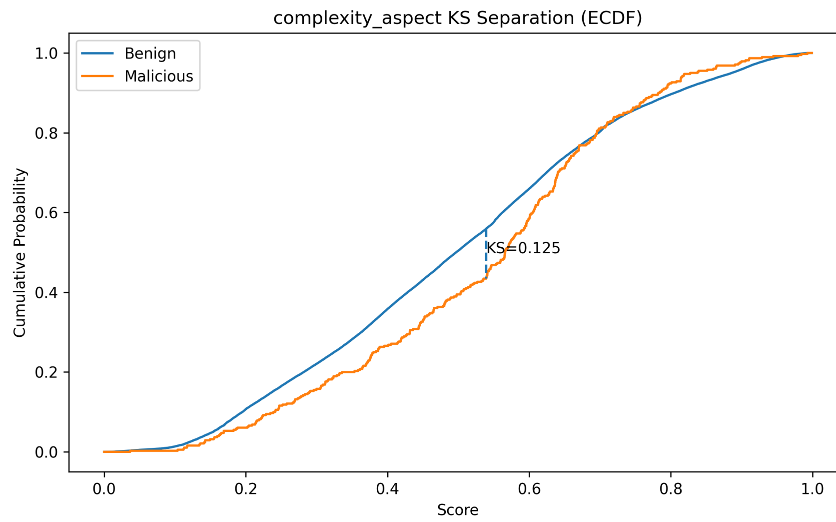


Figure 5.12: ECDF of complexity scores for malicious and benign packages. The maximum vertical separation corresponds to the KS statistic.

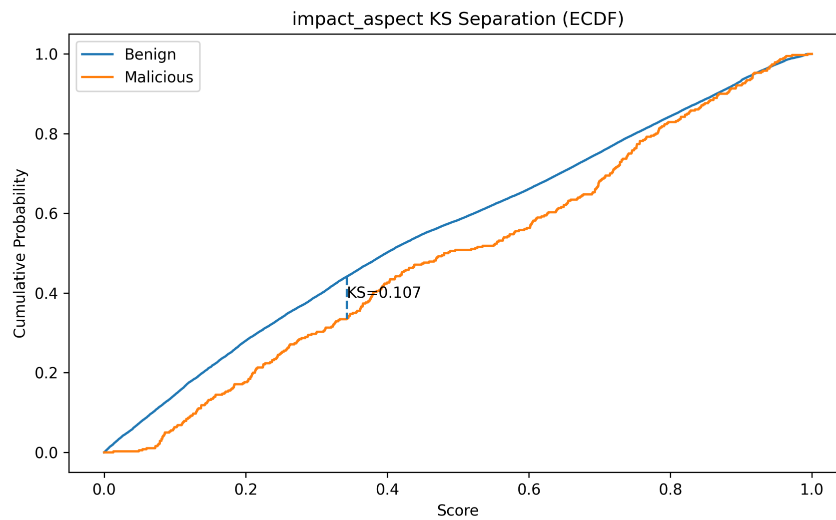


Figure 5.13: ECDF of impact scores for malicious and benign packages. The maximum vertical separation corresponds to the KS statistic.

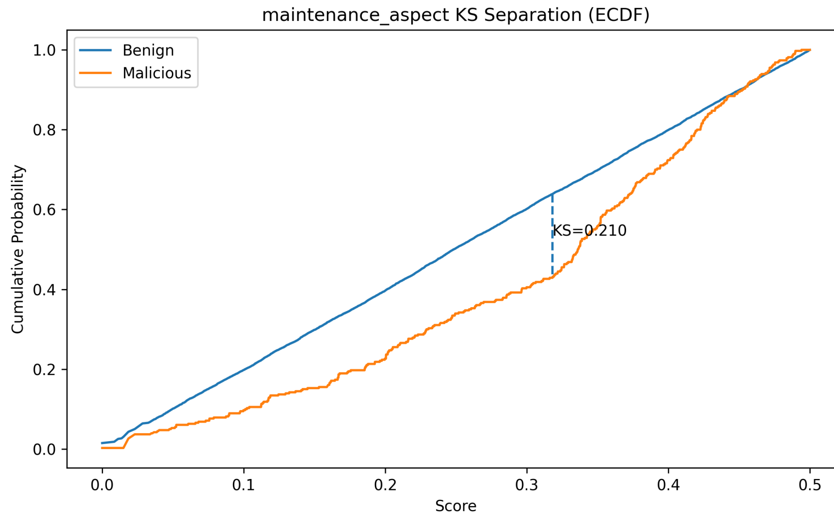


Figure 5.14: ECDF of maintenance scores for malicious and benign packages. The maximum vertical separation corresponds to the KS statistic.

The results once again show that the maintenance aspect exhibits the strongest separation (KS = 0.210), indicating that maintenance-related signals provide the most discriminative power among the three aspects. This aligns with earlier observations from the enrichment analysis, where maintenance metrics demonstrated relatively stronger individual performance.

The impact aspect shows moderate separation (KS = 0.107), suggesting that while impact contributes to identifying high-value targets, it is less effective in distinguishing malicious from benign packages in isolation. Similarly, the complexity aspect demonstrates limited separation (KS = 0.125), indicating weaker discriminative capability compared to maintenance.

Overall, these findings reinforce that maintenance-related signals play a more significant role in differentiating malicious and benign packages, while impact and

complexity contribute more to prioritisation rather than direct separation.

5.4 Summary

The evaluation results indicate that the proposed scoring framework achieves moderate separation between malicious and benign packages (KS = 0.175), with consistent enrichment observed across top-ranked segments (5%–10%). At the aspect level, the KS-based analysis further shows that maintenance provides the strongest discriminatory signal (KS = 0.210), followed by impact (KS = 0.107), while complexity exhibits comparatively weaker separation (KS = 0.125).

The results demonstrate that the framework captures meaningful socio-technical signals for prioritising higher-risk packages. In particular, maintenance-related signals contribute more strongly to distinguishing malicious from benign packages, whereas impact and complexity play a more complementary role in shaping prioritisation rather than direct separation.

This addresses our research question (Section 1.2), demonstrating that socio-technical metadata signals can be used to capture susceptibility to software supply chain attacks from an attacker-centric perspective. A complete separation for high susceptibility risk packages is neither expected nor observed, and we find that substantial overlap exists between the two distributions. Despite this, our framework achieves meaningful differentiation at the distribution level, where higher-ranked segments exhibit a greater concentration of malicious packages compared to lower-ranked regions. This directly addresses our research question, as it reflects that socio-technical metadata signals can be used to capture susceptibility from an attacker-centric perspective.

Further in our quantitative analysis, we observe that maintenance-related signals provide the strongest discriminatory power, followed by the impact aspect, while contributions from complexity aspects are secondary. These signals are integrated in final composite score, enabling us to combine multiple dimensions of risk into a unified representation of susceptibility. Overall, our findings suggest that while susceptibility cannot be determined deterministically, it can be effectively characterised in probabilistic terms using observable socio-technical metadata.

To further examine how these signals manifest in real-world scenarios, the following chapter presents case studies of compromised packages and analyses their behaviour under the proposed scoring framework.

Chapter 6

Case Study Analysis and Discussion

This chapter discusses real-world software supply chain compromise in the RubyGems ecosystem using our risk scoring framework and interprets the findings.

The quantitative evaluation in Chapter 5 provided us with an overall assessment of our framework. However, the analysis is limited by the lack of datasets documenting software supply chain attacks in the RubyGems ecosystem. In particular, there is no comprehensive dataset of malicious RubyGems packages that captures a wide range of attack techniques suitable for large-scale quantitative analysis. The Backstabbers dataset, used in this study as a proxy for risk susceptibility, has already been utilised during the risk score computation process and also for the evaluation of the scoring framework. Yet, it is primarily biased towards dependency confusion or typosquatting attacks (Section 3.2.2.2). As such, it does not fully capture the diversity of real-world attack techniques observed in software supply chain compromises. Hence, to address this limitation, case study analysis

is employed to examine specific, documented incidents involving different types of attacks, including malicious code injection and maintainer compromise.

This chapter begins with a qualitative analysis of selected case studies, followed by a brief discussion of how these results relate to the quantitative analysis in Chapter 5 and what they imply for attacker-centric target selection and socio-technical risk signals.

6.1 Case Studies

We examine three case studies in this section, given the limited number of well-documented software supply chain compromise incidents within the RubyGems ecosystem. These case studies illustrate different compromise mechanisms and demonstrate how the proposed framework captures varying attacker strategies across real-world scenarios.

Table 6.1: Summary of selected real-world compromised RubyGems packages used for case study analysis.

