Anticipating High-Risk Open Source Projects for Supply Chain Subversion

Yassir Laaouissi

OS3 - Research Project 2 - July 2025

Supervisor: Max van der Horst

Abstract

Open-source software (OSS) supply chains face threats from malicious subversion of widely used components. This project investigates structural and governance-related risk factors that can make certain OSS projects vulnerable to supply chain attacks. Using the 2024 backdoor incident [8] in xz/liblzma (an essential Linux compression library) as a model case, to identify patterns of asymmetric trust and structural opacity (e.g., lone maintainers, lack of code review) that often precede such compromises. The methodology combines GitHub metadata analysis (maintainer activity, contribution patterns, security practices) with anomaly detection in commit and release histories to flag projects exhibiting abnormal risk signals. The goal is to detect "subversion-prone" OSS projects before an attack occurs by using human and project health indicators. For this, a scoring system is developed based on these risk signals and applied to critical OSS packages. The results yield a ranked list of currently available high-impact open source projects, such as sgerrand/alpine-pkg-glibc. with potential supply chain weaknesses. Demonstrating that proactive outside-in monitoring of OSS governance can provide early warnings of subversion risk. This research aims to inform maintainers, ecosystem managers, and security teams of predictive indicators for OSS supply chain security, contributing to a more resilient open-source ecosystem.

1. Introduction

The security of open-source software (OSS) supply chains has become a major concern in recent years [2], driven by high-profile compromises such as **SolarWinds** [3], the **event-stream** npm package incident [4], and the **xz/liblzma** backdoor in 2024 [5]. These incidents highlight how attackers can exploit the trust and ubiquity of OSS components to introduce malicious code downstream. As was the case in the xz/liblzma incident, where a malicious contributor injected a backdoor into a core Linux utility, underscored the need for better *anticipation* of such threats.

Unlike traditional software vulnerabilities, supply chain subversions often involve an attacker leveraging insider access or maintainer privileges over an extended period. The xz/liblzma backdoor was only discovered post factum when a user noticed unusual system behavior (slight delays in SSH login) after upgrading to the compromised version. By the time the issue was investigated, the malicious code had already been released and nearly spread to numerous Linux distributions. This reactive discovery illustrates the stakes: *we need methods to spot at-risk OSS projects before the damage is done.*

1.1 Case Study: xz/liblzma Backdoor Incident (Model Case for Signal Identification)

To design and inform our risk assessment approach, we thoroughly analyzed the **xz Utils (liblzma)** backdoor – a real-world OSS supply chain subversion attempt that occurred in early 2024. XZ Utils is a widely-used data compression library included in virtually all

Linux distributions (often as `liblzma`). In version 5.6.0 (released January 2024) and 5.6.1, a backdoor was introduced by the project's new maintainer that could allow remote code execution through the compression routine. The attack was notable because it was not an external compromise; it was an *insider attack* by someone who had become the de facto maintainer of the project.

The original longtime maintainer of xz (author of XZ Utils) had stepped down and transferred maintainership to an individual using the alias "Jia Tan." There was minimal overlap or supervisory period – effectively a single hand-off of control. For a few months, this new maintainer carried out normal-looking development, making legitimate improvements to the code. However, hidden among these commits, they added a malicious code snippet that introduced an authentication bypass in certain scenarios (the backdoor). The changes were subtle enough not to trigger immediate suspicion. The malicious maintainer then promptly created new releases (v5.6.0 and v5.6.1) containing this code and worked to get these versions adopted in downstream repositories and Linux distributions.

The backdoor was discovered only incidentally. A Linux developer (Andres Freund) noticed that after updating XZ Utils, his SSH server's login process became a few milliseconds slower and used slightly more CPU, with no obvious explanation in the changelog. This anomaly led him to inspect the code more closely, ultimately uncovering the injected backdoor. Once public, the revelation triggered a frantic response to ban the malicious version from distributions and investigate the new maintainer's identity (who appears to have been an imposter). The incident could have been far worse; because it was caught relatively quickly (before mass deployment), the damage was limited. Nonetheless, it served as a wake-up call for the community.

Post-mortem analyses of the xz/liblzma compromise [8] revealed that the following risk factors were present *before* the backdoor was discovered, and these directly informed the design of our risk model:

- **Bus Factor:** At the time of the incident, **xz/liblzma had only one active maintainer**, the attacker (Jia Tan). After the handover, no other co-maintainers were actively committing. This is the classic bus factor = 1 scenario. The project's fate rested in one person's hands, which made the subversion possible without any secondary approval.
- **Contributor Churn:** There was a *maintainer turnover* just prior to the incident – the original author (Micah Snyder) became inactive, and a new contributor stepped in as sole maintainer. This kind of churn (especially if abrupt) is exactly when a project is vulnerable: the community likely felt grateful someone volunteered to take over, and may have been less critical or review-heavy toward that person's contributions. The long period of stable maintenance by a trusted author followed by a complete change in personnel created an opening for exploitation.
- **Pull Request Review Patterns:** Under the new maintainer, **code changes were merged with little to no external review**. Many commits were directly pushed or PRs self-approved by the attacker. Since no other active maintainers were participating in code review, the usual checks and balances were absent. This allowed malicious code to slip in under the guise of routine updates. In retrospect, the lack of review comments or oversight on significant changes (like those in 5.6.0) was an anomaly that could have been noted.
- **Maintainer Privilege Transitions:** The critical event enabling the attack was the **granting of commit and release privileges to a new, unverified maintainer**. The project did not have a formal process or community vote for this transition – it was an informal handoff. Thus, an outsider rapidly gained full control. Our signal for maintainer transition would have fired in this scenario, highlighting that a permission change occurred.

- **Branch Protection:** The fact that the backdoor could be injected implies that **there were no strong branch protection rules in effect**. Specifically, there was likely no requirement for a second maintainer to review or approve changes before they were merged into the main branch. If such protections had been in place (and another trusted maintainer existed to do reviews), the malicious code might have been caught or at least delayed.
- **Commit Frequency Spikes:** Prior to the backdoor's introduction, the repository's commit activity showed some **unusual spikes**. For example, the attacker might have made a flurry of commits to prepare the new release, possibly to obfuscate the malicious change among others. XZ Utils had historically been relatively stable, so any sudden burst of commits in a short time frame stood out statistically. Our analysis of commit timestamps would have flagged this deviation as a possible sign of orchestrated activity.
- **Release Cadence Anomalies:** Correspondingly, the project's release pattern changed. The attacker hurriedly pushed out backdoored releases and urged adoption. This **sudden acceleration in release cadence** – two releases in quick succession (5.6.0 and 5.6.1) – was abnormal for XZ Utils. It suggested an ulterior motive (getting the malicious version widely deployed) rather than the typical pace of development. Our release interval check captures such irregular timing.
- **Signed Commits/Releases:** XZ's malicious releases were **not signed with any maintainer GPG key** (or signing was used inconsistently). This meant there was no cryptographic attestation of the commits' origin beyond the attacker's GitHub account, and users had no easy way to detect tampering. A vigilant observer might have been alarmed that a critical package suddenly delivered unsigned releases or that the signing key wasn't the one used by the previous maintainer.
- **Top Contributor Dominance:** During the lead-up to the discovery, essentially **100% of commits in XZ Utils were coming from the attacker**. By definition, the top contributor (Jia Tan) was responsible for nearly all recent changes, which correlates with the bus factor of 1. Such dominance meant there was little diversity in code contributions—another sign of single-threaded control.
- **Lack of Formal Governance:** Finally, XZ Utils had **no public documentation of how maintainers are chosen or how the project is governed**. The transfer of maintainership happened quietly through personal communication. There was no community process to vet the new maintainer or to oversee the transition. This lack of transparency and policy made it easier for an attacker to assume control without raising alarms. In projects with well-defined governance (e.g., requiring multiple endorsers for a new maintainer, or a transition period), such an abrupt change might have been met with more scrutiny.

