

# An LLM-based Quantitative Framework for Evaluating High-Stealthy Backdoor Risks in OSS Supply Chains

Zihe Yan<sup>1\*</sup>, Kai Luo<sup>2\*</sup>, Haoyu Yang<sup>3</sup>, Yang Yu<sup>3</sup>, Zhuosheng Zhang<sup>1†</sup>, Guancheng Li<sup>3†</sup>

<sup>1</sup>School of Computer Science, Shanghai Jiao Tong University, China

<sup>2</sup>Institute for Interdisciplinary Information Sciences, Tsinghua University, China

<sup>3</sup>Tencent Xuanwu Lab, Tencent, China

yangtuomao, zhangzs@sjtu.edu.cn

luok22@mails.tsinghua.edu.cn

haoyuyang, tkyu, atumli@tencent.com

## Abstract

In modern software development workflows, the open-source software supply chain significantly contributes to efficient and convenient engineering practices. With increasing system complexity, it has become a common practice to use open-source software as third-party dependencies. However, due to the lack of maintenance for underlying dependencies and insufficient community auditing, ensuring the security of source code and the legitimacy of repository maintainers has become a challenge, particularly in the context of high-stealthy backdoor attacks such as the XZ-Util incident. To address these problems, we propose a fine-grained project evaluation framework for backdoor risk assessment in open-source software. Our evaluation framework models highly stealthy backdoor attacks from the attacker’s perspective and defines targeted metrics for each attack stage. Moreover, to overcome the limitations of static analysis in assessing the reliability of repository maintenance activities, such as irregular commit privilege escalation and insufficient review participation, we employ large language models (LLMs) to perform semantic evaluation of code repositories while avoiding reliance on manually crafted patterns. The effectiveness of our framework is validated on 66 high-priority packages in the Debian ecosystem, and the experimental results reveal that the current open-source software supply chain is exposed to a series of security risks.

Code —

<https://github.com/XuanwuLab/HSBRiskEvaluator>

## 1 Introduction

As the cornerstone of modern software engineering, open-source software has been deeply integrated into a wide range of systems, from desktop applications (Kamau 2025) and mobile platforms (Håkegård et al. 2024) to cloud computing infrastructures. By leveraging collaborative development, code reuse, and community-driven models, open-source software significantly lowers technical barriers and accelerates software innovation and iteration. Developers

worldwide can share components and reuse modules to efficiently build complex application ecosystems. Many mainstream architectures, such as Linux, Docker, and AI frameworks, heavily rely on the openness and reusability of open-source software.

However, the openness and extensive dependency propagation of open-source software have also introduced increasing risks to the software supply chain. In recent years, attackers have gradually evolved traditional poisoning attacks into more targeted and persistent advanced threats. A typical example is the **XZ Utils incident** (Väyrynen 2025), where the adversary did not merely rely on a one-off code injection but instead maintained a long-term presence within the community, steadily building trust and waiting for the right moment to strike. These attacks exhibit distinctive characteristics:

- **High stealth**, achieved by leveraging techniques such as binary payload embedding (Lin et al. 2024), formatting obfuscation, and pseudo-random naming to deeply bury backdoors within critical dependency paths;
- **High complexity**, as the attack chains often span multiple repositories and downstream build systems (Shao et al. 2024), making detection through standard review processes extremely difficult;
- **Human-oriented tactics** (Fu et al. 2025), where attackers use social engineering to integrate into maintainer circles and masquerade as trusted contributors, significantly lowering defenders’ vigilance against potential threats.

As open-source dependencies become increasingly intertwined and automated build tools proliferate, once such backdoors are merged into the main branch, their impact can rapidly propagate across projects and platforms, creating systemic risks that are extremely hard to contain.

To address these threats, existing studies have explored defense mechanisms based on static code analysis, security auditing, software bill of materials (SBOM) generation (Xia et al. 2023), and dependency tracking (Williams et al. 2025). However, these approaches often suffer from two limitations: (1) the lack of attacker-centric modeling to anticipate which repositories are likely targets; and (2) the reliance on known patterns, which fails to detect evolving stealth strate-

\*These authors contributed equally.

gies. Consequently, there is an urgent need for a systematic and quantifiable risk assessment method to help developers and platform maintainers identify *high-value poisoning targets* and formulate effective defense strategies.

In this work, we propose a systematic **risk assessment framework for highly stealthy poisoning attacks** targeting open-source repositories. From the attacker’s perspective, our framework models the target selection process based on four core dimensions: (1) **Dependency Impact**, which measures the propagation effect of the component in the software ecosystem; (2) **Payload Concealment**, which assesses the feasibility of embedding and disguising malicious code in the project; (3) **Community Quality**, which captures the strength of community mechanisms in defending against malicious code contributions from attackers; and (4) **Continuous Integration**. Based on this metric system, we integrate LLMs to perform static and semantic analysis of repositories, ultimately building an automated assessment tool. This tool generates a multi-dimensional *risk matrix* to provide structured, quantitative scores of poisoning risk in open-source software.