Package	**Compromise Type**	**Composite Score**	**Rank (%)**	**Maint. (%)**	**Impact (%)**	**Complexity (%)**	**Year**
bootstrap-sass	RCE backdoor	0.626	96.76	36.66	99.96	79.96	2019 [82]
rest-client	Maintainer hijack	0.527	82.11	1.49	99.89	64.24	2019 [26]
strong_password	RCE payload	0.510	78.04	39.73	96.93	33.65	2019 [88]

6.1.1 bootstrap-sass

Bootstrap is a popular front-end framework for web applications, and bootstrap-sass[1] is a Sass-based RubyGems package for easier customisation and reuse in Rails applications. It has widespread adoption as a dependency (73M downloads), and a compromise would impact many downstream applications that rely on it.

In March 2019, RubyGems bootstrap-sass was compromised (CVE-2019-10842) by the introduction of a hidden remote code execution (RCE) backdoor in version 3.2.0.3[82]. The breach was discovered when a developer had a build failure in their project and found that an earlier version (v3.2.0.2) had been removed and quickly replaced by v3.2.0.3. It raised suspicion because the new version was uploaded directly to the RubyGems registry without corresponding changes in the GitHub repository. The issue was promptly reported to the maintainers, and the malicious version was identified and yanked on the same day. It was found later that one of the maintainer's credentials had leaked (old and reused password across multiple platforms), which led to the compromise. The observed impact of the bootstrap-sass compromise appears to have been limited due to the immediate detection and removal of the malicious version. However, the potential impact would have been significant with large scale propagation of the injected backdoor across dependent applications.

In our framework, the bootstrap-sass package has a composite score of 0.626 and ranks at the 96.76th percentile (Table 6.1), placing it within the high-risk range. This indicates a high level of susceptibility to supply-chain compromise. The ranking, as we see, is primarily driven by the impact aspect, where the package

[1] `https://rubygems.org/gems/bootstrap-sass/versions/3.4.1`

reaches an extremely high percentile (99.96), reflecting its widespread adoption and the substantial downstream consequences associated with its compromise. This aligns with the quantitative analysis, where impact-related signals contribute to the identification of high-value targets, particularly in widely used packages with large propagation potential (Figure 5.5, Figure 5.9).

The maintenance percentile (36.66) indicates moderate activity, suggesting that the package was not an obvious target based on inactivity. This differs from the quantitative analysis, where maintenance-related signals showed stronger separation between malicious and benign packages (Figure 5.6, Figure 5.10). This case reflects a scenario where maintenance does not act as a dominant driver of susceptibility. This difference can be attributed to the nature of the attack, which was enabled by compromised maintainer credentials rather than weaknesses in repository activity. In such cases, high or moderate maintenance does not necessarily prevent compromise, as the attack originates from external access rather than observable socio-technical signals.

The complexity percentile (79.96) indicates above-average structural complexity, contributing to the overall risk score. However, its influence remains secondary compared to the impact aspect, consistent with the quantitative analysis showing weaker separation and prioritisation performance for complexity (Figure 5.4, Figure 5.8)

Taken together, this case reflects a risk profile primarily driven by the impact aspect, with maintenance and complexity contributing to a lesser extent. It demonstrates how the proposed framework captures high-impact packages as susceptible to supply chain compromise in real-world scenarios.

6.1.2 rest-client

The rest-client[2] gem is a widely used HTTP and REST client library in Ruby, enabling applications to interact with external web services. Due to its role in handling network communication, it is widely used as a dependency (4.6B downloads) in many Ruby-based applications.

The rest-client gem was compromised (CVE-2019-15224) in 2019 through a maintainer account hijack [26]. An attacker gained unauthorised access to the RubyGems account of a maintainer and published malicious versions (from 1.6.10 to 1.6.13) of the package. These versions contained obfuscated code designed to exfiltrate sensitive information from affected systems. The compromise was discovered, the malicious versions were yanked, and the compromised account was locked on the same day, but the incident demonstrated the risks associated with compromised maintainer credentials in widely used packages.

According to our scoring framework, the rest-client package has a composite score of 0.527 and ranks at the 82.11th percentile (Table 6.1), placing it within the high-risk region of the distribution. While its overall ranking is lower than bootstrap-sass, it still falls within the upper range of scores, indicating elevated susceptibility.

The maintenance percentile (1.49) is extremely low, indicating that the package is very well-maintained as per our framework. This contradicts the quantitative analysis, where maintenance-related signals showed stronger separation between malicious and benign packages (Figure 5.6, Figure 5.10). This demonstrates that for a different attack vector, maintenance does not act as a dominant driver of

[2] `https://rubygems.org/gems/rest-client/versions/1.8.0`

susceptibility.

The impact percentile (99.89), however, is extremely high, reflecting the widespread use of the package and the substantial downstream consequences of a compromise. This behaviour is consistent with the quantitative findings, where impact-related signals contribute to identifying high-value targets with large propagation potential, as represented in (Figure 5.5, Figure 5.9).

The complexity percentile (64.24) indicates above-average structural complexity, contributing to the overall risk score. This is similar to the bootstrap-sass case study, where the complexity aspect contribution is moderate, although it is not the primary factor. This aligns with the quantitative analysis showing limited separation and lower prioritisation performance for complexity during the evaluation (Figure 5.4, Figure 5.8)

From the framework's perspective, rest-client is a high-impact package whose susceptibility is primarily driven by impact, while maintenance contributes relatively little to the overall risk, reflecting a different attack vector that is not captured by our framework's malicious label.

6.1.3 strong_password

The strong_password[3] gem is a Ruby library used to enforce password strength requirements in applications, helping developers implement basic security policies for user authentication. Although its download scale is lower (3M+ downloads) compared to previous study cases, it was considered because of its role in application-level security mechanisms.

[3] `https://rubygems.org/gems/strong_password`

strong_password breach (CVE-2019-13354) in July 2019 was discovered by a developer during their own project's dependency audit. During the audit, the change log from version 0.0.6 to 0.0.7 was missing. This raised a suspicion, and on further analysis, 0.0.7 was found to be a malicious version that introduced a staged remote code execution payload that fetched and executed code from an external source [88]. The package's purpose in security aspects of an application makes it a critical component in a software supply chain.

Our scoring framework assigns this package a composite score of 0.510 and ranks it at the 78.04th percentile (Table 6.1), indicating a moderate-to-high level of susceptibility within the distribution. The maintenance percentile (39.73) and complexity percentile (33.65) indicate moderate contributions from both aspects, with neither acting as a dominant driver of risk. In contrast to the quantitative analysis, where maintenance-related signals showed stronger separation between malicious and benign packages, this case reflects a scenario where maintenance does not significantly influence susceptibility, while complexity remains a secondary signal with weaker separation and prioritisation performance (Figure 5.6, Figure 5.4, Table 5.4)

The impact percentile (96.93), however, is notably high, indicating that the package still presents meaningful downstream consequences despite the smaller-scale downloads compared to the previous two case studies. However, as observed in the previous cases, impact-related signals contribute to identifying packages with potential breach propagation reflected in their distributional shift and enrichment above the random baseline (Figure 5.5, Figure 5.9)

From the framework's perspective, this case reflects a risk profile primarily driven by impact, while other aspects contribute to a lesser extent for a different

attack technique(malicious injection).

6.1.4 Cross-Case Analysis

The case study analysis presents an interesting picture. In each of the cases, it is the impact aspect that plays a dominant role in driving the overall risk ranking, with each package exhibiting very high impact percentiles (above 96), reflecting their potential for downstream propagation in the open-source RubyGems ecosystem.

The contribution of maintenance in all study cases varies and also contradicts the quantitative evaluation in Chapter 5, which identified maintenance as a strong discriminator between malicious and benign packages. The case studies, on the other hand, show that maintenance does not consistently act as a dominant factor in real-world compromises.

This disconnect between the results of the quantitative analysis and the case study analysis can be attributed to the composition of the Backstabbers dataset on which the analysis of the previous chapter was conducted. The Backstabbers dataset overwhelmingly features typosquatting-type attacks, while the packages in the case studies are all of malicious version injection type. This in turn may indicate that high maintenance is a signal associated with typosquatting type attacks while high impact is a signal associated with malicious version injection type attacks. These observations highlight that different attack mechanisms may influence which socio-technical signals are most relevant.

Taken together, the case studies illustrate that while the framework effectively captures high-impact targets, the relative importance of other aspects, particularly

maintenance, depends on the nature of the attack vector. This reinforces the need to interpret risk scores in the context of different compromise mechanisms rather than relying on a single dominant signal. To illustrate this point, we discuss a set of compromised packages from the Backstabbers dataset in the next section.

6.1.5 Malicious Packages from Backstabbers dataset

The previous case studies were selected because they were well documented real world incidents. However, they lacked insight into the most prevalent type of software supply chain attack vector, namely dependency name confusion (also known as typosquatting, brand hijacking, etc.). In this subsection we examine three malicious packages from the Backstabbers dataset that specifically relate to this attack vector. These packages represent different types of utility libraries and tools but all exhibit high maintenance score which we posit is a consequence of the attack vector.