In hindsight, each of these factors contributed to the success of the subversion attempt. Importantly, none of them are vulnerabilities in the code itself; they are weaknesses in the project's structure and processes. If a system had been in place to monitor OSS projects for these warning signs, **xz/liblzma would potentially have been flagged as a high-risk project in late 2023** (when the original maintainer stepped down and activity patterns shifted). **Indeed, if our model were applied to historical data for xz/liblzma leading up to the incident, it would likely have yielded a score of 12/13, triggering nearly all defined risk signals.** The xz case study thus provides a blueprint of signals that a pre-compromise detector should look for, and our risk model encapsulates those signals.

Research Questions: This study seeks to answer several key questions.

- First, how generalizable is the xz/liblzma case to other OSS components in different ecosystems? In other words, are there common warning signs that preceded the xz attack which might also appear in other projects?
- Second, what structural, social, and governance factors make OSS projects more susceptible to supply chain subversion?

  - This study hypothesizes, based on a case study of the liblzma/xz incident documented in this report, that certain project characteristics – e.g., a "bus factor" of one (only a single maintainer), lack of peer review, or recent turnover in maintainers – can create opportunities for malicious contributors. The aim is to identify any early warning signals from prior cases that indicate heightened risk.
- Third, which widely-used OSS components currently exhibit these risk factors, and thus could be considered susceptible to such attacks? Answering this involves surveying critical OSS projects for risk factors and producing a ranked list of those most at risk.

Addressing these questions is important because modern software and infrastructure heavily depend on open-source dependencies [6]. A compromise in a low-level library or tool (as with xz utils) can cascade into thousands of products and systems.

Being able to anticipate OSS subversion allows stakeholders to harden processes, scrutinize code changes, or even intervene with maintainers *before* an incident occurs. In contrast to existing supply chain security efforts that are largely reactive (post-mortem analysis of incidents) or reliant on project self-reporting, this study explores an approach which is *proactive* and "outside-in". It uses only publicly observable data, requiring no consent or instrumentation from the project being evaluated.

In summary, this research sets out to demonstrate that by monitoring certain human and structural signals in open-source projects, one can obtain an early warning system for supply chain attacks, thus improving the trust and resilience of the OSS ecosystem.

2. Literature Review

2.1 Industry Initiatives

A number of tools and frameworks have emerged to assess OSS project security and integrity. The Open Source Security Foundation's **Scorecard** project [1], for example, evaluates repositories against best practices such as the use of branch protection rules, dependency update automation, and continuous integration tests. Google's **SLSA** (Supply-chain Levels for Software Artifacts) framework [7] provides guidelines for build systems and artifact provenance to ensure software hasn't been tampered with during build and release processes. These initiatives set valuable baseline standards; however, they primarily focus on *technical hygiene* and infrastructure (e.g., enforcing code signing, reproducible builds) rather than the *social trust* dynamics of a project.

These approaches often lack deep introspection into governance risks or maintainer trust relationships. This is, for example, due to the fact that tools like Scorecard and SLSA typically require the project's participation or integration (an "inside-in" approach), meaning they function best for projects that have already adopted certain practices.

This makes them less effective for *outsiders* trying to identify risk in a project that has not opted into these frameworks. In short, existing industry scorecards can tell if a project follows best practices, but they may miss subtler warning signs of intentional subversion, especially if an attacker is following the rules superficially while plotting a backdoor.

2.2 Dependency Metadata Services

Other efforts focus on ecosystem-wide metadata. Platforms like **libraries.io** and **deps.dev** aggregate information on package dependencies, version histories, maintainers, and licensing. These services help identify critical projects (for example, a library that many others depend on) and can flag potential issues like outdated maintainers or abandoned projects. They enable reasoning about project centrality and even *abandonment risk*. However, they offer limited insight into the *maintainer behaviors or governance processes* within a project.

For instance, libraries.io might show that a given library has had no new release in a long time (a maintenance signal), but it won't reveal if the lone maintainer of that library quietly handed over control to a new, unknown individual. In the context of supply chain attacks, dependency graphs alone do not capture who is in control or how code is contributed and reviewed.

2.3 Academic Research on OSS Trust

The research community has examined open-source trust and sustainability from various angles. Gkortzis and Spinellis [11], for example, analyzed project evolution to detect signs of *software aging* such as reduced activity or increasingly centralized control. Those factors are relevant for subversion risk – e.g., a project with one remaining active contributor (centralized control) or irregular maintenance (potentially looking for help) might be more vulnerable to a malicious volunteer. However, much of this work has focused on code quality or vulnerability discovery rather than malicious *maintainer behavior*.

For instance, Gkortzis & Spinellis's methods center on code and repository metrics to predict project abandonment, not on detecting an *infiltrator* who is actively contributing code. Overall, there is a gap in academic literature on pre-mortem detection of OSS supply chain attacks – most studies and tools analyze incidents after they occur or focus on technical vulnerabilities inside the code, rather than subtle shifts in the social governance of the project.

2.4 Case Studies of Past Incidents

Several documented supply chain compromises shed light on what existing tools miss. Cox [8] provides an analysis of the xz/liblzma backdoor, revealing how a malicious maintainer sustained a long-term deception to land malicious code into a critical library.

This incident highlighted weaknesses in trust delegation and oversight: the original maintainer, after years of work, unwittingly handed control to an attacker with little community fanfare. Another notable case is **event-stream (2018)**, where an npm library author transferred ownership of his widely-used project to a stranger who then injected malware – an attack dissected by Wayne [9]. That case demonstrated how *social engineering* (someone volunteering to help an overworked maintainer) combined with the absence of code review led to a supply chain exploit.

Similarly, the **PHP source repository compromise (2021)** involved attackers directly breaching the self-hosted Git server for PHP and adding a backdoor, bypassing GitHub's protections entirely (an incident reported by Sharma [10]).

Each of these cases underscores the human factor: maintainers can be fooled or malicious, code review can be bypassed if not enforced, and infrastructure can be targeted. Crucially, these attacks were not flagged by static analyzers or dependency scans, since the code *appeared legitimate until its hidden purpose was discovered.* They were only caught after-the-fact, demonstrating the need for proactive monitoring of *who is doing what* in critical OSS projects.