The main contributions of this work are as follows:

1. A unified evaluation framework is proposed to quantify and assess supply chain poisoning risks in open-source Linux packages hosted on GitHub. The framework integrates both structural and behavioral risk factors.
2. A fully automated assessment tool is developed and released. The tool leverages both static heuristics and LLM-based semantic analysis to support large-scale, code-aware risk evaluation.
3. We empirically evaluate 66 high-priority packages in the Debian ecosystem, revealing that **Community Quality** remains the most prevalent attack surface across projects. This insight is further validated through a case study on the `xz` repository, a real-world incident that reflects our framework’s key risk indicators.

## 2 Related Work

### 2.1 Software Supply Chain Attack

Although most work has focused on framework design for efficiency improvement (Vazquez Melendez, Bergey, and Smith 2024), security issues have recently attracted increasing attention due to the rising trend of software supply chain attack incidents. Such vulnerability can exist throughout the entire software development lifecycle and pose threats to various dependency environments (e.g., npm, PyPI, GitHub). Williams et al. (2025) studied how attackers exploit package managers to achieve arbitrary code execution in open-source software supply chains. Their study analyzed seven ecosystems, including npm and PyPI, and identified three install-time and four runtime techniques that attackers use to execute malicious code. Delicheh and Mens (2024) systematically analyzed the software supply chain security issues in GitHub Actions, identifying risks such as dependency confusion, poor permission management, command injection, and credential exposure in reusable Actions and workflows. Because such attacks are often **stealthily**

embedded in dependencies or build processes, they can impact a wide range of downstream projects and users, resulting in extensive and severe consequences.

### 2.2 Backdoor Detection and Risk Assessment in Open Source Repositories

To systematize the detection of risks in open source software (OSS) (Aberdour 2007), numerous risk assessment frameworks have been proposed, including OpenSSF Scorecard (Zahan et al. 2023), NIST 800-161 (Barabanov, Markov, and Tsirlov 2020), and BSIMM (McGraw 2015). These frameworks support both pre-release warnings and post-release issue detection (Barabanov, Markov, and Tsirlov 2020). Building on them, Hamer et al. (2025) mapped the metrics of ten supply chain frameworks to ATT&CK attack techniques (Yousaf and Zhou 2024) and analyzed incidents such as SolarWinds (Anisa and Widi-aningsih 2024), Log4j (Martínez and Durán 2021), and XZ Utils (Przymus and Durieux 2025). However, their approach only identified attack-related tasks and could not serve as a general standard for evaluating OSS security. It also overlooked highly stealthy backdoor threats, particularly when attackers infiltrate project communities. To mitigate this gap, Tan et al. (2025) introduced human factors as an attack vector, while Kalu et al. (2025) further proposed reputation-based metrics to assess maintainer trustworthiness. Nevertheless, such reputation scores remain vulnerable to manipulation, as demonstrated by the XZ Utils incident, which undermines their reliability.

## 3 Method

### 3.1 Threat Modeling and Attack Framework

We model the poisoning threat from the perspective of a stealthy and persistent adversary, inspired by real-world incidents such as the `xz` backdoor. The attacker’s primary objective is to inject malicious payloads into upstream open-source repositories in a way that maximizes downstream propagation and minimizes detection. To support this goal, we define both the adversary’s capabilities and a four-stage attack framework that drives our risk evaluation design.

**Adversary Assumptions** The adversary is assumed to possess full access to publicly available metadata of open-source repositories, including dependency trees, version histories, CI configurations, and contribution patterns. The attacker does not control downstream users but can continuously observe ecosystem-wide trends.

Meanwhile, the adversary is capable of submitting well-crafted pull requests (PRs), building trust through benign contributions, and crafting hidden payloads that evade conventional static analysis. We further assume the use of advanced language models to assist in semantic manipulation and stealth injection. However, the attacker does not have knowledge of specific security review policies used by downstream integrators.

**The Attack Framework** As illustrated in Figure 1, the poisoning process consists of four stages reflecting core structural weaknesses.