Table 6.2: Selected dataset-derived malicious packages with high maintenance risk

Package	**Composite (%)**	**Maint. (%)**	**Impact (%)**	**Complexity (%)**
activerecord-duplicate	91.42	98.86	92.61	55.99
capistrano-copy-subdir	98.27	99.74	99.43	84.33
aasm_history	69.95	97.47	64.05	47.66

6.1.5.1 activerecord-duplicate

The activerecord-duplicate gem, a utility extension for ActiveRecord, exhibits a high composite score (above the 90th percentile) driven by both maintenance and impact aspects (Table 6.2). From our perspective, this reflects weak maintenance

characteristics, such as infrequent updates and limited monitoring, which may increase the likelihood of unpatched vulnerabilities. At the same time, its naming and association with the widely used ActiveRecord framework may introduce elements of dependency confusion or naming-based selection, where developers may include the package based on familiarity rather than careful verification.

Within our framework, this combination places the package in a higher susceptibility region, where attackers may find a favourable balance between effort and payoff. In real-world scenarios, utility libraries extending core frameworks are often implicitly trusted by developers, and this trust, combined with naming similarity and weak maintenance, may allow malicious changes to persist undetected and propagate through dependent applications.

6.1.5.2 capistrano-copy-subdir

Another gem we considered to analyse is capistrano-copy-subdir, which is used in deployment automation workflows. It demonstrates one of the highest composite scores (above the 98th percentile) (Table 6.2). In addition to its extremely high maintenance percentile, the package also exhibits very high impact and complexity percentiles, indicating that it lies in a critical region of the ecosystem where both propagation potential and structural exposure are elevated. Within our framework, this combination reflects a convergence of multiple risk factors, placing the package among the most susceptible components and representing a high-value target across all aspects.

From an attacker's perspective, deployment tools are particularly attractive because they operate within trusted build and release pipelines. If compromised, such components could enable multiple attack vectors, including malicious code injec-

tion during deployment, tampering with build artefacts, or introducing backdoors into downstream applications. In addition, packages within well-known ecosystems such as Capistrano may also be susceptible to dependency confusion or naming-based attacks, where similarly named extensions are unintentionally selected during installation. This further increases their attractiveness, as attackers can exploit both their high impact and their position within trusted dependency resolution processes.

6.1.5.3 aasm_history

The aasm_history gem, an extension within the AASM ecosystem, exhibits a moderately high composite score along with a very high maintenance percentile (Table 6.2). While its impact and complexity percentiles are lower compared to the previous cases, its position within the AASM ecosystem highlights a different form of susceptibility. Within our framework, this suggests that the package is not primarily driven by propagation potential or structural exposure, but by its association with a widely used parent library.

From an attacker's perspective, such extension packages can be attractive due to brandjacking-type risks, where attackers exploit the reputation of established libraries. Developers may rely on the library name as a signal of legitimacy and include extensions without extensive verification. In some cases, this may also favour dependency resolution or naming-based attacks, where similarly named packages are unintentionally selected. This creates an opportunity for attackers to introduce malicious functionality through seemingly legitimate add-ons, where the decision to use the package is influenced more by perceived trust than by its individual characteristics.

These examples reinforce the observation that maintenance-related signals play a significant role in identifying susceptible packages. However, this may be strongly co-related to the attack vector used for the attack, in this case the vector being typosquatting.

6.2 Discussion

In this section, we interpret the key results of our work in the context of software supply chain security. It includes insights from both quantitative evaluation and case study analysis to examine attacker-centric target selection and the role of socio-technical risk signals.

6.2.1 Metric Discriminatory Strength

Our scoring framework integrates multiple metrics representing different socio-technical risk signals into a unified composite score through a hierarchical bottom-up architecture. To examine the contribution of individual metrics, we analysed their ability to distinguish between malicious and benign packages.

Overall, we observed that individual metrics provide varying levels of discriminatory strength, with no single metric showing strong separation on its own. While all metrics of structural complexity exhibit positive discriminatory behaviour, their influence remains limited, which can be seen in risk distribution analysis. However, the commit-gap metric reflects stronger discriminatory power, capturing maintenance inactivity and indicating potential ease of compromise. Similarly, for the impact aspect, both forks and total downloads show positive discriminatory power. Although the influence of forks is relatively low, it suggests that targeting forked

repositories is an attractive option for adversaries. Total downloads reflect ecosystem impact, thereby highlighting high-value targets.

6.2.2 Distribution of Risk Score

The distribution of the final composite score, as well as aspect scores, suggests that, contrary to our expectations, software supply chain risk signals are not concentrated in higher scores but are instead distributed across the score range. From observing the separation between malicious and benign packages, we observe a gradual shift in scores instead of a steep rise, indicating that risk exists along a spectrum. This indicates that attackers' target selection, in practice, does not operate by relying on a single signal but instead considers a combination of factors such as package popularity, maintenance conditions, and the ease of exploiting trust relationships or gaining privileged access.

Attackers appear to make trade-offs between impact and effort, selecting targets that offer a balance between reach and feasibility of compromise rather than strictly maximising one dimension. This also reinforces that the scoring framework captures relative susceptibility and not deterministic risk, where higher scores indicate increased likelihood but do not guarantee that a package will be targeted or compromised.

6.2.3 Role of Socio-Technical Signals

Our evaluation at the aspect level highlights how different socio-technical signals contribute to susceptibility, but more importantly, how they behave differently when viewed from an attacker's perspective. From our quantitative analysis on

the Backstabbers dataset, we observe that maintenance-related signals show the strongest association with malicious packages. This aligns with our expectation that weaker maintenance, such as infrequent updates or limited community engagement, etc. creates opportunities for attackers by reducing oversight and increasing the likelihood of unnoticed compromise. In this sense, maintenance can be interpreted as a proxy for opportunity, particularly in scenarios involving long-term attacks or social engineering, where adversaries gradually exploit trust relationships.

However, when we relate this to real-world scenarios, as we did in the case studies, we also observe a deviation from this expectation. Maintenance does not consistently act as a dominant factor across all attack types. This suggests that while maintenance is a strong statistical signal, it does not fully capture all attack vectors, highlighting a limitation of relying solely on observable repository activity. Our case studies showed that impact-related signals are also a strong indicator of attractiveness for a compromise. We interpret the role as these indicators as capturing the potential payoff from a successful compromise. While high impact alone does not guarantee that a package will be targeted, we observe that widely used packages remain consistently attractive due to their propagation potential and downstream consequences. From an attacker's perspective, this reflects a clear incentive, where even moderately accessible packages may be prioritised if they offer significant reach within the ecosystem.

Structural complexity, in comparison, appears to play a secondary role in our framework. We observe that while complexity contributes to the attack surface through dependency relationships and hidden structural pathways, it does not strongly drive target selection in isolation.

It must be noted though that certain attack vectors, such as credential compromise or external account access, may bypass signals derived from ecosystem metadata. These are opportunistic attacks that follow an attack opportunity rather than repository conditions observed in reconnaissance and therefore are not captured in this work.

6.2.4 Composite Score Formulation and Design Choice

In our framework, we constructed the composite score by combining the three aspect-level scores, namely maintenance, impact, and complexity, using a weighted aggregation approach (as described in Section 4.4.2.3 Equation 4.9).

However, from our earlier observations, we noticed that maintenance-related signals showed stronger separation in the quantitative analysis, while impact-related signals appeared more prominent in the case studies. This led us to explore whether a single dominant aspect could better differentiate malicious from benign packages. To examine this, we experimented with an alternative formulation where the composite score was computed as the maximum of the three aspect scores, instead of the mean-based aggregation used in our framework.

When evaluated using the KS test, this alternative formulation resulted in weaker separation (KS = 0.122) compared to our proposed composite score (KS = 0.175). This can be seen in Figure 6.1. This indicates that while individual aspects may appear dominant in specific scenarios, relying on a single signal does not capture the broader interaction between socio-technical factors. In contrast, combining signals through our composite formulation provides a more consistent representation of susceptibility, aligning better with how attackers consider multi-

ple factors when selecting targets.