In summary, existing tools provide pieces of the puzzle (best practices checks, dependency criticality, etc.) but leave a significant gap in detecting the kinds of long-con, **inside threat** scenarios that characterize supply chain subversion. This project attempts to bridge that gap with an "outside-in" approach that monitors OSS projects for early warning signals visible from their public activity.

3. Methodology

The approach taken uses an **outside-in view** of open-source projects, using publicly available data (primarily from GitHub) to assess the health and integrity of projects without requiring any action from the project's maintainers apart from publishing the project to the public. The methodology can be summarized in four stages: (1) Project selection, (2) Risk signal identification, (3) Data collection and analysis using Python, and (4) Risk scoring.

3.1 Risk Signal Identification

Drawing from the xz/liblzma case (detailed in Section 1.1) and insights from literature [8], a set of OSS risk signals was defined. These signals serve as measurable attributes that might indicate a project's vulnerability to a supply chain attack. The key attributes examined and their motivations are as follows:

- **Bus Factor (Maintainer Redundancy):** The bus factor is the number of active maintainers or the share of contributions by the top contributor. A high bus factor risk means the project relies on very few individuals. For example, if only one person is effectively maintaining the project, the bus factor is 1 – a single point of failure. This study quantified this by checking what fraction of commits come from the top contributor. In the implementation used in this study, if one developer accounts for more than ≈80% of the total contributions, it flags the bus factor as dangerously low. The 80% threshold is grounded in empirical practice, as a 2025 study defines "bus factor = 1" when a single developer contributes ≥**80%** of all commits—demonstrating that this cutoff reliably identifies projects with critically low maintainer redundancy [12].
- **Contributor Churn:** Contributor Churn refers to turnover in maintainers or core contributors over time. Healthy projects may gain and lose contributors naturally, but low churn (the same maintainer(s) for many years with no new blood) could mean a lack of oversight or that a departure of one person creates a vacuum. Conversely, a sudden change (an old maintainer leaves and a new one takes over) could also be a risk pointer. This study tracks how many distinct years contributors joined the project to gauge churn; if all maintainers joined in the same year (indicating no turnover), a project is considered to have a low contributor churn. The xz case exemplified this: a long-time maintainer became inactive, and an attacker stepped in as the sole new maintainer, meaning there was effectively a single hand-off rather than ongoing healthy rotation.
- **Branch Protection and Review Process:** Whether the repository requires pull request reviews before code can be merged into the main branch. Strong governance would enforce that no single maintainer can push code unilaterally without review (via GitHub's branch protection rules requiring approvals). This study used the GitHub API to check if the default branch (e.g., "main") has required pull request reviews enabled. If no such protection is in place, the project is flagged for lack of enforced peer review. Additionally, pull request review patterns were examined: how many pull requests were merged without any discussion or review comments. A project where a large fraction of PRs are self-approved or have zero review comments indicates poor review hygiene. The proof of concept analysis computes the percentage of merged PRs that had no comments; if more than 50% of recent merged PRs had zero commentary or review, they were deemed to have an insufficient review process. This suggests code changes may not be getting proper oversight. In xz/liblzma, for instance, the malicious maintainer could merge their own code with minimal or no review from others, implying either a permissive process or the absence of other reviewers. The **50% threshold** is based on empirical findings showing that, in very small teams (≤5 committers), **over**

half of pull requests are merged without any code review, indicating a lack of enforced peer oversight and thus flagging insufficient review hygiene [13].

- **Maintainer Privilege Transitions:** Changes in who has commit or release authority. Supply chain attacks often exploit moments of transition [14] – e.g., when a project founder hands over control to a new maintainer, or when someone new is granted commit rights without thorough vetting. The proof of concept leveraged GitHub event logs to detect recent changes like additions or removals of collaborators with elevated permissions. If keywords such as "added" or "removed" were found in repository events regarding admin roles or permissions, they flagged a maintainer transition. The rationale is that a project that recently underwent a leadership change deserves scrutiny, especially if the new maintainer is not well known. In the xz case, the original maintainer stepped away and the project was effectively handed to an attacker ("Jia Tan") who then used that position to insert a backdoor.
- **Commit Frequency Spikes:** Unusual bursts of commit activity in short time spans. Attackers may attempt to push through malicious changes among bursts of ostensibly normal updates, or a sudden spike might indicate an urgent push (as was the case when the xz maintainer rushed out releases with the backdoor). This study analyzed the commit history timeline for anomalies using simple statistical checks. Specifically, the average commits per day were calculated and flagged any day with a commit count more than 3 standard deviations above the mean as a **commit spike**. A project with normally low activity that suddenly has a frenzy of commits could be cause for attention. Prior to the xz backdoor insertion, there were indeed sudden bursts of repository activity that were out of character for the project.
- **Release Cadence Anomalies:** Similarly, the timing of software releases is analyzed. Consistent release cadence (e.g., one release every few months) is normal for many projects. If a project suddenly makes an unusually timed release or a rapid series of releases, it might signal an attempt to propagate a change quickly. This study computed the intervals between successive releases and flagged cases where any interval deviated significantly ($>3\sigma$) from the project's mean release interval. In xz/liblzma, after the malicious code was added, there was a sudden push to tag new releases and get them included in downstream Linux distributions quickly – a deviation from its prior slower release schedule. Release cadence anomalies were detected by calculating the mean (μ) and standard deviation (σ) of inter-release intervals. Any interval deviating by more than 3σ from the mean was flagged as anomalous, following standard outlier detection practice. This method highlights irregular release bursts, such as the one observed in the XZ/liblzma backdoor case.
- **Signed Commits/Releases:** The use of cryptographic signatures on commits or release tags is a supply chain security best practice. While not foolproof (a malicious maintainer can sign their bad commits), a lack of signing means that if an account is compromised, malicious commits could be made without detection. The proof of concept checked what fraction of recent commits were GPG-signed and considered it a risk signal if the rate was very low. In this study's implementation, if fewer than 20% of sampled commits were signed, it flags "low commit signing rate". xz's attacker did not consistently sign tags or commits, so consumers had no cryptographic provenance to detect something amiss. Since empirical studies show that only around 10% of commits in popular open-source projects are GPG-signed [15] and software signing adoption remains low even in industry [16], setting a conservative threshold of 20% provides a reasonable benchmark for flagging projects with insufficient signing hygiene.
- **Top Contributor Dominance:** This metric overlaps with bus factor but specifically quantifies if one contributor is doing the vast majority of work. Even if there are multiple maintainers on paper, one person might still author, say, 95% of all commits –

meaning effectively they dominate the project's development. This study calculated the share of contributions by the top author; if the top author >85% of commits, the proof of concept flagged it. This condition often coincides with a bus factor of one, but it can also catch cases where, for example, there are a few maintainers but one is far more active than others. Such dominance might lead to less oversight and busyness that an attacker could exploit. In the selected case study, "Jia Tan" was essentially the only active contributor during the backdoor period, so both bus factor and top-contributor dominance were triggered.