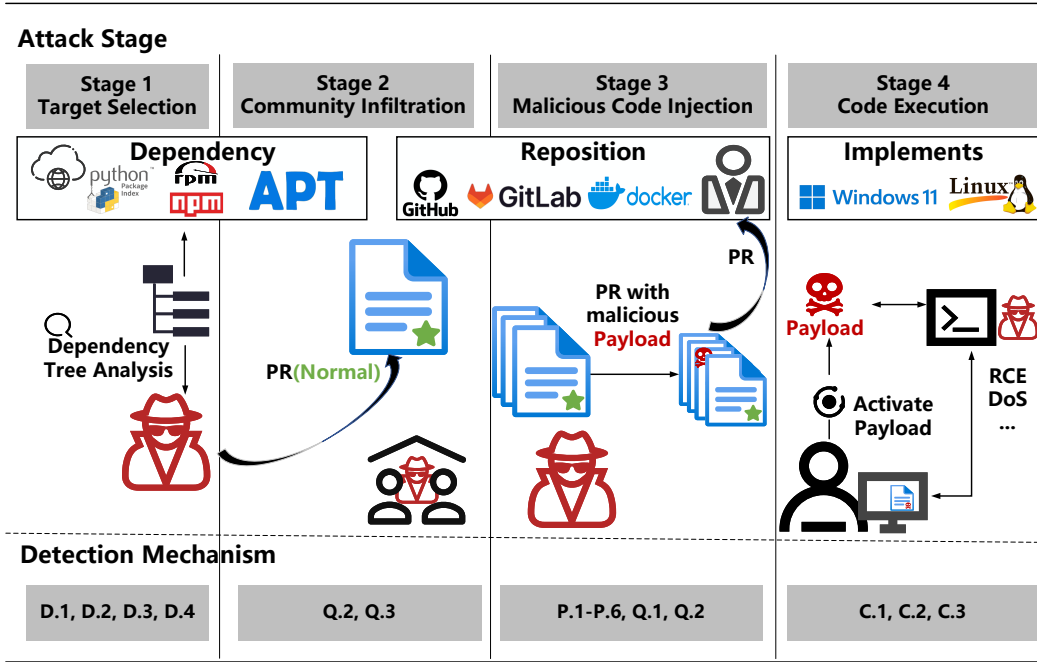


Figure 1: The attack framework and detection mechanism.

- **Stage 1: Reconnaissance and Target Selection.** The attacker analyzes dependency graphs to identify upstream components with high transitive influence.
- **Stage 2: Initial Access via Social Infiltration.** The attacker joins the community by submitting low-risk PRs and builds trust.
- **Stage 3: Hidden Payload Injection.** Carefully crafted triggers are embedded into accepted PRs or auxiliary files, often across multiple repositories.
- **Stage 4: Downstream Execution and Exploitation.** The malicious code is activated during build or runtime in downstream environments.

Each of these stages introduces unique risks, which are systematically captured in our framework via four corresponding dimensions: Dependency Impact (DI), Payload Concealment (PC), Community Quality (CQ), and Continuous Integration (CI). These dimensions form the core of our automated risk evaluation method.

### 3.2 Risk Evaluation Framework Design

**Design Objectives** The proposed framework aims to quantify and automate the evaluation of poisoning risks in open-source software supply chains, with a particular emphasis on identifying stealthy and human-facilitated backdoor injection scenarios that are often overlooked by traditional vulnerability scanners. Instead of focusing solely on code patterns, our design leverages LLMs to incorporate both semantic code analysis and human-oriented signals from community dynamics and dependency structures. The framework is driven by the following three objectives:

(1) To establish a reproducible and transparent assessment process integrating technical and social aspects of repository security, enabling consistent evaluation across diverse projects.

(2) To construct a set of granular, attack-stage-aligned risk metrics that capture critical adversarial behaviors such as dependency hijacking, privilege escalation, semantic obfuscation, and triggerability.

(3) To support scalable and automated repository scanning, producing structured risk matrices and aggregate risk scores that can inform early warning systems and downstream supply chain protection strategies.

**Unified Scoring Scheme** To ensure comparability and automation across heterogeneous repositories, all individual risk metrics are scored using a unified scheme composed of quantile normalization followed by weighted aggregation. Specifically, each raw metric value is mapped to a percentile score within the empirical distribution observed across a large corpus of repositories. For count-based indicators, a  $\log_{10}$  transformation is applied prior to normalization to reduce skew and enhance interpretability.

The scoring of these dimensions follows the unified quantile normalization and weighted aggregation strategy introduced in Section 3.3. In the following paragraphs, we detail the design rationale and highlight the high-impact features of each dimension.

A representative scoring function is:

$$R = \sum_{i=1}^n w_i \cdot Q_i, \quad (1)$$

Dimension	Index	Individual Risk Metric	Score	Analysis Method	Attack Stage
Dependency Impact	D.1	Self Priority Exposure	Normalized	Static	S.1
	D.2	Self Essential Exposure	Normalized	Static	S.1
	D.3	Dependency Priority Exposure	Normalized	Static	S.1
	D.4	Dependency Essential Exposure	Normalized	Static	S.1
Payload Concealment	P.1	Binary in Test Files	Boolean	Static and Semantic	S.3
	P.2	Binary in Documentation	Boolean	Static and Semantic	S.3
	P.3	Binary in Code Files	Boolean	Static and Semantic	S.3
	P.4	Binary in Asset Files	Boolean	Static and Semantic	S.3
	P.5	Binary in Other Files	Boolean	Static and Semantic	S.3
	P.6	Total Binary File Count (log-normalized)	Numeric	Static	S.3
Community Quality	Q.1	Community Popularity	Normalized	Static	S.1 / S.3
	Q.2	Community Review	Normalized	Static and Semantic	S.2 / S.3
	Q.3	Community Privilege Barrier	Normalized	Static	S.1 / S.2
Continuous Integration	C.1	Dependabot Enabled	Boolean	Static	S.4
	C.2	Dangerous Action Provider	Proportion	Static	S.4
	C.3	Dangerous Action Pin	Proportion	Static	S.4

Table 1: Structure of the Risk Evaluation Framework and Its Mapping to Attack Stages

where  $Q_i$  denotes the quantile-normalized value of the  $i$ -th metric and  $w_i$  is its assigned weight.