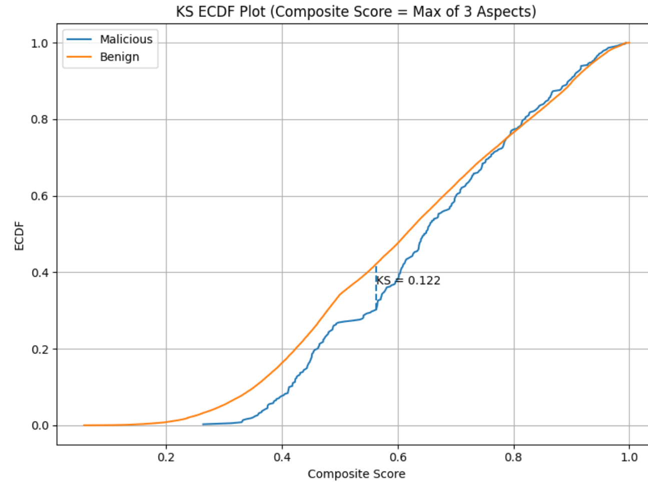


Figure 6.1: ECDF comparison for the alternative composite score (maximum of aspect scores), showing reduced separation between malicious and benign distributions (KS = 0.122) compared to the proposed composite formulation.

6.2.5 Practical Interpretation of Results

The evaluation results show moderate effect sizes, including AUC values slightly above 0.6, a KS statistic of 0.175, and enrichment of approximately 1.5×. While these values are not strong in a classification sense, we expect this behaviour in highly imbalanced open-source ecosystems, where malicious packages represent only a very small fraction of the total population. In such settings, even if the model successfully shifts malicious packages towards higher scores, the overlap with benign packages remains substantial, resulting in moderate separation rather than a clearly distinct boundary.

From our perspective, the key value of the framework lies not in precise clas-

sification but in its ability to support prioritisation. As shown in the enrichment analysis (Section 5.2), the model captures a higher number of malicious packages within the top 5–10% of ranked results compared to a random baseline. This means that, in practice, security efforts can be focused on a smaller subset of packages while still covering a larger share of potentially high-risk components, which is particularly useful in large-scale software supply chains where exhaustive inspection is not feasible.

We also observe that the overlap between malicious and benign distributions is expected, as many benign packages share characteristics with those that are later compromised. This reflects the uncertainty under which attackers operate, where target selection is based on relative advantages rather than clearly separable signals. As a result, the framework captures susceptibility in probabilistic terms, where higher scores indicate increased likelihood of being targeted rather than guaranteed compromise.

Importantly, the absence of extreme tail concentration further supports this interpretation. If malicious packages were concentrated only at the highest end of the score distribution, the framework could be interpreted as a near-deterministic classifier. However, since malicious packages are distributed across a broader range of elevated scores, we interpret this as evidence that risk is not confined to a narrow set of extreme cases. Instead, it reinforces the view that the framework functions as a risk profiling tool, capturing varying degrees of susceptibility across the ecosystem rather than producing a strict binary distinction between benign and malicious packages.

6.2.6 Comparison with Existing Approaches

Existing supply chain security approaches often focus on known vulnerabilities (e.g., CVE-based scanning) or general project health metrics [97, 25, 1, 16]. In practice, many industry tools and frameworks, such as OpenSSF Scorecard [65, 73] and related risk assessment platforms [8], rely on predefined metrics and expert-driven weighting schemes to evaluate project security and health. While these approaches are useful for identifying existing weaknesses, they do not fully capture attacker behaviour at the package level, particularly during the pre-compromise phase.

In contrast, our work models susceptibility using pre-compromise socio-technical metadata signals and adopts an attacker-centric perspective. The use of AUC-based weighting allow us to reason about the relative contribution of different factors. This differs from black-box predictive models such as machine learning-based vulnerability prediction approaches [76], where the decision-making process is often less transparent. As a result, our framework enables clearer interpretation of why certain packages are ranked as higher risk, while still capturing meaningful prioritisation behaviour.

Chapter 7

Conclusions, Limitations and Future Work

7.1 Conclusion

The objective of our work was to investigate whether socio-technical metadata signals can be used to capture the susceptibility of open-source packages to software supply chain attacks from an attacker-centric perspective.

In order to do that, we developed a socio-technical risk profiling framework that estimates the susceptibility of open source packages using pre-compromise metadata signals. Unlike existing approaches that focus on post-compromise indicators or general project health, our framework is designed to capture attacker-relevant factors before an attack occurs. We built this framework using complexity, impact and maintenance-related features into a hierarchical scoring model. The weights for individual metrics were derived using ROC-AUC analysis, fine-tuned to known malicious labels, enabling a data-driven aggregation of signals. We evaluated the

framework on a dataset of 79,000 RubyGems packages, including 380 known malicious instances, and further validated the results using distributional analysis, enrichment analysis, and statistical testing through the Kolmogorov–Smirnov (KS) test. Case study analysis was also conducted to examine how the framework behaves under different real-world attack scenarios.

The results show that the framework provides meaningful prioritisation of higher-risk packages. The susceptibility risk is not concentrated solely at the extreme high end of the distribution but is instead spread across a broader high-risk range. Moreover, malicious packages are moderately enriched in higher-ranked score regions, and the framework achieves measurable separation between malicious and benign distributions (KS = 0.175). We also observe that maintenance and impact-related signals contribute most strongly to distinguishing malicious packages compared to complexity.

These findings allow us to answer our initial research question, that is, to what extent can socio-technical metadata signals be used to capture the susceptibility of open source packages to software supply chain attacks from an attacker-centric perspective? Our results demonstrate that to some extent, these socio-technical metadata signals can be used to capture the susceptibility of open-source packages to software supply chain attacks from an attacker-centric perspective. However, this susceptibility is not deterministic. Instead, it is probabilistic, where higher scores indicate an increased likelihood of being targeted rather than a guaranteed compromise. We were able to determine that, based on attack vectors, both impact and maintenance signals can be strong discriminators, although more exploration may be needed to find a better composite score computation methodology. Overall, our research does demonstrate that packages with higher risk exhibit statistically

higher composite scores, but with a gradual shift rather than a steep rise

In conclusion, this work demonstrates that while software supply chain risk cannot be fully predicted, it can be systematically understood and prioritised using observable socio-technical signals. By adopting an attacker-centric perspective and combining multiple dimensions of risk, the proposed framework provides a practical approach for identifying susceptible packages in large-scale open-source ecosystems. This shifts the focus from reactive vulnerability detection to proactive risk awareness, supporting more effective and scalable security practices in modern software supply chains.

7.2 Limitations

Our proposed framework provides an approach to prioritising packages based on supply chain attack susceptibility, but there are several limitations that we acknowledge here.

The primary limitation of our study lies in the availability of a comprehensive malicious dataset. We rely on existing malicious packages from Backstabber's collection as a proxy for susceptibility, which does not fully capture the diversity of software supply chain attack techniques. As a result, our evaluation is constrained to the patterns represented in the dataset, and this also does not allow us to explicitly examine the relationship between specific attack types and observed score behaviour.

Our analysis is further restricted to the fact that the socio- technical metadata are purely from the RubyGems ecosystem. While this provides a realistic and relevant evaluation setting, it may limit the applicability of our findings to other

open-source ecosystems with different structures, practices, and attack patterns.

In addition, during our initial data pre-processing stages, we excluded packages with more than 20% missing or incomplete repository information, which may introduce bias and affect the representativeness of the results. Furthermore, the use of log transformation and normalisation during preprocessing may compress score variations, reducing the visibility of extreme values and limiting our ability to clearly identify outlier packages.

Finally, since our framework relies entirely on observable repository-level and registry-level metadata signals, attack vectors involving external factors, such as credential compromise or social engineering, are not directly captured by our approach. Such attacks may bypass indicators derived from repository behaviour. Addressing these limitations in future work has the potential to improve the effectiveness of the framework and enable more robust modelling of software supply chain risk.

7.3 Future Work

Several directions can be explored to extend the work on this risk scoring framework. First, we can evaluate the framework using temporal validation, where risk scores are computed on historical data and assessed against future compromise events. This would provide a stronger indication of generalisability beyond the current dataset.

Second, we can examine the framework in other package ecosystems, such as npm or PyPI, which would enable comparison of risk patterns across different environments and assess the broader applicability of the approach. Extending the

framework across ecosystems would also allow investigation of whether the relative importance of socio-technical signals remains consistent or varies across different development contexts.

In addition, further experimentation with alternative score computation strategies could be explored. This includes evaluating different aggregation methods beyond the current weighted formulation to better understand how score construction influences separation and prioritisation performance.

Finally, incorporating additional dynamic signals, such as maintainer activity trends or real-time repository behaviour, may improve the responsiveness of the framework and capture evolving risk conditions more effectively.

These extensions would enhance both the robustness and practical relevance of the proposed risk profiling framework.

References

[1] Rabe Abdalkareem et al. "On the impact of using trivial packages: An empirical case study on npm and PyPI." *Empirical Software Engineering* 25 (2020), pp. 1168–1204. DOI: 10.1007/s10664-019-09792-9.

[2] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. "An empirical study on the survival rate of GitHub projects." In: *Proceedings of the 19th International Conference on Mining Software Repositories*. Association for Computing Machinery, 2022, pp. 365–375. ISBN: 9781450393034. DOI: 10.1145/3524842.3527941.