- **Maintainership Policy Transparency:** This research also checked whether an OSS project documents any governance or maintainership policies (e.g., a CONTRIBUTING.md or GOVERNANCE file that outlines how new maintainers are added or how code gets reviewed). The presence of a formal policy can indicate that the project has thought about trust and process, whereas absence might correlate with ad-hoc or opaque decision-making. The proof of concept scanned for files named `CONTRIBUTING`, `GOVERNANCE`, `MAINTAINERS`, or similar, and looked for keywords like "maintain", "govern", or "policy" in them. If no evidence of a maintainer governance policy was found, it flagged the project for having *no visible maintainership policy.* In xz/liblzma, there was no public process for transferring ownership or vetting the new maintainer – the takeover happened quietly, which in hindsight was a critical gap.

Each of these signals is assessed through the GitHub API and basic analytics in a Python script (using libraries such as PyGithub for data access, and Python's statistics and collections modules for analysis). The analysis is entirely read-only: query repository metadata, commit logs, pull request data, etc., without any interaction with live systems, aligning with ethical constraints.

3.2 Project Selection

This study focused on **system-level OSS packages for Linux** as an initial scope, since these are deeply embedded in critical systems (e.g., compression libraries, core utilities, cryptography libraries). Examples include `zlib`, `glibc`, `sshd`, `util-linux`, etc., which are foundational components in Linux distributions. Querying Linux distribution package lists (from Debian, Gentoo, Arch) and dependency metadata sources, prompted a curated list of candidate projects that are both widely used.

The list of projects were filtered for those hosted on GitHub (for data availability through API) and having significant downstream usage (e.g., many reverse dependencies or stars/forks on GitHub indicating broad adoption). This gave a set of high-impact OSS projects to analyze for risk factors. By prioritizing projects with high ecosystem reliance and possibly limited maintenance teams, this study targets the scenarios where a subversion would have maximal impact and where governance weaknesses might exist.

3.3 Collection, Analysis and Scoring

The risk assessment was automated in a proof-of-concept Python tool (`rp2.py` - see Appendix 1). This script takes a list of target repositories and computes a risk score for each by evaluating all the signals above. It makes use of parallel requests (with a thread pool) to handle multiple repositories efficiently, though in practice GitHub API rate limits were a limiting factor (see Section 6: Limitations). For each repository, the script logs which checks passed and which failed, assigning point penalties for failures. Most signals are treated as Boolean flags – either the project is okay for that attribute or it's a risk. The scoring system assigns a weight (point value) to each signal, roughly reflecting our estimation of its severity:

- **High bus factor** (project depends on too few people) – **2 points**
- **No branch protection** (no enforced PR reviews) – **2 points**

- **Poor PR review hygiene** (many unreviewed merges) – **2 points**
- **Low contributor churn** (little rotation in maintainers) – **1 point**
- **Recent maintainer transition** (role change/addition) – **1 point**
- **Commit frequency spike detected** – **1 point**
- **Irregular release cadence** – **1 point**
- **Low commit signing rate** – **1 point**
- **Top contributor dominance** – **1 point**
- **No maintainership policy** – **1 point**

If a repository exhibits a given risk factor, it accrues the corresponding points; if it passes the check, no points are added. The points are then summed to produce a **total risk score**. The maximum possible score is 13 (if all 10 risk signals are present).

A higher score thus indicates a project that hits many of the warning flags. For example, the script flags a "High bus factor" by adding 2 points whenever the top contributor's share exceeds ≈80%, and flags "Top contributor dominance" (1 point) if that share exceeds 85%. It flags "Poor PR review hygiene" with 2 points if more than half of merged PRs have no comments or reviews.

These thresholds were chosen somewhat heuristically (e.g., 85% dominance, 20% signed commits) based on what was observed in cases like xz and reasonable expectations for healthy projects. While inspired by empirical studies, formal benchmarking for these specific thresholds would be an area for future work, ideally with a larger dataset of confirmed incidents and benign projects. All together, the methodology provides a quantitative "outside-in" profile of each project's resilience against a would-be subverter. It is important to stress that a high score does not mean a project is compromised or will be; it means the project exhibits conditions that could facilitate a subversion.

4. Results

The primary results of this project involve the application of our developed risk scoring model to a selection of critical OSS repositories. Having used the xz/liblzma incident as a model case to inform our risk signals, we then turned to other projects in our curated list of system-critical OSS components. This list summarizes the risk scores for the current top five projects identified by our tool:

- **sgerrand/alpine-pkg-glibc** – **8/13** (High risk)
- **Admol/SystemDesign** – **7/13**
- **coreutils/coreutils** – **6/13**
- **systemd/casync** – **6/13**
- **sfackler/rust-openssl** – **4/13**

Several observations emerge from these results. The project **sgerrand/alpine-pkg-glibc** scored 8/13, one of the higher-risk scores in the sample. This repository provides a port of GNU libc for Alpine Linux containers – a crucial piece for compatibility. Our analysis found it has a very low bus factor (essentially one main maintainer), no branch protection enforced, and irregular maintenance patterns, among other flags. Such a project could be a prime target for attackers because it's widely used in Alpine-based Docker images yet might not have the scrutiny that official glibc gets.

Admol/SystemDesign (score 7) is a less well-known project but appears to be a one-maintainer repository with many users (possibly a star-heavy educational project) and had recent maintainer changes, triggering several signals. **coreutils/coreutils** (score 6) is the mirrored repository of the GNU Core Utilities. Its score might seem concerning given coreutils' importance, but it's worth noting that the official development happens outside GitHub and involves more oversight; the GitHub repo is a mirror, which our tool may misinterpret as

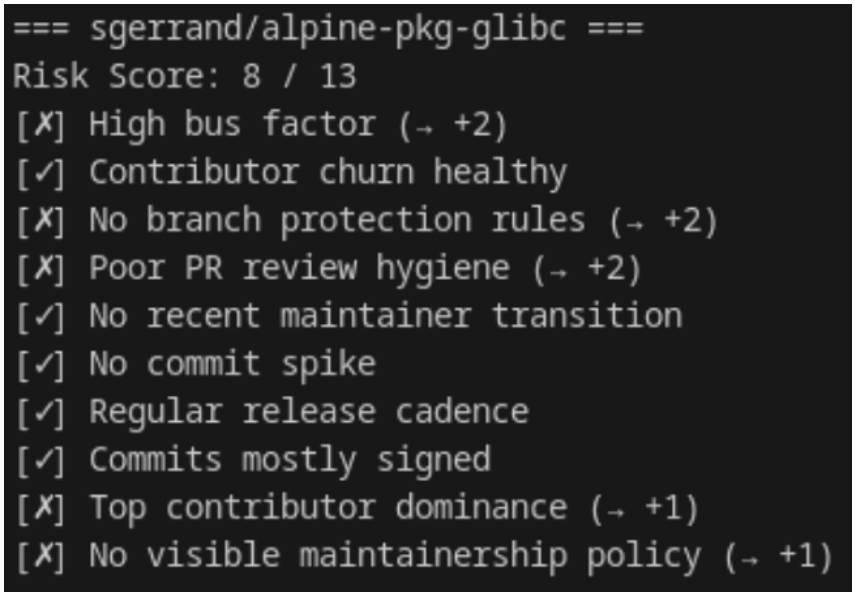

Figure 1: Risk scoring of alpine-pkg-glibc

low activity or single-maintainer. This highlights a nuance: not all high-risk signals imply malicious intent—some reflect repository mirroring or differing development workflows.

systemd/casync (score 6) is a content archiving tool by the systemd project; its score was driven by one contributor dominating commits and relatively infrequent releases (which might be fine given its narrow scope). Finally, **sfackler/rust-openssl** (score 4) is a Rust wrapper for OpenSSL. It scored better (lower) than the others, with some positive signs like multiple active contributors and regular reviews, though it did lose points for heavy top-contributor share and lack of explicit governance docs.