### 3.3 Risk Dimensions

To comprehensively assess the feasibility and potential impact of backdoor poisoning in software supply chains, the overall risk is decomposed into four orthogonal dimensions, each corresponding to a critical stage or behavioral aspect of the poisoning process. As shown in Table 1, each dimension consists of multiple sub-metrics, and scores are computed using a unified quantile normalization and weighted aggregation strategy (see Section 3.3). The following discussion briefly outlines the design rationale of each dimension and highlights the most representative high-risk features.

**Dependency Impact** This dimension quantifies the potential impact of poisoning a given repository by analyzing its position within the software dependency graph. Repositories that are deeply embedded in critical infrastructure are inherently more attractive targets for adversaries due to their wide transitive influence, such as system-level packages, cloud SDKs, or build toolchains.

To model this, we define four sub-metrics reflecting both upstream and downstream exposure:

- **Self Priority Exposure (D.1):** Counts how many packages built from the repository are labeled as `required`, `important`, or `standard` in target distributions.
- **Self Essential Exposure (D.2):** Measures how many of those packages are classified as `essential` under Debian’s definition.
- **Dependency Priority Exposure (D.3):** Indicates how many high-priority packages depend on this repository.
- **Dependency Essential Exposure (D.4):** Counts essential packages that transitively depend on it.

Metrics D.3 and D.4 carry relatively higher weights in this dimension, as they reflect the repository’s downstream influence and the difficulty of isolating the impact once poisoned. These metrics directly correspond to the attacker’s initial stage of target selection (S.1).

**Payload Concealment** This dimension assesses the repository’s potential to host concealed malicious payloads, especially those embedded in binary files that are hard to inspect or review. Unlike source code, modifications within binary files do not produce a human-readable “diff” in pull requests, which makes a thorough review of their changes nearly impossible. This opacity turns binary files into ideal containers for hiding information. Attackers often leverage such files to hide exploit logic, which may later be triggered through indirect means, as observed in the `xz` backdoor case.

We define six binary-related sub-metrics that reflect different functional contexts where binary files may appear:

- **P.1–P.5:** Boolean indicators of whether binary files are present in test files, documentation, source code, asset files, or other custom file types.
- **P.6:** A log-normalized count of all binary files found in the repository.

Among these, P.1 (binary in test files), P.3 (binary in code files), and P.6 (overall binary count) are assigned higher weights. This is due to their significant potential for concealing payloads and diminishing auditability, which corresponds to the S.3 attack stage (evading detection during code review). These vectors are particularly difficult to defend against because they are embedded within legitimate and high-frequency development activities. Since the submission of code and test files constitutes a normal part of the engineering workflow, malicious binaries can be camouflaged within these routine commits. Consequently, such backdoors often bypass conventional code reviews, as differentiating them from benign updates requires a level of scrutiny that is not always feasible.

**Community Quality** This dimension spans multiple stages of the poisoning attack lifecycle, including S.2 (infiltration) and S.3 (code merging), as it captures both the difficulty of obtaining elevated privileges and the rigor of code review practices. A key challenge for adversaries is to gain sufficient influence in the project, either by submitting

pull requests that are accepted with minimal scrutiny or by being promoted to maintainer after prolonged contribution.

Given the complexity of this threat surface, we organize community-related risk into three higher-level sub-dimensions:

- **Community Popularity (Q.1):** Reflects the overall exposure and external interest of the project, including stargazers, forks, and the volume of active participants. Less popular projects tend to attract fewer reviewers, making covert infiltration easier.
- **Community Review (Q.2):** Measures the enforcement strength of peer review mechanisms, including the prevalence of direct commits bypassing the pull request process, unreviewed pull request merges, and pull requests with descriptions that mismatch the actual modifications.
- **Community Privilege Barrier (Q.3):** Captures the difficulty of obtaining elevated privileges (e.g., maintainer or reviewer) by analyzing how many PRs were typically submitted before promotion.

This dimension was directly inspired by the stealthy escalation strategy employed in the recent `xz` backdoor incident, where the attacker infiltrated the project over an extended period through legitimate contributions and ultimately gained commit rights. However, we found that existing open-source risk scoring frameworks such as OpenSSF Scorecard, Snyk Advisor, and `deps.dev` largely overlook this type of social engineering risk, as they focus more on dependency freshness, CI safety, and artifact hygiene.