[3] Iosif Arvanitis et al. "A systematic analysis of the event-stream incident." In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec '22. Association for Computing Machinery, 2022, pp. 22–28. DOI: 10.1145/3517208.3523753.

[4] Guilherme Avelino et al. "On the Abandonment and Survival of Open Source Projects: An Empirical Investigation." In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12. DOI: 10.1109/ESEM.2019.8870181.

[5] Lingfeng Bao et al. “A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects.” *IEEE Transactions on Software Engineering* 47.6 (2021), pp. 1277–1298. DOI: 10.1109/TSE.2019.2918536.

[6] Alex Birsan. *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies*. Medium Blog. Available online at: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610. 2021.

[7] Bitergia. *Bitergia Risk Radar*. urlhttps://bitergia.com/risk-radar/. Accessed: 2026-04-03. 2024.

[8] BitSight Technologies. *BitSight Security Ratings*. https://www.bitsight.com/security-ratings. Accessed: 2026-04-03. 2024.

[9] BitSight Technologies. *Understanding BitSight Ratings and Score Ranges*. https://help.bitsighttech.com/hc/en-us/articles/28651033784855-Vendor-Score-Trust-Score. Accessed: 2026-04-03. 2024.

[10] Black Duck Software. *Managing Security Risk in Black Duck*. https://abb.app.blackduck.com/doc/Risk/Managing_Security_Risk.html. Accessed: 2026-04-03. 2024.

[11] Black Duck Software. *Open Source Vulnerability Management and Reporting*. https://www.blackduck.com/software-composition-analysis-tools/vulnerability-reporting.html. Accessed: 2026-04-03. 2024.

[12] Hudson Borges, André Hora, and Marco Tulio Valente. “Predicting the Popularity of GitHub Repositories.” In: *Proceedings of the 12th ACM International Conference on Predictive Models and Data Analytics in Software*

Engineering (PROMISE). PROMISE '16. 2016, pp. 33–42. DOI: 10.1145/2972958.2972966.

[13] Alan Cao and Brendan Dolan-Gavitt. "What the Fork? Finding and Analyzing Malware in GitHub Forks." In: *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. 2022. DOI: 10.14722/madweb.2022.23001.

[14] Alan Cao and Brendan Dolan-Gavitt. "What the Fork? Finding and Analyzing Malware in GitHub Forks." In: *Proceedings 2022 Workshop on Measurements, Attacks, and Defenses for the Web*. Internet Society, 2022. DOI: 10.14722/madweb.2022.23001.

[15] CHAOSS Project. *Practitioner Guide - Contributor Sustainability*. Accessed 2026-03-25. July 2024. URL: https://chaoss.community/practitioner-guide-contributor-sustainability/.

[16] Md Atique Reza Chowdhury et al. "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages." *IEEE Transactions on Software Engineering* 48 (2022), pp. 2695–2708. DOI: 10.1109/TSE.2021.3068901.

[17] Codecov. *Bash Uploader Security Update*. https://about.codecov.io/security-update/. Accessed: Mar. 14, 2026. Apr. 2021.

[18] Jailton Coelho et al. "Identifying unmaintained projects in GitHub." In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Association for Computing Machinery, 2018. DOI: 10.1145/3239235.3240501.

[19] Jailton Coelho et al. "Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects." *Information and Software Technology* 122 (2020), p. 106274. DOI: 10.1016/j.infsof.2020.106274.

[20] Eleni Constantinou and Tom Mens. "Socio-technical evolution of the Ruby ecosystem in GitHub." *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 34–44. DOI: 10.1109/SANER.2017.7884607.

[21] Russ Cox. "Fifty Years of Open Source Software Supply-Chain Security." *ACM Queue* 68.10 (Sept. 2025), pp. 88–95. DOI: 10.1145/3762635.

[22] Matteo Crosignani, Marco Macchiavelli, and André F. Silva. "Pirates Without Borders: The Propagation of Cyberattacks through Firms' Supply Chains." *Journal of Financial Economics* 147 (2022), pp. 432–448. DOI: 10.1016/j.jfineco.2022.12.002.

[23] Cybersecurity and Infrastructure Security Agency (CISA). *Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations*. https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a. Alert (AA20-352A). Apr. 2021. (Visited on 04/07/2026).

[24] Alexandre Decan, Tom Mens, and Maeick Claes. "On the topology of package dependency networks: a comparison of three programming language ecosystems." In: *Proccedings of the 10th European Conference on Software Architecture Workshops*. ECSAW 16. Association for Computing Machinery, 2016, pp. 1–4. DOI: 10.1145/2993412.3003382.

[25] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems." *Empirical Software Engineering* 24 (2018), pp. 381–416. DOI: 10.1007/s10664-017-9589-y.

[26] Hayley Denbraver. *Code execution back door found in Ruby's rest-client library.* Aug. 2019. URL: https://snyk.io/blog/code-execution-back-door-found-in-rubys-rest-client-library/ (visited on 04/05/2026).

[27] Sowmiya Dhandapani. *Enhancing Software Supply Chain Security Through STRIDE-Based Threat Modelling of CI/CD Pipelines.* June 2025. URL: https://arxiv.org/abs/2506.06478.

[28] Ruian Duan et al. "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS 2021).* Internet Society, 2021. DOI: 10.14722/ndss.2021.23055.

[29] William Enck and Laurie Williams. "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations." *IEEE Secur. Priv.* 20.2 (2022), pp. 96–100. DOI: 10.1109/MSEC.2022.3142338.

[30] Sharifah Yaqoub A. Fayi. "What Petya/NotPetya Ransomware Is and What Its Remediations Are." In: *Information Technology – New Generations (ITNG 2018).* Vol. 738. Advances in Intelligent Systems and Computing. Springer International Publishing, 2018, pp. 93–100. DOI: 10.1007/978-3-319-77028-4_15.

[31] Jenn Gile. *npm Account Takeovers are a Growing Malware Trend*. Endorlabs blog. Retrieved: 2026-03-31. 2026. URL: https://www.endorlabs.com/learn/npm-account-takeovers-are-a-growing-malware-trend.

[32] GitHub. *Saving repositories with stars.* [Online; accessed 17-March-2026]. GitHub Documentation. Mar. 2026.

[33] Sean Goggins, Kevin Lumbard, and Matt Germonprez. "Open Source Community Health: Analytical Metrics and Their Corresponding Narratives." In: *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*. 2021, pp. 25–33. DOI: 10.1109/SoHeal52568.2021.00010.

[34] Danny Grander. *Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months*. Snyk Security Blog. Available online: https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/. Nov. 2018.

[35] Sajal Halder et al. "Malicious Package Detection using Metadata Information." In: *Proceedings of the ACM Web Conference 2024*. WWW '24. Association for Computing Machinery, 2024, pp. 1779–1789. DOI: 10.1145/3589334.3645543.

[36] Aviatl Harel. *NPM Supply Chain Attack: Massive Compromise of debug, chalk and 16 other packages*. https://www.upwind.io/feed/npm-supply-chain-attack-massive-compromise-of-debug-chalk-and-16-other-packages. Accessed: Mar. 14, 2026. Sept. 2025.

[37] Thomas Hastings and Kristen R. Walcott. "Continuous Verification of Open Source Components in a World of Weak Links." In: *2022 IEEE International*

Symposium on Software Reliability Engineering Workshops (ISSREW). 2022, pp. 201–207. DOI: 10.1109/ISSREW55968.2022.00068.

[38] Eman Abu Ishgair, Marcela S. Melara, and Santiago Torres-Arias. *SoK: A Defense-Oriented Evaluation of Software Supply Chain Security*. 2024. DOI: 10.48550/arXiv.2405.14993. arXiv: 2405.14993.

[39] Slinger Jansen. "Measuring the Health of Open Source Software Ecosystems: Beyond the Scope of Project Health." *Inf. Softw. Technol.* 56.11 (2014), pp. 1508–1519. DOI: 10.1016/j.infsof.2014.04.006.

[40] Jing Jiang et al. "Why and how developers fork what from whom in GitHub." *Empirical Software Engineering* 22.1 (2017), pp. 547–578. DOI: 10.1007/s10664-016-9436-6. URL: https://link.springer.com/article/10.1007/s10664-016-9436-6.

[41] Sha Jiang, Jian Cao, and Mukesh Prasad. "The Metrics to Evaluate the Health Status of OSS Projects Based on Factor Analysis." In: *Communications in Computer and Information Science*. Springer Singapore, 2019, pp. 723–737. DOI: 10.1007/978-981-15-1377-0_56.

[42] Kalu Kelechi G. et al. "An Industry Interview Study of Software Signing for Supply Chain Security." In: *Proceedings of the 34th USENIX Security Symposium (USENIX Security '25)*. USENIX Association, 2025. ISBN: 978-1-939133-52-6. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/kalu.