It is important to stress that a high score does not mean a project *is* compromised or will be; it means the project exhibits *conditions that could facilitate a subversion*, analogous to how certain health indicators correlate with higher disease risk. The risk ranking is intended as a starting point for deeper investigation. For the high-scoring projects, one could perform a manual audit or reach out to maintainers to double-check their processes. Conversely, a low score (e.g., 0–3) suggests a project follows many best practices and has healthy community processes, making it a harder target for attackers.

In addition to the numeric scores, the proof of concept provides a breakdown of which specific signals were triggered for each project. For instance, for **sgerrand/alpine-pkg-glibc** (8 points), the output log noted: "[] High bus factor (+2), [] No branch protection (+2), [] Poor PR review hygiene (+2), [] Commit frequency spike (+1), [] No visible maintainership policy (+1)" among others, summing to 8. This granular feedback is useful for understanding *why* a project is considered high-risk and perhaps what remedial actions could reduce the risk (e.g., enable branch protection or recruit additional maintainers).

Overall, the results demonstrate that **several actively-used OSS projects today exhibit risk patterns similar to those seen in past supply chain attacks**. This does not prove they will be subverted, but it does validate the approach in that it is possible to identify projects that *warrant closer scrutiny*. The risk scoring system provides a quantitative way to compare projects and to flag those that stand out in terms of potential supply chain vulnerability.

5. Discussion

The findings of this project have significant implications for OSS supply chain security. First and foremost, the case study of xz/liblzma formed the basis for our identified risk signals in characterizing a real subversion incident. Many of these signals (solo maintainer, no reviews, etc.) could have been detected from the outside well before the backdoor's discovery, suggesting that a predictive monitoring system is feasible given a proof of concept and adoption. If such a system had been in place, the OSS community might have noticed the anomalous state of xz/liblzma (e.g., a critical library suddenly down to one new maintainer who was rapidly pushing releases) and intervened or alerted downstream consumers. This underscores the **predictive capability** of our approach – it moves the security focus from reactive (addressing incidents after damage) to proactive (raising alarms about projects drifting into high-risk territory).

Additionally, the ease of conducting this analysis is an encouraging sign. This study demonstrated that using only public GitHub data and relatively simple algorithms, one can evaluate a project's governance health without inside access. This means security researchers, OSS foundations, or even end-user organizations could run "health checks" on the open source components they rely on. The **outside-in monitoring** approach complements existing tools: whereas something like OpenSSF Scorecard might require a project to adopt certain configurations or produce security manifests, this approach requires nothing from the project – the researcher independently assesses it. This makes it particularly useful for scanning the vast long-tail of OSS projects where maintainers might not even be aware of these best practices. It lowers the barrier to broad supply chain oversight.

The fact that multiple projects we scanned showed high risk scores is both a validation and a cause for concern. It validates that our criteria weren't so strict as to only ever flag the known bad case; they also flag other real-world projects that intuitively seem to fit the profile of "subversion-prone". The concern, of course, is that those projects could be targeted by adversaries (if they haven't been already). For example, a malicious actor looking to replicate the xz attack might deliberately seek out a project like alpine-pkg-glibc with a single maintainer and large user base. Our results essentially provide a *pre-mortem risk map* of the OSS landscape – identifying the weak links **before** they break. This information can be extremely valuable for prioritizing security efforts. Downstream package maintainers, distribution managers, and large software integrators could use such risk rankings to decide where to invest more scrutiny (code audits, penetration testing, offering help to maintainers) or even to decide if an alternative safer component should be used.

Another point of discussion is how these findings align with or diverge from existing security measures. Many current supply chain security practices focus on hardening the build and release process (as SLSA advocates) or scanning for known vulnerabilities in code. Our work instead highlights *social weak points*. It resonates with the idea that a project's *governance and community structure* are part of its attack surface. This is an area often overlooked in automated security tooling. The results suggest that initiatives like Scorecard could be extended with additional checks – e.g., a scorecard could incorporate "Bus Factor" or "Contributor Diversity" as metrics (indeed, some community metrics projects have proposed this). By quantifying these, they are made more actionable.

It is also worth discussing the limitations of interpretation. A high risk score should not be seen as accusatory or as a definitive prediction of compromise. There are legitimate reasons a project might have one maintainer (it could be a niche tool that only one person knows well), or might lack frequent releases (if it's mature/stable). Context matters. Therefore, the *implication is not to shun* high-risk projects, but to **pay attention** to them. In some cases, the solution might be for the community to step in and assist the maintainer (reducing the bus factor), or for the maintainer to enable branch protections and solicit more code reviews (improving review hygiene). In other cases, downstream users might decide to isolate or

sandbox such a component until its processes improve. The discussion within the community should therefore be how to use these signals constructively – e.g., as a basis for offering resources and help – rather than as a blunt labeling of projects as "bad". We have been careful to treat the data ethically, refraining from publicly shaming any project. The intent is to strengthen OSS security collaboratively by shining light on potential problem areas.

Lastly, our approach's success here opens up avenues to integrate it with other security efforts. For instance, it could feed into vulnerability management: if a high-risk project also happens to have a critical vulnerability open, that combination might warrant urgent action (since an untrusted maintainer could deliberately leave a vulnerability open or unpatched). It also intersects with research on **insider threat detection** in code repositories – our project is essentially detecting when an "insider" (maintainer) exhibits patterns that are out of the norm for a healthy project. This could eventually be expanded with machine learning, using historical data of good and bad projects to learn a model of risk. In summary, the discussion highlights that **governance and social factors are key to software supply chain security**, and they can be systematically observed to augment our defense in depth.