To address this gap, we constructed a fine-grained Community Quality model comprising **15 sub-metrics** spanning project popularity, review dynamics, and privilege escalation barriers. For brevity, only the high-level grouping is shown in Table 1.

Among these, sub-metrics in Q.2 and Q.3 are assigned higher weights. Weak peer review (Q.2) and low privilege elevation barriers (Q.3) are direct enablers for malicious code injection and long-term infiltration, respectively, making them particularly critical in high-stealth threat scenarios.

**Continuous Integration** This dimension evaluates the security posture of a repository’s automated workflows, particularly those defined in continuous integration (CI) pipelines. In many real-world poisoning attacks, CI configurations serve as the execution trigger for embedded payloads—either during build, test, or deployment stages.

We define three sub-metrics to assess CI-related risk:

- **Dependabot Enabled (C.1):** A boolean indicator of whether the repository has enabled GitHub’s Dependabot to keep dependencies up to date and reduce stale or vulnerable components.
- **Dangerous Action Provider (C.2):** The proportion of workflow steps that use untrusted or community-maintained GitHub Actions, which may introduce unintended behaviors.
- **Dangerous Action Pin (C.3):** The proportion of Actions that are not pinned to specific commit hashes, making them susceptible to upstream changes or hijacking.

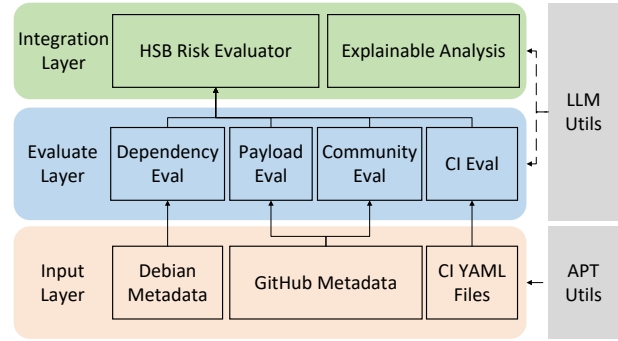


Figure 2: System architecture of the proposed risk assessment tool.

This dimension corresponds to attack stage S.4 (triggering), where malicious logic is activated via automated execution paths. Notably, C.2 and C.3 are assigned higher weights due to their direct influence on execution determinism and trust boundaries. Even though CI-related signals contribute less to the overall score, a small misconfiguration in CI can still serve as a stealthy trigger vector for payload activation, underscoring its relevance in end-to-end attack modeling.

**HSBR Score** To quantify the overall poisoning risk of a repository, we aggregate the scores of all four dimensions into a unified metric: the High-Stealth Backdoor Risk (HSBR) Score. This composite score is designed to prioritize cases where the backdoor may propagate widely or evade detection for extended periods.

Although all four dimensions contribute meaningfully to the final risk assessment, we assign higher weights to **Dependency Impact (DI)** and **Community Quality (CQ)** based on their critical roles in real-world attack scenarios. DI reflects the potential blast radius through transitive dependencies, while CQ captures human-centered vulnerabilities in project governance—both of which were pivotal in the `xz` incident.

Formally, the overall HSBR score is computed as:

$$R_{\text{total}} = w_{\text{DI}} \cdot R_{\text{DI}} + w_{\text{PC}} \cdot R_{\text{PC}} + w_{\text{CQ}} \cdot R_{\text{CQ}} + w_{\text{CI}} \cdot R_{\text{CI}}. \quad (2)$$

In our implementation, we empirically set parameters  $(w_{\text{DI}}, w_{\text{PC}}, w_{\text{CQ}}, w_{\text{CI}}) = (0.3, 0.2, 0.3, 0.2)$  to emphasize the propagation and infiltration risks. These weights can be adjusted under different threat modeling assumptions.

### 3.4 Implementation of the Risk Assessment Tool

**System Overview** We develop an end-to-end automated tool named **HSBRE** to assess high-stealthy poisoning risks in open-source repositories. As shown in Figure 2, the system adopts a three-layered modular architecture: (1) the *Input Layer*, which collects structured metadata from Debian packages, GitHub repositories, and CI configurations; (2) the *Evaluate Layer*, which performs metric-specific analysis aligned with our four risk dimensions—Dependency Impact (DI), Payload Concealment (PC), Community Quality

(CQ), and Continuous Integration (CI); and (3) the *Integration Layer*, which aggregates scores into a unified HSBR score and generates explainable summaries.

**Input Layer: Metadata Collection** Metadata collection begins by identifying high-priority Debian packages and resolving their full transitive dependencies using APT-based utilities. We extract priority levels and essential status to support DI metrics (D.1 to D.4). Repository-level metadata, including PR history, issue discussions, contributor graphs, and commit activity, is collected via the GitHub API. CI configurations (YAML files) are parsed to extract workflow triggers and action providers. The collected metadata is aggregated into a unified repository-level representation, which serves as input to the Evaluate Layer and supports metric computation across all four dimensions.