[43] Riivo Kikas et al. "Structure and Evolution of Package Dependency Networks." In: *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 102–112. DOI: 10.1109/MSR.2017.55.

[44] Raula Gaikovina Kula et al. "Selecting third-party libraries: the practitioners' perspective." In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. 2020, pp. 108–119. DOI: 10.1145/3368089.3409711.

[45] Piergiorgio Ladisa et al. "Journey to the Centre of Software Supply Chain Attacks." *IEEE Security & Privacy* 21.6 (2023). DOI: 10.1109/MSEC.2023.3302066.

[46] Piergiorgio Ladisa et al. "Risk Explorer for Software Supply Chains: Understanding the Attack Surface of Open-Source based Software Development." In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED'22. Association for Computing Machinery, 2022, pp. 35–36. ISBN: 9781450398855. DOI: 10.1145/3560835.3564546.

[47] Piergiorgio Ladisa et al. "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains." In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 1509–1526. DOI: 10.1109/SP46215.2023.10179304.

[48] Tobias Lauinger et al. "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. *arXiv preprint arXiv:1811.00918* (2018). NDSS 2017. DOI: 10.48550/arXiv.1811.00918.

[49] Xiaozhou Li et al. "OSSARA: Abandonment Risk Assessment for Embedded Open Source Components." *IEEE Software* 39.4 (2022), pp. 48–53. DOI: 10.1109/MS.2022.3163011.

[50] Zhifang Liao et al. "Exploring the Characteristics of Issue-Related Behaviors in GitHub Using Visualization Techniques." *IEEE Access* 6 (2018), pp. 24003–24015. DOI: 10.1109/ACCESS.2018.2810295.

[51] Chengwei Liu et al. "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Association for Computing Machinery, 2022, pp. 672–684. DOI: 10.1145/3510003.3510142.

[52] Raphina Liu et al. "Dirty-Waters: Detecting Software Supply Chain Smells." In: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. FSE Companion '25. Association for Computing Machinery, 2025, pp. 1045–1049. DOI: 10.1145/3696630.3728578.

[53] Masike Malatji, Sune Von Solms, and Annlize Marnewick. "Socio-technical systems cybersecurity framework." *Inf. Comput. Secur.* 27.2 (2019), pp. 233–255. DOI: 10.1108/ICS-03-2018-0031.

[54] Jeferson Martínez and M Javier Durán. "Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study." *International Journal of Safety and Security Engineering* 11.5 (2021), pp. 537–545. DOI: 10.18280/ijsse.110505.

[55] Frank J. Massey Jr. "The Kolmogorov-Smirnov Test for Goodness of Fit." *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78.

[56] MITRE Corporation. *ATT&CK Technique T1195: Supply Chain Compromise*. https://attack.mitre.org/techniques/T1195/. Accessed: 2026-03-31. 2024.

[57] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. "Using complexity metrics to improve software security." *Computer Fraud & Security* 2013.5 (2013), pp. 8–17. DOI: 10.1016/S1361-3723(13)70045-9.

[58] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. "What are the characteristics of highly-selected packages? A case study on the npm ecosystem." *Journal of Systems and Software* 198 (2023), p. 111588. DOI: 10.1016/j.jss.2022.111588.

[59] National Institute of Standards and Technology. *Exploratory Data Analysis*. Accessed April 2026. 2016. URL: https://www.itl.nist.gov/div898/handbook/eda/eda.htm (visited on 04/05/2026).

[60] Thanh-Cong Nguyen, Duc-Ly Vu, and Narayan C. Debnath. "An Analysis of Malicious Behaviors of Open-Source Packages Using Dynamic Analysis." In: *International Conference on Computer Applications in Industry and Engineering*. Vol. 2242. Communications in Computer and Information Science. Springer Nature, 2024, pp. 102–114. DOI: 10.1007/978-3-031-76273-4_8.

[61] Marc Ohm et al. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks." In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer International Publishing, 2020, pp. 23–43. DOI: 10.1007/978-3-030-52683-2_2.

[62] Chinenye L. Okafor et al. "SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties." In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED'22. Association for Computing Machinery, 2022, pp. 15–24. DOI: 10.1145/3560835.3564556.

[63] Olumide Shittu. *A Complete Guide to ROC Curves and AUC*. Accessed: 2026-03-26. Feb. 2026. URL: https://www.statology.org/a-complete-guide-to-roc-curves-and-auc/.

[64] OpenSCRM. *Open Source Supply Chain Risk Management (OpenSCRM).* urlhttps://openscrm.org/. Accessed: 2026-04-03. 2024.

[65] OpenSSF. *OpenSSF Scorecard*. urlhttps://scorecard.dev/. Accessed: 2026-04-03. 2024.

[66] OWASP Foundation. *OWASP Top 10 (2025): A03: Software Supply Chain Failures*. https://owasp.org/Top10/2025/A03_2025-Software_Supply_Chain_Failures/. Accessed: 2026-03-31. 2025.

[67] Ivan Pashchenko et al. "Vulnerable open source dependencies: counting those that matter." In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Association for Computing Machinery, 2018. DOI: 10.1145/3239235.3268920.

[68] Ivan Pashchenko et al. "Vulnerable open source dependencies: counting those that matter." In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. 2018. DOI: 10.1145/3239235.3268920.

[69] Andrey Polkovnychenko and Brian Moussalli. *Revival Hijack - PyPI hijack technique exploited in the wild, puts 22K packages at risk*. Accessed: March 23, 2026. Sept. 2024. URL: https://jfrog.com/blog/revival-hijack-pypi-hijack-technique-exploited-22k-packages-at-risk/.

[70] Piotr Przymus and Thomas Durieux. "Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack." In: *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 2025, pp. 91–102. DOI: 10.1109/MSR66628.2025.00026.

[71] Datadog Security Research. *MUT-8694: An NPM and PyPI Malicious Campaign Targeting Windows Users.* https://securitylabs.datadoghq.com/articles/mut-8964-an-npm-and-pypi-malicious-campaign-targeting-windows-users/. Retrieved: 2026-03-31. Nov. 2024.

[72] ReversingLabs. *ReversingLabs Annual Software Supply Chain Security Report Spotlights Mounting Attacks on AI, Crypto, Open Source, and Commercial Software.* Accessed: Mar. 31, 2026. 2025. URL: https://ntsc.org/wp-content/uploads/2025/03/The-2025-Software-Supply-Chain-Security-Report-RL-compressed.pdf.

[73] SecurityScorecard. *SecurityScorecard Glossary.* https://support.securityscorecard.com/hc/en-us/articles/4410784989083-SecurityScorecard-glossary. Accessed: 2026-04-03. 2024.

[74] Multi-State Information Sharing and Analysis Center. *Cyber Alert: CCleaner Software Supply Chain Compromise*. Tech. rep. Center for Internet Security, Oct. 2017. URL: https://www.cisecurity.org/ms-isac/cyber-alert-ccleaner-software-supply-chain-compromise.

[75] Ax Sharma. *Dependency Hijacking Attack Hits Over 35 Tech Companies.* Sonatype Security Blog. Available online: https://www.sonatype.com/blog/dependency-hijacking-software-supply-chain-attack-hits-more-than-35-organizations. Feb. 2021. (Visited on 03/30/2026).

[76] Yonghee Shin et al. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities." *IEEE Transactions on Software Engineering* 37.6 (2011), pp. 772–787. DOI: 10.1109/TSE.2010.81.

[77] Sonatype. *Sonatype's 10th Annual State of the Software Supply Chain Report Reveals 156% Surge in OSS Malware*. Available: https://www.sonatype.com/press-releases/sonatypes-10th-annual-state-of-the-software-supply-chain-report. Oct. 2024.

[78] Sonatype. *State of the Software Supply Chain 2025*. https://www.sonatype.com/state-of-the-software-supply-chain. Accessed: Mar. 31, 2026. 2026.

[79] Vince Steckler and Ondrej Vlcek. *Update to the CCleaner 5.33.6162 Security Incident*. Avast Security Blog. Sept. 2017. URL: https://blog.avast.com/update-to-the-ccleaner-5.33.6162-security-incident.

[80] Lindsay Sterle and Suman Bhunia. "On SolarWinds Orion Platform Security Breach." In: *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation*. 2021, pp. 636–641. DOI: 10.1109/SWC50871.2021.00094.

[81] Kara Struckman. *How to Secure Open Source Software: The Dilemma of the XZ Utils Backdoor*. Wilson Center Blog. Available online at: https://www.wilsoncenter.org/blog-post/how-secure-open-source-software-dilemma-xz-utils-backdoor. Apr. 2024. (Visited on 03/30/2026).

[82] Liran tal. *Malicious remote code execution backdoor discovered in the popular bootstrap-sass Ruby gem*. Apr. 2019. URL: https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/ (visited on 04/05/2026).