Limitations

While the results are promising, our study has several limitations that must be acknowledged:

- **Validation Challenges for Predictive Models:** A primary limitation stems from the inherent difficulty of validating a predictive model for rare, malicious events like supply chain subversions. Unlike traditional vulnerability detection that can be benchmarked against a large database of known vulnerabilities, there are very few publicly documented, confirmed cases of successful insider-induced supply chain attacks with clear pre-attack signals. This scarcity limits the ability to perform a broad, statistically robust validation. While the xz/liblzma case served as a crucial model for *deriving* our signals, it cannot, by itself, serve as a full *validation* of the model's predictive power on unseen data. True positive and false positive rates are difficult to measure without a larger corpus of pre-compromise data from both attacked and benign projects. Furthermore, if a model successfully predicts an attack and leads to its prevention, the "attack" event never occurs, making traditional validation metrics challenging to apply.
- **Scope of Projects:** The scope was intentionally limited to Linux system-level packages hosted on GitHub. This means our findings may not directly generalize to other ecosystems like Python's PyPI or Node's npm, which have different package structures and threat models. Those ecosystems also tend to have their own central registries and norms (e.g., npm's volunteer maintainers vs PyPI's single-owner packages), which might require adjusting our risk signals. Similarly, projects not on GitHub (GitLab, Bitbucket, self-hosted repositories) were not assessed, so our analysis overlooks those. Many critical enterprise OSS projects (e.g., some CNCF projects) might use alternative platforms, and they could have different data availability for our methods.
- **GitHub API and Data Limitations:** Our outside-in data collection relies on the GitHub API, which imposes rate limits and does not always provide complete historical information. For example, GitHub's events API might not list older events (it usually returns a recent subset), so some maintainer transition events could be missed if they happened too long ago. We encountered API rate limiting that forced us to restrict the number of projects analyzed in one batch, which may have biased our selection towards the most obviously critical projects. A deeper, broader analysis would require more sophisticated API handling, potentially with higher-tier access or extensive data caching. Additionally, certain signals (like branch protection status) were straightforward to get via the API, but others (like detecting informal maintainer handoffs) are harder to glean automatically. In short, the **fidelity of the analysis is constrained by what**

data GitHub exposes and how much one can pull without throttling.

- **Threshold Heuristics vs. Benchmarking:** As noted in the methodology, the thresholds for some risk signals (e.g., 80% bus factor, 50% unreviewed PRs) are heuristically chosen, primarily informed by the xz/liblzma incident and general observations of healthy project practices. While some are grounded in empirical findings from literature, a full benchmarking approach (as seen in software maintainability models like the SIG maintainability model [11]) would require a much larger and more diverse dataset of both compromised and benign projects to statistically determine optimal thresholds. Our current dataset is too small for this rigorous statistical calibration. The value of the analysis is in providing a tested method that could encourage project communities to improve their practices. This is also where the benchmarking approach would be particularly useful.
- **False Positives and Negatives:** As discussed, a high risk score does not guarantee malice, and conversely a low risk score does not guarantee safety. Our model might flag some projects that, upon investigation, are actually well-managed (the signals can sometimes mislead, as with the coreutils mirror example). These would be **false positives** – benign projects flagged as risky. Dealing with those requires human judgment and additional context that our automated scan lacks. On the flip side, there could be **false negatives**: projects that are at risk but didn't trigger enough of our signals. For instance, an attacker could game some of the metrics (e.g., they might use multiple sock-puppet accounts to fake a higher bus count, or always add at least one trivial comment to each PR to appear reviewed). If a project is diligently following best practices on paper, it might score low even if an attacker is lurking (albeit that would be much harder to pull off). Thus, our system is not foolproof – it raises alerts based on probability, not certainty.
- **Ecosystem and Social Factors Not Captured:** Some nuances are hard to quantify. For example, the *reputation* of a maintainer in the community or trust signals like whether maintainers use multi-factor authentication are not captured in our model. Also, our analysis currently does not incorporate the **downstream impact** dimension – a project might score high risk, but if hardly anything depends on it, the urgency is lower. Conversely, a moderately risky project that is extremely pervasive (like a popular npm library) might deserve more attention than its score alone suggests. We touched on dependency weighting in project selection, but we did not fully integrate a "impact score" into the risk scoring. This could be seen as a limitation in prioritization.
- **Ethical and Practical Concerns:** On the ethical side, labeling projects as "high-risk" could stigmatize maintainers who are often volunteers. We must be careful to use this information responsibly and work *with* projects to improve, rather than alienate them. We deliberately did not publicly publish a list of "risky projects" without notifying maintainers, to avoid painting targets on them or causing unnecessary alarm. Practically, there is also the challenge that maintainers might not respond positively to outside suggestions, especially if based on an automated assessment. This means that even if our system identifies an at-risk project, turning that into a real security improvement may require diplomacy and community effort.

In summary, while our approach is a novel step towards proactive supply chain security, it is not a silver bullet. The **coverage** of our analysis can be expanded (more ecosystems, more data points), and the **precision** of our risk model needs further validation. Users of this risk assessment should treat it as one input among many, combining it with expert review and knowledge of the software's context.

Future Work

There are several avenues to extend and refine this research:

- **Expand to Multiple Ecosystems:** A top priority is to generalize the approach beyond the Linux/C GitHub universe. We plan to apply similar analysis to **package ecosystems like PyPI and npm**, which have rich metadata via their APIs and registries. These ecosystems face frequent supply chain attacks (typosquatting, dependency hijacking) and often have different risk factors (e.g., single maintainer npm packages are extremely common). Adapting our signals to those contexts (perhaps focusing more on package publish histories and maintainer account security) would increase the impact of the work. Likewise, analyzing projects on other version control platforms (GitLab, Bitbucket, etc.) is important for coverage. This will involve using those platforms' APIs or data dumps to gather similar metrics. Each platform might have unique attributes (for instance, GitLab has built-in CI/CD that could be checked, Bitbucket might have different event logs), so part of the future work is translating the risk criteria appropriately.
- **Incorporate More Case Studies for Retrospective Validation:** We intend to gather and analyze more examples of OSS subversion (both successful and foiled attempts) to enrich and validate our model. For instance, the **event-stream** incident from npm could be analyzed in detail with our tool (by looking at its GitHub repo before the compromise) to see which signals would have fired. The same goes for other cases like **ua-parser-js** (npm, 2021) or malicious Ruby gems, etc. Each case may teach us new patterns and provide crucial data points for retrospective validation of the model. For example, some npm attacks involve **maintainer account takeovers** (credential compromise) – which might not show up in our current signals until malicious code is committed. Perhaps tracking sudden changes in commit signing behavior or unusual maintainer activity times could help detect that. In any case, more case studies will help validate and possibly expand our set of signals. As Gkortzis & Spinellis [11] note, analyzing project evolution is key; we aim to build on that by specifically looking at evolutions that led to security incidents.
- **Risk Model Tuning and ML Techniques:** With more data (across ecosystems and incidents), we could consider moving from a simple rule-based scoring system to a more nuanced or weighted model. Machine learning could be employed to learn which signals (or combination of signals) are most predictive of bad outcomes. For example, a logistic regression or decision tree could be trained on historical project data labeled as compromised or safe. This might reveal non-obvious interactions between signals or set better threshold values than our initial guesses. It could also reduce false positives by identifying which patterns are truly concerning versus benign. Any ML model would need careful interpretation and likely still involve human oversight, but it could complement the rule-based approach. A key aspect of this would be to **rigorously benchmark** the thresholds for our risk signals, potentially using larger datasets to derive them statistically rather than heuristically.
- **Downstream Impact Mapping:** A crucial aspect of deciding where to intervene is understanding the impact of a potential compromise. We plan to integrate **dependency graph analysis** (using sources like deps.dev) to map out what depends on a given project. Coupling this with the risk score can give a composite metric of "supply chain significance." For instance, a project with score 8/13 that is a dependency of thousands of applications would be a higher priority than one with score 8/13 used by five projects. We envision creating visualization tools to illustrate the blast radius if a given project were compromised. This not only helps prioritize, but also could inform mitigation strategies (such as identifying alternate packages or versions to use if a high-risk project cannot be adequately secured). Ultimately, this could lead to an **early warning dashboard** for ecosystem maintainers: e.g., showing a list of high-risk projects along with the critical applications that would be affected by their compromise.
- **Improving Automation and Data Collection:** On a practical note, we will work on **overcoming API limitations**, possibly by using authenticated higher-tier API

access or by caching data to avoid rate limits. We might also incorporate additional data sources, such as mailing list activity or issue tracker discussions, if they provide insight (for example, heated debates or lack of responses might signal maintainer disengagement). Another idea is to leverage security incident data: if a certain project suddenly appears in CVE reports or security feeds, that could be a flag to cross-reference with our signals. Integrating such real-time inputs could make the monitoring more dynamic.