**Evaluate Layer: Dimension-specific Scoring** For each risk dimension, we implement a dedicated evaluation module: (1) *Dependency Eval* scores DI metrics based on package criticality and dependency graph position; (2) *Payload Eval* flags high-risk binary usage contexts (e.g., test/code folders); (3) *Community Eval* analyzes review patterns, privilege elevation, and social activity using LLM-powered semantic modules; and (4) *CI Eval* inspects YAML workflows for unpinned or untrusted actions. Notably, LLM-based interpretation is used in CQ and PC dimensions to detect semantic inconsistencies (e.g., misleading PR titles, suspicious binary usage). Each module outputs a structured risk vector which is passed to the Integration Layer.

**Integration Layer: HSBR Calculation and Explainability** This layer aggregates the normalized scores from all dimensions using the HSBR Score formulation (Equation 2) and corresponding weights. The final score reflects stealthiness-aware poisoning risk and is used for both ranking and alerting. In addition, the layer leverages LLM-generated rationale traces to provide explanation snippets for high-risk signals (e.g., User gained maintainer role with only 2 PRs, CI runs unpinned third-party actions). These explanations support transparency and auditability for downstream stakeholders.

## 4 Experimental Evaluation

### 4.1 Setup

To assess supply chain poisoning risks in open-source software, we focus on high-priority packages within the Debian ecosystem (Onagh and Nayebi 2025), which provides a widely adopted and hierarchically structured foundation for system-level risk evaluation. A total of 319 Debian packages labeled as *required*, *important*, or *standard* were selected, as they typically constitute the functional backbone of the system. Among these, 66 packages were found to have valid public GitHub repositories, enabling repository-level risk analysis using our proposed LLM-based evaluation framework. GPT-4o from OpenAI (Widder, Whittaker, and West 2024) was adopted as the language model backend to assist in semantic-aware evaluations.

It is worth noting that, although our assessment was conducted on Debian packages, the proposed framework can

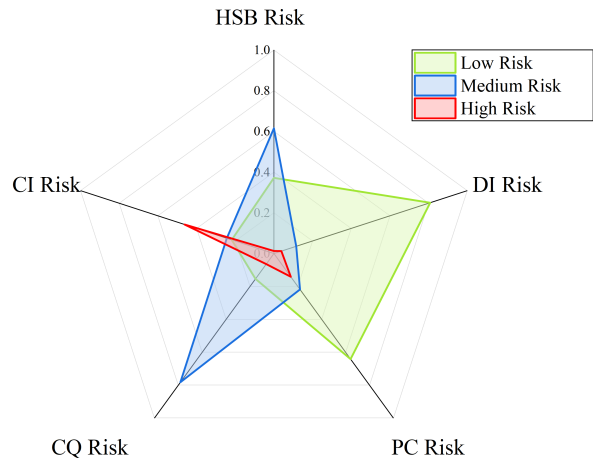


Figure 3: Distribution of repository risk levels across four assessment dimensions as well as the overall HSB Risk.

be readily adapted to other Linux distributions with similar package structures, such as Ubuntu or Arch Linux, making the methodology broadly applicable across open-source ecosystems.

### 4.2 Result

Our analysis reveals several key findings regarding how supply chain poisoning risks manifest across repositories (Figure 3, Table 2). The computation of each dimension-level risk score is based on a collection of weighted individual risk metrics.

- **(i) Community Quality poses the most widespread and actionable threat surface.** Although only 1.5% of repositories are classified as High risk in this dimension, over 80% fall into the Medium-risk range. Given that community structures directly govern permission control and review processes, which are key enablers in real-world attacks, this dimension constitutes the broadest and most impactful risk surface.
- **(ii) Continuous Integration presents the most concentrated high-risk segment.** The CI dimension has the highest proportion of High-risk repositories (39.4%), primarily due to misconfigured workflows (e.g., disabled Dependabot, unpinned or untrusted GitHub Actions). These insecure automation practices offer direct exploit paths during build or deployment.
- **(iii) Payload Concealment risks are non-negligible and structurally embedded.** Around 13.6% of repositories are High risk in this dimension. Many allow binary files in test and asset areas, enabling covert payload storage without triggering code-level alarms.
- **(iv) Dependency Impact appears structurally less critical.** Over 78% of repositories are rated Low risk in this dimension, suggesting that most targets are not deeply embedded in critical dependency chains. This implies limited reachability from transitive infection paths.