[83] Socket Security Team. *Malicious Ruby Gems Exfiltrate Telegram Tokens Following Vietnam Ban*. Socket.dev Blog. Accessed: Mar. 17, 2026. June 2025. URL: https://socket.dev/blog/malicious-ruby-gems-exfiltrate-telegram-tokens-and-messages-following-vietnam-ban.

[84] Adam Thomas and Jérôme Segura. *Electrum Bitcoin wallets under siege.* [Online; accessed 18-March-2026]. Apr. 2019. URL: https://www.malwarebytes.com/blog/news/2019/04/electrum-bitcoin-wallets-under-siege.

[85] Ken Thompson. “Reflections on Trusting Trust.” *Communications of the ACM* 27.8 (1984), pp. 761–763. DOI: 10.1145/358198.358210.

[86] Glenn Thorpe. *NPM Library (ua-parser-js) Hijacked: What You Need to Know*. Rapid7 Blog. 2021. URL: https://www.rapid7.com/blog/post/2021/10/25/npm-library-ua-parser-js-hijacked-what-you-need-to-know/.

[87] Bill Toulas. *Malicious RubyGems pose as Fastlane to steal Telegram API data*. BleepingComputer Blog. Accessed: Mar. 17, 2026. July 2025. URL: https://www.bleepingcomputer.com/news/security/malicious-rubygems-pose-as-fastlane-to-steal-telegram-api-data/.

[88] Elisa Velarde. *Sonatype Intelligence Insights: CVE-2019-13354: strong_password*. July 2019. URL: https://www.sonatype.com/blog/cve-2019-13354-strong-password (visited on 04/05/2026).

[89] Huta Veronikal. “When to use hierarchical linear modeling.” *The Quantitative Methods for Psychology* 10.1 (2014), pp. 13–28. DOI: 10.20982/tqmp.10.1.p013.

[90] Chris Williams. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. [Online; accessed 17-March-2026]. Mar. 2016. URL: https://www.theregister.com/2016/03/23/npm_left_pad_chaos/.

[91] Laurie Williams et al. “Research Directions in Software Supply Chain Security.” *ACM Trans. Softw. Eng. Methodol.* 34.5 (2025). DOI: 10.1145/3714464.

[92] Heather Woltman et al. “An Introduction to Hierarchical Linear Modelling.” *Tutorials in quantitative methods for psychology* 8.1 (2012), pp. 52–69. DOI: 10.20982/tqmp.08.1.p052.

[93] Nusrat Zahan et al. “What are weak links in the npm supply chain?” In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’22. Association for Computing Machinery, 2022, pp. 331–340. DOI: 10.1145/3510457.3513044.

[94] Ahmed Zerouali et al. “An Empirical Analysis of Technical Lag in npm Package Dependencies.” *Proc. Int. Conf. Softw. Reuse (ICSR)*. 2018, pp. 95–110. DOI: 10.1007/978-3-319-90421-4_6.

[95] Xunhui Zhang et al. “Pull Request Decisions Explained: An Empirical Overview.” *IEEE Transactions on Software Engineering* 49.2 (2023), pp. 849–871. DOI: 10.1109/TSE.2022.3165056.

[96] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. "How has forking changed in the last 20 years?: a study of hard forks on GitHub." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 2020, pp. 445–456. DOI: 10.1145/3377811.3380412.

[97] Markus Zimmermann et al. "Small world with high risks: a study of security threats in the npm ecosystem." In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. USENIX Association, 2019, pp. 995–1010. URL: https://dl.acm.org/doi/10.5555/3361338.3361407.

Appendices

A Risk Metrics and Directionality

Table A.1: Risk Metrics with Directionality and Adversary Rationale

Metric	CHAOSS-reference/Novel	Definition	Directionality	Adversary Rationale
Dependency	CHAOSS (Upstream Code Dependencies)	Direct and transitive dependency count	High ↑	More prone to dependency confusion and compromise
Dependency Depth	Novel	Maximum transitive depth	High ↑	More blind spots - obscure monitoring
File count	Novel	Number of files in GitHub repository	High ↑	Less scrutiny – easier to hide malicious payload
Directory depth	Novel	Directory hierarchy of GitHub repository	High ↑	More blind spots - obscure monitoring
Lines of code	Novel	Total lines of code in source files	High ↑	Easier to hide malicious payload (larger attack surface)
Downloads	CHAOSS (Number of Downloads)	Number of downloads from RubyGems registry	High ↑	Significant compromised downstream consumers
Dependents	CHAOSS (Project Popularity)	Number of packages using it as upstream dependency	High ↑	Significant compromised downstream consumers
Stars	CHAOSS (Project Popularity)	GitHub project popularity	High ↑	Social engineering vectors
Forks	CHAOSS (Technical Fork)	Independent copies on same platform	High ↑	Significant compromised downstream consumers
Commit gap	CHAOSS (Project Velocity/Issue Resolution Duration)	Project inactivity	High ↑	Less scrutiny – unmonitored repository
Issues (6 months)	CHAOSS (Issues New)	New issues opened in past 6 months	Low ↓	Lack of community scrutiny
PR/MRs (6 months)	CHAOSS (Change Requests)	New PRs/MRs in past 6 months	Low ↓	Lack of activity and maintenance
Contributors	CHAOSS (Contributors)	Number of project contributors	Low ↓	Lack of community activity

B Notation Guide

$y \in \{0,1\}$ Ground truth label from Backstabbers dataset: $y = 1$ (malicious gem), $y = 0$ (benign gem)

X_k k-th leaf metric score for a gem (normalized to $[0,1]$), where $k \in \{1,\ldots,m\}$

$\textbf{AUC}_k$ Area Under ROC Curve for metric k:

$$\text{AUC}_k = P(X_k(\text{malicious}) > X_k(\text{benign}))$$

s_k **Metric strength** measuring excess discriminatory power:

$$s_k = \max(\text{AUC}_k - 0.5, 0)$$

M^* Surviving metrics with AUC > 0.5: $M^* = \{k \in M \mid s_k > 0\}$

M_p Set of all metrics belonging to attribute p

M_p^* Surviving metrics within attribute p: $M_p^* = M_p \cap M^*$

w_{pk} Weight of metric k within attribute p:

$$w_{pk} = \frac{s_k}{\sum_{j \in M_p} s_j}$$

A_p p-th attribute score (weighted mean of normalised metric values within the attribute):

$$A_p = \sum_{k \in M_p^*} w_{pk} \cdot X_k$$

S_s s-th aspect score, where $s \in S = \{1,2,3\}$:

$$S_s = \frac{1}{|A_s|} \sum_{p \in A_s} A_p$$

Aspect mappings

- Complexity: $A_{\text{Dependency}}, A_{\text{Codebase}}$
- Impact: $A_{\text{Usage}}, A_{\text{Popularity}}$
- Maintenance: $A_{\text{Activity}}, A_{\text{Stakeholders}}$

C Final composite score:

$$C = \frac{1}{|S|} \sum_{s \in S} S_s$$

C Metric-Level AUC Results

Table C.1: Metric-Level ROC-AUC Performance, Strength, and Normalised Weights

Aspect	**Metric**	**AUC**	**Strength (s_k)**	**Weight**	**Direction**
Complexity	total_dep	0.545	0.045	0.530	High ↑
Complexity	dep_depth	0.540	0.040	0.470	High ↑
Complexity	file_count	0.517	0.017	0.333	High ↑
Complexity	dir_depth	0.530	0.030	0.585	High ↑
Complexity	total_loc	0.504	0.004	0.082	High ↑
Impact	total_downloads	0.595	0.095	1.000	High ↑
Impact	dependents_count	0.488	0.000	0.000	High ↑
Impact	stars	0.481	0.000	0.000	High ↑
Impact	forks	0.520	0.020	1.000	High ↑
Maintenance	commit_gap_days	0.604	0.104	1.000	High ↑
Maintenance	issues_6m	0.500	0.000	0.000	Low ↓
Maintenance	prs_6m	0.500	0.000	0.000	Low ↓
Maintenance	contributors_count	0.500	0.000	0.000	Low ↓

D Full Enrichment Analysis Across Ranking Thresholds

Table D.1: Full enrichment analysis across inspection thresholds from 1% to 100%.