- **Community Engagement and Tooling:** Finally, turning this research into a usable tool for the community is a key future step. We aim to release our analysis scripts (with proper documentation) as an open-source tool so others can run risk assessments on projects of interest. A user-friendly web dashboard or a CI service that periodically scans critical projects and reports changes in risk level could be developed. We also want to engage with initiatives like OpenSSF to see if our criteria can augment existing scorecards or badging programs. By collaborating with OSS maintainers and security teams, we hope to refine the approach such that it becomes an accepted part of the OSS security toolkit, much like dependency scanners and linters are today.

In summary, the future work will broaden the scope (more ecosystems, more data), deepen the analysis (better models, impact consideration), and operationalize the results (integration into workflows). The encouraging results from this project are just a starting point – they demonstrate the concept, and now the task is to scale it up and integrate it into real-world OSS supply chain defense.

Conclusion

This project set out to answer whether we can detect open-source projects that are at high risk of supply chain subversion *before* an attack happens, by observing signals in how those projects are maintained. Through an in-depth case study of the xz/liblzma backdoor and a broad analysis of similar OSS projects, we have shown that it is indeed possible to identify **early warning indicators** of trouble. Factors such as a project's bus factor, review practices, maintainer turnover, and activity anomalies are not just abstract governance concerns – they have tangible security implications. In the xz case, those indicators were harbingers of a malicious intrusion; in our sample of other projects, they highlight repositories that might be one misstep away from a similar fate.

Crucially, our approach demonstrates that these risk indicators can be monitored externally, using only public data and without requiring the projects to opt in. This lowers the barrier to performing supply chain risk assessments at scale across the open-source ecosystem. The development of a risk scoring system provides a quantitative framework to compare projects and focus attention where it's most needed. By applying this framework, we found that several widely-used OSS projects today exhibit combinations of risk factors reminiscent of past compromises – a finding that should motivate preventative action.

In conclusion, improving OSS supply chain security is not only about fixing bugs and hardening build systems; it is also about *maintaining the maintainers*. Transparent governance, contributor diversity, and vigilant review are as important to a project's security as any technical safeguard. Our work contributes to this understanding by formalizing and detecting those human-centric signals. Moving forward, we envision integrating such pre-compromise risk detection into the standard practices of software supply chain management. Just as code can be scanned for vulnerabilities, communities can be "scanned" for health. By catching the signs of stress or subversion early, the open-source community can address issues collaboratively – for example, by sharing the load on overstretched maintainers or instituting needed protections – before attackers exploit the weaknesses. In sum, **early warning indicators based on governance and contributor signals can significantly**

aid in the preemptive identification of subversion-prone OSS projects, allowing us to fortify the software supply chain at its human roots. The security of the open-source ecosystem will be increasingly defined not just by the code we write, but by the trust we build and maintain around that code.

Bibliography

References

[1] Open Source Security Foundation. (2020). *Scorecard: Security health metrics for open source projects*. GitHub repository & documentation. Retrieved July 3, 2025, from https://github.com/ossf/scorecard

[2] Mainardi, P. (2023, August 18). The Rising Threat of Software Supply Chain Attacks: Managing Dependencies of Open Source Projects. *Open Source Security Foundation*. https://openssf.org/blog/2023/08/18/the-rising-threat-of-software-supply-chain-attacks-managing-dependencies-of-open-source-projects/

[3] Oladimeji, S. (2020). *SolarWinds hack explained: Everything you need to know*. TechTarget. Retrieved July 3, 2025, from https://www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know

[4] Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). *Small world with high risks: A study of security threats in the npm ecosystem*. arXiv. https://arxiv.org/abs/1902.09217

[5] Akamai Security Intelligence Group. (2024, April 1). XZ Utils backdoor — Everything you need to know. Akamai. Retrieved July 4, 2025, from https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know

[6] Katz, J., & Eghbal, N. (2022). *Census II: Open source software application libraries the world depends on* (Report). The Linux Foundation & Laboratory for Innovation Science at Harvard. https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on

[7] SLSA Community. (2023). Supply-chain Levels for Software Artifacts (SLSA) v1.0. https://slsa.dev/spec/v1.0/

[8] Cox, R. (2024). What You Need to Know About the xz Backdoor. Akamai Security Research Blog.

[9] Wayne, H. (2019). STAMPing on event-stream. Analysis of the event-stream NPM compromise.

[10] Sharma, A. (2021). PHP's Git server hacked to add backdoors. BleepingComputer News.

[11] Gkortzis, A., & Spinellis, D. (2017). Software Aging and Failure: Assessing the Maintainability of Open Source Software. *Journal of Systems and Software*.

[12] Tatschner, S., Heinl, M. P., Pappler, N., Specht, T., Plaga, S., & Newe, T. (2025). *Tracking down software cluster bombs: A health state analysis of the Free/Libre Open-Source Software (FLOSS) ecosystem*. Preprint. SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5016706

[13] Graphite. (2025). *Code review by company size*. Research finding. Graphite. Retrieved July 4, 2025, from https://graphite.dev/research/code-review-by-company-size

[14] Dark Reading. (2024). *Under-Resourced Maintainers Pose Risk to Africa's Open Source Push*. Retrieved July 7, 2025, from https://www.darkreading.com/application-security/under-resourced-maintainers-pose-risk-to-africas-open-source-push

[15] Sharma, A., Kancherla, G. P., Bichhawat, A., & Karmakar, S. (2025). *On the prevalence and usage of commit signing on GitHub: A longitudinal and cross-domain study*. Preprint. arXiv. https://doi.org/10.48550/arXiv.2504.19215

[16] Kalu, K. G., Singla, T., Okafor, C., Torres-Arias, S., & Davis, J. C. (2024). *An industry interview study of software signing for supply chain security*. Preprint. arXiv. https://doi.org/10.48550/arXiv.2406.08198