Index	Risk Metric	HS Ratio (%)
C.1	Dependabot Disabled	62.12
Q.3	Community Privilege Barrier	53.03
C.2	Dangerous Action Provider	39.39
P.1	Binary in Test Files	34.85
C.3	Dangerous Action Pin	31.82
P.4	Binary in Asset Files	28.79
P.2	Binary in Documentation	27.27
P.6	Total Binary File Count	19.70
D.1	Self Priority Exposure	13.64
P.5	Binary in Other Files	13.64
D.3	Dependency Priority Exposure	12.12
D.4	Dependency Essential Exposure	6.06
Q.1	Community Popularity	6.06
D.2	Self Essential Exposure	4.55
P.3	Binary in Code Files	3.03
Q.2	Community Review	0.00

Table 2: High-risk ratio for each individual metric across all repositories. HS Ratio is short for High-risk ratio.

## 5 Discussion

### 5.1 Case Study: XZ Backdoor Attack

To validate the effectiveness of our evaluation framework, we conduct an in-depth case study on the `xz` repository<sup>1</sup>, which was involved in a high-profile supply chain backdoor incident in 2024. The attack targeted the `liblzma5` and `xz-utils` packages, which are critical components in many Linux distributions, and subsequently propagated into downstream systems such as OpenSSH.

**Stage 1: Reconnaissance and Target Selection.** The `xz` repository plays a pivotal role in system-level functionality, as confirmed by its maximum score in both *Dependency Priority Exposure* (1.0) and *Dependency Essential Exposure* (1.0). These scores indicate that it is widely depended upon by high-priority and essential Debian packages, making it a strategically valuable target from a transitive influence perspective.

**Stage 2: Initial Access via Social Infiltration.** Despite having moderate community activity (e.g., *Community Popularity* = 0.21), the repository exhibits weak barriers to privilege escalation. It scores relatively high on *Community Privilege Barrier* (0.64), suggesting that users could acquire elevated permissions (e.g., merge rights) without extensive scrutiny. Moreover, a non-negligible *Community Review* score (0.58) implies inconsistent PR vetting, potentially allowing social infiltration via legitimate contributions.

**Stage 3: Hidden Payload Injection.** A high score in *Binary in Test Files* (1.0) and a near-maximum value in *Total Binary File Count* (0.88) indicate that binary content was prevalent, especially in test scaffolding. This aligns with the attacker’s use of disguised payloads embedded within auxiliary files. Although no binary was detected in core code or documentation areas, the permissive handling of binaries increases the risk of covert injection.

**Stage 4: Downstream Execution and Exploitation.** This phase was characterized by the inclusion of subtle build-time

<sup>1</sup><https://github.com/tukaani-project/xz>

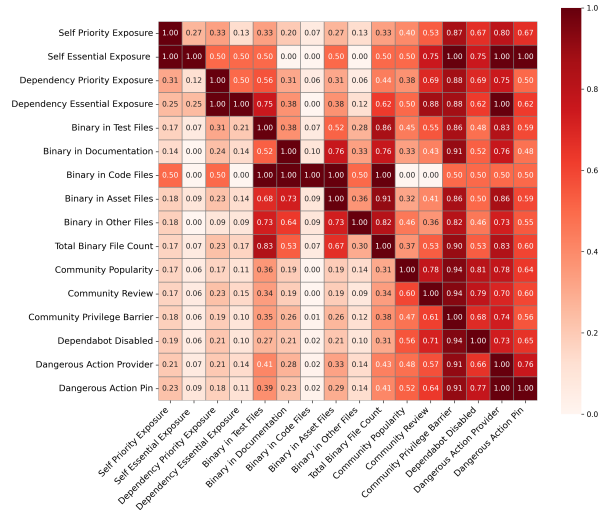


Figure 4: Heatmap of cross-metric correlation.

triggers. The repository shows a maximum risk score in *Dependabot Disabled* (1.0), weakening automated dependency alerts, and a high value in *Dangerous Action Provider* (0.78), indicating that unverified GitHub Actions were used. These insecure CI practices facilitated undetected activation of the payload in downstream environments.

Overall, the `xz` attack aligns closely with our four-stage poisoning model, and each stage is reflected by corresponding high-risk signals in our metric system.

### 5.2 Cross-Metric Correlation Analysis

Figure 4 presents the pairwise correlations among all risk metrics, revealing several clear clusters. Binary-related metrics show consistently strong mutual correlations, with all pairwise values above 0.7, validating their grouping under Payload Concealment. CI-related signals, including *Dependabot Disabled*, *Dangerous Action Provider*, and *Dangerous Action Pin*, form a tight cluster, with all pairwise correlations above 0.7, reflecting shared CI security practices. By contrast, dependency and community metrics correlate weakly with other dimensions, confirming that they capture complementary aspects of risk and motivating a multi-dimensional evaluation model.

## 6 Conclusion

In this work, we present an automated, attacker-centric framework for multi-dimensional assessment of high-stealth backdoor risks in open-source software supply chains. Experiments on 66 high-priority Debian packages and the `xz` backdoor case show that weaknesses in community governance and CI configurations are the most pervasive risk factors, highlighting the necessity and practical value of multi-dimensional risk modeling.