Top % Inspected	Score Range Covered	Number of Gems	Expected Malicious (Random)	Actual Malicious Captured	Precision (%)	Enrichment (×)	Capture Rate (%)
1%	0.6680–0.8953	790	3.8	1	0.127	0.263	0.263
2%	0.6438–0.8953	1580	7.6	7	0.443	0.921	1.842
3%	0.6290–0.8953	2370	11.4	11	0.464	0.965	2.895
4%	0.6171–0.8953	3160	15.2	23	0.728	1.513	6.053
5%	0.6073–0.8953	3950	19.0	30	0.759	1.579	7.895
6%	0.5982–0.8953	4740	22.8	36	0.759	1.579	9.474
7%	0.5897–0.8953	5529	26.6	43	0.778	1.617	11.316
8%	0.5815–0.8953	6320	30.4	48	0.759	1.579	12.632
9%	0.5747–0.8953	7110	34.2	54	0.759	1.579	14.211
10%	0.5679–0.8953	7899	38.0	60	0.760	1.579	15.789
11%	0.5621–0.8953	8690	41.8	68	0.783	1.627	17.895
12%	0.5562–0.8953	9480	45.6	73	0.770	1.601	19.211
13%	0.5510–0.8953	10270	49.4	80	0.779	1.619	21.053
14%	0.5462–0.8953	11060	53.2	85	0.769	1.598	22.368
15%	0.5411–0.8953	11850	57.0	88	0.743	1.544	23.158
16%	0.5360–0.8953	12640	60.8	90	0.712	1.480	23.684
17%	0.5312–0.8953	13430	64.6	96	0.715	1.486	25.263
18%	0.5263–0.8953	14220	68.4	104	0.731	1.520	27.368
19%	0.5217–0.8953	15010	72.2	108	0.720	1.496	28.421

Top % Inspected	Score Range Covered	Number of Gems	Expected Malicious (Random)	Actual Malicious Captured	Precision (%)	Enrichment (×)	Capture Rate (%)
20%	0.5185–0.8953	15800	76.0	112	0.709	1.474	29.474
21%	0.5146–0.8953	16590	79.8	116	0.699	1.454	30.526
22%	0.5102–0.8953	17380	83.6	121	0.696	1.447	31.842
23%	0.5057–0.8953	18170	87.4	125	0.688	1.430	32.895
24%	0.5011–0.8953	18960	91.2	134	0.707	1.469	35.263
25%	0.4967–0.8953	19750	95.0	139	0.704	1.463	36.579
26%	0.4924–0.8953	20540	98.8	145	0.706	1.468	38.158
27%	0.4880–0.8953	21330	102.6	150	0.703	1.462	39.474
28%	0.4837–0.8953	22120	106.4	158	0.714	1.485	41.579
29%	0.4794–0.8953	22910	110.2	165	0.720	1.497	43.421
30%	0.4754–0.8953	23700	114.0	167	0.705	1.465	43.947
31%	0.4713–0.8953	24490	117.8	170	0.694	1.443	44.737
32%	0.4674–0.8953	25280	121.6	179	0.708	1.472	47.105
33%	0.4631–0.8953	26070	125.4	183	0.702	1.459	48.158
34%	0.4591–0.8953	26860	129.2	185	0.689	1.432	48.684
35%	0.4549–0.8953	27650	133.0	188	0.680	1.414	49.474
36%	0.4505–0.8953	28440	136.8	192	0.675	1.404	50.526
37%	0.4463–0.8953	29230	140.6	193	0.660	1.373	50.789
38%	0.4423–0.8953	30020	144.4	193	0.643	1.337	50.789
39%	0.4381–0.8953	30810	148.2	199	0.646	1.343	52.368
40%	0.4342–0.8953	31600	152.0	201	0.636	1.322	52.895

Top % Inspected	**Score Range Covered**	**Number of Gems**	**Expected Malicious (Random)**	**Actual Malicious Captured**	**Precision (%)**	**Enrichment (×)**	**Capture Rate (%)**
41%	0.4298–0.8953	32390	155.8	203	0.627	1.303	53.421
42%	0.4256–0.8953	33180	159.6	206	0.621	1.291	54.211
43%	0.4214–0.8953	33970	163.4	213	0.627	1.304	56.053
44%	0.4175–0.8953	34760	167.2	219	0.630	1.310	57.632
45%	0.4135–0.8953	35550	171.0	224	0.630	1.310	58.947
46%	0.4095–0.8953	36340	174.8	229	0.630	1.310	60.263
47%	0.4057–0.8953	37130	178.6	231	0.622	1.293	60.789
48%	0.4018–0.8953	37920	182.4	233	0.614	1.277	61.316
49%	0.3979–0.8953	38710	186.2	239	0.617	1.284	62.895
50%	0.3939–0.8953	39500	190.0	241	0.610	1.268	63.421
51%	0.3903–0.8953	40290	193.8	244	0.606	1.259	64.211
52%	0.3864–0.8953	41080	197.6	248	0.604	1.255	65.263
53%	0.3828–0.8953	41870	201.4	252	0.602	1.251	66.316
54%	0.3790–0.8953	42660	205.2	257	0.602	1.252	67.632
55%	0.3752–0.8953	43450	209.0	261	0.601	1.249	68.684
56%	0.3713–0.8953	44240	212.8	268	0.606	1.259	70.526
57%	0.3683–0.8953	45030	216.6	272	0.604	1.256	71.579
58%	0.3648–0.8953	45820	220.4	275	0.600	1.248	72.368
59%	0.3611–0.8953	46610	224.2	279	0.599	1.244	73.421
60%	0.3571–0.8953	47400	228.0	282	0.595	1.237	74.211
61%	0.3532–0.8953	48190	231.8	286	0.593	1.234	75.263

Top % Inspected	**Score Range Covered**	**Number of Gems**	**Expected Malicious (Random)**	**Actual Malicious Captured**	**Precision (%)**	**Enrichment (×)**	**Capture Rate (%)**
62%	0.3495–0.8953	48980	235.6	290	0.592	1.231	76.316
63%	0.3457–0.8953	49770	239.4	295	0.593	1.232	77.632
64%	0.3419–0.8953	50560	243.2	300	0.593	1.234	78.947
65%	0.3380–0.8953	51350	247.0	304	0.592	1.231	80.000
66%	0.3341–0.8953	52140	250.8	310	0.595	1.236	81.579
67%	0.3303–0.8953	52930	254.6	311	0.588	1.222	81.842
68%	0.3264–0.8953	53720	258.4	318	0.592	1.231	83.684
69%	0.3224–0.8953	54510	262.2	324	0.594	1.236	85.263
70%	0.3181–0.8953	55300	266.0	328	0.593	1.233	86.316
71%	0.3140–0.8953	56090	269.8	332	0.592	1.231	87.368
72%	0.3099–0.8953	56880	273.6	338	0.594	1.235	88.947
73%	0.3055–0.8953	57670	277.4	342	0.593	1.233	90.000
74%	0.3014–0.8953	58460	281.2	342	0.585	1.216	90.000
75%	0.2969–0.8953	59250	285.0	342	0.577	1.200	90.000
76%	0.2926–0.8953	60040	288.8	346	0.576	1.198	91.053
77%	0.2883–0.8953	60830	292.6	348	0.572	1.189	91.579
78%	0.2840–0.8953	61620	296.4	352	0.571	1.188	92.632
79%	0.2795–0.8953	62410	300.2	353	0.566	1.176	92.895
80%	0.2751–0.8953	63200	304.0	355	0.562	1.168	93.421
81%	0.2705–0.8953	63990	307.8	359	0.561	1.166	94.474
82%	0.2657–0.8953	64780	311.6	362	0.559	1.162	95.263

Top % Inspected	Score Range Covered	Number of Gems	Expected Malicious (Random)	Actual Malicious Captured	Precision (%)	Enrichment (×)	Capture Rate (%)
83%	0.2610–0.8953	65570	315.4	367	0.560	1.164	96.579
84%	0.2562–0.8953	66360	319.2	368	0.555	1.153	96.842
85%	0.2512–0.8953	67150	323.0	369	0.550	1.142	97.105
86%	0.2458–0.8953	67940	326.8	372	0.548	1.138	97.895
87%	0.2406–0.8953	68730	330.6	373	0.543	1.128	98.158
88%	0.2351–0.8953	69520	334.4	373	0.537	1.115	98.158
89%	0.2292–0.8953	70310	338.2	375	0.533	1.109	98.684
90%	0.2229–0.8953	71100	342.0	375	0.527	1.096	98.684
91%	0.2168–0.8953	71890	345.8	377	0.524	1.090	99.211
92%	0.2101–0.8953	72680	349.6	378	0.520	1.081	99.474
93%	0.2028–0.8953	73470	353.4	380	0.517	1.075	100.000
94%	0.1948–0.8953	74260	357.2	380	0.512	1.064	100.000
95%	0.1858–0.8953	75050	361.0	380	0.506	1.053	100.000
96%	0.1759–0.8953	75840	364.8	380	0.501	1.042	100.000
97%	0.1645–0.8953	76630	368.6	380	0.496	1.031	100.000
98%	0.1504–0.8953	77420	372.4	380	0.491	1.020	100.000
99%	0.1269–0.8953	78210	376.2	380	0.486	1.010	100.000
100%	0.0309–0.8953	79000	380.0	380	0.481	1.000	100.000