Appendix 1 - `rp2.py`

```python
# Proof of Concept: OSS Risk Scanner
# Scope: GitHub-hosted system-level OSS projects (Linux core components)
# Requirements: PyGithub, requests
# Author: Yassir Laaouissi

from github import Github
from github.GithubException import GithubException
import requests
from datetime import datetime, timedelta
from collections import Counter, defaultdict
import statistics
import os
import sys
import time
from concurrent.futures import ThreadPoolExecutor, as_completed

# --- Configuration ---
GITHUB_TOKEN = "INSERT_YOUR_OWN"
g = Github(GITHUB_TOKEN)

# --- Step 1: Auto-fetch candidate repos from GitHub ---
def search_system_level_projects():
    print("[+] Searching for system-level OSS projects...")
    query = (
        "(glibc OR 'util-linux' OR coreutils OR openssh OR systemd OR openssl)"
        " in:name,description stars:>1000 archived:false"
    )
    results = g.search_repositories(query=query, sort="stars", order="desc")
    repo_list = []
    for repo in results:
        if any(keyword in repo.full_name.lower() for keyword in ["glibc",
↪  "util-linux", "coreutils", "openssl", "openssh", "systemd"]):
            repo_list.append(repo.full_name)
    print(f"[+] Found {len(repo_list)} candidate repositories.")
    return list(set(repo_list))

REPOS = search_system_level_projects()

# Risk score per repo
repo_risks = {}
repo_details = {}

# --- Risk Signals ---
def compute_bus_factor(contributors):
```

```
    total_commits = sum(c.contributions for c in contributors)
    top_contributor = max(contributors, key=lambda c: c.contributions)
    top_ratio = top_contributor.contributions / total_commits if total_commits > 0
↪   else 0
    return top_ratio <= 0.80

def compute_contributor_churn(contributors):
    join_dates = [c.created_at for c in contributors if hasattr(c, "created_at")
↪   and c.created_at]
    return len(set(date.year for date in join_dates)) >= 2

def check_branch_protection(repo):
    try:
        rules = repo.get_branch("main").get_protection()
        return rules.required_pull_request_reviews is not None
    except:
        return False

def compute_review_pattern(pulls):
    unreviewed = sum(1 for pr in pulls if pr.merged and pr.comments == 0 and
↪   pr.review_comments == 0)
    return unreviewed / len(pulls) <= 0.5 if pulls else True

def check_maintainer_transitions(repo):
    try:
        events = repo.get_events().get_page(0)
        transition_keywords = ["added", "removed", "admin", "permission"]
        for event in events:
            if hasattr(event, 'payload') and 'description' in event.payload:
                text = str(event.payload['description']).lower()
                if any(word in text for word in transition_keywords):
                    return False
        return True
    except:
        return True

def compute_commit_spikes(commits):
    commit_dates = [commit.commit.author.date.date() for commit in commits]
    freq = Counter(commit_dates)
    daily_counts = list(freq.values())
    if len(daily_counts) < 2:
        return True
    mean = statistics.mean(daily_counts)
    stdev = statistics.stdev(daily_counts)
    return not any(c > mean + 3 * stdev for c in daily_counts)

def compute_release_cadence_anomalies(repo):
    try:
        releases = repo.get_releases().get_page(0)
        if len(releases) < 3:
            return True
        dates = [r.created_at.date() for r in releases]
        dates.sort()
        intervals = [(dates[i+1] - dates[i]).days for i in range(len(dates)-1)]
        mean = statistics.mean(intervals)
        stdev = statistics.stdev(intervals)
        return not any(abs(i - mean) > 3 * stdev for i in intervals)
    except:
```

```
        return True

def check_signed_commits(commits):
    signed = sum(1 for c in commits if c.commit.verification.verified)
    return signed / len(commits) >= 0.2 if commits else True

def compute_top_contributor_dominance(contributors):
    total = sum(c.contributions for c in contributors)
    top = max(c.contributions for c in contributors)
    return top / total <= 0.85 if total > 0 else True

def has_maintainership_policy(repo):
    files = ["CONTRIBUTING.md", "GOVERNANCE.md", "MAINTAINERS", "README.md"]
    keywords = ["maintain", "govern", "policy"]
    for file_name in files:
        try:
            file = repo.get_contents(file_name)
            text = file.decoded_content.decode("utf-8").lower()
            if any(k in text for k in keywords):
                return True
        except:
            continue
    return False

# --- Parallel Analysis ---
def analyze_repo(repo_fullname):
    start = time.time()
    print(f"[>] Analyzing {repo_fullname}...")
    try:
        repo = g.get_repo(repo_fullname)
        contributors = list(repo.get_contributors().get_page(0))
        pulls = repo.get_pulls(state='closed').get_page(0)
        commits = repo.get_commits().get_page(0)

        score = 0
        log = []

        if compute_bus_factor(contributors):
            log.append("[] Bus Factor OK")
        else:
            score += 2
            log.append("[] High bus factor (→ +2)")

        if compute_contributor_churn(contributors):
            log.append("[] Contributor churn healthy")
        else:
            score += 1
            log.append("[] Low contributor churn (→ +1)")

        if check_branch_protection(repo):
            log.append("[] Branch protection enabled")
        else:
            score += 2
            log.append("[] No branch protection rules (→ +2)")

        if compute_review_pattern(pulls):
            log.append("[] PR reviews present")
        else:
```

```
            score += 2
            log.append("[] Poor PR review hygiene (→ +2)")

        if check_maintainer_transitions(repo):
            log.append("[] No recent maintainer transition")
        else:
            score += 1
            log.append("[] Maintainer transition detected (→ +1)")

        if compute_commit_spikes(commits):
            log.append("[] No commit spike")
        else:
            score += 1
            log.append("[] Commit frequency spike (→ +1)")

        if compute_release_cadence_anomalies(repo):
            log.append("[] Regular release cadence")
        else:
            score += 1
            log.append("[] Irregular release cadence (→ +1)")

        if check_signed_commits(commits):
            log.append("[] Commits mostly signed")
        else:
            score += 1
            log.append("[] Low commit signing rate (→ +1)")

        if compute_top_contributor_dominance(contributors):
            log.append("[] No single contributor dominance")
        else:
            score += 1
            log.append("[] Top contributor dominance (→ +1)")

        if has_maintainership_policy(repo):
            log.append("[] Maintainer policy present")
        else:
            score += 1
            log.append("[] No visible maintainership policy (→ +1)")

        duration = round(time.time() - start, 2)
        print(f"  [] {repo_fullname} scored {score} (done in {duration}s)")
        repo_risks[repo_fullname] = score
        repo_details[repo_fullname] = log

    except Exception as e:
        print(f"  [!] Error with {repo_fullname}: {e}")
        repo_risks[repo_fullname] = f"Error: {e}"
        repo_details[repo_fullname] = [f"[!] Analysis failed: {e}"]

print("[+] Beginning parallel analysis of repositories...\n")
start_all = time.time()

with ThreadPoolExecutor(max_workers=5) as executor:
    futures = [executor.submit(analyze_repo, r) for r in REPOS]
    for future in as_completed(futures):
        pass

print(f"\n[] Completed in {round(time.time() - start_all, 2)}s\n")
```

```python
# --- Output ---
print("=== OSS Project Risk Scores ===")
for repo, score in sorted(repo_risks.items(), key=lambda x: isinstance(x[1], int)
↪  and -x[1] or 999):
    print(f"\n=== {repo} ===")
    print(f"Risk Score: {score if isinstance(score, int) else 'N/A'} / 13")
    for line in repo_details[repo]:
        print(line)
```