## Acknowledgments

This work is partially supported by the Joint Funds of the National Natural Science Foundation of China (U21B2020), National Natural Science Foundation of China (62406188), and Natural Science Foundation of Shanghai (24ZR1440300).

## References

- Aberdour, M. 2007. Achieving quality in open-source software. *IEEE software*, 24(1): 58–64.
- Anisa, G.; and Widianingsih, F. 2024. SolarWinds attack: Stages, implications, and mitigation strategies in the cyber age. *Electronic Integrated Computer Algorithm Journal*, 2(1): 47–52.
- Barabanov, A.; Markov, A.; and Tsirlov, V. 2020. On systematics of the information security of software supply chains. In *Proceedings of the Computational Methods in Systems and Software*, 115–129. Springer.
- Delicheh, H. O.; and Mens, T. 2024. Mitigating security issues in github actions. In *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, 6–11.
- Fu, Z.-H.; Ge, M.-F.; Ding, T.-F.; and Liu, Z.-W. 2025. Human-Oriented Opportunistic Control for Time-Varying Formation of NAVs with Opportunistic-Moving Leader and Intermittent Human-Computer Interactions. *IEEE Transactions on Vehicular Technology*.
- Håkegård, J. E.; Lundkvist, H.; Rauniyar, A.; and Morris, P. 2024. Performance evaluation of an open source implementation of a 5G standalone platform. *IEEE Access*, 12: 25809–25819.
- Hamer, S.; Bowen, J.; Haque, M. N.; Hines, R.; Madden, C.; and Williams, L. 2025. Closing the Chain: How to reduce your risk of being SolarWinds, Log4j, or XZ Utils. *arXiv preprint arXiv:2503.12192*.
- Kalu, K. G.; Okorafor, S.; Durak, B.; Laine, K.; Moreno, R. C.; Torres-Arias, S.; and Davis, J. C. 2025. ARMS: A Vision for Actor Reputation Metric Systems in the Open-Source Software Supply Chain. *arXiv preprint arXiv:2505.18760*.
- Kamau, J. W. 2025. Investigating the Influence of User Training on the Adoption of Desktop Open Source Software: A Case of Students in Selected Universities in Kenya. *African Journal of Empirical Research*, 6(2): 361–369.
- Lin, C.-C.; Mirzaei, M.; Chu, E.-T.; and Cheng, C. C. 2024. HPDH-MI: A High Payload Data Hiding Technique for Medical Images Based on AMBTC. *Symmetry*, 16(12): 1634.
- Martínez, J.; and Durán, J. M. 2021. Software supply chain attacks, a threat to global cybersecurity: SolarWinds’ case study. *International Journal of Safety and Security Engineering*, 11(5): 537–545.
- McGraw, G. 2015. Software security and the building security in maturity model (BSIMM). *Journal of Computing Sciences in Colleges*, 30(3): 7–8.
- Onagh, E.; and Nayebi, M. 2025. Extension Decisions in Open Source Software Ecosystem. *Available at SSRN 4975545*.
- Przymus, P.; and Durieux, T. 2025. 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)*, 91–102. IEEE.
- Shao, C.; Li, G.; Wu, J.; and Zheng, X. 2024. Exploring semantic redundancy using backdoor triggers: A complementary insight into the challenges facing dnn-based software vulnerability detection. *ACM Transactions on Software Engineering and Methodology*, 33(4): 1–28.
- Tan, Z.; Parambath, S. P.; Anagnostopoulos, C.; Singer, J.; and Marnerides, A. K. 2025. Advanced persistent threats based on supply chain vulnerabilities: Challenges, solutions & future directions. *IEEE Internet of Things Journal*.
- Väyrynen, V. 2025. The Human Factor in OSS Security: risks and mitigation strategies.
- Vazquez Melendez, E. I.; Bergey, P.; and Smith, B. 2024. Blockchain technology for supply chain provenance: increasing supply chain efficiency and consumer trust. *Supply chain management: An international journal*, 29(4): 706–730.
- Widder, D. G.; Whittaker, M.; and West, S. M. 2024. Why ‘open’ AI systems are actually closed, and why this matters. *Nature*, 635(8040): 827–833.
- Williams, L.; Benedetti, G.; Hamer, S.; Paramitha, R.; Rahman, I.; Tamanna, M.; Tystahl, G.; Zahan, N.; Morrison, P.; Acar, Y.; et al. 2025. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology*, 34(5): 1–38.
- Xia, B.; Bi, T.; Xing, Z.; Lu, Q.; and Zhu, L. 2023. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2630–2642. IEEE.
- Yousaf, A.; and Zhou, J. 2024. From sinking to saving: MITRE ATT &CK and D3FEND frameworks for maritime cybersecurity. *International Journal of Information Security*, 23(3): 1603–1618.
- Zahan, N.; Kanakiya, P.; Hambleton, B.; Shohan, S.; and Williams, L. 2023. Openssf scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy*, 21(6): 76–88.